<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2  The Sage Category Framework</td>
<td>3</td>
</tr>
<tr>
<td>3  Maps and Morphisms</td>
<td>157</td>
</tr>
<tr>
<td>4  Individual Categories</td>
<td>249</td>
</tr>
<tr>
<td>5  Functorial constructions</td>
<td>1345</td>
</tr>
<tr>
<td>6  Examples of parents using categories</td>
<td>1391</td>
</tr>
<tr>
<td>7  Internals</td>
<td>1487</td>
</tr>
<tr>
<td>8  Indices and Tables</td>
<td>1503</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>1505</td>
</tr>
<tr>
<td>Index</td>
<td>1509</td>
</tr>
</tbody>
</table>
CHAPTER ONE

INTRODUCTION

1.1 Sage categories quickref

- `sage.categories.primer?` a primer on Elements, Parents, and Categories
- `sage.categories.tutorial?` a tutorial on Elements, Parents, and Categories
- `Category?` technical background on categories
- `Sets(), Semigroups(), Algebras(QQ)` some categories
- `SemiGroups().example()` sample implementation of a semigroup
- `Hom(A, B), End(A, Algebras())` homomorphisms sets
- `tensor, cartesian_product` functorial constructions

Module layout:

- `sage.categories.basic` the basic categories
- `sage.categories.all` all categories
- `sage.categories.semigroups` the Semigroups() category
- `sage.categories.examples.semigroups` the example of Semigroups()
- `sage.categories.homset` morphisms,...
- `sage.categories.map`
- `sage.categories.morphism`
- `sage.categories.functors`
- `sage.categories.cartesian_product` functorial constructions
- `sage.categories.tensor`
- `sage.categories.dual`
THE SAGE CATEGORY FRAMEWORK

2.1 Elements, parents, and categories in Sage: a primer

Contents

- Elements, parents, and categories in Sage: a primer
  - Abstract
  - Introduction: Sage as a library of objects and algorithms
  - A bit of help from abstract algebra
  - A bit of help from computer science
  - Sage categories
  - Case study
  - Specifying the category of a parent
  - Scaling further: functorial constructions, axioms, …
  - Writing a new category

2.1.1 Abstract

The purpose of categories in Sage is to translate the mathematical concept of categories (category of groups, of vector spaces, …) into a concrete software engineering design pattern for:

- organizing and promoting generic code
- fostering consistency across the Sage library (naming conventions, doc, tests)
- embedding more mathematical knowledge into the system

This design pattern is largely inspired from Axiom and its followers (Aldor, Fricas, MuPAD, …). It differs from those by:

- blending in the Magma inspired concept of Parent/Element
- being built on top of (and not into) the standard Python object oriented and class hierarchy mechanism.

This did not require changing the language, and could in principle be implemented in any language supporting the creation of new classes dynamically.
The general philosophy is that Building mathematical information into the system yields more expressive, more conceptual and, at the end, easier to maintain and faster code (within a programming realm; this would not necessarily apply to specialized libraries like gmp!).

**One line pitch for mathematicians**

Categories in Sage provide a library of interrelated bookshelves, with each bookshelf containing algorithms, tests, documentation, or some mathematical facts about the objects of a given category (e.g. groups).

**One line pitch for programmers**

Categories in Sage provide a large hierarchy of abstract classes for mathematical objects. To keep it maintainable, the inheritance information between the classes is not hardcoded but instead reconstructed dynamically from duplication free semantic information.

### 2.1.2 Introduction: Sage as a library of objects and algorithms

The Sage library, with more than one million lines of code, documentation, and tests, implements:

- Thousands of different kinds of objects (classes):
  - Integers, polynomials, matrices, groups, number fields, elliptic curves, permutations, morphisms, languages, … and a few racoons …
- Tens of thousands methods and functions:
  - Arithmetic, integer and polynomial factorization, pattern matching on words, …

**Some challenges**

- How to organize this library?
  
  One needs some bookshelves to group together related objects and algorithms.
- How to ensure consistency?

  Similar objects should behave similarly:

  ```sage
  sage: Permutations(5).cardinality()
  120
  ```

  ```sage
  sage: GL(2,2).cardinality()
  → # needs sage.libs.gap sage.modules
  6
  ```

  ```sage
  sage: A = random_matrix(ZZ, 6, 3, x=7)
  → # needs sage.modules
  ```

  ```sage
  sage: L = LatticePolytope(A.rows())
  → # needs sage.geometry.polyhedron sage.modules
  ```

  ```sage
  sage: L.npoints()
  # oops! # random
  → # needs sage.geometry.polyhedron sage.modules
  37
  ```
```python
>>> from sage.all import *
>>> Permutations(Integer(5)).cardinality()
120

>>> GL(Integer(2), Integer(2)).cardinality() → # needs sage.libs.gap sage.modules
6

>>> A = random_matrix(ZZ, Integer(6), Integer(3), x=Integer(7)) → # needs sage.modules

>>> L = LatticePolytope(A.rows()) → needs sage.geometry.polyhedron sage.modules

>>> L.npoints() → needs sage.geometry.polyhedron sage.modules
37

• How to ensure robustness?
• How to reduce duplication?

Example: binary powering:

```python
sage: m = 3
sage: m^8 == m*m*m*m*m*m*m == ((m^2)^2)^2
True

>>> from sage.all import *
>>> m = Integer(3)
>>> m**Integer(8) == m*m*m*m*m*m*m == ((m**Integer(2))**Integer(2))**Integer(2)
True

sage: # needs sage.modules
sage: m = random_matrix(QQ, 4, algorithm='echelonizable', rank=3, upper_bound=60)
sage: m^8 == m*m*m*m*m*m*m == ((m^2)^2)^2
True

>>> from sage.all import *
>>> # needs sage.modules
>>> m = random_matrix(QQ, Integer(4), algorithm='echelonizable', rank=Integer(3), upper_bound=Integer(60))
>>> m**Integer(8) == m*m*m*m*m*m*m == ((m**Integer(2))**Integer(2))**Integer(2)
True
```

We want to implement binary powering only once, as *generic* code that will apply in all cases.
2.1.3 A bit of help from abstract algebra

The hierarchy of categories

What makes binary powering work in the above examples? In both cases, we have a set endowed with a multiplicative binary operation which is associative and which has a unit element. Such a set is called a monoid, and binary powering (to a non-negative power) works generally for any monoid.

Sage knows about monoids:

```python
sage: Monoids()
Category of monoids
```

and sure enough, binary powering is defined there:

```python
sage: m._pow_int.__module__
'needs sage.modules'.
```

That’s our bookshelf! And it’s used in many places:

```python
sage: GL(2, ZZ) in Monoids()
True
sage: NN in Monoids()
True
```

For a less trivial bookshelf we can consider euclidean rings: once we know how to do euclidean division in some set \( R \), we can compute gcd’s in \( R \) generically using the Euclidean algorithm.

We are in fact very lucky: abstract algebra provides us right away with a large and robust set of bookshelves which is the result of centuries of work of mathematicians to identify the important concepts. This includes for example:

```python
sage: Sets()
Category of sets
sage: Groups()
Category of groups
sage: Rings()
```
Each of the above is called a category. It typically specifies what are the operations on the elements, as well as the axioms satisfied by those operations. For example the category of groups specifies that a group is a set endowed with a binary operation (the multiplication) which is associative and admits a unit and inverses.

Each set in Sage knows which bookshelf of generic algorithms it can use, that is to which category it belongs:

```python
sage: G = GL(2, ZZ)  # needs sage.modules

sage: G.category()  # needs sage.modules
Category of infinite groups
```

In fact a group is a semigroup, and Sage knows about this:

```python
sage: Groups().is_subcategory(Semigroups())
True

sage: G in Semigroups()  # needs sage.modules
True
```

2.1. Elements, parents, and categories in Sage: a primer
Altogether, our group gets algorithms from a bunch of bookshelves:

```python
sage: G.categories()
# needs sage.modules
[Category of infinite groups, Category of groups, Category of monoids, ...
, Category of magmas,
Category of infinite sets, ...]
```

Those can be viewed graphically:

```python
sage: g = Groups().category_graph()
# needs sage.graphs
sage: g.set_latex_options(format="dot2tex")
# needs sage.graphs sage.modules sage.plot
sage: view(g) # not tested
```

In case `dot2tex` is not available, you can use instead:

```python
sage: g.show(vertex_shape=\texttt{None}, figsize=20)
# needs sage.graphs sage.modules sage.plot
```

Here is an overview of all categories in Sage:
Wrap-up: generic algorithms in Sage are organized in a hierarchy of bookshelves modelled upon the usual hierarchy of categories provided by abstract algebra.

**Elements, Parents, Categories**

**Parent**

A *parent* is a Python instance modelling a set of mathematical elements together with its additional (algebraic) structure. Examples include the ring of integers, the group $S_3$, the set of prime numbers, the set of linear maps between two given vector spaces, and a given finite semigroup.

These sets are often equipped with additional structure: the set of all integers forms a ring. The main way of encoding this information is specifying which categories a parent belongs to.

It is completely possible to have different Python instances modelling the same set of elements. For example, one might want to consider the ring of integers, or the poset of integers under their standard order, or the poset of integers under divisibility, or the semiring of integers under the operations of maximum and addition. Each of these would be a different instance, belonging to different categories.

For a given model, there should be a unique instance in Sage representing that parent:

```python
sage: IntegerRing() is IntegerRing()
True
```

**Element**

An *element* is a Python instance modelling a mathematical element of a set.

Examples of element include 5 in the integer ring, $x^3 - x$ in the polynomial ring in $x$ over the rationals, $4 + O(3^3)$ in the 3-adics, the transposition $(12)$ in $S_3$, and the identity morphism in the set of linear maps from $\mathbb{Q}^3$ to $\mathbb{Q}^3$.

Every element in Sage has a parent. The standard idiom in Sage for creating elements is to create their parent, and then provide enough data to define the element:

```python
sage: R = PolynomialRing(ZZ, name='x')
sage: R([1, 2, 3])
3*x^2 + 2*x + 1
```

```python
>>> from sage.all import *

>>> R = PolynomialRing(ZZ, name='x')
>>> R([Integer(1), Integer(2), Integer(3)])
3*x^2 + 2*x + 1
```
One can also create elements using various methods on the parent and arithmetic of elements:

```sage
sage: x = R.gen()
sage: 1 + 2*x + 3*x^2
3*x^2 + 2*x + 1
```

```python
>>> from sage.all import *

>>> x = R.gen()

>>> Integer(1) + Integer(2)*x + Integer(3)*x**Integer(2)
3*x^2 + 2*x + 1
```

Unlike parents, elements in Sage are not necessarily unique:

```sage
sage: ZZ(5040) is ZZ(5040)
False
```

```python
>>> from sage.all import *

>>> ZZ(Integer(5040)) is ZZ(Integer(5040))
False
```

Many parents model algebraic structures, and their elements support arithmetic operations. One often further wants to do arithmetic by combining elements from different parents: adding together integers and rationals for example. Sage supports this feature using coercion (see `sage.structure.coerce` for more details).

It is possible for a parent to also have simultaneously the structure of an element. Consider for example the monoid of all finite groups, endowed with the Cartesian product operation. Then, every finite group (which is a parent) is also an element of this monoid. This is not yet implemented, and the design details are not yet fixed but experiments are underway in this direction.

**Todo:** Give a concrete example, typically using `ElementWrapper`.

## Category

A *category* is a Python instance modelling a mathematical category.

Examples of categories include the category of finite semigroups, the category of all (Python) objects, the category of $\mathbb{Z}$-algebras, and the category of Cartesian products of $\mathbb{Z}$-algebras:

```sage
sage: FiniteSemigroups()
Category of finite semigroups
sage: Objects()
Category of objects
sage: Algebras(ZZ)
Category of algebras over Integer Ring
sage: Algebras(ZZ).CartesianProducts()
Category of Cartesian products of algebras over Integer Ring
```

```python
>>> from sage.all import *

>>> FiniteSemigroups()
Category of finite semigroups
>>> Objects()
Category of objects
>>> Algebras(ZZ)
Category of algebras over Integer Ring
```

(continues on next page)
Mind the 's' in the names of the categories above; GroupAlgebra and GroupAlgebras are distinct things.

Every parent belongs to a collection of categories. Moreover, categories are interrelated by the super categories relation. For example, the category of rings is a super category of the category of fields, because every field is also a ring.

A category serves two roles:

- to provide a model for the mathematical concept of a category and the associated structures: homsets, morphisms, functorial constructions, axioms.
- to organize and promote generic code, naming conventions, documentation, and tests across similar mathematical structures.

**CategoryObject**

Objects of a mathematical category are not necessarily parents. Parent has a superclass that provides a means of modeling such.

For example, the category of schemes does not have a faithful forgetful functor to the category of sets, so it does not make sense to talk about schemes as parents.

**Morphisms, Homsets**

As category theorists will expect, Morphisms and Homsets will play an ever more important role, as support for them will improve.

Much of the mathematical information in Sage is encoded as relations between elements and their parents, parents and their categories, and categories and their super categories:

```
sage: 1.parent()
Integer Ring

sage: ZZ
Integer Ring

sage: ZZ.category()
Join of Category of Dedekind domains
    and Category of euclidean domains
    and Category of noetherian rings
    and Category of infinite enumerated sets
    and Category of metric spaces

sage: ZZ.categories()
[Join of Category of Dedekind domains
    and Category of euclidean domains
    and Category of noetherian rings
    and Category of infinite enumerated sets
    and Category of metric spaces,
    Category of Dedekind domains,
    Category of euclidean domains, Category of principal ideal domains,
    Category of unique factorization domains, Category of gcd domains,
```
Category of integral domains, Category of domains, ...
Category of commutative rings, Category of rings, ...
Category of magmas and additive magmas, ...
Category of monoids, Category of semigroups,
Category of commutative magmas, Category of unital magmas, Category of magmas,
Category of commutative additive groups, ..., Category of additive magmas,
Category of infinite enumerated sets, Category of enumerated sets,
Category of infinite sets, Category of metric spaces,
Category of topological spaces, Category of sets,
Category of sets with partial maps,
Category of objects]

```
sage: g = EuclideanDomains().category_graph()  # needs sage.graphs
 benefited from sage.graphs
sage: g.set_latex_options(format="dot2tex")   # needs sage.graphs sage.plot
sage: view(g)  # not tested  # needs sage.graphs sage.plot
```

```python
>>> from sage.all import *
>>> Integer(1).parent()
Integer Ring

>>> ZZ
Integer Ring

>>> ZZ.category()
Join of Category of Dedekind domains
   and Category of euclidean domains
   and Category of noetherian rings
   and Category of infinite enumerated sets
   and Category of metric spaces

>>> ZZ.categories()
[Join of Category of Dedekind domains
   and Category of euclidean domains
   and Category of noetherian rings
   and Category of infinite enumerated sets
   and Category of metric spaces,
Category of Dedekind domains,
Category of euclidean domains, Category of principal ideal domains,
Category of unique factorization domains, Category of gcd domains,
Category of integral domains, Category of domains, ...
Category of commutative rings, Category of rings, ...
Category of magmas and additive magmas, ...
Category of monoids, Category of semigroups,
Category of commutative magmas, Category of unital magmas, Category of magmas,
Category of commutative additive groups, ..., Category of additive magmas,
Category of infinite enumerated sets, Category of enumerated sets,
Category of infinite sets, Category of metric spaces,
Category of topological spaces, Category of sets,
Category of sets with partial maps,
Category of objects]
```

```
>>> g = EuclideanDomains().category_graph()  # needs sage.graphs
 benefited from sage.graphs
```
2.1.4 A bit of help from computer science

Hierarchy of classes

How are the bookshelves implemented in practice?

Sage uses the classical design paradigm of Object Oriented Programming (OOP). Its fundamental principle is that any object that a program is to manipulate should be modelled by an instance of a class. The class implements:

- a data structure: which describes how the object is stored,
- methods: which describe the operations on the object.

The instance itself contains the data for the given object, according to the specified data structure.

Hence, all the objects mentioned above should be instances of some classes. For example, an integer in Sage is an instance of the class Integer (and it knows about it!):

```python
sage: i = 12
sage: type(i)
<class 'sage.rings.integer.Integer'>
```

Applying an operation is generally done by calling a method:

```python
sage: i.factor()
2^2 * 3

sage: # needs sage.symbolic
sage: x = var('x')

sage: p = 6*x^2 + 12*x + 6
sage: type(p)
<class 'sage.symbolic.expression.Expression'>

sage: p.factor()
6*(x + 1)^2

sage: # needs sage.symbolic
sage: R.<x> = PolynomialRing(QQ, sparse=True)

sage: pQ = R(p)

sage: type(pQ)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category.element_class'>

sage: pQ.factor()
(6) * (x + 1)^2
```

(continues on next page)
Factoring integers, expressions, or polynomials are distinct tasks, with completely different algorithms. Yet, from a user (or caller) point of view, all those objects can be manipulated alike. This illustrates the OOP concepts of polymorphism, data abstraction, and encapsulation.

Let us be curious, and see where some methods are defined. This can be done by introspection:

```
sage: i._mul_?
# not tested
```

For plain Python methods, one can also just ask in which module they are implemented:

```
sage: i._pow_.__module__  # not tested (Issue #24275)
'sage.categories.semisrroups'

sage: pQ._mul_.__module__  # needs sage.symbolic
'sage.rings.polynomial.polynomial_element_generic'

sage: pQ._pow_.__module__  # not tested (Issue #24275)
# needs sage.symbolic
```
We see that integers and polynomials have each their own multiplication method: the multiplication algorithms are indeed unrelated and deeply tied to their respective datastructures. On the other hand, as we have seen above, they share the same powering method because the set \( \mathbb{Z} \) of integers, and the set \( \mathbb{Q}[x] \) of polynomials are both semigroups. Namely, the class for integers and the class for polynomials both derive from an *abstract class* for semigroup elements, which factors out the *generic* methods like \_pow\_. This illustrates the use of hierarchy of classes to share common code between classes having common behaviour.

OOP design is all about isolating the objects that one wants to model together with their operations, and designing an appropriate hierarchy of classes for organizing the code. As we have seen above, the design of the class hierarchy is easy since it can be modelled upon the hierarchy of categories (bookshelves). Here is for example a piece of the hierarchy of classes for an element of a group of permutations:

```plaintext
sage: P = Permutations(4)
sage: m = P.an_element()
sage: for cls in m.__class__.mro(): print(cls)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
<class 'sage.combinat.permutation.StandardPermutations_n.Element'>
<class 'sage.combinat.permutation.Permutation'>
...  
<class 'sage.categories.groups.Groups.element_class'>
<class 'sage.categories.monoids.Monoids.element_class'>
...  
<class 'sage.categories.semigroups.Semigroups.element_class'>
...

On the top, we see concrete classes that describe the data structure for matrices and provide the operations that are tied to this data structure. Then follow abstract classes that are attached to the hierarchy of categories and provide generic algorithms.

2.1. Elements, parents, and categories in Sage: a primer
The full hierarchy is best viewed graphically:

```python
sage: g = class_graph(m.__class__) #--
"needs sage.combinat sage.graphs"
sage: g.set_latex_options(format="dot2tex") #--
"needs sage.combinat sage.graphs sage.plot"
sage: view(g) # not tested #--
"needs sage.combinat sage.graphs sage.plot"
```

Parallel hierarchy of classes for parents

Let us recall that we do not just want to compute with elements of mathematical sets, but with the sets themselves:

```python
sage: ZZ.one()
1
sage: R = QQ['x,y']
sage: R.krull_dimension()
2
sage: A = R.quotient( R.ideal(x^2 - 2) )
sage: A.krull_dimension() # todo: not implemented
```

Here are some typical operations that one may want to carry on various kinds of sets:

- The set of permutations of 5, the set of rational points of an elliptic curve: counting, listing, random generation
- A language (set of words): rationality testing, counting elements, generating series
- A finite semigroup: left/right ideals, center, representation theory
- A vector space, an algebra: Cartesian product, tensor product, quotient

Hence, following the OOP fundamental principle, parents should also be modelled by instances of some (hierarchy of) classes. For example, our group $G$ is an instance of the following class:

```python
sage: G = GL(2, ZZ) #--
"needs sage.modules"
sage: type(G) #--
"<class 'sage.groups.matrix_gps.linear_gap.LinearMatrixGroup_gap_with_category'>"
```
Here is a piece of the hierarchy of classes above it:

```
sage: for cls in G._class__.mro(): print(cls)  # needs sage.libs.gap sage.modules
<class 'sage.groups.matrix_gps.linear_gap.LinearMatrixGroup_gap_with_category'>
...
<class 'sage.categories.groups.Groups.parent_class'>
<class 'sage.categories.monoids.Monoids.parent_class'>
<class 'sage.categories.semigroups.Semigroups.parent_class'>
...
```

Note that the hierarchy of abstract classes is again attached to categories and parallel to that we had seen for the elements. This is best viewed graphically:

```
sage: g = class_graph(m.__class__)  # needs sage.combinat sage.graphs sage.modules sage.plot
sage: g.relabel(lambda x: x.replace("_",r"\_")))  # not tested
sage: view(g)
```

Note: This is a progress upon systems like Axiom or MuPAD where a parent is modelled by the class of its elements; this oversimplification leads to confusion between methods on parents and elements, and makes parents special; in particular it prevents potentially interesting constructions like “groups of groups”.

2.1. Elements, parents, and categories in Sage: a primer 17
2.1.5 Sage categories

Why this business of categories? And to start with, why don't we just have a good old hierarchy of classes `Group`, `Semigroup`, `Magma`, ...?

Dynamic hierarchy of classes

As we have just seen, when we manipulate groups, we actually manipulate several kinds of objects:

- groups
- group elements
- morphisms between groups
- and even the category of groups itself!

Thus, on the group bookshelf, we want to put generic code for each of the above. We therefore need three, parallel hierarchies of abstract classes:

- `Group`, `Monoid`, `Semigroup`, `Magma`, ...
- `GroupElement`, `MonoidElement`, `SemigroupElement`, `MagmaElement`, ...
- `GroupMorphism`, `MonoidMorphism`, `SemigroupMorphism`, `MagmaMorphism`, ...

(and in fact many more as we will see).

We could implement the above hierarchies as usual:

```python
class Group(Monoid):
    # generic methods that apply to all groups

class GroupElement(MonoidElement):
    # generic methods that apply to all group elements

class GroupMorphism(MonoidMorphism):
    # generic methods that apply to all group morphisms
```

And indeed that's how it was done in Sage before 2009, and there are still many traces of this. The drawback of this approach is duplication: the fact that a group is a monoid is repeated three times above!

Instead, Sage now uses the following syntax, where the `Groups` bookshelf is structured into units with nested classes:

```python
class Groups(Category):
    def super_categories(self):
        return [Monoids(), ...]

class ParentMethods:
    # generic methods that apply to all groups

class ElementMethods:
    # generic methods that apply to all group elements

class MorphismMethods:
    # generic methods that apply to all group morphisms (not yet implemented)

class SubcategoryMethods:
    # generic methods that apply to all subcategories of Groups()
```
With this syntax, the information that a group is a monoid is specified only once, in the `Category.super_categories()` method. And indeed, when the category of inverse unital magmas was introduced, there was a single point of truth to update in order to reflect the fact that a group is an inverse unital magma:

```
sage: Groups().super_categories()
[Category of monoids, Category of inverse unital magmas]
```

The price to pay (there is no free lunch) is that some magic is required to construct the actual hierarchy of classes for parents, elements, and morphisms. Namely, `Groups.ElementMethods` should be seen as just a bag of methods, and the actual class `Groups().element_class` is constructed from it by adding the appropriate super classes according to `Groups().super_categories()`:

```
sage: Groups().element_class
<class 'sage.categories.groups.Groups.element_class'>
sage: Groups().element_class.__bases__
(<class 'sage.categories.monoids.Monoids.element_class'>,
 <class 'sage.categories.magmas.Magmas.Unital.Inverse.element_class'>)
```

We now see that the hierarchy of classes for parents and elements is parallel to the hierarchy of categories:

```
sage: Groups().all_super_categories()
[Category of groups,  
 Category of monoids,  
 Category of semigroups,  
 ...  
 Category of magmas,  
 Category of sets,  
 ...]
```

```
sage: for cls in Groups().element_class.mro(): print(cls)
<class 'sage.categories.groups.Groups.element_class'>
<class 'sage.categories.monoids.Monoids.element_class'>
<class 'sage.categories.semigroups.Semigroups.element_class'>
...
<class 'sage.categories.magmas.Magmas.element_class'>
```

```
sage: for cls in Groups().parent_class.mro(): print(cls)
<class 'sage.categories.groups.Groups.parent_class'>
<class 'sage.categories.monoids.Monoids.parent_class'>
<class 'sage.categories.semigroups.Semigroups.parent_class'>
...
<class 'sage.categories.magmas.Magmas.parent_class'>
```
Another advantage of building the hierarchy of classes dynamically is that, for parametrized categories, the hierarchy may depend on the parameters. For example an algebra over $\mathbb{Q}$ is a $\mathbb{Q}$-vector space, but an algebra over $\mathbb{Z}$ is not (it is just a $\mathbb{Z}$-module)!

**Note:** At this point this whole infrastructure may feel like overdesigning, right? We felt like this too! But we will see later that, once one gets used to it, this approach scales very naturally.

From a computer science point of view, this infrastructure implements, on top of standard multiple inheritance, a dynamic composition mechanism of mixin classes (Wikipedia article Mixin), governed by mathematical properties.

For implementation details on how the hierarchy of classes for parents and elements is constructed, see `Category`.

### On the category hierarchy: subcategories and super categories

We have seen above that, for example, the category of sets is a super category of the category of groups. This models the fact that a group can be unambiguously considered as a set by forgetting its group operation. In object-oriented parlance, we want the relation “a group is a set”, so that groups can directly inherit code implemented on sets.

Formally, a category $\mathcal{C}$ is a super category of a category $\mathcal{D}$ if Sage considers any object of $\mathcal{D}$ to be an object of $\mathcal{C}$, up to an implicit application of a canonical functor from $\mathcal{D}$ to $\mathcal{C}$. This functor is normally an inclusion of categories or a forgetful functor. Reciprocally, $\mathcal{D}$ is said to be a subcategory of $\mathcal{C}$.

**Warning:** This terminology deviates from the usual mathematical definition of subcategory and is subject to change. Indeed, the forgetful functor from the category of groups to the category of sets is not an inclusion of categories, as it is not injective: a given set may admit more than one group structure. See Issue #16183 for more details. The name supercategory is also used with a different meaning in certain areas of mathematics.
Categories are instances and have operations

Note that categories themselves are naturally modelled by instances because they can have operations of their own. An important one is:

```python
sage: Groups().example() #...
← needs sage.modules
General Linear Group of degree 4 over Rational Field
```

which gives an example of object of the category. Besides illustrating the category, the example provides a minimal template for implementing a new object in the category:

```python
sage: S = Semigroups().example(); S
An example of a semigroup: the left zero semigroup
```

Its source code can be obtained by introspection:

```python
sage: ...
# not tested
>>> from sage.all import *
>>> S
# not tested
```

This example is also typically used for testing generic methods. See `Category.example()` for more.

Other operations on categories include querying the super categories or the axioms satisfied by the operations of a category:

```python
sage: Groups().super_categories()
[Category of monoids, Category of inverse unital magmas]
sage: Groups().axioms()
frozenset({'Associative', 'Inverse', 'Unital'})
```

or constructing the intersection of two categories, or the smallest category containing them:

```python
sage: Groups() & FiniteSets()
Category of finite groups
sage: Algebras(QQ) | Groups()
Category of monoids
```

(continues on next page)
Specifications and generic documentation

Categories do not only contain code but also the specifications of the operations. In particular a list of mandatory and optional methods to be implemented can be found by introspection with:

```python
sage: Groups().required_methods()
{'element': {'optional': [ '_mul_' ], 'required': []},
 'parent': {'optional': [], 'required': [ '__contains__' ]})
```

Documentation about those methods can be obtained with:

```python
sage: G = Groups()
sage: G.element_class._mul_?
# not tested
sage: G.parent_class.one?
# not tested
```

See also the `abstract_method()` decorator.

**Warning:** Well, more precisely, that’s how things should be, but there is still some work to do in this direction. For example, the inverse operation is not specified above. Also, we are still missing a good programmatic syntax to specify the input and output types of the methods. Finally, in many cases the implementer must provide at least one of two methods, each having a default implementation using the other one (e.g. listing or iterating for a finite enumerated set); there is currently no good programmatic way to specify this.

Generic tests

Another feature that parents and elements receive from categories is generic tests; their purpose is to check (at least to some extent) that the parent satisfies the required mathematical properties (is my semigroup indeed associative?) and is implemented according to the specifications (does the method `an_element` indeed return an element of the parent?):

```python
sage: S = FiniteSemigroups().example(alphabet=('a', 'b'))
sage: TestSuite(S).run( verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
```
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass

>>> from sage.all import *
>>> S = FiniteSemigroups().example(alphabet=('a', 'b'))
>>> TestSuite(S).run(verbosity = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
  running ._test_elements_eq_reflexive() . . . pass
  running ._test_elements_eq_symmetric() . . . pass
  running ._test_elements_eq_transitive() . . . pass
  running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
Tests can be run individually:
sage: S._test_associativity()

>>> from sage.all import *

>>> S._test_associativity()

Here is how to access the code of this test:

2.1. Elements, parents, and categories in Sage: a primer 23
Here is how to run the test on all elements:

```python
sage: L = S.list()
sage: S._test_associativity(elements=L)
```

See `TestSuite` for more information.

Let us see what happens when a test fails. Here we redefine the product of \( S \) to something definitely not associative:

```python
sage: S.product = lambda x, y: S("({}, {}, {})")
```

And rerun the test:

```python
sage: S._test_associativity(elements=L)
```

We can recover instantly the actual values of \( x, y, z \), that is, a counterexample to the associativity of our broken semigroup, using post mortem introspection with the Python debugger `pdb` (this does not work yet in the notebook):

```python
sage: import pdb
```

(continues on next page)
\[((aa)a)\]
(Pdb) p x * (y * z)
'(a(aa))''

```
>>> from sage.all import *
>>> import pdb
>>>
pdb.pm()  # not tested
```

```
> /opt/sage-5.11.rc1/local/lib/python/unittest/case.py(424)assertTrue()
-> raise self.failureException(msg)
(Pdb) u
> /opt/sage-5.11.rc1/local/lib/python2.7/site-packages/sage/categories/semigroups.
-> testerassertTrue((x * y) * z == x * (y * z))
(Pdb) p x, y, z
('a', 'a', 'a')
(Pdb) p (x * y) * z
'(aa)a'
(Pdb) p x * (y * z)
'(a(aa))'
```

**Wrap-up**

- Categories provide a natural hierarchy of bookshelves to organize not only code, but also specifications and testing tools.
- Everything about, say, algebras with a distinguished basis is gathered in `AlgebrasWithBasis` or its super categories. This includes properties and algorithms for elements, parents, morphisms, but also, as we will see, for constructions like Cartesian products or quotients.
- The mathematical relations between elements, parents, and categories translate dynamically into a traditional hierarchy of classes.
- This design enforces robustness and consistency, which is particularly welcome given that Python is an interpreted language without static type checking.

### 2.1.6 Case study

In this section, we study an existing parent in detail; a good followup is to go through the `sage.categories.tutorial` or the thematic tutorial on coercion and categories ("How to implement new algebraic structures in Sage") to learn how to implement a new one!

We consider the example of finite semigroup provided by the category:

```
sage: S = FiniteSemigroups().example(); S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
```

```
>>> S
```

```
>>> from sage.all import *
>>> S = FiniteSemigroups().example(); S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
>>> S
```

2.1. Elements, parents, and categories in Sage: a primer 25
Where do all the operations on $S$ and its elements come from?

```python
sage: x = S('a')

>>> from sage.all import *
>>> x = S('a')
```

_repr_ is a technical method which comes with the data structure (ElementWrapper); since it’s implemented in Cython, we need to use Sage’s introspection tools to recover where it’s implemented:

```python
sage: x._repr__module__
sage: sage.misc.sageinspect.sage_getfile(x._repr_)
'.../sage/structure/element_wrapper.pyx'

>>> from sage.all import *
>>> x._repr__module__
>>> sage.misc.sageinspect.sage_getfile(x._repr_)
'.../sage/structure/element_wrapper.pyx'
```

_pow_int_ is a generic method for all finite semigroups:

```python
sage: x._pow_int__module__
'sage.categories.semigroups'

>>> from sage.all import *
>>> x._pow_int__module__
'sage.categories.semigroups'
```

_mul_ is a generic method provided by the Magmas category (a magma is a set with an inner law $*$, not necessarily associative). If the two arguments are in the same parent, it will call the method _mul_, and otherwise let the coercion model try to discover how to do the multiplication:

```python
sage: x._mul__??
# not tested

>>> from sage.all import *
>>> x._mul__??
# not tested
```

Since it is a speed critical method, it is implemented in Cython in a separate file:

```python
sage: x._mul__module__
'sage.categories.coercion_methods'

>>> from sage.all import *
>>> x._mul__module__
'sage.categories.coercion_methods'
```

_mul_ is a default implementation, also provided by the Magmas category, that delegates the work to the method product of the parent (following the advice: if you do not know what to do, ask your parent); it’s also a speed critical method:

```python
sage: x._mul__??
# not tested
sage: x._mul__module__
'sage.categories.coercion_methods'
sage: x._mul__func__ is Magmas.ElementMethods._mul_parent
True
```
>>> from sage.all import *
>>> x._mul_?
# not tested
>>> x._mul_.__module__
sage.categories.coercion_methods
>>> x._mul_.__func__ is Magmas.ElementMethods._mul_parent
True

product is a mathematical method implemented by the parent:

sage: S.product.__module__
sage.categories.examples.finite_semigroups

>>> from sage.all import *
>>> S.product.__module__
sage.categories.examples.finite_semigroups

cayley_graph is a generic method on the parent, provided by the FiniteSemigroups category:

sage: S.cayley_graph.__module__
sage.categories.semigroups

>>> from sage.all import *
>>> S.cayley_graph.__module__
sage.categories.semigroups

multiplication_table is a generic method on the parent, provided by the Magmas category (it does not require associativity):

sage: S.multiplication_table.__module__
sage.categories.magas

>>> from sage.all import *
>>> S.multiplication_table.__module__
sage.categories.magas

Consider now the implementation of the semigroup:

sage: S  
# not tested

>>> from sage.all import *
>>> S  
# not tested

This implementation specifies a data structure for the parents and the elements, and makes a promise: the implemented parent is a finite semigroup. Then it fulfills the promise by implementing the basic operation product. It also implements the optional method semigroup_generators. In exchange, S and its elements receive generic implementations of all the other operations. S may override any of those by more efficient ones. It may typically implement the element method is_idempotent to always return True.

A (not yet complete) list of mandatory and optional methods to be implemented can be found by introspection with:

sage: FiniteSemigroups().required_methods()
{'element': {'optional': ['_mul_'], 'required': []}, 'parent': {'optional': ['semigroup_generators'], 'required': ['__contains__']})
>>> from sage.all import *
>>> FiniteSemigroups().required_methods()
{'element': {'optional': ['_mul_'], 'required': []},
'parent': {'optional': ['semigroup_generators'],
'required': ['__contains__']})

product does not appear in the list because a default implementation is provided in term of the method \texttt{\_mul\_} on elements. Of course, at least one of them should be implemented. On the other hand, a default implementation for \texttt{\_contains\_} is provided by \texttt{Parent}.

Documentation about those methods can be obtained with:

```
sage: C = FiniteSemigroups().element_class
sage: C._mul_  # not tested
```

See also the \texttt{abstract\_method()} decorator.

Here is the code for the finite semigroups category:

```
sage: FiniteSemigroups  # not tested
```

2.1.7 Specifying the category of a parent

Some parent constructors (not enough!) allow to specify the desired category for the parent. This can typically be used to specify additional properties of the parent that we know to hold a priori. For example, permutation groups are by default in the category of finite permutation groups (no surprise):

```
sage: P = PermutationGroup([[(1,2,3)]]); P  # needs sage.groups
Permutation Group with generators [(1,2,3)]
sage: P.category()  # needs sage.groups
Category of finite enumerated permutation groups
```

```
>>> from sage.all import *
>>> P = PermutationGroup([[(Integer(1),Integer(2),Integer(3))]]); P  # needs sage.groups
Permutation Group with generators [(1,2,3)]
>>> P.category()  # needs sage.groups
Category of finite enumerated permutation groups
```

In this case, the group is commutative, so we can specify this:

```
sage: P = PermutationGroup([[(1,2,3)]],  # needs sage.groups
....: category=PermutationGroups().Finite().Commutative()); P
Permutation Group with generators [(1,2,3)]
```

(continues on next page)
This feature can even be used, typically in experimental code, to add more structure to existing parents, and in particular to add methods for the parents or the elements, without touching the code base:

```python
sage: class Foos(Category):
    ....: def super_categories(self):
    ....:     return [PermutationGroups().Finite().Commutative()]
    ....: class ParentMethods:
    ....:     def foo(self): print("foo")
    ....: class ElementMethods:
    ....:     def bar(self): print("bar")

sage: P = PermutationGroup([[1,2,3]], category=Foos())
sage: P.foo()
foo
sage: p = P.an_element()
sage: p.bar()
bar
```

In the long run, it would be thinkable to use this idiom to implement forgetful functors; for example the above group could be constructed as a plain set with:

```python
sage: P = PermutationGroup([[1,2,3]], category=Sets())  # not implemented, needs...
```

2.1. Elements, parents, and categories in Sage: a primer
At this stage though, this is still to be explored for robustness and practicality. For now, most parents that accept a category argument only accept a subcategory of the default one.

2.1.8 Scaling further: functorial constructions, axioms, ...

In this section, we explore more advanced features of categories. Along the way, we illustrate that a large hierarchy of categories is desirable to model complicated mathematics, and that scaling to support such a large hierarchy is the driving motivation for the design of the category infrastructure.

Functorial constructions

Sage has support for a certain number of so-called covariant functorial constructions which can be used to construct new parents from existing ones while carrying over as much as possible of their algebraic structure. This includes:

- Cartesian products: See cartesian_product.
- Tensor products: See tensor.
- Subquotients / quotients / subobjects / isomorphic objects: See:
  - Sets().Subquotients,
  - Sets().Quotients,
  - Sets().Subobjects,
  - Sets().IsomorphicObjects
- Dual objects: See Modules().DualObjects.
- Algebras, as in group algebras, monoidal algebras, ...: See: Sets.ParentMethods.algebra().

Let for example $A$ and $B$ be two parents, and let us construct the Cartesian product $A \times B \times B$:

```python
sage: A = AlgebrasWithBasis(QQ).example(); A.rename("A")
# needs sage.combinat sage.modules
sage: B = HopfAlgebrasWithBasis(QQ).example(); B.rename("B")
# needs sage.combinat sage.modules
sage: C = cartesian_product([A, B, B]); C
# needs sage.combinat sage.modules
```

In which category should this new parent be? Since $A$ and $B$ are vector spaces, the result is, as a vector space, the direct sum $A \oplus B \oplus B$, hence the notation. Also, since both $A$ and $B$ are monoids, $A \times B \times B$ is naturally endowed with a monoid structure for pointwise multiplication:

```python
>>> from sage.all import *
>>> P = PermutationGroup([[(Integer(1),Integer(2),Integer(3))]], category=Sets())
# not implemented, needs sage.groups
```
the unit being the Cartesian product of the units of the operands:

```
sage: C.one()  #...
needs sage.combinat sage.groups sage.modules
B[(0, word: )] + B[(1, ())] + B[(2, ())]
sage: cartesian_product([A.one(), B.one(), B.one()])  #...
needs sage.combinat sage.groups sage.modules
B[(0, word: )] + B[(1, ())] + B[(2, ())]
```

The pointwise product can be implemented generically for all magmas (i.e. sets endowed with a multiplicative operation) that are constructed as Cartesian products. It's thus implemented in the `Magmas` category:

```
sage: C.product.__module__  #...
needs sage.combinat sage.groups sage.modules
'sage.categories.magmas'
```

More specifically, keeping on using nested classes to structure the code, the product method is put in the nested class `Magmas.CartesianProducts.ParentMethods`:

```python
class Magmas(Category):
    class ParentMethods:
        # methods for magmas
class ElementMethods:
        # methods for elements of magmas
class CartesianProduct(CartesianProductCategory):
    class ParentMethods:
        # methods for magmas that are constructed as Cartesian products
def product(self, x, y):
            # ...
class ElementMethods:
            # ...
```

**Note:** The support for nested classes in Python is relatively recent. Their intensive use for the category infrastructure did
reveal some glitches in their implementation, in particular around class naming and introspection. Sage currently works around the more annoying ones but some remain visible. See e.g. sage.misc.test_nested_class.

Let us now look at the categories of \( C \):

```python
sage: C.categories()

[Category of finite dimensional Cartesian products of algebras with basis over
Rational Field, ...]
```

This reveals the parallel hierarchy of categories for Cartesian products of semigroups magmas, ... We are thus glad that Sage uses its knowledge that a monoid is a semigroup to automatically deduce that a Cartesian product of monoids is a Cartesian product of semigroups, and build the hierarchy of classes for parents and elements accordingly.

In general, the Cartesian product of \( A \) and \( B \) can potentially be an algebra, a coalgebra, a differential module, and be finite dimensional, or graded, or .... This can only be decided at runtime, by introspection into the properties of \( A \) and \( B \); furthermore, the number of possible combinations (e.g. finite dimensional differential algebra) grows exponentially with the number of properties.

**Axioms**

**First examples**

We have seen that Sage is aware of the axioms satisfied by, for example, groups:

```python
sage: Groups().axioms()
frozenset({'Associative', 'Inverse', 'Unital'})
```

```python
>>> from sage.all import *

>>> Groups().axioms()
frozenset({'Associative', 'Inverse', 'Unital'})
```

In fact, the category of groups can be defined by stating that a group is a magma, that is a set endowed with an internal binary multiplication, which satisfies the above axioms. Accordingly, we can construct the category of groups from the category of magmas:

```python
sage: Magmas().Associative().Unital().Inverse()
Category of groups
```
In general, we can construct new categories in Sage by specifying the axioms that are satisfied by the operations of the super categories. For example, starting from the category of magmas, we can construct all the following categories just by specifying the axioms satisfied by the multiplication:

```python
>>> from sage.all import *
>>> Magmas().Associative().Unital().Inverse()
Category of groups
```

```python
sage: Magmas()
Category of magmas
sage: Magmas().Unital()
Category of unital magmas
```

```python
>>> from sage.all import *
>>> Magmas().Commutative().Unital()
Category of commutative unital magmas
```

```python
sage: Magmas().Commutative().Unital()
Category of commutative unital magmas
```

```python
>>> from sage.all import *
>>> Magmas().Commutative().Unital()
Category of commutative unital magmas
```

```python
sage: Magmas().Associative()
Category of semigroups
```

```python
>>> from sage.all import *
>>> Magmas().Associative()
Category of semigroups
```

```python
sage: Magmas().Associative().Unital()
Category of monoids
```

```python
>>> from sage.all import *
>>> Magmas().Associative().Unital()
Category of monoids
```

```python
sage: Magmas().Associative().Unital().Commutative()
Category of commutative monoids
```

```python
>>> from sage.all import *
>>> Magmas().Associative().Unital().Commutative()
Category of commutative monoids
```

```python
sage: Magmas().Associative().Unital().Inverse()
Category of groups
```

2.1. Elements, parents, and categories in Sage: a primer
Axioms and categories with axioms

Here, Associative, Unital, Commutative are axioms. In general, any category Cs in Sage can declare a new axiom A. Then, the category with axiom Cs.A() models the subcategory of the objects of Cs satisfying the axiom A. Similarly, for any subcategory Ds of Cs, Ds.A() models the subcategory of the objects of Ds satisfying the axiom A. In most cases, it’s a full subcategory (see Wikipedia article Subcategory).

For example, the category of sets defines the Finite axiom, and this axiom is available in the subcategory of groups:

```
sage: Sets().Finite()
Category of finite sets
sage: Groups().Finite()
Category of finite groups
```

The meaning of each axiom is described in the documentation of the corresponding method, which can be obtained as usual by introspection:

```
sage: C = Groups()
sage: C.Finite
# not tested
```

The purpose of categories with axioms is no different from other categories: to provide bookshelves of code, documentation, mathematical knowledge, tests, for their objects. The extra feature is that, when intersecting categories, axioms are automatically combined together:

```
sage: C = Magmas().Associative() & Magmas().Unital().Inverse() & Sets().Finite(); C
Category of finite groups
sage: sorted(C.axioms())
['Associative', 'Finite', 'Inverse', 'Unital']
```

For a more advanced example, Sage knows that a ring is a set C endowed with a multiplication which distributes over addition, such that (C, +) is a commutative additive group and (C, *) is a monoid:

```
sage: C = (CommutativeAdditiveGroups() & Monoids()).Distributive(); C
Category of rings
```

(continues on next page)
sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive', 'Unital']

```python
definition
>>> from sage.all import *

>>> C = (CommutativeAdditiveGroups() & Monoids()).Distributive(); C
Category of rings

>>> sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive', 'Unital']
```

The infrastructure allows for specifying further deduction rules, in order to encode mathematical facts like Wedderburn's theorem:

```python
definition
sage: DivisionRings() & Sets().Finite()
Category of finite enumerated fields

>>> from sage.all import *

>>> DivisionRings() & Sets().Finite()
Category of finite enumerated fields
```

**Note:** When an axiom specifies the properties of some operations in Sage, the notations for those operations are tied to this axiom. For example, as we have seen above, we need two distinct axioms for associativity: the axiom "Additive-Associative" is about the properties of the addition $+$, whereas the axiom "Associative" is about the properties of the multiplication $\ast$.

We are touching here an inherent limitation of the current infrastructure. There is indeed no support for providing generic code that is independent of the notations. In particular, the category hierarchy about additive structures (additive monoids, additive groups, …) is completely duplicated by that for multiplicative structures (monoids, groups, …).

As far as we know, none of the existing computer algebra systems has a good solution for this problem. The difficulty is that this is not only about a single notation but a bunch of operators and methods: $+$, $-$, zero, summation, sum, … in one case, $\ast$, $\times$, one, product, prod, factor, … in the other. Sharing something between the two hierarchies of categories would only be useful if one could write generic code that applies in both cases; for that one needs to somehow automatically substitute the right operations in the right spots in the code. That's kind of what we are doing manually between e.g. `AdditiveMagmas.ParentMethods.addition_table()` and `Magmas.ParentMethods.multiplication_table()`, but doing this systematically is a different beast from what we have been doing so far with just usual inheritance.

### Single entry point and name space usage

A nice feature of the notation `Cs.A()` is that, from a single entry point (say the category `Magmas` as above), one can explore a whole range of related categories, typically with the help of introspection to discover which axioms are available, and without having to import new Python modules. This feature will be used in Issue #15741 to unclutter the global name space from, for example, the many variants of the category of algebras like:

```python
definition
sage: FiniteDimensionalAlgebrasWithBasis(QQ)
Category of finite dimensional algebras with basis over Rational Field
```
There will of course be a deprecation step, but it's recommended to prefer right away the more flexible notation:

```python
>>> from sage.all import *
>>> FiniteDimensionalAlgebrasWithBasis(QQ)
Category of finite dimensional algebras with basis over Rational Field
```

```python
>>> from sage.all import *
>>> Algebras(QQ).WithBasis().FiniteDimensional()
Category of finite dimensional algebras with basis over Rational Field
```

**Design discussion**

How far should this be pushed? *Fields* should definitely stay, but should *FiniteGroups* or *DivisionRings* be removed from the global namespace? Do we want to further completely deprecate the notation `FiniteGroups()` in favor of `Groups().Finite()`?

**On the potential combinatorial explosion of categories with axioms**

Even for a very simple category like *Magmas*, there are about $2^5$ potential combinations of the axioms! Think about what this becomes for a category with two operations $+$ and $*$:

```python
sage: C = (Magmas() & AdditiveMagmas()).Distributive(); C
Category of distributive magmas and additive magmas

sage: CAA = C.Associative().AdditiveAssociative()
sage: CAA.AdditiveCommutative().AdditiveUnital().AdditiveInverse()
Category of rngs

sage: CAA.AdditiveCommutative().AdditiveUnital().Unital()
Category of semirings

sage: CAA.AdditiveCommutative().AdditiveUnital().AdditiveInverse().Unital()
Category of rings

sage: Rings().Division()
Category of division rings

sage: Rings().Division().Commutative()
Category of fields

sage: Rings().Division().Finite()
Category of finite enumerated fields
```

```python
>>> from sage.all import *
>>> C = (Magmas() & AdditiveMagmas()).Distributive(); C
Category of distributive magmas and additive magmas

>>> CAA = C.Associative().AdditiveAssociative()
>>> CAA.AdditiveCommutative().AdditiveUnital().AdditiveInverse()
```
Category of rngs

```python
>>> CAA.AdditiveCommutative().AdditiveUnital().Unital()
```

Category of semirings

```python
>>> CAA.AdditiveCommutative().AdditiveUnital().AdditiveInverse().Unital()
```

Category of rings

```python
>>> Rings().Division()
```

Category of division rings

```python
>>> Rings().Division().Commutative()
```

Category of fields

```python
>>> Rings().Division().Finite()
```

Category of finite enumerated fields

or for more advanced categories:

```python
sage: g = HopfAlgebras(QQ).WithBasis().Graded().Connected().category_graph()  # needs sage.graphs
sage: g.set_latex_options(format="dot2tex")  # needs sage.graphs sage.plot
sage: view(g)  # not tested  # needs sage.graphs sage.plot
```

Difference between axioms and regressive covariant functorial constructions

Our running examples here will be the axiom `FiniteDimensional` and the regressive covariant functorial construction `Graded`. Let `Cs` be some subcategory of `Modules`, say the category of modules itself:

```python
sage: Cs = Modules(QQ)
```

Then, `Cs.FiniteDimensional()` (respectively `Cs.Graded()`) is the subcategory of the objects \( O \) of `Cs` which are finite dimensional (respectively graded).

Let also `Ds` be a subcategory of `Cs`, say:

```python
sage: Ds = Algebras(QQ)
```
A finite dimensional algebra is also a finite dimensional module:

```
sage: Algebras(QQ).FiniteDimensional().is_subcategory(Modules(QQ).FiniteDimensional())
True
```

Similarly a graded algebra is also a graded module:

```
sage: Algebras(QQ).Graded().is_subcategory(Modules(QQ).Graded())
True
```

This is the covariance property: for an axiom or a covariant functorial construction, if \( \mathcal{D} \) is a subcategory of \( \mathcal{C} \), then \( \mathcal{D}.A() \) is a subcategory of \( \mathcal{C}.A() \).

What happens if we consider reciprocally an object of \( \mathcal{C}.A() \) which is also in \( \mathcal{D} \)? A finite dimensional module which is also an algebra is a finite dimensional algebra:

```
sage: Modules(QQ).FiniteDimensional() & Algebras(QQ)
Category of finite dimensional algebras over Rational Field
```

On the other hand, a graded module \( \mathcal{O} \) which is also an algebra is not necessarily a graded algebra! Indeed, the grading on \( \mathcal{O} \) may not be compatible with the product on \( \mathcal{O} \):

```
sage: Modules(QQ).Graded() & Algebras(QQ)
Join of Category of algebras over Rational Field
and Category of graded vector spaces over Rational Field
```

The relevant difference between \( \text{FiniteDimensional} \) and \( \text{Graded} \) is that \( \text{FiniteDimensional} \) is a statement about the properties of \( \mathcal{O} \) seen as a module (and thus does not depend on the given category), whereas \( \text{Graded} \) is a statement about the properties of \( \mathcal{O} \) and all its operations in the given category.

In general, if a category satisfies a given axiom, any subcategory also satisfies that axiom. Another formulation is that, for an axiom \( A \) defined in a super category \( \mathcal{C} \) of \( \mathcal{D} \), \( \mathcal{D}.A() \) is the intersection of the categories \( \mathcal{D} \) and \( \mathcal{C}.A() \):

```
sage: As = Algebras(QQ).FiniteDimensional(); As
Category of finite dimensional algebras over Rational Field
sage: Bs = Algebras(QQ) & Modules(QQ).FiniteDimensional(); As
Category of finite dimensional algebras over Rational Field
sage: As is Bs
True
```
An immediate consequence is that, as we have already noticed, axioms commute:

```python
sage: As = Algebras(QQ).FiniteDimensional().WithBasis(); As
Category of finite dimensional algebras with basis over Rational Field
sage: Bs = Algebras(QQ).WithBasis().FiniteDimensional(); Bs
Category of finite dimensional algebras with basis over Rational Field
sage: As is Bs
True
```

On the other hand, axioms do not necessarily commute with functorial constructions, even if the current printout may missuggest so:

```python
sage: As = Algebras(QQ).Graded().WithBasis(); As
Category of graded algebras with basis over Rational Field
sage: Bs = Algebras(QQ).WithBasis().Graded(); Bs
Category of graded algebras with basis over Rational Field
sage: As is Bs
False
```

This is because $B$s is the category of algebras endowed with basis, which are further graded; in particular the basis must respect the grading (i.e. be made of homogeneous elements). On the other hand, $A$s is the category of graded algebras, which are further endowed with some basis; that basis need not respect the grading. In fact $A$s is really a join category:

```python
sage: type(As)
<class 'sage.categories.category.JoinCategory_with_category'>
sage: As._repr_(as_join=True)
'Join of Category of algebras with basis over Rational Field and Category of graded algebras over Rational Field'
```
Todo: Improve the printing of functorial constructions and joins to raise this potentially dangerous ambiguity.

Further reading on axioms

We refer to sage.categories.category_with_axiom for how to implement axioms.

Wrap-up

As we have seen, there is a combinatorial explosion of possible classes. Constructing by hand the full class hierarchy would not scale unless one would restrict to a very rigid subset. Even if it was possible to construct automatically the full hierarchy, this would not scale with respect to system resources.

When designing software systems with large hierarchies of abstract classes for business objects, the difficulty is usually to identify a proper set of key concepts. Here we are lucky, as the key concepts have been long identified and are relatively few:

- Operations (+, *, …)
- Axioms on those operations (associativity, …)
- Constructions (Cartesian products, …)

Better, those concepts are sufficiently well known so that a user can reasonably be expected to be familiar with the concepts that are involved for his own needs.

Instead, the difficulty is concentrated in the huge number of possible combinations, an unpredictable large subset of which being potentially of interest; at the same time, only a small – but moving – subset has code naturally attached to it.

This has led to the current design, where one focuses on writing the relatively few classes for which there is actual code or mathematical information, and lets Sage compose dynamically and lazily those building blocks to construct the minimal hierarchy of classes needed for the computation at hand. This allows for the infrastructure to scale smoothly as bookshelves are added, extended, or reorganized.

2.1.9 Writing a new category

Each category \( C \) must be provided with a method \( C\.super_categories() \) and can be provided with a method \( C\.subcategory_hook_(D) \). Also, it may be needed to insert \( C \) into the output of the \( super_categories() \) method of some other category. This determines the position of \( C \) in the category graph.

A category may provide methods that can be used by all its objects, respectively by all elements of its objects.

Each category should come with a good example, in sage.categories.examples.
Inserting the new category into the category graph

\[ \text{C.super_categories()} \] must return a list of categories, namely the immediate super categories of \( C \). Of course, if you know that your new category \( C \) is an immediate super category of some existing category \( D \), then you should also update the method \( D\.\text{super_categories} \) to include \( C \).

The immediate super categories of \( C \) should not be join categories. Furthermore, one always should have:

\begin{verbatim}
Cs().is_subcategory( Category.join(Cs().super_categories()) )
Cs()._cmp_key > other._cmp_key for other in Cs().super_categories()
\end{verbatim}

This is checked by \texttt{\_test\_category()}.

In several cases, the category \( C \) is directly provided with a generic implementation of \texttt{\_super\_categories}; a typical example is when \( C \) implements an axiom or a functorial construction; in such a case, \( C \) may implement \texttt{\_extra\_super\_categories()} to complement the super categories discovered by the generic implementation. This method needs not return immediate super categories; instead it’s usually best to specify the largest super category providing the desired mathematical information. For example, the category \texttt{Magmas.Combinette.Algebras} states that the algebra of a commutative magma is a commutative magma. This is sufficient to let Sage deduce that it’s in fact a commutative algebra.

Methods for objects and elements

Different objects of the same category share some algebraic features, and very often these features can be encoded in a method, in a generic way. For example, for every commutative additive monoid, it makes sense to ask for the sum of a list of elements. Sage’s category framework allows to provide a generic implementation for all objects of a category.

If you want to provide your new category with generic methods for objects (or elements of objects), then you simply add a nested class called \texttt{\_\_ParentMethods} (or \texttt{\_\_ElementMethods}). The methods of that class will automatically become methods of the objects (or the elements). For instance:

\begin{verbatim}
sage: P.<x,y> = ZZ[]
sage: P.prod([x,y,2])
2*x*y
sage: P.prod.__module__
sage.categories.monoids
sage: P.prod.__func__
raw_getattr(Monoids().\_\_ParentMethods, "prod")
True

>>> from sage.all import *
>>> P = ZZ[\'x, y\']; (x, y) = P._\_first\_ngens(2)
>>> P.prod({x,y,Integer(2)})
2*x*y
>>> P.prod.__module__
sage.categories.monoids'
>>> P.prod.__func__
raw_getattr(Monoids().\_\_ParentMethods, "prod")
True
\end{verbatim}

We recommend to study the code of one example:

\begin{verbatim}
sage: C = CommutativeAdditiveMonoids()
sage: C
\end{verbatim}
On the order of super categories

The generic method \texttt{C.all_super_categories()} determines recursively the list of all super categories of \( C \).

The order of the categories in this list does influence the inheritance of methods for parents and elements. Namely, if \( P \) is an object in the category \( C \) and if \( C_1 \) and \( C_2 \) are both super categories of \( C \) defining some method \texttt{foo} in \texttt{ParentMethods}, then \( P \) will use \( C_1 \)'s version of \texttt{foo} if and only if \( C_1 \) appears before \( C_2 \) in \texttt{C.all_super_categories()}.

However this must be considered as an implementation detail: if \( C_1 \) and \( C_2 \) are incomparable categories, then the order in which they appear must be mathematically irrelevant: in particular, the methods \texttt{foo} in \( C_1 \) and \( C_2 \) must have the same semantic. Code should not rely on any specific order, as it is subject to later change. Whenever one of the implementations is preferred in some common subcategory of \( C_1 \) and \( C_2 \), for example for efficiency reasons, the ambiguity should be resolved explicitly by defining a method \texttt{foo} in this category. See the method \texttt{some_elements} in the code of the category \texttt{FiniteCoxeterGroups} for an example.

Since Issue \#11943, \texttt{C.all_super_categories()} is computed by the so-called C3 algorithm used by Python to compute Method Resolution Order of new-style classes. Thus the order in \texttt{C.all_super_categories()}, \texttt{C.parent_class.mro()} and \texttt{C.element_class.mro()} are guaranteed to be consistent.

Since Issue \#13589, the C3 algorithm is put under control of some total order on categories. This order is not necessarily meaningful, but it guarantees that C3 always finds a consistent Method Resolution Order. For background, see \texttt{sage.misc.c3_controlled}. A visible effect is that the order in which categories are specified in \texttt{C.super_categories()}, or in a join category, no longer influences the result of \texttt{C.all_super_categories()}.

Subcategory hook (advanced optimization feature)

The default implementation of the method \texttt{C.is_subcategory(D)} is to look up whether \( D \) appears in \texttt{C.all_super_categories()}. However, building the list of all the super categories of \( C \) is an expensive operation that is sometimes best avoided. For example, if both \( C \) and \( D \) are categories defined over a base, but the bases differ, then one knows right away that they cannot be subcategories of each other.

When such a short-path is known, one can implement a method \texttt{_subcategory_hook}. Then, \( C \).is_subcategory(D) first calls \( D._subcategory_hook(C) \). If this returns Unknown, then \( C \).is_subcategory(D) tries to find \( D \) in \texttt{C.all_super_categories()}. Otherwise, \( C \).is_subcategory(D) returns the result of \( D._subcategory_hook(C) \).

By default, \( D._subcategory_hook(C) \) tests whether \texttt{issubclass(C.parent_class,D.parent_class)}, which is very often giving the right answer:

\begin{verbatim}
sage: Rings()._subcategory_hook_(Algebras(QQ))
True
sage: HopfAlgebras(QQ)._subcategory_hook_(Algebras(QQ))
False
sage: Algebras(QQ)._subcategory_hook_(HopfAlgebras(QQ))
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> Rings()._subcategory_hook_(Algebras(QQ))
True
\end{verbatim}
2.2 Categories

AUTHORS:

• David Kohel, William Stein and Nicolas M. Thiery

Every Sage object lies in a category. Categories in Sage are modeled on the mathematical idea of category, and are distinct from Python classes, which are a programming construct.

In most cases, typing \texttt{x.category()} returns the category to which \texttt{x} belongs. If \texttt{C} is a category and \texttt{x} is any object, \texttt{C(x)} tries to make an object in \texttt{C} from \texttt{x}. Checking if \texttt{x} belongs to \texttt{C} is done as usually by \texttt{x in C}.

See \texttt{Category} and \texttt{sage.categories.primer} for more details.

EXAMPLES:

We create a couple of categories:

\begin{verbatim}
sage: Sets() Category of sets
sage: GSets(AbelianGroup([2, 4, 9])) # needs sage.groups
    Category of G-sets for Multiplicative Abelian group isomorphic to C2 x C4 x C9
sage: Semigroups() Category of semigroups
sage: VectorSpaces(FiniteField(11)) Category of vector spaces over Finite Field of size 11
sage: Ideals(IntegerRing()) Category of ring ideals in Integer Ring
\end{verbatim}

Let's request the category of some objects:

\begin{verbatim}
>>> from sage.all import *
>>> Sets() Category of sets
>>> GSets(AbelianGroup([Integer(2), Integer(4), Integer(9)]))  # needs sage.groups
    Category of G-sets for Multiplicative Abelian group isomorphic to C2 x C4 x C9
>>> Semigroups() Category of semigroups
>>> VectorSpaces(FiniteField(Integer(11))) Category of vector spaces over Finite Field of size 11
>>> Ideals(IntegerRing()) Category of ring ideals in Integer Ring
\end{verbatim}
over (number fields and quotient fields and metric spaces)

\begin{verbatim}
    sage: G = SymmetricGroup(9)  # needs sage.groups
    sage: G.category()  # needs sage.groups
    Join of
    Category of finite enumerated permutation groups and
    Category of finite Weyl groups and
    Category of well generated finite irreducible complex reflection groups

    sage: P = PerfectMatchings(3)  # needs sage.combinat
    sage: P.category()  # needs sage.combinat
    Category of finite enumerated sets

    >>> from sage.all import *
    >>> V = VectorSpace(RationalField(), Integer(3))  # needs sage.modules
    >>> V.category()  # needs sage.modules
    Category of finite dimensional vector spaces with basis
    over (number fields and quotient fields and metric spaces)

    >>> G = SymmetricGroup(Integer(9))  # needs sage.groups
    >>> G.category()  # needs sage.groups
    Join of
    Category of finite enumerated permutation groups and
    Category of finite Weyl groups and
    Category of well generated finite irreducible complex reflection groups

    >>> P = PerfectMatchings(Integer(3))  # needs sage.combinat
    >>> P.category()  # needs sage.combinat
    Category of finite enumerated sets

    Let's check some memberships:

    sage: V in VectorSpaces(QQ)  # needs sage.modules
    True
    sage: V in VectorSpaces(FiniteField(11))  # needs sage.modules
    False
    sage: G in Monoids()  # needs sage.groups
    True
    sage: P in Rings()  # needs sage.combinat
    False
\end{verbatim}
For parametrized categories one can use the following shorthand:

```python
sage: V in VectorSpaces
#—
needs sage.modules
True
sage: G in VectorSpaces
#—
needs sage.groups
False
```

A parent \( P \) is in a category \( C \) if \( P.category() \) is a subcategory of \( C \).

**Note:** Any object of a category should be an instance of `CategoryObject`.

For backward compatibility this is not yet enforced:

```python
sage: class A:
     ....   def category(self):
     ....       return Fields()

sage: A() in Rings()
True
```

By default, the category of an element \( x \) of a parent \( P \) is the category of all objects of \( P \) (this is dubious and may be deprecated):

```python
sage: v = V.gen(1)
(continues on next page)```
needs sage.modules

sage: v.category()
# needs sage.modules
Category of elements of Vector space of dimension 3 over Rational Field

>>> from sage.all import *
>>> V = VectorSpace(RationalField(), Integer(3))
# needs sage.modules
>>> v = V.gen(Integer(1))
# needs sage.modules
>>> v.category()
# needs sage.modules
Category of elements of Vector space of dimension 3 over Rational Field

class sage.categories.category.Category

    Base classes: UniqueRepresentation, SageObject

The base class for modeling mathematical categories, like for example:

    • Groups(): the category of groups
    • EuclideanDomains(): the category of euclidean rings
    • VectorSpaces(QQ): the category of vector spaces over the field of rationals

See sage.categories.primer for an introduction to categories in Sage, their relevance, purpose, and usage. The documentation below will focus on their implementation.

Technically, a category is an instance of the class Category or some of its subclasses. Some categories, like VectorSpaces, are parametrized: VectorSpaces(QQ) is one of many instances of the class VectorSpaces. On the other hand, EuclideanDomains() is the single instance of the class EuclideanDomains.

Recall that an algebraic structure (say, the ring \( \mathbb{Q}[x] \)) is modelled in Sage by an object which is called a parent. This object belongs to certain categories (here EuclideanDomains() and Algebras()). The elements of the ring are themselves objects.

The class of a category (say EuclideanDomains) can define simultaneously:

    • Operations on the category itself (what is its super categories? its category of morphisms? its dual category?).
    • Generic operations on parents in this category, like the ring \( \mathbb{Q}[x] \).
    • Generic operations on elements of such parents (e. g., the Euclidean algorithm for computing gcds).
    • Generic operations on morphisms of this category.

This is achieved as follows:

sage: from sage.categories.category import Category
sage: class EuclideanDomains(Category):
    ...
    # operations on the category itself
    ...
    def super_categories(self):
        ...
        [Rings()]
    ...
    ...
    def dummy(self): # TODO: find some good examples
        ...
        pass
    ...
    ...
    class ParentMethods: # holds the generic operations on parents
        ...
        # TODO: find a good example of an operation

(continues on next page)
Note that the nested class \texttt{ParentMethods} is merely a container of operations, and does not inherit from anything. Instead, the hierarchy relation is defined once at the level of the categories, and the actual hierarchy of classes is built in parallel from all the \texttt{ParentMethods} nested classes, and stored in the attribute \texttt{parent_class}. Then, a parent in a category \texttt{C} receives the appropriate operations from all the super categories by usual class inheritance from \texttt{C.parent_class}.

Similarly, two other hierarchies of classes, for elements and morphisms respectively, are built from all the \texttt{ElementMethods} and \texttt{MorphismMethods} nested classes.

**EXAMPLES:**

We define a hierarchy of four categories \texttt{As()}, \texttt{Bs()}, \texttt{Cs()}, \texttt{Ds()} with a diamond inheritance. Think for example:

- \texttt{As()}: the category of sets
- \texttt{Bs()}: the category of additive groups
- \texttt{Cs()}: the category of multiplicative monoids
- \texttt{Ds()}: the category of rings
sage: from sage.categories.category import Category
sage: from sage.misc.lazy_attribute import lazy_attribute
sage: class As (Category):
    ....: def super_categories(self):
    ....:     return []
    ....:
    ....:     class ParentMethods:
    ....:         def fA(self):
    ....:             return "A"
    ....:     f = fA
sage: class Bs (Category):
    ....: def super_categories(self):
    ....:     return [As()]
    ....:
    ....:     class ParentMethods:
    ....:         def fB(self):
    ....:             return "B"
    ....:
sage: class Cs (Category):
    ....: def super_categories(self):
    ....:     return [As()]
    ....:
    ....:     class ParentMethods:
    ....:         def fC(self):
    ....:             return "C"
    ....:     f = fC
sage: class Ds (Category):
    ....: def super_categories(self):
    ....:     return [Bs(),Cs()]
    ....:
    ....:     class ParentMethods:
    ....:         def fD(self):
    ....:             return "D"

>>> from sage.all import *
>>> from sage.categories.category import Category
>>> from sage.misc.lazy_attribute import lazy_attribute
>>> class As (Category):
...     def super_categories(self):
...         return []
...:
>>> class ParentMethods:
...     def fA(self):
...         return "A"
...     f = fA
>>> class Bs (Category):
...     def super_categories(self):
...         return [As()]
...:
>>> class ParentMethods:
...     def fB(self):
...         return "B"
...:
>>> class Cs (Category):
...     def super_categories(self):
...         return [As()]
...:
>>> class ParentMethods:
...     def fC(self):
...         return "C"
...     f = fC
>>> class Ds (Category):
...     def super_categories(self):
...         return [Bs(),Cs()]
...:
>>> class ParentMethods:
...     def fD(self):
...         return "D"
...    def super_categories(self):
...        return [As()]
...
>>> class ParentMethods:
...    def fC(self):
...        return "C"
...
    f = fC

>>> class Ds (Category):
...    def super_categories(self):
...        return [Bs(),Cs()]
...
>>> class ParentMethods:
...    def fD(self):
...        return "D"
...
    return "D"

Categories should always have unique representation; by Issue #12215, this means that it will be kept in cache, but only if there is still some strong reference to it.

We check this before proceeding:

```
sage: import gc
sage: idAs = id(As())
sage: _ = gc.collect()
sage: n == id(As())
False
sage: a = As()
sage: id(As()) == id(As())
True
sage: As().parent_class == As().parent_class
True

>>> from sage.all import *
>>> import gc
>>> idAs = id(As())
>>> _ = gc.collect()
>>> n == id(As())
False
>>> a = As()
>>> id(As()) == id(As())
True
>>> As().parent_class == As().parent_class
True
```

We construct a parent in the category $Ds()$ (that, is an instance of $Ds().parent_class$), and check that it has access to all the methods provided by all the categories, with the appropriate inheritance order:

```
sage: D = Ds().parent_class()
sage: [ D.fA(), D.fB(), D.fC(), D.fD() ]
['A', 'B', 'C', 'D']
sage: D.f()
'C'

>>> from sage.all import *
>>> D = Ds().parent_class()
>>> [ D.fA(), D.fB(), D.fC(), D.fD() ]
```

(continues on next page)
Here is the parallel hierarchy of classes which has been built automatically, together with the method resolution order (.mro()):

```python
sage: As().parent_class
<class '__main__.As.parent_class'>
sage: As().parent_class.__bases__
(... 'object'...)
sage: As().parent_class.mro()
[<class '__main__.As.parent_class'>, <... 'object'>]

>>> from sage.all import *

sage: Bs().parent_class
<class '__main__.Bs.parent_class'>
sage: Bs().parent_class.__bases__
(... 'object'...)
sage: Bs().parent_class.mro()
[<class '__main__.Bs.parent_class'>, <class '__main__.As.parent_class'>, <... 'object'>]

>>> from sage.all import *

sage: Cs().parent_class
<class '__main__.Cs.parent_class'>
```

(continues on next page)
sage: Cs().parent_class.__bases__
(<class '__main__.As.parent_class'>,)
sage: Cs().parent_class.__mro__
(<class '__main__.Cs.parent_class'>, <class '__main__.As.parent_class'>, <...
< 'object'>)

>>> from sage.all import *
>>> Cs().parent_class
<class '__main__.Cs.parent_class'>

>>> Cs().parent_class.__bases__
(<class '__main__.As.parent_class'>,)

>>> Cs().parent_class.__mro__
(<class '__main__.Cs.parent_class'>, <class '__main__.As.parent_class'>, <...
< 'object'>)

sage: Ds().parent_class
<class '__main__.Ds.parent_class'>
sage: Ds().parent_class.__bases__
(<class '__main__.Cs.parent_class'>, <class '__main__.Bs.parent_class'>)

sage: Ds().parent_class.mro()
[<class '__main__.Ds.parent_class'>, <class '__main__.Cs.parent_class'>,
<class '__main__.Bs.parent_class'>, <class '__main__.As.parent_class'>, <...
< 'object'>]

Note that two categories in the same class need not have the same super_categories. For example, Algebras(QQ) has VectorSpaces(QQ) as super category, whereas Algebras(ZZ) only has Modules(ZZ) as super category. In particular, the constructed parent class and element class will differ (inheriting, or not, methods specific for vector spaces):

sage: Algebras(QQ).parent_class is Algebras(ZZ).parent_class
False
sage: issubclass(Algebras(QQ).parent_class, VectorSpaces(QQ).parent_class)
True

>>> from sage.all import *
>>> Algebras(QQ).parent_class is Algebras(ZZ).parent_class
False

>>> issubclass(Algebras(QQ).parent_class, VectorSpaces(QQ).parent_class)
True

On the other hand, identical hierarchies of classes are, preferably, built only once (e.g. for categories over a base ring):

sage: Algebras(GF(5)).parent_class is Algebras(GF(7)).parent_class
True
We now construct a parent in the usual way:

```python
sage: class myparent(Parent):
    ....:    def __init__(self):
    ....:        Parent.__init__(self, category=ds())
    ....:    def g(self):
    ....:        return "myparent"
    ....:    class Element():
    ....:        pass
sage: D = myparent()
```

```python
sage: D.__class__
<class '__main__.myparent_with_category'>
sage: D.__class__.__bases__
(<class '__main__.myparent'>, <class '__main__.Ds.parent_class'>)
sage: D.__class__.mro()
[<class '__main__.myparent_with_category'>,
<class '__main__.myparent'>,
<class '__main__.Ds.parent_class'>,
<class 'sage.structure.parent.Parent'>,
<class 'sage.structure.category_object.CategoryObject'>,
<class 'sage.structure.sage_object.SageObject'>,
<class '__main__.Ds.parent_class'>,
<class '__main__.Cs.parent_class'>,
<class '__main__.Bs.parent_class'>,
<class '__main__.As.parent_class'>,
<... 'object'>]
```

```python
sage: D.fA()
'A'
sage: D.fB()
'B'
sage: D.fC()
'C'
sage: D.fD()
'D'
sage: D.f()
'C'
sage: D.g()
'myparent'
```
...  
D = myparent()  
D._class__
<class '__main__.myparent_with_category'>  
D._class__._bases__
(<class '__main__.myparent'>, <class '__main__.Ds.parent_class'>)  
D._class__._mro__
<class '__main__.myparent_with_category'>,  
<class '__main__.myparent'>,  
<class 'sagestructure.parent.Parent'>,  
<class 'sagestructure.category_object.CategoryObject'>,  
<class 'sagestructure.sage_object.SageObject'>,  
<class '__main__.Ds.parent_class'>,  
<class '__main__.Cs.parent_class'>,  
<class '__main__.Bs.parent_class'>,  
<class '__main__.As.parent_class'>,  
<... 'object'>)  
D.fA()
'A'
D.fB()
'B'
D.fC()
'C'
D.fD()
'D'
D.f()
'C'
D.g()
'myparent'

sage: D.element_class
<class '__main__.myparent_with_category.element_class'>
sage: D.element_class.mro()
[[<class '__main__.myparent_with_category.element_class'>],
<class ...__main__....Element...>,
<class '__main__.Ds.element_class'>,
<class '__main__.Cs.element_class'>,
<class '__main__.Bs.element_class'>,
<class '__main__.As.element_class'>,
<... 'object'>]

>>> from sage.all import *

D.element_class
<class '__main__.myparent_with_category.element_class'>
D.element_class.mro()
[[<class '__main__.myparent_with_category.element_class'>],
<class ...__main__....Element...>,
<class '__main__.Ds.element_class'>,
<class '__main__.Cs.element_class'>,
<class '__main__.Bs.element_class'>,
<class '__main__.As.element_class'>,
<... 'object'>]

_super_categories__

The immediate super categories of this category.

This lazy attribute caches the result of the mandatory method super_categories() for speed. It also
does some mangling (flattening join categories, sorting, …).

Whenever speed matters, developers are advised to use this lazy attribute rather than calling `super_categories()`.

Note: This attribute is likely to eventually become a tuple. When this happens, we might as well use `Category._sort()`, if not `Category._sort_uniq()`.

EXAMPLES:

```python
sage: Rings()._super_categories
[Category of rngs, Category of semirings]

>>> from sage.all import *
>>> Rings()._super_categories
[Category of rngs, Category of semirings]
```

__super_categories_for_classes__()  
The super categories of this category used for building classes.

This is a close variant of `__super_categories()` used for constructing the list of the bases for `parent_class()`, `element_class()`, and friends. The purpose is ensure that Python will find a proper Method Resolution Order for those classes. For background, see `sage.misc.c3_controlled`.

See also:

__cmp_key__().

Note: This attribute is calculated as a by-product of computing `__all_super_categories()`.

EXAMPLES:

```python
sage: Rings().__super_categories_for_classes
[Category of rngs, Category of semirings]

>>> from sage.all import *
>>> Rings().__super_categories_for_classes
[Category of rngs, Category of semirings]
```

__all_super_categories__()  
All the super categories of this category, including this category.

Since Issue #11943, the order of super categories is determined by Python’s method resolution order C3 algorithm.

See also:

`all_super_categories()`

Note: this attribute is likely to eventually become a tuple.

Note: this sets `__super_categories_for_classes()` as a side effect
EXAMPLES:

sage: C = Rings(); C
Category of rings
sage: C._all_super_categories
[Category of rings, Category of rngs, Category of semirings, ...
 Category of monoids, ...
 Category of commutative additive groups, ...
 Category of sets, Category of sets with partial maps,
 Category of objects]

>>> from sage.all import *

>>> C = Rings(); C
Category of rings

>>> C._all_super_categories
[Category of rings, Category of rngs, Category of semirings, ...
 Category of monoids, ...
 Category of commutative additive groups, ...
 Category of sets, Category of sets with partial maps,
 Category of objects]

_all_super_categories_proper()

All the proper super categories of this category.

Since Issue #11943, the order of super categories is determined by Python’s method resolution order C3
algorithm.

See also:

all_super_categories()

Note: this attribute is likely to eventually become a tuple.

EXAMPLES:

sage: C = Rings(); C
Category of rings
sage: C._all_super_categories_proper
[Category of rngs, Category of semirings, ...
 Category of monoids, ...
 Category of commutative additive groups, ...
 Category of sets, Category of sets with partial maps,
 Category of objects]

>>> from sage.all import *

>>> C = Rings(); C
Category of rings

>>> C._all_super_categories_proper
[Category of rngs, Category of semirings, ...
 Category of monoids, ...
 Category of commutative additive groups, ...
 Category of sets, Category of sets with partial maps,
 Category of objects]

_set_of_super_categories()

The frozen set of all proper super categories of this category.
Note: this is used for speeding up category containment tests.

See also:

all_super_categories()

EXAMPLES:

```
sage: sorted(Groups()._set_of_super_categories, key=str)
[Category of inverse unital magmas,
 Category of magmas,
 Category of monoids,
 Category of objects,
 Category of semigroups,
 Category of sets,
 Category of sets with partial maps,
 Category of unital magmas]
sage: sorted(Groups()._set_of_super_categories, key=str)
[Category of inverse unital magmas, Category of magmas, Category of monoids,
 Category of objects, Category of semigroups, Category of sets,
 Category of sets with partial maps, Category of unital magmas]
```

```python
>>> from sage.all import *
>>> sorted(Groups()._set_of_super_categories, key=str)
[Category of inverse unital magmas,
 Category of magmas,
 Category of monoids,
 Category of objects,
 Category of semigroups,
 Category of sets,
 Category of sets with partial maps,
 Category of unital magmas]
```

_make_named_class (name, method_provider, cache=False, picklable=True)

Construction of the parent/element/… class of self.

INPUT:

- name – a string; the name of the class as an attribute of self. E.g. “parent_class”
- method_provider – a string; the name of an attribute of self that provides methods for the new class (in addition to those coming from the super categories). E.g. “ParentMethods”
- cache – a boolean or ignore_reduction (default: False) (passed down to dynamic_class; for internal use only)
- picklable – a boolean (default: True)

ASSUMPTION:

It is assumed that this method is only called from a lazy attribute whose name coincides with the given name.

OUTPUT:

A dynamic class with bases given by the corresponding named classes of self’s super_categories, and methods taken from the class getattr(self, method_provider).
Note:

- In this default implementation, the reduction data of the named class makes it depend on `self`. Since the result is going to be stored in a lazy attribute of `self` anyway, we may as well disable the caching in `dynamic_class` (hence the default value `cache=False`).
- `CategoryWithParameters` overrides this method so that the same parent/element/… classes can be shared between closely related categories.
- The bases of the named class may also contain the named classes of some indirect super categories, according to `_super_categories_for_classes()`. This is to guarantee that Python will build consistent method resolution orders. For background, see `sage.misc.c3_controlled`.

See also:

`CategoryWithParameters._make_named_class()`

EXAMPLES:

```python
sage: PC = Rings()._make_named_class("parent_class", "ParentMethods"); PC
<class 'sage.categories.rings.Rings.parent_class'>
sage: type(PC)
<class 'sage.structure.dynamic_class.DynamicMetaClass'>
sage: PC.__bases__
(<class 'sage.categories.rings.Rngs.Rngs.parent_class'>,
 <class 'sage.categories.semirings.Semirings.parent_class'>)
```

```python
>>> from sage.all import *
>>> PC = Rings()._make_named_class("parent_class", "ParentMethods"); PC
<class sage.categories.rings.Rings.parent_class>
```

Notethat, by default, the result is not cached:

```python
sage: PC is Rings()._make_named_class("parent_class", "ParentMethods")
False
```

```python
>>> from sage.all import *
>>> PC is Rings()._make_named_class("parent_class", "ParentMethods")
False
```

Indeed this method is only meant to construct lazy attributes like `parent_class` which already handle this caching:

```python
sage: Rings().parent_class
<class 'sage.categories.rings.Rings.parent_class'>
```

```python
>>> from sage.all import *
>>> Rings().parent_class
<class 'sage.categories.rings.Rings.parent_class'>
```

Reduction for pickling also assumes the existence of this lazy attribute:
sage: PC._reduction
(built-in function getattr), (Category of rings, 'parent_class'))
sage: loads(dumps(PC)) is Rings().parent_class
True

>>> from sage.all import *
>>> PC._reduction
(built-in function getattr), (Category of rings, 'parent_class'))
>>> loads(dumps(PC)) is Rings().parent_class
True

_repr_()
Return the print representation of this category.

EXAMPLES:

sage: Sets()  
# indirect doctest
Category of sets

>>> from sage.all import *
>>> Sets()  
# indirect doctest
Category of sets

_repr_object_names()
Return the name of the objects of this category.

EXAMPLES:

sage: FiniteGroups()._repr_object_names()
'finite groups'
sage: AlgebrasWithBasis(QQ)._repr_object_names()
'algebras with basis over Rational Field'

>>> from sage.all import *
>>> FiniteGroups()._repr_object_names()
'finite groups'
>>> AlgebrasWithBasis(QQ)._repr_object_names()
'algebras with basis over Rational Field'

_test_category(**options)
Run generic tests on this category

See also:
TestSuite.

EXAMPLES:

sage: Sets()._test_category()

>>> from sage.all import *
>>> Sets()._test_category()

Let us now write a couple broken categories:
```python
sage: class MyObjects(Category):
    ....:    pass
MyObjects()._test_category()
Traceback (most recent call last):
  ...,
NotImplementedError: <abstract method super_categories at ...>

sage: class MyObjects(Category):
    ....:    def super_categories(self):
    ....:        return tuple()
MyObjects()._test_category()
Traceback (most recent call last):
  ...,
AssertionError: Category of my objects.super_categories() should return a list

sage: class MyObjects(Category):
    ....:    def super_categories(self):
    ....:        return []
MyObjects()._test_category()
Traceback (most recent call last):
  ...,
AssertionError: Category of my objects is not a subcategory of Objects()
```

```python
>>> from sage.all import *
>>> class MyObjects(Category):
...    pass
>>> MyObjects()._test_category()
Traceback (most recent call last):
  ...,
NotImplementedError: <abstract method super_categories at ...>

>>> class MyObjects(Category):
...    def super_categories(self):
...        return tuple()
>>> MyObjects()._test_category()
Traceback (most recent call last):
  ...,
AssertionError: Category of my objects.super_categories() should return a list

>>> class MyObjects(Category):
...    def super_categories(self):
...        return []
>>> MyObjects()._test_category()
Traceback (most recent call last):
  ...,
AssertionError: Category of my objects is not a subcategory of Objects()
```

`_with_axiom(axiom)`

Return the subcategory of the objects of `self` satisfying the given `axiom`.

**INPUT:**

- `axiom` – a string, the name of an axiom

**EXAMPLES:**

```python
sage: Sets()._with_axiom("Finite")
Category of finite sets
```

(continues on next page)
When `axiom` is not defined for `self`, `self` is returned:

```
sage: Sets()._with_axiom("Associative")
Category of sets
```

```
>>> from sage.all import *
>>> Sets()._with_axiom("Associative")
Category of sets
```

**Warning:** This may be changed in the future to raising an error.

`_with_axiom_as_tuple(axiom)`

Return a tuple of categories whose join is `self._with_axiom()`.

**INPUT:**

- `axiom` – a string, the name of an axiom

This is a lazy version of `_with_axiom()` which is used to avoid recursion loops during join calculations.

**Note:** The order in the result is irrelevant.

**EXAMPLES:**

```
sage: Sets()._with_axiom_as_tuple('Finite')
(Category of finite sets,)
sage: Magmas()._with_axiom_as_tuple('Finite')
(Category of magmas, Category of finite sets)
sage: Rings().Division()._with_axiom_as_tuple('Finite')
(Category of division rings,
Category of finite monoids,
Category of commutative magmas,
```

(continues on next page)
Category of finite additive groups

sage: HopfAlgebras(QQ)._with_axiom_as_tuple('FiniteDimensional')
(Category of Hopf algebras over Rational Field,
  Category of finite dimensional vector spaces over Rational Field)

>>> from sage.all import *

>>> Sets()._with_axiom_as_tuple('Finite')
(Category of finite sets,)

>>> Magmas()._with_axiom_as_tuple('Finite')
(Category of magmas, Category of finite sets)

>>> Rings().Division()._with_axiom_as_tuple('Finite')
(Category of division rings,
  Category of finite monoids,
  Category of commutative magmas,
  Category of finite additive groups)

>>> HopfAlgebras(QQ)._with_axiom_as_tuple('FiniteDimensional')
(Category of Hopf algebras over Rational Field,
  Category of finite dimensional vector spaces over Rational Field)

_without_axioms (named=False)

Return the category without the axioms that have been added to create it.

INPUT:

- named – a boolean (default: False)

Todo: Improve this explanation.

If named is True, then this stops at the first category that has an explicit name of its own. See category_with_axiom.CategoryWithAxiom._without_axioms()

EXAMPLES:

sage: Sets()._without_axioms()
Category of sets

sage: Semigroups()._without_axioms()
Category of magmas

sage: Algebras(QQ).Commutative().WithBasis()._without_axioms()
Category of magmatic algebras over Rational Field

sage: Algebras(QQ).Commutative().WithBasis()._without_axioms(named=True)
Category of algebras over Rational Field

>>> from sage.all import *

>>> Sets()._without_axioms()
Category of sets

>>> Semigroups()._without_axioms()
Category of magmas

>>> Algebras(QQ).Commutative().WithBasis()._without_axioms()
Category of magmatic algebras over Rational Field

>>> Algebras(QQ).Commutative().WithBasis()._without_axioms(named=True)
Category of algebras over Rational Field

static _sort (categories)

Return the categories after sorting them decreasingly according to their comparison key.
See also:

_cmp_key()

INPUT:

• categories – a list (or iterable) of non-join categories

OUTPUT:

A sorted tuple of categories, possibly with repeats.

Note: The auxiliary function _flatten_categories used in the test below expects a second argument, which is a type such that instances of that type will be replaced by its super categories. Usually, this type is JoinCategory.

EXAMPLES:

```
sage: Category._sort([Sets(), Objects(), Coalgebras(QQ), Monoids(), Sets().Finite()])
(Category of monoids,
Category of coalgebras over Rational Field,
Category of finite sets,
Category of sets,
Category of objects)
sage: Category._sort([Sets().Finite(), Semigroups().Finite(), Sets().Facade(),
˓→Magmas().Commutative()])
(Category of finite semigroups,
Category of commutative magmas,
Category of finite sets,
Category of facade sets)
sage: Category._sort(Category._flatten_categories([Sets().Finite(),
˓→Algebras(QQ).WithBasis(), Semigroups().Finite(),
˓→Algebras(QQ).Commutative(), Algebras(QQ).Graded().WithBasis(),
˓→JoinCategory]))
(Category of algebras with basis over Rational Field,
Category of algebras with basis over Rational Field,
Category of graded algebras over Rational Field,
Category of commutative algebras over Rational Field,
Category of finite semigroups,
Category of finite sets,
Category of facade sets)
```

```bash
>>> from sage.all import *

>>> Category._sort([Sets(), Objects(), Coalgebras(QQ), Monoids(), Sets().Finite()])
(Category of monoids,
Category of coalgebras over Rational Field,
Category of finite sets,
Category of sets,
Category of objects)

>>> Category._sort([Sets().Finite(), Semigroups().Finite(), Sets().Facade(),
˓→Magmas().Commutative()])
(Category of finite semigroups,
Category of commutative magmas,
Category of finite sets,
Category of facade sets)
```

(continues on next page)
Category of facade sets)

>>> Category._sort(Category._flatten_categories([Sets().Finite(),
                                                           Algebras(QQ).WithBasis(), Semigroups().Finite(),
                                                           Sets().Facade(),
                                                           Algebras(QQ).Commutative(), Algebras(QQ).Graded().WithBasis()],
                                                       sage.categories.category.JoinCategory))
(Category of algebras with basis over Rational Field,
Category of algebras with basis over Rational Field,
Category of graded algebras over Rational Field,
Category of commutative algebras over Rational Field,
Category of finite semigroups,
Category of finite sets,
Category of facade sets)

static _sort_uniq(categories)

Return the categories after sorting them and removing redundant categories.

Redundant categories include duplicates and categories which are super categories of other categories in the input.

INPUT:

- categories – a list (or iterable) of categories

OUTPUT: a sorted tuple of mutually incomparable categories

EXAMPLES:

sage: Category._sort_uniq([Rings(), Monoids(), Coalgebras(QQ)])
(Category of rings, Category of coalgebras over Rational Field)

Note that, in the above example, Monoids() does not appear in the result because it is a super category of Rings().

static __classcall__(*args, **options)

Input mangling for unique representation.

Let $C = Cs(\ldots)$ be a category. Since Issue #12895, the class of $C$ is a dynamic subclass $Cs\_with\_category$ of $Cs$ in order for $C$ to inherit code from the SubcategoryMethods nested classes of its super categories.

The purpose of this __classcall__ method is to ensure that reconstructing $C$ from its class with $Cs\_with\_category(\ldots)$ actually calls properly $Cs(\ldots)$ and gives back $C$.

See also:

subcategory_class()

EXAMPLES:

sage: A = Algebras(QQ)
sage: A._class_
<class 'sage.categories.algebras.Algebras_with_category'>
sage: A is Algebras(QQ)
True

```
sage: A is A.__class__(QQ)
True
```

```
>>> from sage.all import *
>>> A = Algebras(QQ)
>>> A.__class__
<class 'sage.categories.algebras.Algebras_with_category'>
>>> A is Algebras(QQ)
True
>>> A is A.__class__(QQ)
True
```

**__init__()**

Initialize this category.

**EXAMPLES:**

```
sage: class SemiprimitiveRings(Category):
....:     def super_categories(self):
....:         return [Rings()]
....:     class ParentMethods:
....:         def jacobson_radical(self):
....:             return self.ideal(0)

sage: C = SemiprimitiveRings()
sage: C
Category of semiprimitive rings
sage: C.__class__
<class '__main__.SemiprimitiveRings_with_category'>
```

```
>>> from sage.all import *

>>> class SemiprimitiveRings(Category):
...     def super_categories(self):
...         return [Rings()]
...     class ParentMethods:
...         def jacobson_radical(self):
...             return self.ideal(Integer(0))

>>> C = SemiprimitiveRings()

>>> C
Category of semiprimitive rings
>>> C.__class__
<class '__main__.SemiprimitiveRings_with_category'>
```

*Note:* If the default name of the category (built from the name of the class) is not adequate, please implement `_repr_object_names()` to customize it.

**Realizations()**

Return the category of realizations of the parent `self` or of objects of the category `self`

**INPUT:**

- `self` -- a parent or a concrete category

*Note:* this function is actually inserted as a method in the class `Category` (see `Realizations()`). It
is defined here for code locality reasons.

EXEMPLARY:

The category of realizations of some algebra:

```
sage: Algebras(QQ).Realizations()
Join of Category of algebras over Rational Field
    and Category of realizations of unital magmas
```

```
>>> from sage.all import *
>>> Algebras(QQ).Realizations()
Join of Category of algebras over Rational Field
    and Category of realizations of unital magmas
```

The category of realizations of a given algebra:

```
sage: A = Sets().WithRealizations().example(); A
    → # needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
sage: A.Realizations()
    → # needs sage.modules
Category of realizations of
The subset algebra of {1, 2, 3} over Rational Field
```

```
sage: C = GradedHopfAlgebrasWithBasis(QQ).Realizations(); C
Join of Category of graded Hopf algebras with basis over Rational Field
    and Category of realizations of Hopf algebras over Rational Field
sage: C.super_categories()
[Category of graded Hopf algebras with basis over Rational Field,
    Category of realizations of Hopf algebras over Rational Field]
```

```
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
    → # needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
>>> A.Realizations()
    → # needs sage.modules
Category of realizations of
The subset algebra of {1, 2, 3} over Rational Field
```

```
>>> C = GradedHopfAlgebrasWithBasis(QQ).Realizations(); C
Join of Category of graded Hopf algebras with basis over Rational Field
    and Category of realizations of Hopf algebras over Rational Field
>>> C.super_categories()
[Category of graded Hopf algebras with basis over Rational Field,
    Category of realizations of Hopf algebras over Rational Field]
```

```
>>> TestSuite(C).run()
```

See also:

- `Sets().WithRealizations`
- `ClasscallMetaClass`

2.2. Categories
WithRealizations()

Return the category of parents in self endowed with multiple realizations.

INPUT:

- self – a category

See also:

- The documentation and code (sage.categories.examples.with_realizations) of Sets().WithRealizations().example() for more on how to use and implement a parent with several realizations.

- Various use cases:
  - SymmetricFunctions
  - QuasiSymmetricFunctions
  - NonCommutativeSymmetricFunctions
  - SymmetricFunctionsNonCommutingVariables
  - DescentAlgebra
  - algebras.Moebius
  - IwahoriHeckeAlgebra
  - ExtendedAffineWeylGroup

- The Implementing Algebraic Structures thematic tutorial.

- sage.categories.realizations

Note: this function is actually inserted as a method in the class Category (see WithRealizations()). It is defined here for code locality reasons.

EXAMPLES:

```python
sage: Sets().WithRealizations()
Category of sets with realizations

>>> from sage.all import *
>>> Sets().WithRealizations()
Category of sets with realizations
```
Let us now explain the concept of realizations. A parent with realizations is a facade parent (see *Sets. Facade*) admitting multiple concrete realizations where its elements are represented. Consider for example an algebra $A$ which admits several natural bases:

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
```

For each such basis $B$ one implements a parent $P_B$ which realizes $A$ with its elements represented by expanding them on the basis $B$:

```python
sage: # needs sage.modules
sage: A.F()
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: A.Out()
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: A.In()
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: A.an_element()
F[{∅}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

If $B$ and $B'$ are two bases, then the change of basis from $B$ to $B'$ is implemented by a canonical coercion between $P_B$ and $P_{B'}$:

```python
sage: # needs sage.modules
sage: F = A.F(); In = A.In(); Out = A.Out()
sage: i = In.an_element(); i
In[{∅}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
sage: F(i)
7*F[{∅}] + 3*F[{1}] + 4*F[{2}] + F[{1, 2}]
sage: F.coerce_map_from(Out)
Generic morphism:
   From: The subset algebra of {1, 2, 3} over Rational Field in the Out basis
   To: The subset algebra of {1, 2, 3} over Rational Field in the...
   Fundamental basis
```

(continues on next page)
allowing for mixed arithmetic:

```python
sage: (1 + Out.from_set(1)) * In.from_set(2,3)  # needs sage.modules
Out[{}] + 2*Out[{1}] + 2*Out[{2}] + 2*Out[{3}] + 2*Out[{1, 2}]
+ 2*Out[{1, 3}] + 4*Out[{2, 3}] + 4*Out[{1, 2, 3}]
```

In our example, there are three realizations:

```
sage: A.realizations()  # needs sage.modules
[The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis, 
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis, 
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis]
```

Instead of manually defining the shorthands \(F\), \(I\), and \(O\), as above one can just do:

```
sage: A.inject_shorthands()  # needs sage.modules
Defining \(F\) as shorthand for 
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
Defining \(I\) as shorthand for 
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis
Defining \(O\) as shorthand for 
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis
```

(continues on next page)
Defining In as shorthand for
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis
Defining Out as shorthand for
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis

Rationale

Besides some goodies described below, the role of \(A\) is threefold:

- To provide, as illustrated above, a single entry point for the algebra as a whole: documentation, access to its properties and different realizations, etc.
- To provide a natural location for the initialization of the bases and the coercions between, and other methods that are common to all bases.
- To let other objects refer to \(A\) while allowing elements to be represented in any of the realizations.

We now illustrate this second point by defining the polynomial ring with coefficients in \(A\):

```
sage: P = A['x']; P
Univariate Polynomial Ring in x over
The subset algebra of \{1, 2, 3\} over Rational Field

sage: x = P.gen()
```

In the following examples, the coefficients turn out to be all represented in the \(F\) basis:

```
sage: P.one()
F[{}

sage: (P.an_element() + 1)^2
F[{}]*x^2 + 2*F[{}]*x + F[{}

>>> from sage.all import *
```
Note how each coefficient involves a single basis which need not be that of the other coefficients. Which basis is used depends on how coercion happened during mixed arithmetic and needs not be deterministic.

One can easily coerce all coefficient to a given basis with:

```python
sage: p.map_coefficients(In)
```

From the natural notation for constructing such polynomials does not yet work:

```python
sage: In[1] * x
```

Alas, the natural notation for constructing such polynomials does not yet work:
The category of realizations of $A$

The set of all realizations of $A$, together with the coercion morphisms is a category (whose class inherits from `Category_realization_of_parent`):

```
sage: A.Realizations()  # needs sage.modules
Category of realizations of
  The subset algebra of {1, 2, 3} over Rational Field
```

The various parent realizing $A$ belong to this category:

```
sage: A.F() in A.Realizations()  # needs sage.modules
True
```

$A$ itself is in the category of algebras with realizations:

```
sage: A in Algebras(QQ).WithRealizations()  # needs sage.modules
True
```

The (mostly technical) `WithRealizations` categories are the analogs of the `*WithSeveralBases` categories in MuPAD-Combinat. They provide support tools for handling the different realizations and the morphisms between them.

Typically, `VectorSpaces(QQ).FiniteDimensional().WithRealizations()` will eventually be in charge, whenever a coercion $\phi : A \rightarrow B$ is registered, to register $\phi^{-1}$ as coercion $B \rightarrow A$ if there is none defined yet. To achieve this, `FiniteDimensionalVectorSpaces` would provide a nested class `WithRealizations` implementing the appropriate logic.
With\text{Realizations} is a \textit{regressive covariant functorial construction}. On our example, this simply means that $A$ is automatically in the category of rings with realizations (covariance):

```python
sage: A in Rings().WithRealizations()
    # needs sage.modules
True
```

and in the category of algebras (regressiveness):

```python
sage: A in Algebras(QQ)
    # needs sage.modules
True
```

\textbf{Note:} For $C$ a category, $C.\text{WithRealizations}()$ in fact calls \texttt{sage.categories.with_realizations.\text{WithRealizations}(C)}$. The later is responsible for building the hierarchy of the categories with realizations in parallel to that of their base categories, optimizing away those categories that do not provide a \texttt{WithRealizations} nested class. See \texttt{sage.categories.covariant_functorial_construction} for the technical details.

\textbf{Note:} Design question: currently \texttt{WithRealizations} is a regressive construction. That is \texttt{self.\text{WithRealizations}()} is a subcategory of \texttt{self} by default:

```python
sage: Algebras(QQ).WithRealizations().super_categories()
[Category of algebras over Rational Field,
 Category of monoids with realizations,
 Category of additive unital additive magmas with realizations]
```

Is this always desirable? For example, \texttt{AlgebrasWithBasis(QQ).WithRealizations()} should certainly be a subcategory of \texttt{Algebras(QQ)}, but not of \texttt{AlgebrasWithBasis(QQ)}. This is because \texttt{AlgebrasWithBasis(QQ)} is specifying something about the concrete realization.

\texttt{additional_structure()}\n\hspace{1em}Return whether \texttt{self} defines additional structure.

\textbf{OUTPUT:}
\begin{itemize}
  \item \texttt{self} if \texttt{self} defines additional structure and \texttt{None} otherwise. This default implementation returns \texttt{self}.
\end{itemize}
A category $C$ defines additional structure if $C$-morphisms shall preserve more structure (e.g. operations) than that specified by the super categories of $C$. For example, the category of magmas defines additional structure, namely the operation $*$ that shall be preserved by magma morphisms. On the other hand the category of rings does not define additional structure: a function between two rings that is both a unital magma morphism and a unital additive magma morphism is automatically a ring morphism.

Formally speaking $C$ defines additional structure, if $C$ is not a full subcategory of the join of its super categories: the morphisms need to preserve more structure, and thus the homsets are smaller.

By default, a category is considered as defining additional structure, unless it is a category with axiom.

EXAMPLES:

Here are some typical structure categories, with the additional structure they define:

```
sage: Sets().additional_structure()
Category of sets
sage: Magmas().additional_structure()       # `*`
Category of magmas
sage: AdditiveMagmas().additional_structure() # `+'nCategory of additive magmas
sage: LeftModules(ZZ).additional_structure()       # left multiplication by scalar
Category of left modules over Integer Ring
sage: Coalgebras(QQ).additional_structure()       # coproduct
Category of coalgebras over Rational Field
sage: Crystals().additional_structure()          # crystal operators
Category of crystals
```

On the other hand, the category of semigroups is not a structure category, since its operation $+$ is already defined by the category of magmas:

```
sage: Semigroups().additional_structure()
```

Most categories with axiom don’t define additional structure:

```
sage: Sets().Finite().additional_structure()
sage: Rings().Commutative().additional_structure()
sage: Modules(QQ).FiniteDimensional().additional_structure()
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(QQ).Unital().additional_structure()
```
As of Sage 6.4, the only exceptions are the category of unital magmas or the category of unital additive magmas (both define a unit which shall be preserved by morphisms):

```
sage: Magmas().Unital().additional_structure()
Category of unital magmas
sage: AdditiveMagmas().AdditiveUnital().additional_structure()
Category of additive unital additive magmas
```

Similarly, *functorial construction categories* don’t define additional structure, unless the construction is actually defined by their base category. For example, the category of graded modules defines a grading which shall be preserved by morphisms:

```
sage: Modules(ZZ).Graded().additional_structure()
Category of graded modules over Integer Ring
```

On the other hand, the category of graded algebras does not define additional structure; indeed an algebra morphism which is also a module morphism is a graded algebra morphism:

```
sage: Algebras(ZZ).Graded().additional_structure()
```

Similarly, morphisms are requested to preserve the structure given by the following constructions:

```
sage: Sets().Quotients().additional_structure()
Category of quotients of sets
sage: Sets().CartesianProducts().additional_structure()
Category of Cartesian products of sets
sage: Modules(QQ).TensorProducts().additional_structure()
```

This might change, as we are lacking enough data points to guarantee that this was the correct design decision.
Note: In some cases a category defines additional structure, where the structure can be useful to manipulate morphisms but where, in most use cases, we don’t want the morphisms to necessarily preserve it. For example, in the context of finite dimensional vector spaces, having a distinguished basis allows for representing morphisms by matrices; yet considering only morphisms that preserve that distinguished basis would be boring.

In such cases, we might want to eventually have two categories, one where the additional structure is preserved, and one where it’s not necessarily preserved (we would need to find an idiom for this).

At this point, a choice is to be made each time, according to the main use cases. Some of those choices are yet to be settled. For example, should by default:

- an euclidean domain morphism preserve euclidean division?

```
sage: EuclideanDomains().additional_structure()
Category of euclidean domains
```

```
>>> from sage.all import *
>>> EuclideanDomains().additional_structure()
Category of euclidean domains
```

- an enumerated set morphism preserve the distinguished enumeration?

```
sage: EnumeratedSets().additional_structure()
```

```
>>> from sage.all import *
>>> EnumeratedSets().additional_structure()
```

- a module with basis morphism preserve the distinguished basis?

```
sage: Modules(QQ).WithBasis().additional_structure()
```

```
>>> from sage.all import *
>>> Modules(QQ).WithBasis().additional_structure()
```

See also:

This method together with the methods overloading it provide the basic data to determine, for a given category, the super categories that define some structure (see `structure()`), and to test whether a category is a full subcategory of some other category (see `is_full_subcategory()`). For example, the category of Coxeter groups is not full subcategory of the category of groups since morphisms need to preserve the distinguished generators:

```
sage: CoxeterGroups().is_full_subcategory(Groups())
False
```

```
>>> from sage.all import *
>>> CoxeterGroups().is_full_subcategory(Groups())
False
```

The support for modeling full subcategories has been introduced in Issue #16340.

`all_super_categories (proper=False)`

Returns the list of all super categories of this category.

INPUT:
• proper—a boolean (default: False); whether to exclude this category.

Since Issue #11943, the order of super categories is determined by Python’s method resolution order C3 algorithm.

**Note:** Whenever speed matters, the developers are advised to use instead the lazy attributes `__all_super_categories()`, `__all_super_categories_proper()`, or `__set_of_super_categories()`, as appropriate. Simply because lazy attributes are much faster than any method.

**EXAMPLES:**

```python
sage: C = Rings(); C
Category of rings
sage: C.all_super_categories()
[Category of rings, Category of rngs, Category of semirings, ...
Category of monoids, ... Category of commutative additive groups, ...
Category of sets, Category of sets with partial maps, Category of objects]

sage: C.all_super_categories(proper = True)
[Category of rngs, Category of semirings, ...
Category of monoids, ...
Category of commutative additive groups, ...
Category of sets, Category of sets with partial maps, Category of objects]

sage: Sets().all_super_categories()
[Category of sets, Category of sets with partial maps, Category of objects]

sage: Sets().all_super_categories(proper= True)
[Category of sets with partial maps, Category of objects]

sage: Sets().all_super_categories(proper= True) is Sets().__all_super_categories
True
sage: Sets().all_super_categories(proper= True) is Sets().__all_super_categories_proper
True
```

```python
>>> from sage.all import *
>>> C = Rings(); C
Category of rings
>>> C.all_super_categories()
[Category of rings, Category of rngs, Category of semirings, ...
Category of monoids, ... Category of commutative additive groups, ...
Category of sets, Category of sets with partial maps, Category of objects]

>>> C.all_super_categories(proper = True)
[Category of rngs, Category of semirings, ...
Category of monoids, ...
Category of commutative additive groups, ...
Category of sets, Category of sets with partial maps, Category of objects]

>>> Sets().all_super_categories()
[Category of sets, Category of sets with partial maps, Category of objects]
```
class method `an_instance()`

Return an instance of this class.

**EXAMPLES:**

```python
sage: Rings.an_instance()
Category of rings
```

Parametrized categories should overload this default implementation to provide appropriate arguments:

```python
sage: Algebras.an_instance()
Category of algebras over Rational Field
sage: Bimodules.an_instance()  # needs sage.rings.real_mpfr
Category of bimodules over Rational Field on the left and Real Field with 53 bits of precision on the right
sage: AlgebraIdeals.an_instance()
Category of algebra ideals in Univariate Polynomial Ring in x over Rational Field
```

`axioms()`

Return the axioms known to be satisfied by all the objects of `self`.

Technically, this is the set of all the axioms `A` such that, if `Cs` is the category defining `A`, then `self` is a subcategory of `Cs().A()`. Any additional axiom `A` would yield a strict subcategory of `self`, at the very least `self & Cs().A()` where `Cs` is the category defining `A`.

**EXAMPLES:**

```python
sage: Monoids().axioms()
frozenset({'Associative', 'Unital'})
sage: (EnumeratedSets().Infinite() & Sets().Facade()).axioms()
frozenset({'Enumerated', 'Facade', 'Infinite'})
```
>>> from sage.all import *
>>> Monoids().axioms()
frozenset(('Associative', 'Unital'))
>>> (EnumeratedSets().Infinite() & Sets().Facade()).axioms()
frozenset(('Enumerated', 'Facade', 'Infinite'))

category()

Return the category of this category. So far, all categories are in the
category of objects.

EXAMPLES:

sage: Sets().category()
Category of objects
sage: VectorSpaces(QQ).category()
Category of objects

>>> from sage.all import *
>>> Sets().category()
Category of objects

>>>
>>> VectorSpaces(QQ).category()
Category of objects

category_graph()

Returns the graph of all super categories of this category

EXAMPLES:

sage: C = Algebras(QQ)
sage: G = C.category_graph()  # needs sage.graphs
sage: G.is_directed_acyclic()  # needs sage.graphs
True

The girth of a directed acyclic graph is infinite, however, the girth of the underlying undirected graph is 4 in
this case:

sage: Graph(G).girth()  # needs sage.graphs
4

>>>

element_class()

A common super class for all elements of parents in this category (and its subcategories).
This class contains the methods defined in the nested class `self.ElementMethods` (if it exists), and has as bases the element classes of the super categories of `self`.

See also:

- `parent_class()`, `morphism_class()`
- `Category` for details

**EXAMPLES:**

```python
sage: C = Algebras(QQ).element_class; C
<class 'sage.categories.algebras.Algebras.element_class'>
sage: type(C)
<class 'sage.structure.dynamic_class.DynamicMetaClass'>
```

By Issue #11935, some categories share their element classes. For example, the element class of an algebra only depends on the category of the base. A typical example is the category of algebras over a field versus algebras over a non-field:

```python
sage: Algebras(GF(5)).element_class is Algebras(GF(3)).element_class
True
sage: Algebras(QQ).element_class is Algebras(ZZ).element_class
False
sage: Algebras(ZZ['t']).element_class is Algebras(ZZ['t','x']).element_class
True
```

These classes are constructed with `__slots__ = ()`, so instances may not have a `__dict__`:

```python
sage: E = FiniteEnumeratedSets().element_class
sage: E.__dictoffset__
0
```

See also:

`parent_class()`
example (*args, **keywords)

Returns an object in this category. Most of the time, this is a parent.

This serves three purposes:

- Give a typical example to better explain what the category is all about. (and by the way prove that the category is non empty :-))
- Provide a minimal template for implementing other objects in this category
- Provide an object on which to test generic code implemented by the category

For all those applications, the implementation of the object shall be kept to a strict minimum. The object is therefore not meant to be used for other applications; most of the time a full featured version is available elsewhere in Sage, and should be used instead.

Technical note: by default FooBar(...).example() is constructed by looking up sage.categories.examples.foo_bar.Example and calling it as Example(). Extra positional or named parameters are also passed down. For a category over base ring, the base ring is further passed down as an optional argument.

Categories are welcome to override this default implementation.

EXAMPLES:

```
sage: Semigroups().example()
An example of a semigroup: the left zero semigroup

sage: Monoids().Subquotients().example()
NotImplemented
```

full_super_categories()

Return the immediate full super categories of self.

See also:

- super_categories()
- is_full_subcategory()

Warning: The current implementation selects the full subcategories among the immediate super categories of self. This assumes that, if $C \subset B \subset A$ is a chain of categories and $C$ is a full subcategory of $A$, then $C$ is a full subcategory of $B$ and $B$ is a full subcategory of $A$.

This assumption is guaranteed to hold with the current model and implementation of full subcategories in Sage. However, mathematically speaking, this is too restrictive. This indeed prevents the complete modelling of situations where any $A$ morphism between elements of $C$ automatically preserves the $B$ structure. See below for an example.

EXAMPLES:

A semigroup morphism between two finite semigroups is a finite semigroup morphism:
sage: Semigroups().Finite().full_super_categories()
[Category of semigroups]

>>> from sage.all import *

>>> Semigroups().Finite().full_super_categories()
[Category of semigroups]

On the other hand, a semigroup morphism between two monoids is not necessarily a monoid morphism (which must map the unit to the unit):

sage: Monoids().super_categories()
[Category of semigroups, Category of unital magmas]

sage: Monoids().full_super_categories()
[Category of unital magmas]

>>> from sage.all import *

>>> Monoids().super_categories()
[Category of semigroups, Category of unital magmas]

>>> Monoids().full_super_categories()
[Category of unital magmas]

Any semigroup morphism between two groups is automatically a monoid morphism (in a group the unit is the unique idempotent, so it has to be mapped to the unit). Yet, due to the limitation of the model advertised above, Sage currently cannot be taught that the category of groups is a full subcategory of the category of semigroups:

sage: Groups().full_super_categories()  # todo: not implemented
[Category of monoids, Category of semigroups, Category of inverse unital → magmas]

sage: Groups().full_super_categories()
[Category of monoids, Category of inverse unital magmas]

>>> from sage.all import *

>>> Groups().full_super_categories()  # todo: not implemented
[Category of monoids, Category of semigroups, Category of inverse unital → magmas]

>>> Groups().full_super_categories()
[Category of monoids, Category of inverse unital magmas]

is_abelian()

Return whether this category is abelian.

An abelian category is a category satisfying:

- It has a zero object;
- It has all pullbacks and pushouts;
- All monomorphisms and epimorphisms are normal.

Equivalently, one can define an increasing sequence of conditions:

- A category is pre-additive if it is enriched over abelian groups (all homsets are abelian groups and composition is bilinear);
- A pre-additive category is additive if every finite set of objects has a biproduct (we can form direct sums and direct products);
- An additive category is pre-abelian if every morphism has both a kernel and a cokernel;
A pre-abelian category is abelian if every monomorphism is the kernel of some morphism and every epimorphism is the cokernel of some morphism.

**EXAMPLES:**

```python
sage: Modules(ZZ).is_abelian()
True
sage: FreeModules(ZZ).is_abelian()
False
sage: FreeModules(QQ).is_abelian()
True
sage: CommutativeAdditiveGroups().is_abelian()
True
sage: Semigroups().is_abelian()
Traceback (most recent call last):
... NotImplementedError: is_abelian
```

```python
>>> from sage.all import *
>>> Modules(ZZ).is_abelian()
True
>>> FreeModules(ZZ).is_abelian()
False
>>> FreeModules(QQ).is_abelian()
True
>>> CommutativeAdditiveGroups().is_abelian()
True
>>> Semigroups().is_abelian()
Traceback (most recent call last):
... NotImplementedError: is_abelian
```

**is_full_subcategory (other)**

Return whether `self` is a full subcategory of `other`.

A subcategory $B$ of a category $A$ is a full subcategory if any $A$-morphism between two objects of $B$ is also a $B$-morphism (the reciprocal always holds: any $B$-morphism between two objects of $B$ is an $A$-morphism).

This is computed by testing whether `self` is a subcategory of `other` and whether they have the same structure, as determined by `structure()` from the result of `additional_structure()` on the super categories.

**Warning:** A positive answer is guaranteed to be mathematically correct. A negative answer may mean that Sage has not been taught enough information (or can not yet within the current model) to derive this information. See `full_super_categories()` for a discussion.

See also:

- `is_subcategory()`
- `full_super_categories()`

**EXAMPLES:**

```python
sage: Magmas().Associative().is_full_subcategory(Magmas())
True
```
Here are two typical examples of false negatives:

```
sage: Groups().is_full_subcategory(Semigroups())
False
sage: Groups().is_full_subcategory(Semigroups()) # todo: not implemented
True
sage: Fields().is_full_subcategory(Rings())
False
sage: Fields().is_full_subcategory(Rings()) # todo: not implemented
True
```

Todo: The latter is a consequence of `EuclideanDomains` currently being a structure category. Is this what we want?

```
sage: EuclideanDomains().is_full_subcategory(Rings())
False
```

### `is_subcategory(c)`

Returns True if self is naturally embedded as a subcategory of c.

**EXAMPLES:**

```
sage: AbGrps = CommutativeAdditiveGroups()
sage: Rings().is_subcategory(AbGrps)
```

(continues on next page)
True
\begin{verbatim}
sage: AbGrps.is_subcategory(Rings())
False
\end{verbatim}

>>> from sage.all import *

\begin{verbatim}
>>>
AbGrps = CommutativeAdditiveGroups()
>>> Rings().is_subcategory(AbGrps)
True
>>> AbGrps.is_subcategory(Rings())
False
\end{verbatim}

The `is_subcategory` function takes into account the base.

\begin{verbatim}
sage: M3 = VectorSpaces(FiniteField(3))
sage: M9 = VectorSpaces(FiniteField(9, 'a'))
# needs sage.rings.finite_rings
sage: M3.is_subcategory(M9)
False
\end{verbatim}

Join categories are properly handled:

\begin{verbatim}
sage: CatJ = Category.join((CommutativeAdditiveGroups(), Semigroups()))
sage: Rings().is_subcategory(CatJ)
True
\end{verbatim}

\begin{verbatim}
>>>
V3 = VectorSpaces(FiniteField(3))
sage: PoSet = PartiallyOrderedSets()
sage: PoV3 = Category.join((V3, PoSet))
sage: AlgebrasWithBasis(FiniteField(3))
sage: A3 = Category.join((V3, PoSet))
sage: PoA3 = Category.join((A3, PoSet))
sage: PoA3.is_subcategory(PoV3)
True
sage: PoV3.is_subcategory(PoV3)
True
sage: PoV3.is_subcategory(PoA3)
False
\end{verbatim}
A3 = AlgebrasWithBasis(FiniteField(Integer(3)))
PoA3 = Category.join((A3, POSet))
PoA3.is_subcategory(PoV3)
True
PoV3.is_subcategory(PoV3)
True
PoV3.is_subcategory(PoA3)
False

static join (categories, as_list=False, ignore_axioms=(), axioms=())

Return the join of the input categories in the lattice of categories.

At the level of objects and morphisms, this operation corresponds to intersection: the objects and morphisms of a join category are those that belong to all its super categories.

INPUT:

• categories – a list (or iterable) of categories

• as_list – a boolean (default: False); whether the result should be returned as a list

• axioms – a tuple of strings; the names of some supplementary axioms

See also:

__and__() for a shortcut

EXAMPLES:

sage: J = Category.join((Groups(), CommutativeAdditiveMonoids())); J
Join of Category of groups and Category of commutative additive monoids
sage: J.super_categories()
[Category of groups, Category of commutative additive monoids]
J.all_super_categories(proper=True)
[Category of groups, ..., Category of magmas,
Category of commutative additive monoids, ..., Category of additive magmas,
Category of sets, ...]

As a short hand, one can use:

sage: Groups() & CommutativeAdditiveMonoids()
Join of Category of groups and Category of commutative additive monoids

This is a commutative and associative operation:
The join of a single category is the category itself:

\[
\text{sage: Category.join([\text{Monoids()}])} \\
\text{Category of monoids}
\]

Similarly, the join of several mutually comparable categories is the smallest one:

\[
\text{sage: Category.join((\text{Sets()}, \text{Rings()}, \text{Monoids()}))} \\
\text{Category of rings}
\]

In particular, the unit is the top category \textit{Objects}:

\[
\text{sage: Groups() & Objects()} \\
\text{Category of groups}
\]
If the optional parameter \texttt{as\_list} is \texttt{True}, this returns the super categories of the join as a list, without constructing the join category itself:

```
sage: Category.join((Groups(), CommutativeAdditiveMonoids()), as_list=True)
[Category of groups, Category of commutative additive monoids]
sage: Category.join((Sets(), Rings(), Monoids()), as_list=True)
[Category of rings]
sage: Category.join((Modules(ZZ), FiniteFields()), as_list=True)
[Category of finite enumerated fields, Category of modules over Integer Ring]
sage: Category.join([], as_list=True)
[]
sage: Category.join([Groups()], as_list=True)
[Category of groups]
sage: Category.join([Groups() & Posets()], as_list=True)
[Category of groups, Category of posets]
```

Support for axiom categories (TODO: put here meaningful examples):

```
sage: Sets().Facade() & Sets().Infinite()
Category of facade infinite sets
sage: Magmas().Infinite() & Sets().Facade()
Category of facade infinite magmas
sage: FiniteSets() & Monoids()
Category of finite monoids
sage: Rings().Commutative() & Sets().Finite()
Category of finite commutative rings
```

```>> from sage.all import *
>>> Category.join((Groups(), CommutativeAdditiveMonoids()), as_list=True)
[Category of groups, Category of commutative additive monoids]
>>> Category.join((Sets(), Rings(), Monoids()), as_list=True)
[Category of rings]
>>> Category.join((Modules(ZZ), FiniteFields()), as_list=True)
[Category of finite enumerated fields, Category of modules over Integer Ring]
>>> Category.join([], as_list=True)
[]
>>> Category.join([Groups()], as_list=True)
[Category of groups]
>>> Category.join([Groups() & Posets()], as_list=True)
[Category of groups, Category of posets]
```
and Category of finite set algebras over Rational Field

\begin{verbatim}
sage: UniqueFactorizationDomains() & Algebras(QQ)
Join of Category of unique factorization domains
and Category of commutative algebras over Rational Field
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> AlgebrasWithBasis(QQ) & FiniteSets().Algebras(QQ)
Join of Category of finite dimensional algebras with basis over Rational Field
and Category of finite set algebras over Rational Field
>>> UniqueFactorizationDomains() & Algebras(QQ)
Join of Category of unique factorization domains
and Category of commutative algebras over Rational Field
\end{verbatim}

\begin{verbatim}
static meet (categories)
Returns the meet of a list of categories

INPUT:

• categories—a non empty list (or iterable) of categories

See also:

__or__() for a shortcut

EXAMPLES:

\begin{verbatim}
sage: Category.meet([Algebras(ZZ), Algebras(QQ), Groups()])
Category of monoids
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> Category.meet([Algebras(ZZ), Algebras(QQ), Groups()])
Category of monoids
\end{verbatim}

That meet of an empty list should be a category which is a subcategory of all categories, which does not make practical sense:

\begin{verbatim}
sage: Category.meet([])
Traceback (most recent call last):
...
ValueError: The meet of an empty list of categories is not implemented
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> Category.meet([])
Traceback (most recent call last):
...
ValueError: The meet of an empty list of categories is not implemented
\end{verbatim}

\begin{verbatim}
morphism_class()
A common super class for all morphisms between parents in this category (and its subcategories).
This class contains the methods defined in the nested class self.MorphismMethods (if it exists), and
has as bases the morphism classes of the super categories of self.

See also:

• parent_class(), element_class()
\end{verbatim}
• Category for details

EXAMPLES:

```python
sage: C = Algebras(QQ).morphism_class; C
<class 'sage.categories.algebras.Algebras.morphism_class'>
sage: type(C)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
```

```python
>>> from sage.all import *

>>> C = Algebras(QQ).morphism_class; C
<class 'sage.categories.algebras.Algebras.morphism_class'>

>>> type(C)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
```

```
or_subcategory(category=None, join=False)

Return category or self if category is None.

INPUT:
• category – a sub category of self, tuple/list thereof, or None
• join – a boolean (default: False)

OUTPUT:
• a category

EXAMPLES:

```python
sage: Monoids().or_subcategory(Groups())
Category of groups
sage: Monoids().or_subcategory(Non)
Category of monoids
```

```python
>>> from sage.all import *

>>> Monoids().or_subcategory(Groups())
Category of groups

>>> Monoids().or_subcategory(Non)
Category of monoids
```

If category is a list/tuple, then a join category is returned:

```python
sage: Monoids().or_subcategory((CommutativeAdditiveMonoids(), Groups()))
Join of Category of groups and Category of commutative additive monoids
```

```python
>>> from sage.all import *

>>> Monoids().or_subcategory((CommutativeAdditiveMonoids(), Groups()))
Join of Category of groups and Category of commutative additive monoids
```

If join is False, an error if raised if category is not a subcategory of self:

```python
sage: Monoids().or_subcategory(EnumeratedSets())
Traceback (most recent call last):
...
ValueError: Subcategory of `Category of monoids` required; got `Category of enumerated sets`
```python
>>> from sage.all import *
>>> Monoids().or_subcategory(EnumeratedSets())
Traceback (most recent call last):
... ValueError: Subcategory of `Category of monoids` required; got `Category of enumerated sets`
```

Otherwise, the two categories are joined together:

```python
sage: Monoids().or_subcategory(EnumeratedSets(), join=True)
Category of enumerated monoids
```

parent_class()

A common super class for all parents in this category (and its subcategories).

This class contains the methods defined in the nested class `self.ParentMethods` (if it exists), and has as bases the parent classes of the super categories of `self`.

See also:

- `element_class()`, `morphism_class()`
- `Category` for details

EXAMPLES:

```python
sage: C = Algebras(QQ).parent_class; C
<class 'sage.categories.algebras.Algebras.parent_class'>
sage: type(C)
<class 'sage.structure.dynamic_class.DynamicMetaClass'>
```

By Issue #11935, some categoriess share their parent classes. For example, the parent class of an algebra only depends on the category of the base ring. A typical example is the category of algebras over a finite field versus algebras over a non-field:

```python
sage: Algebras(GF(7)).parent_class is Algebras(GF(5)).parent_class
True
sage: Algebras(QQ).parent_class is Algebras(ZZ).parent_class
False
sage: Algebras(ZZ['t']).parent_class is Algebras(ZZ['t', 'x']).parent_class
True
```

```python
>>> from sage.all import *
>>> Algebras(GF(Integer(7))).parent_class is Algebras(GF(Integer(5))).parent_class
True
```
(continues on next page)
See `CategoryWithParameters` for an abstract base class for categories that depend on parameters, even though the parent and element classes only depend on the parent or element classes of its super categories. It is used in `Bimodules`, `Category_over_base` and `sage.categories.category.JoinCategory`.

**required_methods()**

Returns the methods that are required and optional for parents in this category and their elements.

**EXAMPLES:**

```python
cat = Algebras(QQ)
cat = Algebras(QQ).required_methods()
cat = {element: {'optional': ['__add__', '__mul__'], 'required': ['__bool__']},
parent: {'optional': ['algebra_generators'], 'required': ['__contains__']}
```

**structure()**

Return the structure self is endowed with.

This method returns the structure that morphisms in this category shall be preserving. For example, it tells that a ring is a set endowed with a structure of both a unital magma and an additive unital magma which satisfies some further axioms. In other words, a ring morphism is a function that preserves the unital magma and additive unital magma structure.

In practice, this returns the collection of all the super categories of self that define some additional structure, as a frozen set.

**EXAMPLES:**

```python
cat = Objects()
cat = cat.structure()
cat = frozenset()
cat = cat = def structure(C):
.....: return Category._sort(C.structure())
cat = cat = structure(Sets())
cat = (Category of sets, Category of sets with partial maps)
cat = cat = structure(Magmas())
cat = (Category of magmas, Category of sets, Category of sets with partial maps)
```
In the following example, we only list the smallest structure categories to get a more readable output:

```python
def structure(C):
    return Category._sort_uniq(C.structure())

structure(Magmas())
structure(Rings())
structure(Fields())
structure(Algebras(QQ))
structure(HopfAlgebras(QQ).Graded().WithBasis().Connected())
```

This method is used in `is_full_subcategory()` for deciding whether a category is a full subcategory of some other category, and for documentation purposes. It is computed recursively from the result of `additional_structure()` on the super categories of `self`.

**subcategory_class()**

A common superclass for all subcategories of this category (including this one).

This class derives from `Dsubcategory_class` for each super category `D` of `self`, and includes all the methods from the nested class `self.SubcategoryMethods`, if it exists.

**See also:**

- Issue #12895
- `parent_class()`
- `element_class()`
• `_make_named_class()`

EXAMPLES:

```python
sage: cls = Rings().subcategory_class; cls
<class 'sage.categories.rings.Rings.subcategory_class'>
sage: type(cls)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
```

```python
>>> from sage.all import *
```

```python
>>>
```

Rings() is an instance of this class, as well as all its subcategories:

```python
sage: isinstance(Rings(), cls)
True
sage: isinstance(AlgebrasWithBasis(QQ), cls)
True
```

```python
>>> from sage.all import *
```

```python
>>>
```

`super_categories()`

Return the immediate super categories of `self`.

OUTPUT:

• a duplicate-free list of categories.

Every category should implement this method.

EXAMPLES:

```python
sage: Groups().super_categories()
[Category of monoids, Category of inverse unital magmas]
sage: Objects().super_categories()
[]
```

```python
>>> from sage.all import *
```

```python
>>>
```

Note: Since Issue #10963, the order of the categories in the result is irrelevant. For details, see On the order of super categories.

Note: Whenever speed matters, developers are advised to use the lazy attribute `_super_categories()` instead of calling this method.
class sage.categories.category.CategoryWithParameters
    Bases: Category

A parametrized category whose parent/element classes depend only on its super categories.

Many categories in Sage are parametrized, like \(C = \text{Algebras}(K)\) which takes a base ring as parameter. In many cases, however, the operations provided by \(C\) in the parent class and element class depend only on the super categories of \(C\). For example, the vector space operations are provided if and only if \(K\) is a field, since \(\text{VectorSpaces}(K)\) is a super category of \(C\) only in that case. In such cases, and as an optimization (see Issue #11935), we want to use the same parent and element class for all fields. This is the purpose of this abstract class.

Currently, JoinCategory, Category_over_base and Bimodules inherit from this class.

EXAMPLES:

```python
sage: C1 = Algebras(GF(5))
sage: C2 = Algebras(GF(3))
sage: C3 = Algebras(ZZ)
sage: from sage.categories.category import CategoryWithParameters
sage: isinstance(C1, CategoryWithParameters)
True
sage: C1.parent_class is C2.parent_class
True
sage: C1.parent_class is C3.parent_class
False
```

```python
>>> from sage.all import *
>>> C1 = Algebras(GF(Integer(5)))
>>> C2 = Algebras(GF(Integer(3)))
>>> C3 = Algebras(ZZ)
>>> from sage.categories.category import CategoryWithParameters
>>> isinstance(C1, CategoryWithParameters)
True
>>> C1.parent_class is C2.parent_class
True
>>> C1.parent_class is C3.parent_class
False
```

Category._make_named_class(name, method_provider, cache=False, picklable=True)
    Construction of the parent/element/... class of self.

INPUT:

- name – a string; the name of the class as an attribute of self. E.g. “parent_class”
- method_provider – a string; the name of an attribute of self that provides methods for the new class (in addition to those coming from the super categories). E.g. “ParentMethods”
- cache – a boolean or ignore_reduction (default: False) (passed down to dynamic_class; for internal use only)
- picklable – a boolean (default: True)

ASSUMPTION:

It is assumed that this method is only called from a lazy attribute whose name coincides with the given name.

OUTPUT:

A dynamic class with bases given by the corresponding named classes of self’s super_categories, and methods taken from the class getattr(self,method_provider).
Note:

- In this default implementation, the reduction data of the named class makes it depend on `self`. Since the result is going to be stored in a lazy attribute of `self` anyway, we may as well disable the caching in `dynamic_class` (hence the default value `cache=False`).

- `CategoryWithParameters` overrides this method so that the same parent/element/… classes can be shared between closely related categories.

- The bases of the named class may also contain the named classes of some indirect super categories, according to `_super_categories_for_classes()`. This is to guarantee that Python will build consistent method resolution orders. For background, see `sage.misc.c3_controlled`.

See also:

`CategoryWithParameters._make_named_class()`

EXAMPLES:

```
sage: PC = Rings()._make_named_class("parent_class", "ParentMethods"); PC
<class 'sage.categories.rings.Rings.parent_class'>
sage: type(PC)
<class 'sage.structure.dynamic_class.DynamicMetaClass'>
sage: PC.__bases__
(<class 'sage.categories.rngs.Rngs.parent_class'>,
 <class 'sage.categories.semirings.Semirings.parent_class'>)
```

```
>>> from sage.all import *
```

```
>>> PC = Rings()._make_named_class("parent_class", "ParentMethods"); PC
<class 'sage.categories.rings.Rings.parent_class'>
```

```
>>> from sage.all import *
```

```
>>> PC = Rings()._make_named_class("parent_class", "ParentMethods"); PC
<class 'sage.categories.ring.Rings.parent_class'>
```

Reduction for pickling also assumes the existence of this lazy attribute:
class sage.categories.category.JoinCategory(super_categories, **kwds)

A class for joins of several categories. Do not use directly; see Category.join instead.

EXAMPLES:

```python
sage: from sage.categories.category import JoinCategory
sage: J = JoinCategory((Groups(), CommutativeAdditiveMonoids())); J
Join of Category of groups and Category of commutative additive monoids
sage: J.super_categories()
[Category of groups, Category of commutative additive monoids]
sage: J.all_super_categories(proper=True)
[Category of groups, ..., Category of magmas, Category of commutative additive monoids, ..., Category of additive magmas, Category of sets, Category of sets with partial maps, Category of objects]
```

By Issue #11935, join categories and categories over base rings inherit from CategoryWithParameters. This allows for sharing parent and element classes between similar categories. For example, since group algebras belong to a join category and since the underlying implementation is the same for all finite fields, we have:

```python
sage: # needs sage.groups sage.rings.finite_rings
sage: G = SymmetricGroup(10)
```

```python
sage: A3 = G.algebra(GF(3))
sage: A5 = G.algebra(GF(5))
sage: type(A3.category())
<class 'sage.categories.category.JoinCategory_with_category'>
sage: type(A3) is type(A5)
True
```

```python
>>> from sage.all import *
```

```python
>>> G = SymmetricGroup(Integer(10))
```

```python
>>> A3 = G.algebra(GF(Integer(3)))
```

(continues on next page)
Category Framework, Release 10.4

>>> A5 = G.algebra(GF(Integer(5)))
>>> type(A3.category())
<class 'sage.categories.category.JoinCategory_with_category'>
>>> type(A3) is type(A5)
True

Category._repr_object_names()
Return the name of the objects of this category.

EXAMPLES:

    sage: FiniteGroups()._repr_object_names()
    'finite groups'
    sage: AlgebrasWithBasis(QQ)._repr_object_names()
    'algebras with basis over Rational Field'

>>> from sage.all import *

>>> FiniteGroups()._repr_object_names()
'finite groups'
>>> AlgebrasWithBasis(QQ)._repr_object_names()
'algebras with basis over Rational Field'

Category._repr()
Return the print representation of this category.

EXAMPLES:

    sage: Sets() # indirect doctest
    Category of sets

>>> from sage.all import *

>>> Sets() # indirect doctest
Category of sets

Category._without_axioms(named=False)
Return the category without the axioms that have been added to create it.

INPUT:

- named -- a boolean (default: False)

Todo: Improve this explanation.

If named is True, then this stops at the first category that has an explicit name of its own. See category_with_axiom.CategoryWithAxiom._without_axioms()

EXAMPLES:

    sage: Sets()._without_axioms()
    Category of sets
    sage: Semigroups()._without_axioms()
    Category of magmas
    sage: Algebras(QQ).Commutative().WithBasis()._without_axioms()
    Category of magmatic algebras over Rational Field
    sage: Algebras(QQ).Commutative().WithBasis()._without_axioms(named=True)
    Category of algebras over Rational Field

2.2. Categories 97
additional_structure()
Return None.
Indeed, a join category defines no additional structure.
See also:
Category.additional_structure()

EXAMPLES:
sage: Modules(ZZ).additional_structure()

is_subcategory(C)
Check whether this join category is subcategory of another category C.
EXAMPLES:
sage: Category.join([Rings(), Modules(QQ)]).is_subcategory(Category.join([Rngs(), Bimodules(QQ, QQ)]))
True

super_categories()
Returns the immediate super categories, as per Category.super_categories().
EXAMPLES:
sage: from sage.categories.category import JoinCategory
sage: JoinCategory({Semigroups(), FiniteEnumeratedSets()}).super_categories()
[Category of semigroups, Category of finite enumerated sets]

sage.categories.category.category_graph(categories=None)
Return the graph of the categories in Sage.
INPUT:
• categories – a list (or iterable) of categories

If categories is specified, then the graph contains the mentioned categories together with all their super categories. Otherwise the graph contains (an instance of) each category in sage.categories.all (e.g. Algebras(QQ) for algebras).

For readability, the names of the category are shortened.

**Todo:** Further remove the base ring (see also Issue #15801).

**EXAMPLES:**

```python
sage: G = sage.categories.category.category_graph(categories=[Groups()])  #...
"needs sage.graphs"
sage: G.vertices(sort=True)  #...
"needs sage.graphs"
['groups', 'inverse unital magmas', 'magmas', 'monoids', 'objects', 'semigroups', 'sets', 'sets with partial maps', 'unital magmas']
sage: G.plot()  #...
"needs sage.graphs sage.plot"
Graphics object consisting of 20 graphics primitives
```

```python
>>> from sage.all import *

>>> G = sage.categories.category.category_graph(categories=[Groups()])  #...
"needs sage.graphs"
>>> G.vertices(sort=True)  #...
"needs sage.graphs"
['groups', 'inverse unital magmas', 'magmas', 'monoids', 'objects', 'semigroups', 'sets', 'sets with partial maps', 'unital magmas']
```

```python
sage: from sage.categories.category import category_sample

sage: sage.categories.category.category_sample()

Return a sample of categories.

It is constructed by looking for all concrete category classes declared in sage.categories.all, calling Category.an_instance() on those and taking all their super categories.

**EXAMPLES:**

```python
sage: from sage.categories.category import category_sample
sage: sorted(category_sample(), key=str)  #...
"needs sage.groups"

[Category of Coxeter groups,
Category of Dedekind domains,
Category of G-sets for Symmetric group of order 8! as a permutation group,
Category of Hecke modules over Rational Field,
Category of Hopf algebras over Rational Field,
```

(continues on next page)
Category of Hopf algebras with basis over Rational Field,
Category of Jacobians over Rational Field,
Category of Lie algebras over Rational Field,
Category of Weyl groups,
Category of abelian varieties over Rational Field,
Category of additive magmas, ...
Category of fields, ...
Category of graded Hopf algebras with basis over Rational Field, ...
Category of modular abelian varieties over Rational Field, ...
Category of simplicial complexes, ...
Category of vector spaces over Rational Field, ...

```python
>>> from sage.all import *
>>> from sage.categories.category import category_sample
>>> sorted(category_sample(), key=str)  # needs sage.categories
[Category of Coxeter groups,
Category of Dedekind domains,
Category of G-sets for Symmetric group of order 8! as a permutation group,
Category of Hecke modules over Rational Field,
Category of Hopf algebras over Rational Field,
Category of Hopf algebras with basis over Rational Field,
Category of Jacobians over Rational Field,
Category of Lie algebras over Rational Field,
Category of Weyl groups,
Category of abelian varieties over Rational Field,
Category of additive magmas, ...
Category of fields, ...
Category of graded Hopf algebras with basis over Rational Field, ...
Category of modular abelian varieties over Rational Field, ...
Category of simplicial complexes, ...
Category of vector spaces over Rational Field, ...
```

`sage.categories.category.is_Category(x)`

Returns True if x is a category.

**EXAMPLES:**

```python
sage: sage.categories.category.is_Category(CommutativeAdditiveSemigroups())
doctest:warning...
DeprecationWarning: the function is_Category is deprecated;
use 'isinstance(..., Category)' instead
See https://github.com/sagemath/sage/issues/37922 for details.
True
sage: sage.categories.category.is_Category(ZZ)
False
```

```python
>>> from sage.all import *
>>> sage.categories.category.is_Category(CommutativeAdditiveSemigroups())
doctest:warning...
DeprecationWarning: the function is_Category is deprecated;
use 'isinstance(..., Category)' instead
See https://github.com/sagemath/sage/issues/37922 for details.
True
>>> sage.categories.category.is_Category(ZZ)
False
```
2.3 Axioms

This documentation covers how to implement axioms and proceeds with an overview of the implementation of the axiom infrastructure. It assumes that the reader is familiar with the category primer, and in particular its section about axioms.

2.3.1 Implementing axioms

Simple case involving a single predefined axiom

Suppose that one wants to provide code (and documentation, tests, ...) for the objects of some existing category $C_s()$ that satisfy some predefined axiom $A$.

The first step is to open the hood and check whether there already exists a class implementing the category $C_s() \cdot A()$. For example, taking $C_s=$Semigroups and the Finite axiom, there already exists a class for the category of finite semigroups:

\begin{verbatim}
sage: Semigroups().Finite()
Category of finite semigroups
sage: type(Semigroups().Finite())
<class 'sage.categories.finite_semigroups.FiniteSemigroups_with_category'>
\end{verbatim}

In this case, we say that the category of semigroups implements the axiom Finite, and code about finite semigroups should go in the class FiniteSemigroups (or, as usual, in its nested classes ParentMethods, ElementMethods, and so on).

On the other hand, there is no class for the category of infinite semigroups:

\begin{verbatim}
sage: Semigroups().Infinite()
Category of infinite semigroups
sage: type(Semigroups().Infinite())
<class 'sage.categories.category.JoinCategory_with_category'>
\end{verbatim}

This category is indeed just constructed as the intersection of the categories of semigroups and of infinite sets respectively:

\begin{verbatim}
>>> from sage.all import *
>>> Semigroups().Infinite()
Category of infinite semigroups
>>> type(Semigroups().Infinite())
<class 'sage.categories.category.JoinCategory_with_category'>
\end{verbatim}

In this case, one needs to create a new class to implement the axiom Infinite for this category. This boils down to adding a nested class Semigroups.Infinite inheriting from CategoryWithAxiom.
In the following example, we implement a category $\text{Cs}$, with a subcategory for the objects satisfying the $\text{Finite}$ axiom defined in the super category $\text{Sets}$ (we will see later on how to define new axioms):

```python
sage: from sage.categories.category_with_axiom import CategoryWithAxiom
sage: class Cs(Category):
....:     def super_categories(self):
....:         return [Sets()]
....:     class Finite(CategoryWithAxiom):
....:         class ParentMethods:
....:             def foo(self):
....:                 print("I am a method on finite Cs'\'s")
```

```python
>>> from sage.all import *
>>> from sage.categories.category_with_axiom import CategoryWithAxiom
>>> class Cs(Category):
...     def super_categories(self):
...         return [Sets()]
...     class Finite(CategoryWithAxiom):
...         class ParentMethods:
...             def foo(self):
...                 print("I am a method on finite Cs'\'s")
```

```python
sage: Cs().Finite()
Category of finite cs
sage: Cs().Finite().super_categories()
[Category of finite sets, Category of cs]
sage: Cs().Finite().all_super_categories()
[Category of finite cs, Category of finite sets, Category of cs, Category of sets, ...]
sage: Cs().Finite().axioms()
frozenset({'Finite'})
```

```python
>>> from sage.all import *
>>> Cs().Finite()
Category of finite cs
>>> Cs().Finite().super_categories()
[Category of finite sets, Category of cs]
>>> Cs().Finite().all_super_categories()
[Category of finite cs, Category of finite sets, Category of cs, Category of sets, ...]
>>> Cs().Finite().axioms()
frozenset({'Finite'})
```

Now a parent declared in the category $\text{Cs}().\text{Finite}()$ inherits from all the methods of finite sets and of finite $\text{Cs}$'s, as desired:

```python
sage: P = Parent(category=Cs().Finite())
sage: P.is_finite()  # Provided by Sets.Finite.ParentMethods
True
sage: P.foo()  # Provided by Cs.Finite.ParentMethods
I am a method on finite Cs'\'s
```

```python
>>> from sage.all import *
>>> P = Parent(category=Cs().Finite())
>>> P.is_finite()  # Provided by Sets.Finite.ParentMethods
True
```

(continues on next page)
Note:

- This follows the same idiom as for **Covariant Functorial Constructions**.

- From an object oriented point of view, any subcategory `Cs()` of `Sets` inherits a `Finite` method. Usually `Cs` could complement this method by overriding it with a method `Cs.Finite` which would make a super call to `Sets.Finite` and then do extra stuff.

In the above example, `Cs` also wants to complement `Sets.Finite`, though not by doing more stuff, but by providing it with an additional mixin class containing the code for finite `Cs`. To keep the analogy, this mixin class is to be put in `Cs.Finite`.

- By defining the axiom `Finite, Sets` fixes the semantic of `Cs.Finite()` for all its subcategories `Cs`: namely “the category of `Cs` which are finite as sets”. Hence, for example, `Modules.Free.Finite` cannot be used to model the category of free modules of finite rank, even though their traditional name “finite free modules” might suggest it.

- It may come as a surprise that we can actually use the same name `Finite` for the mixin class and for the method defining the axiom; indeed, by default a class does not have a binding behavior and would completely override the method. See the section **Defining a new axiom** for details and the rationale behind it.

An alternative would have been to give another name to the mixin class, like `FiniteCategory`. However this would have resulted in more namespace pollution, whereas using `Finite` is already clear, explicit, and easier to remember.

- Under the hood, the category `Cs().Finite()` is aware that it has been constructed from the category `Cs()` by adding the axiom `Finite`:

```python
sage: Cs().Finite()._base_category()
Category of cs
sage: Cs().Finite()._axiom
'Finite'
```

```python
>>> from sage.all import *

sage: Cs().Finite()._base_category()
Category of cs
sage: Cs().Finite()._axiom
'Finite'
```

Over time, the nested class `Cs.Finite` may become large and too cumbersome to keep as a nested subclass of `Cs`. Or the category with axiom may have a name of its own in the literature, like `semigroups` rather than `associative magmas`, or `fields` rather than `commutative division rings`. In this case, the category with axiom can be put elsewhere, typically in a separate file, with just a link from `Cs`:

```python
sage: class Cs(Category):
    ....:    def super_categories(self):
    ....:        return [Sets()]

sage: class FiniteCs(CategoryWithAxiom):
    ....:    def foo(self):
    ....:        print("I am a method on finite C's")

sage: Cs.Finite = FiniteCs
```

(continues on next page)
For a real example, see the code of the class `FiniteGroups` and the link to it in `Groups`. Note that the link is implemented using `LazyImport`; this is highly recommended: it makes sure that `FiniteGroups` is imported after `Groups` it depends upon, and makes it explicit that the class `Groups` can be imported and is fully functional without importing `FiniteGroups`.

**Note:** Some categories with axioms are created upon Sage's startup. In such a case, one needs to pass the `at_startup=True` option to `LazyImport`, in order to quiet the warning about that lazy import being resolved upon startup. See for example `Sets.Finite`.

This is undoubtedly a code smell. Nevertheless, it is preferable to stick to lazy imports, first to resolve the import order properly, and more importantly as a reminder that the category would be best not constructed upon Sage's startup. This is to spur developers to reduce the number of parents (and therefore categories) that are constructed upon startup. Each `at_startup=True` that will be removed will be a measure of progress in this direction.

**Note:** In principle, due to a limitation of `LazyImport` with nested classes (see Issue #15648), one should pass the option `as_name` to `LazyImport`:

```python
Finite = LazyImport('sage.categories.finite_groups', 'FiniteGroups', as_name='Finite')
```

in order to prevent `Groups.Finite` to keep on reimporting `FiniteGroups`.

Given that passing this option introduces some redundancy and is error prone, the axiom infrastructure includes a little workaround which makes the `as_name` unnecessary in this case.

### Making the category with axiom directly callable

If desired, a category with axiom can be constructed directly through its class rather than through its base category:

```python
sage: Semigroups()
Category of semigroups
sage: Semigroups() is Magmas().Associative()
True
sage: FiniteGroups()
Category of finite groups
```
For this notation to work, the class `Semigroups` needs to be aware of the base category class (here, `Magmas`) and of the axiom (here, `Associative`):

```
sage: Semigroups._base_category_class_and_axiom
(<class 'sage.categories.magmas.Magmas'>, 'Associative')
sage: Fields._base_category_class_and_axiom
(<class 'sage.categories.division_rings.DivisionRings'>, 'Commutative')
sage: FiniteGroups._base_category_class_and_axiom
(<class 'sage.categories.groups.Groups'>, 'Finite')
sage: FiniteDimensionalAlgebrasWithBasis._base_category_class_and_axiom
(<class 'sage.categories.algebras_with_basis.AlgebrasWithBasis'>, 'FiniteDimensional')
```

In our example, the attribute `_base_category_class_and_axiom` was set upon calling `Cs().Finite()`, which makes the notation seemingly work:

```
sage: FiniteCs()
Category of finite cs
sage: FiniteCs._base_category_class_and_axiom
(<class '__main__.Cs'>, 'Finite')
sage: FiniteCs._base_category_class_and_axiom_origin
'set by __classget__'
```

But calling `FiniteCs()` right after defining the class would have failed (try it!). In general, one needs to set the attribute explicitly:
```
sage: class FiniteCs(CategoryWithAxiom):
    ....:    _base_category_class_and_axiom = (Cs, 'Finite')
    ....:    class ParentMethods:
    ....:        def foo(self):
    ....:            print("I am a method on finite C's")
```

```
>>> from sage.all import *
>>> class FiniteCs(CategoryWithAxiom):
...    _base_category_class_and_axiom = (Cs, 'Finite')
...    class ParentMethods:
...        def foo(self):
...            print("I am a method on finite C's")
```

Having to set explicitly this link back from `FiniteCs` to `Cs` introduces redundancy in the code. It would therefore be desirable to have the infrastructure set the link automatically instead (a difficulty is to achieve this while supporting lazy imported categories with axiom).

As a first step, the link is set automatically upon accessing the class from the base category class:

```
sage: Algebras.WithBasis._base_category_class_and_axiom
<class 'sage.categories.algebras.Algebras'>, 'WithBasis')
sage: Algebras.WithBasis._base_category_class_and_axiom_origin
'set by __classget__'
```

```
>>> from sage.all import *

`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
>>> from sage.all import *
>>> FiniteCoxeterGroups()
Category of finite Coxeter groups
>>> FiniteCoxeterGroups() is CoxeterGroups().Finite()
True
>>> FiniteCoxeterGroups()._base_category_class_and_axiom_origin
'deduced by base_category_class_and_axiom'

>>> FiniteDimensionalAlgebrasWithBasis(QQ)
Category of finite dimensional algebras with basis over Rational Field
>>> FiniteDimensionalAlgebrasWithBasis(QQ) is Algebras(QQ).FiniteDimensional().
˓
True
If the heuristic succeeds, the result is guaranteed to be correct. If it fails, typically because the category has a name of its own like Fields, the attribute _base_category_class_and_axiom should be set explicitly. For more examples, see the code of the classes Semigroups or Fields.

Note: When printing out a category with axiom, the heuristic determines whether a category has a name of its own by checking out how _base_category_class_and_axiom was set:

sage: Fields._base_category_class_and_axiom_origin
'hardcoded'

>>> from sage.all import *
>>> Fields._base_category_class_and_axiom_origin
'hardcoded'
See CategoryWithAxiom._without_axioms(), CategoryWithAxiom._repr_object_names_static().

In our running example FiniteCs, Sage failed to deduce automatically the base category class and axiom because the class Cs is not in the standard location sage.categories.cs.

Design discussion

The above deduction, based on names, is undoubtedly inelegant. But it's safe (either the result is guaranteed to be correct, or an error is raised), it saves on some redundant information, and it is only used for the simple shorthands like FiniteGroups() for Groups().Finite(). Finally, most if not all of these shorthands are likely to eventually disappear (see Issue #15741 and the related discussion in the primer).

Defining a new axiom

We describe now how to define a new axiom. The first step is to figure out the largest category where the axiom makes sense. For example Sets for Finite, Magmas for Associative, or Modules for FiniteDimensional. Here we define the axiom Green for the category Cs and its subcategories:

sage: from sage.categories.category_with_axiom import CategoryWithAxiom
c sage: class Cs(Category):
....:    def super_categories(self):
....:        return [Sets()]
(continues on next page)
With the current implementation, the name of the axiom must also be added to a global container:

```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: all_axioms += ("Green",)
```

We can now use the axiom as usual:

```python
sage: Cs().Green()
Category of green cs

sage: P = Parent(category=Cs().Green())
sage: P.foo()
I am a method on green C's
```

Compared with our first example, the only newcomer is the method `.Green()` that can be used by any subcategory `Ds()` of `Cs()` to add the axiom `Green`. Note that the expression `Ds().Green` always evaluates to this method, regardless of whether `Ds` has a nested class `Ds.Green` or not (an implementation detail):

```python
sage: Cs().Green
<bound method Cs.SubcategoryMethods.Green of Category of cs>
```
>>> from sage.all import *
>>> Cs().Green
<bound method Cs.SubcategoryMethods.Green of Category of cs>

Thanks to this feature (implemented in \texttt{CategoryWithAxiom.__classget__()}), the user is systematically referred to the documentation of this method when doing introspection on \texttt{Ds().Green}:

\begin{verbatim}
sage: C = Cs()
sage: C.Green# not tested
sage: Cs().Green.__doc__
'documentation of the axiom Green'
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> C = Cs()
>>> C.Green# not tested
>>> Cs().Green.__doc__
'documentation of the axiom Green'
\end{verbatim}

It is therefore the natural spot for the documentation of the axiom.

\textbf{Note:} The presence of the nested class \texttt{Green} in \texttt{Cs} is currently mandatory even if it is empty.

\textbf{Todo:} Specify whether or not one should systematically use \texttt{@cached_method} in the definition of the axiom. And make sure all the definition of axioms in Sage are consistent in this respect!

\textbf{Todo:} We could possibly define an \texttt{@axiom} decorator? This could hide two little implementation details: whether or not to make the method a cached method, and the call to \texttt{.with_axiom(...) under the hood.} It could do possibly do some more magic. The gain is not obvious though.

\textbf{Note:} \texttt{all_axioms} is only used marginally, for sanity checks and when trying to derive automatically the base category class. The order of the axioms in this tuple also controls the order in which they appear when printing out categories with axioms (see \texttt{CategoryWithAxiom._repr_object_names_static()}).

During a Sage session, new axioms should only be added at the \textit{end} of \texttt{all_axioms}, as above, so as to not break the cache of \texttt{axioms_rank()}. Otherwise, they can be inserted statically anywhere in the tuple. For axioms defined within the Sage library, the name is best inserted by editing directly the definition of \texttt{all_axioms} in \texttt{sage.categories.category_with_axiom}.

\textbf{Design note}

Let us state again that, unlike what the existence of \texttt{all_axioms} might suggest, the definition of an axiom is local to a category and its subcategories. In particular, two independent categories \texttt{Cs()} and \texttt{Ds()} can very well define axioms with the same name and different semantics. As long as the two hierarchies of subcategories don't intersect, this is not a problem. And if they do intersect naturally (that is if one is likely to create a parent belonging to both categories), this probably means that the categories \texttt{Cs} and \texttt{Ds} are about related enough areas of mathematics that one should clear the ambiguity by having either the same semantic or different names.

This caveat is no different from that of name clashes in hierarchy of classes involving multiple inheritance.
Todo: Explore ways to get rid of this global `all_axioms` tuple, and/or have automatic registration there, and/or having a register_axiom(...) method.

Special case: defining an axiom depending on several categories

In some cases, the largest category where the axiom makes sense is the intersection of two categories. This is typically the case for axioms specifying compatibility conditions between two otherwise unrelated operations, like `Distributive` which specifies a compatibility between `*` and `+`. Ideally, we would want the `Distributive` axiom to be defined by:

```sage
sage: Magmas() & AdditiveMagmas()
Join of Category of magmas and Category of additive magmas
```

The current infrastructure does not support this perfectly: indeed, defining an axiom for a category $C$ requires $C$ to have a class of its own; hence a `JoinCategory` as above won’t do; we need to implement a new class like `MagmasAndAdditiveMagmas`; furthermore, we cannot yet model the fact that `MagmasAndAdditiveMagmas()` is the intersection of `Magmas()` and `AdditiveMagmas()` rather than a mere subcategory:

```sage
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: Magmas() & AdditiveMagmas() is MagmasAndAdditiveMagmas()
False
```

Still, there is a workaround to get the natural notations:

```sage
sage: (Magmas() & AdditiveMagmas()).Distributive()
Category of distributive magmas and additive magmas
```

The trick is to define `Distributive` as usual in `MagmasAndAdditiveMagmas`, and to add a method `Magmas.SubcategoryMethods.Distributive()` which checks that `self` is a subcategory of both `Magmas()` and `AdditiveMagmas()`, complains if not, and otherwise takes the intersection of `self` with `MagmasAndAdditiveMagmas()` before calling `Distributive`.

The downsides of this workaround are:
• Creation of an otherwise empty class `MagmasAndAdditiveMagmas`.

• Pollution of the namespace of `Magmas()` (and subcategories like `Groups()`) with a method that is irrelevant (but safely complains if called).

• `C._with_axiom('Distributive')` is not strictly equivalent to `C.Distributive()`, which can be unpleasantly surprising:

```
sage: (Monoids() & CommutativeAdditiveGroups()).Distributive()
Category of rings

sage: (Monoids() & CommutativeAdditiveGroups())._with_axiom('Distributive')
Join of Category of monoids and Category of commutative additive groups
```

```python
>>> from sage.all import *

>>> (Monoids() & CommutativeAdditiveGroups()).Distributive()
Category of rings

>>> (Monoids() & CommutativeAdditiveGroups())._with_axiom('Distributive')
Join of Category of monoids and Category of commutative additive groups
```

**Todo:** Other categories that would be better implemented via an axiom depending on a join category include:

• **Algebras:** defining an associative unital algebra as a ring and a module satisfying the suitable compatibility axiom between inner multiplication and multiplication by scalars (bilinearity). Of course this should be implemented at the level of `MagmaticAlgebras`, if not higher.

• **Bialgebras:** defining a bialgebra as an algebra and coalgebra where the coproduct is a morphism for the product.

• **Bimodules:** defining a bimodule as a left and right module where the two actions commute.

**Todo:**

• Design and implement an idiom for the definition of an axiom by a join category.

• Or support more advanced joins, through some hook or registration process to specify that a given category is the intersection of two (or more) categories.

• Or at least improve the above workaround to avoid the last issue; this possibly could be achieved using a class `Magmas.Distributive` with a bit of `__classcall__` magic.

**Handling multiple axioms, arborescence structure of the code**

**Prelude**

Let us consider the category of magmas, together with two of its axioms, namely `Associative` and `Unital`. An associative magma is a semigroup and a unital semigroup is a monoid. We have also seen that axioms commute:

```
sage: Magmas().Unital()
Category of unital magmas

sage: Magmas().Associative()
Category of semigroups

sage: Magmas().Associative().Unital()
Category of monoids
```

(continues on next page)
At the level of the classes implementing these categories, the following comes as a general naturalization of the previous section:

```python
sage: Magmas().Unital().Associative()
Category of monoids

>>> from sage.all import *

>>> Magmas().Unital()
Category of unital magmas

>>> Magmas().Associative()
Category of semigroups

>>> Magmas().Associative().Unital()
Category of monoids

>>> Magmas().Unital().Associative()
Category of monoids
```

However, the following may look suspicious at first:

```python
sage: Magmas.Unital
<class 'sage.categories.magmas.Magmas.Unital'>

sage: Magmas.Associative
<class 'sage.categories.semigroups.Semigroups'>

sage: Magmas.Associative.Unital
<class 'sage.categories.monoids.Monoids'>

>>> from sage.all import *

>>> Magmas.Unital
<class 'sage.categories.magmas.Magmas.Unital'>

>>> Magmas.Associative
<class 'sage.categories.semigroups.Semigroups'>

>>> Magmas.Associative.Unital
<class 'sage.categories.monoids.Monoids'>
```

The purpose of this section is to explain the design of the code layout and the rationale for this mismatch.
Abstract model

As we have seen in the Primer, the objects of a category $\text{Cs}(\cdot)$ can usually satisfy, or not, many different axioms. Out of all combinations of axioms, only a small number are relevant in practice, in the sense that we actually want to provide features for the objects satisfying these axioms.

Therefore, in the context of the category class $\text{Cs}$, we want to provide the system with a collection $(D_S)_{S \subseteq \mathcal{S}}$ where each $S$ is a subset of the axioms and the corresponding $D_S$ is a class for the subcategory of the objects of $\text{Cs}(\cdot)$ satisfying the axioms in $S$. For example, if $\text{Cs}(\cdot)$ is the category of magmas, the pairs $(S, D_S)$ would include:

| $\{\text{Associative}\}$ | : Semigroups |
| $\{\text{Associative, Unital}\}$ | : Monoids |
| $\{\text{Associative, Unital, Inverse}\}$ | : Groups |
| $\{\text{Associative, Commutative}\}$ | : Commutative Semigroups |
| $\{\text{Unital, Inverse}\}$ | : Loops |

Then, given a subset $T$ of axioms, we want the system to be able to select automatically the relevant classes $(D_S)_{S \subseteq \mathcal{S}, S \cap T}$, and build from them a category for the objects of $\text{Cs}$ satisfying the axioms in $T$, together with its hierarchy of super categories. If $T$ is in the indexing set $\mathcal{S}$, then the class of the resulting category is directly $D_T$:

```
sage: C = Magmas().Unital().Inverse().Associative(); C
Category of groups
sage: type(C)
<class 'sage.categories.groups.Groups_with_category'>
```

Otherwise, we get a join category:

```
sage: C = Magmas().Infinite().Unital().Associative(); C
Category of infinite monoids
sage: type(C)
<class 'sage.categories.category.JoinCategory_with_category'>
sage: C.super_categories()
[Category of monoids, Category of infinite sets]
```

```
sage: from sage.all import *
>>> C = Magmas().Infinite().Unital().Associative(); C
Category of infinite monoids
>>> type(C)
<class 'sage.categories.category.JoinCategory_with_category'>
>>> C.super_categories()
[Category of monoids, Category of infinite sets]
```

2.3. Axioms
Concrete model as an arborescence of nested classes

We further want the construction to be efficient and amenable to laziness. This led us to the following design decision: the collection \( (D_S)_{S \in \mathcal{S}} \) of classes should be structured as an arborescence (or equivalently a rooted forest). The root is \( C_\emptyset \), corresponding to \( S = \emptyset \). Any other class \( D_S \) should be the child of a single class \( D_{S'} \) where \( S' \) is obtained from \( S \) by removing a single axiom \( A \). Of course, \( D_{\emptyset} \) and \( A \) are respectively the base category class and axiom of the category with axiom \( D_{\emptyset} \) that we have met in the first section.

At this point, we urge the reader to explore the code of Magmas and DistributiveMagmasAndAdditiveMags and see how the arborescence structure on the categories with axioms is reflected by the nesting of category classes.

Discussion of the design

Performance

Thanks to the arborescence structure on subsets of axioms, constructing the hierarchy of categories and computing intersections can be made efficient with, roughly speaking, a linear/quadratic complexity in the size of the involved category hierarchy multiplied by the number of axioms (see Section Algorithms). This is to be put in perspective with the manipulation of arbitrary collections of subsets (aka boolean functions) which can easily raise NP-hard problems.

Furthermore, thanks to its locality, the algorithms can be made suitably lazy: in particular, only the involved category classes need to be imported.

Flexibility

This design also brings in quite some flexibility, with the possibility to support features such as defining new axioms depending on other axioms and deduction rules. See below.

Asymmetry

As we have seen at the beginning of this section, this design introduces an asymmetry. It’s not so bad in practice, since in most practical cases, we want to work incrementally. It’s for example more natural to describe FiniteFields as Fields with the axiom Finite rather than Magmas and AdditiveMags with all (or at least sufficiently many) of the following axioms:

```
sage: sorted(Fields().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Commutative', 'Distributive',
 'Division', 'NoZeroDivisors', 'Unital']
```

```
>>> from sage.all import *
>>> sorted(Fields().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Commutative', 'Distributive',
 'Division', 'NoZeroDivisors', 'Unital']
```

The main limitation is that the infrastructure currently imposes to be incremental by steps of a single axiom.

In practice, among the roughly 60 categories with axioms that are currently implemented in Sage, most admitted a (rather) natural choice of a base category and single axiom to add. For example, one usually thinks more naturally of a monoid as a semigroup which is unital rather than as a unital magma which is associative. Modeling this asymmetry in the code actually brings a bonus: it is used for printing out categories in a (heuristically) mathematician-friendly way:
Only in a few cases is a choice made that feels mathematically arbitrary. This is essentially in the chain of nested classes `distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas.AdditiveAssociative.AdditiveCommutative.AdditiveUnital.Associative`.

**Placeholder classes**

Given that we can only add a single axiom at a time when implementing a `CategoryWithAxiom`, we need to create a few category classes that are just placeholders. For the worst example, see the chain of nested classes `distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas.AdditiveAssociative.AdditiveCommutative.AdditiveUnital.Associative`.

This is suboptimal, but fits within the scope of the axiom infrastructure which is to reduce a potentially exponential number of placeholder category classes to just a couple.

Note also that, in the above example, it's likely that some of the intermediate classes will grow to non-placeholder ones, as people will explore more weaker variants of rings.

**Mismatch between the arborescence of nested classes and the hierarchy of categories**

The fact that the hierarchy relation between categories is not reflected directly as a relation between the classes may sound suspicious at first! However, as mentioned in the primer, this is actually a big selling point of the axioms infrastructure: by calculating automatically the hierarchy relation between categories with axioms one avoids the nightmare of maintaining it by hand. Instead, only a rather minimal number of links needs to be maintained in the code (one per category with axiom).

Besides, with the flexibility introduced by runtime deduction rules (see below), the hierarchy of categories may depend on the parameters of the categories and not just their class. So it’s fine to make it clear from the onset that the two relations do not match.

**Evolutivity**

At this point, the arborescence structure has to be hardcoded by hand with the annoyances we have seen. This does not preclude, in a future iteration, to design and implement some idiom for categories with axioms that adds several axioms at once to a base category; maybe some variation around:

```python
class DistributiveMagmasAndAdditiveMagmas:
    ...

    @category_with_axiom(
        AdditiveAssociative,
        AdditiveCommutative,
        AdditiveUnital,
        AdditiveInverse,
        Associative)
    def __(): return LazyImport('sage.categories.rngs', 'Rngs', at_startup=True)
```
or:

```python
register_axiom_category(DistributiveMagmasAndAdditiveMagmas,
    {AdditiveAssociative,
     AdditiveCommutative,
     AdditiveUnital,
     AdditiveInverse,
     Associative},
    'sage.categories.rngs', 'Rngs', at_startup=True)
```

The infrastructure would then be in charge of building the appropriate arborescence under the hood. Or rely on some database (see discussion on Issue #10963, in particular at the end of comment 332).

**Axioms defined upon other axioms**

Sometimes an axiom can only be defined when some other axiom holds. For example, the axiom `NoZeroDivisors` only makes sense if there is a zero, that is if the axiom `AdditiveUnital` holds. Hence, for the category `MagmasAndAdditiveMagmas`, we consider in the abstract model only those subsets of axioms where the presence of `NoZeroDivisors` implies that of `AdditiveUnital`. We also want the axiom to be only available if meaningful:

```python
sage: Rings().NoZeroDivisors()
Category of domains
sage: Rings().Commutative().NoZeroDivisors()
Category of integral domains
sage: Semirings().NoZeroDivisors()
Traceback (most recent call last):
...
AttributeError: 'Semirings_with_category' object has no attribute 'NoZeroDivisors'...
```

Concretely, this is to be implemented by defining the new axiom in the (SubcategoryMethods nested class of the) appropriate category with axiom. For example the axiom `NoZeroDivisors` would be naturally defined in `magmas_and_additive_magmas.MagmasAndAdditiveMagmas.Distributive.AdditiveUnital`.

**Note:** The axiom `NoZeroDivisors` is currently defined in `Rings`, by simple lack of need for the feature; it should be lifted up as soon as relevant, that is when some code will be available for parents with no zero divisors that are not necessarily rings.
**Deduction rules**

A similar situation is when an axiom $A$ of a category $Cs$ implies some other axiom $B$, with the same consequence as above on the subsets of axioms appearing in the abstract model. For example, a division ring necessarily has no zero divisors:

```python
sage: 'NoZeroDivisors' in Rings().Division().axioms()
True
sage: 'NoZeroDivisors' in Rings().axioms()
False
```

This deduction rule is implemented by the method `Rings.Division.extra_super_categories()`:

```python
sage: Rings().Division().extra_super_categories()
(Category of domains,)
```

In general, this is to be implemented by a method $Cs.A.extra_super_categories$ returning a tuple $(Cs().B(),)$, or preferably $(Ds().B(),)$ where $Ds$ is the category defining the axiom $B$.

This follows the same idiom as for deduction rules about functorial constructions (see `covariant_functorial_construction.CovariantConstructionCategory.extra_super_categories()`). For example, the fact that a Cartesian product of associative magmas (i.e. of semigroups) is an associative magma is implemented in `Semigroups.CartesianProducts.extra_super_categories()`:

```python
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Associative().CartesianProducts().extra_super_categories()

>>> from sage.all import *

>>> Magmas().Associative()
Category of semigroups
>>> Magmas().Associative().CartesianProducts().extra_super_categories()
[Category of semigroups]
```

Similarly, the fact that the algebra of a commutative magma is commutative is implemented in `Magmas.Combinative.Algebras.extra_super_categories()`:

```python
sage: Magmas().Commutative().Algebras(QQ).extra_super_categories()

[Category of commutative magmas]
```

2.3. Axioms
Warning: In some situations this idiom is inapplicable as it would require to implement two classes for the same category. This is the purpose of the next section.

**Special case**

In the previous examples, the deduction rule only had an influence on the super categories of the category with axiom being constructed. For example, when constructing `Rings().Division()`, the rule `Rings.Division.extra_super_categories()` simply adds `Rings().NoZeroDivisors()` as a super category thereof.

In some situations this idiom is inapplicable because a class for the category with axiom under construction already exists elsewhere. Take for example Wedderburn’s theorem: any finite division ring is commutative, i.e. is a finite field. In other words, `DivisionRings().Finite()` coincides with `Fields().Finite()`:

```python
sage: DivisionRings().Finite()
Category of finite enumerated fields
sage: DivisionRings().Finite() is Fields().Finite()
True
```

Therefore we cannot create a class `DivisionRings.Finite` to hold the desired `extra_super_categories` method, because there is already a class for this category with axiom, namely `Fields.Finite`.

A natural idiom would be to have `DivisionRings.Finite` be a link to `Fields.Finite` (locally introducing an undirected cycle in the arborescence of nested classes). It would be a bit tricky to implement though, since one would need to detect, upon constructing `DivisionRings().Finite()`, that `DivisionRings.Finite` is actually `Fields.Finite`, in order to construct appropriately `Fields().Finite()`; and reciprocally, upon computing the super categories of `Fields().Finite()`, to not try to add `DivisionRings().Finite()` as a super category.

Instead the current idiom is to have a method `DivisionRings.Finite_extra_super_categories` which mimics the behavior of the would-be `DivisionRings.Finite.extra_super_categories`:

```python
sage: DivisionRings().Finite_extra_super_categories()
(Category of commutative magmas,)
```

This idiom is admittedly rudimentary, but consistent with how mathematical facts specifying non trivial inclusion relations between categories are implemented elsewhere in the various `extra_super_categories` methods of axiom categories and covariant functorial constructions. Besides, it gives a natural spot (the docstring of the method) to document and test the modeling of the mathematical fact. Finally, Wedderburn’s theorem is arguably a theorem about division rings (in the context of division rings, finiteness implies commutativity) and therefore lives naturally in `DivisionRings.Finite`.

An alternative would be to implement the category of finite division rings (i.e. finite fields) in a class `DivisionRings.Finite` rather than `Fields.Finite`:
 sage: class MyDivisionRings(Category):
....:    def super_categories(self):
....:        return [Rings()]

 sage: class MyFields(Category):
....:    def super_categories(self):
....:        return [MyDivisionRings()]

 sage: class MyFiniteFields(CategoryWithAxiom):
....:    _base_category_class_and_axiom = (MyDivisionRings, "Finite")
....:    def extra_super_categories(self): # Wedderburn’s theorem
....:        return [MyFields()]

 sage: MyDivisionRings.Finite = MyFiniteFields

 sage: MyDivisionRings().Finite()
Category of my finite fields
 sage: MyFields().Finite() is MyDivisionRings().Finite()
True

>>> from sage.all import *
>>> from sage.categories.category_with_axiom import CategoryWithAxiom

>>> class MyDivisionRings(Category):
...    def super_categories(self):
...        return [Rings()]

>>> class MyFields(Category):
...    def super_categories(self):
...        return [MyDivisionRings()]

>>> class MyFiniteFields(CategoryWithAxiom):
...    _base_category_class_and_axiom = (MyDivisionRings, "Finite")
...    def extra_super_categories(self): # Wedderburn’s theorem
...        return [MyFields()]

>>> MyDivisionRings.Finite = MyFiniteFields

>>> MyDivisionRings().Finite()
Category of my finite fields
>>> MyFields().Finite() is MyDivisionRings().Finite()
True

In general, if several categories $C_1s()$, $C_2s()$, … are mapped to the same category when applying some axiom $A$ (that is $C_1s().A() == C_2s().A() == ...$), then one should be careful to implement this category in a single class $Cs.A$, and set up methods extra_super_categories or $A_extra_super_categories$ methods as appropriate. Each such method should return something like $[C_2s()]$ and not $[C_2s().A()]$ for the latter would likely lead to an infinite recursion.

**Design discussion**

Supporting similar deduction rules will be an important feature in the future, with quite a few occurrences already implemented in upcoming issues. For the time being though there is a single occurrence of this idiom outside of the tests. So this would be an easy thing to refactor after Issue #10963 if a better idiom is found.
Larger synthetic examples

We now consider some larger synthetic examples to check that the machinery works as expected. Let us start with a category defining a bunch of axioms, using \texttt{axiom()} for conciseness (don’t do it for real axioms; they deserve a full documentation):

```python
sage: from sage.categories.category_singleton import Category_singleton
sage: from sage.categories.category_with_axiom import axiom
sage: import sage.categories.category_with_axiom
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: all_axioms += ('B', 'C', 'D', 'E', 'F')

sage: class As(Category_singleton):
    ....: def super_categories(self):
    ....:     return [Objects()]
    ....:
    ....:     class SubcategoryMethods:
    ....:         B = axiom('B')
    ....:         C = axiom('C')
    ....:         D = axiom('D')
    ....:         E = axiom('E')
    ....:         F = axiom('F')
    ....:
    ....:         class B(CategoryWithAxiom):
    ....:             pass
    ....:         class C(CategoryWithAxiom):
    ....:             pass
    ....:         class D(CategoryWithAxiom):
    ....:             pass
    ....:         class E(CategoryWithAxiom):
    ....:             pass
    ....:         class F(CategoryWithAxiom):
    ....:             pass

>>> from sage.all import *
>>> from sage.categories.category_singleton import Category_singleton
>>> from sage.categories.category_with_axiom import axiom
>>> import sage.categories.category_with_axiom
>>> all_axioms = sage.categories.category_with_axiom.all_axioms
>>> all_axioms += ('B', 'C', 'D', 'E', 'F')

>>> class As(Category_singleton):
...     def super_categories(self):
...         return [Objects()]
...     ...
...     class SubcategoryMethods:
...         B = axiom('B')
...         C = axiom('C')
...         D = axiom('D')
...         E = axiom('E')
...         F = axiom('F')
...     ...
...     class B(CategoryWithAxiom):
...         pass
...     ...
...     class C(CategoryWithAxiom):
...         pass
...     ...
...     class D(CategoryWithAxiom):
...         pass
...     ...
...     class E(CategoryWithAxiom):
...         pass
...     ...
...     class F(CategoryWithAxiom):
...         pass
```

(continues on next page)
Now we construct a subcategory where, by some theorem of William, axioms $B$ and $C$ together are equivalent to $E$ and $F$ together:

```python
sage: class A1s(Category_singleton):
    def super_categories(self):
        return [As()]

sage: class B(CategoryWithAxiom):
    def C_extra_super_categories(self):
        return [As().E(), As().F()]

sage: class E(CategoryWithAxiom):
    def F_extra_super_categories(self):
        return [As().B(), As().C()]

sage: A1s().B().C()
Category of e f a1s
```

The axioms $B$ and $C$ do not show up in the name of the obtained category because, for concision, the printing uses some heuristics to not show axioms that are implied by others. But they are satisfied:

```python
>>> from sage.all import *
>>> A1s().B().C()
Category of e f a1s
```

```python
sage: sorted(A1s().B().C().axioms())
['B', 'C', 'E', 'F']
```

Note also that this is a join category:

```python
sage: type(A1s().B().C())
<class 'sage.categories.category.JoinCategory_with_category'>
sage: A1s().B().C().super_categories()
[Category of e a1s,
```

2.3. Axioms

121
As desired, William's theorem holds:

```
sage: A1s().B().C() is A1s().E().F()
True
```

and propagates appropriately to subcategories:

```
sage: C = A1s().E().F().D().B().C()
sage: C is A1s().B().C().E().F().D(). # commutativity
True
sage: C is A1s().E().F().E().F().D(). # William's theorem
True
sage: C is A1s().E().E().F().F().D(). # commutativity
True
sage: C is A1s().E().F().D(). # idempotency
True
```

In this quick variant, we actually implement the category of b c a2s, and choose to do so in A2s.B.C:

```
sage: class A2s(Category_singleton):
    ....:     def super_categories(self):
    ....:         return [As()]
```

(continues on next page)
As desired, William’s theorem and its consequences hold:

```python
sage: A2s().B().C() is A2s().E().F()
True
sage: C = A2s().E().F().D().B().C()
sage: C is A2s().B().C().E().F().D()  # commutativity
True
sage: C is A2s().E().F().E().F().D()  # William’s theorem
True
sage: C is A2s().E().E().F().F().D()  # commutativity
True
sage: C is A2s().E().F().D()          # idempotency
True
sage: C is A2s().D().E().F()
```

2.3. Axioms
Finally, we “accidentally” implement the category of \( \mathcal{B} \mathcal{C} \mathcal{A}_1 \mathcal{S} \), both in \( \mathcal{A}_3 \mathcal{B} \mathcal{C} \) and \( \mathcal{A}_3 \mathcal{E} \mathcal{F} \):

```python
sage: class A3s(Category_singleton):
    ....:     def super_categories(self):
    ....:         return [As()]
    ....:
    ....:     class B(CategoryWithAxiom):
    ....:         class C(CategoryWithAxiom):
    ....:             def extra_super_categories(self):
    ....:                 return [As().E(), As().F()]
    ....:
    ....:     class E(CategoryWithAxiom):
    ....:         class F(CategoryWithAxiom):
    ....:             def extra_super_categories(self):
    ....:                 return [As().B(), As().C()]
```

We can still construct, say:

```python
sage: A3s().B()
Category of b a3s
sage: A3s().C()
Category of c a3s
```
However,

```
>>> from sage.all import *
```

```
>>> A3s().B().C()  # not tested
```

runs into an infinite recursion loop, as `A3s().B().C()` wants to have `A3s().E().F()` as super category and reciprocally.

**Todo:** The above example violates the specifications (a category should be modelled by at most one class), so it's appropriate that it fails. Yet, the error message could be usefully complemented by some hint at what the source of the problem is (a category implemented in two distinct classes). Leaving a large enough piece of the backtrace would be useful though, so that one can explore where the issue comes from (e.g. with post mortem debugging).

### 2.3.2 Specifications

After fixing some vocabulary, we summarize here some specifications about categories and axioms.

**The lattice of constructible categories**

A mathematical category $C$ is implemented if there is a class in Sage modelling it; it is constructible if it is either implemented, or is the intersection of implemented categories; in the latter case it is modelled by a `JoinCategory`. The comparison of two constructible categories with the `Category.is_subcategory()` method is supposed to model the comparison of the corresponding mathematical categories for inclusion of the objects (see *On the category hierarchy: subcategories and super categories* for details). For example:

```
sage: Fields().is_subcategory(Rings())
True
```

```
>>> from sage.all import *
```

```
>>> Fields().is_subcategory(Rings())
True
```

However this modelling may be incomplete. It can happen that a mathematical fact implying that a category $A$ is a subcategory of a category $B$ is not implemented. Still, the comparison should endow the set of constructible categories with a poset structure and in fact a lattice structure.

In this lattice, the join of two categories (`Category.join()`) is supposed to model their intersection. Given that we compare categories for inclusion, it would be more natural to call this operation the *meet*; blames go to me (Nicolas) for originally comparing categories by *amount of structure* rather than by *inclusion*. In practice, the join of two categories may be a strict super category of their intersection; first because this intersection might not be constructible; second because Sage might miss some mathematical information to recover the smallest constructible super category of the intersection.

### 2.3. Axioms
Axioms

We say that an axiom \( A \) is defined by a category \( Cs() \) if \( Cs \) defines an appropriate method \( Cs().SubcategoryMethods.A() \), with the semantic of the axiom specified in the documentation; for any subcategory \( Ds() \), \( Ds().A() \) models the subcategory of the objects of \( Ds() \) satisfying \( A \). In this case, we say that the axiom \( A \) is defined for the category \( Ds() \). Furthermore, \( Ds() \) implements the axiom \( A \) if \( Ds \) has a category with axiom as nested class \( Ds().A() \). The category \( Ds() \) satisfies the axiom if \( Ds() \) is a subcategory of \( Cs().A() \) (meaning that all the objects of \( Ds() \) are known to satisfy the axiom \( A \)).

A digression on the structure of fibers when adding an axiom

Consider the application \( \phi_A \) which maps a category to its category of objects satisfying \( A \). Equivalently, \( \phi_A \) is computing the intersection with the defining category with axiom of \( A \). It follows immediately from the latter that \( \phi_A \) is a regressive endomorphism of the lattice of categories. It restricts to a regressive endomorphism \( Cs() \mapsto Cs().A() \) on the lattice of constructible categories.

This endomorphism may have non-trivial fibers, as in our favorite example: \( \text{DivisionRings()} \) and \( \text{Fields()} \) are in the same fiber for the axiom \( \text{Finite} \):

```python
sage: DivisionRings().Finite() is Fields().Finite()
True
```

Consider the intersection \( S \) of such a fiber of \( \phi_A \) with the upper set \( I_A \) of categories that do not satisfy \( A \). The fiber itself is a sublattice. However \( I_A \) is not guaranteed to be stable under intersection (though exceptions should be rare). Therefore, there is a priori no guarantee that \( S \) would be stable under intersection. Also it’s presumably finite, in fact small, but this is not guaranteed either.

Specifications

- Any constructible category \( C \) should admit a finite number of larger constructible categories.

- The methods `super_categories`, `extra_super_categories`, and friends should always return strict supercategories.

  For example, to specify that a finite division ring is a finite field, \( \text{DivisionRings().Finite()}.\text{super_categories()} \), without returning \( \text{Fields().Finite()} \) if it could possibly return \( \text{Fields()} \); but it’s preferable to return the largest category that contains the relevant information, in this case \( \text{Magmas().Commutative()} \), and to let the infrastructure apply the derivations.

- The base category of a `CategoryWithAxiom` should be an implemented category (i.e. not a `JoinCategory`). This is checked by `CategoryWithAxiom._test_category_with_axiom()`.

- Arborescent structure: Let \( Cs() \) be a category, and \( S \) be some set of axioms defined in some super categories of \( Cs() \) but not satisfied by \( Cs() \). Suppose we want to provide a category with axiom for the elements of \( Cs() \) satisfying the axioms in \( S \). Then, there should be a single enumeration \( A_1, A_2, \ldots, A_k \) without repetition of axioms in \( S \) such that \( Cs().A_1.A_2.\ldots.A_k \) is an implemented category. Furthermore, every intermediate step \( Cs().A_1.A_2.\ldots.A_i \) with \( i \leq k \) should be a category with axiom having \( A_i \) as axiom and \( Cs().A_1.A_2.\ldots.A_i \) as base category class; this base category class should not satisfy \( A_i \). In particular, when some axioms of \( S \) can be deduced from previous ones by deduction rules, they should not appear in the enumeration \( A_1, A_2, \ldots, A_k \).
• In particular, if \( Cs() \) is a category that satisfies some axiom \( A \) (e.g. from one of its super categories), then it should not implement that axiom. For example, a category class \( Cs \) can never have a nested class \( Cs.A.A \). Similarly, applying the specification recursively, a category satisfying \( A \) cannot have a nested class \( Cs.A1.A2.A3.A \) where \( A1, A2, A3 \) are axioms.

• A category can only implement an axiom if this axiom is defined by some super category. The code has not been systematically checked to support having two super categories defining the same axiom (which should of course have the same semantic). You are welcome to try, at your own risk. :-)

• When a category defines an axiom or functorial construction \( A \), this fixes the semantic of \( A \) for all the subcategories. In particular, if two categories define \( A \), then these categories should be independent, and either the semantic of \( A \) should be the same, or there should be no natural intersection between the two hierarchies of subcategories.

• Any super category of a \( \text{CategoryWithParameters} \) should either be a \( \text{CategoryWithParameters} \) or a \( \text{Category_singleton} \).

• A \( \text{CategoryWithAxiom} \) having a \( \text{Category_singleton} \) as base category should be a \( \text{CategoryWithAxiom_singleton} \). This is handled automatically by \( \text{CategoryWithAxiom.__init__()} \) and checked in \( \text{CategoryWithAxiom._test_category_with_axiom()} \).

• A \( \text{CategoryWithAxiom} \) having a \( \text{Category_over_base_ring} \) as base category should be a \( \text{Category_over_base_ring} \). This currently has to be handled by hand, using \( \text{CategoryWithAxiom_over_base_ring} \). This is checked in \( \text{CategoryWithAxiom._test_category_with_axiom()} \).

**Todo:** The following specifications would be desirable but are not yet implemented:

• A functorial construction category (Graded, CartesianProducts, ...) having a \( \text{Category_singleton} \) as base category should be a \( \text{CategoryWithAxiom_singleton} \).

Nothing difficult to implement, but this will need to rework the current “no subclass of a concrete class” assertion test of \( \text{Category_singleton.__classcall__()} \).

• Similarly, a covariant functorial construction category having a \( \text{Category_over_base_ring} \) as base category should be a \( \text{Category_over_base_ring} \).

The following specification might be desirable, or not:

• A join category involving a \( \text{Category_over_base_ring} \) should be a \( \text{Category_over_base_ring} \). In the mean time, a \( \text{base_ring} \) method is automatically provided for most of those by \( \text{Modules.SubcategoryMethods.base_ring()} \).

---

### 2.3.3 Design goals

As pointed out in the primer, the main design goal of the axioms infrastructure is to subdue the potential combinatorial explosion of the category hierarchy by letting the developer focus on implementing a few bookshelves for which there is actual code or mathematical information, and let Sage compose dynamically and lazily these building blocks to construct the minimal hierarchy of classes needed for the computation at hand. This allows for the infrastructure to scale smoothly as bookshelves are added, extended, or reorganized.

Other design goals include:

• Flexibility in the code layout: the category of, say, finite sets can be implemented either within the Sets category (in a nested class \( \text{Sets.Finite} \)), or in a separate file (typically in a class \( \text{FiniteSets} \) in a lazily imported module \( \text{sage.categories.finite_sets} \)).

• Single point of truth: a theorem, like Wedderburn’s, should be implemented in a single spot.

---

### 2.3. Axioms

127
• Single entry point: for example, from the entry `Rings`, one can explore a whole range of related categories just by applying axioms and constructions:

```python
sage: Rings().Commutative().Finite().NoZeroDivisors()
Category of finite integral domains
sage: Rings().Finite().Division()
Category of finite enumerated fields
```

```python
>>> from sage.all import *
>>> Rings().Commutative().Finite().NoZeroDivisors()
Category of finite integral domains
>>> Rings().Finite().Division()
Category of finite enumerated fields
```

This will allow for progressively getting rid of all the entries like `GradedHopfAlgebrasWithBasis` which are polluting the global namespace.

Note that this is not about precluding the existence of multiple natural ways to construct the same category:

```python
sage: Groups().Finite()
Category of finite groups
sage: Monoids().Finite().Inverse()
Category of finite groups
sage: Sets().Finite() & Monoids().Inverse()
Category of finite groups
```

```python
>>> from sage.all import *
>>> Groups().Finite()
Category of finite groups
>>> Monoids().Finite().Inverse()
Category of finite groups
>>> Sets().Finite() & Monoids().Inverse()
Category of finite groups
```

• Concise idioms for the users (adding axioms, …)
• Concise idioms and well highlighted hierarchy of bookshelves for the developer (especially with code folding)
• Introspection friendly (listing the axioms, recovering the mixins)

**Note:** The constructor for instances of this class takes as input the base category. Hence, they should in principle be constructed as:

```python
sage: FiniteSets(Sets())
Category of finite sets
```

```python
sage: Sets.Finite(Sets())
Category of finite sets
```

```python
>>> from sage.all import *
>>> FiniteSets(Sets())
Category of finite sets
>>> Sets.Finite(Sets())
Category of finite sets
```

None of these idioms are really practical for the user. So instead, this object is to be constructed using any of the following idioms:
The later two are implemented using respectively `CategoryWithAxiom.__classcall__()` and `CategoryWithAxiom.__classget__()`. 

### 2.3.4 Upcoming features

#### Todo:

- Implement compatibility axiom / functorial constructions. For example, one would want to have:

\[
A \text{.CartesianProducts()} \ & \ B \text{.CartesianProducts()} = (A \& B) \text{.CartesianProducts()}
\]

- Once full subcategories are implemented (see Issue #10668), make the relevant categories with axioms be such. This can be done systematically for, e.g., the axioms Associative or Commutative, but not for the axiom Unital: a semigroup morphism between two monoids need not preserve the unit.

Should all full subcategories be implemented in term of axioms?

### 2.3.5 Algorithms

#### Computing joins

The workhorse of the axiom infrastructure is the algorithm for computing the join $J$ of a set $C_1, \ldots, C_k$ of categories (see `Category.join()`). Formally, $J$ is defined as the largest constructible category such that $J \subseteq C_i$ for all $i$, and $J \subseteq C.A()$ for every constructible category $C \supset J$ and any axiom $A$ satisfied by $J$.

The join $J$ is naturally computed as a closure in the lattice of constructible categories: it starts with the $C_i$’s, gathers the set $S$ of all the axioms satisfied by them, and repeatedly adds each axiom $A$ to those categories that do not yet satisfy $A$ using `Category._with_axiom()`. Due to deduction rules or (extra) super categories, new categories or new axioms may appear in the process. The process stops when each remaining category has been combined with each axiom. In practice, only the smallest categories are kept along the way; this is correct because adding an axiom is covariant: $C.A()$ is a subcategory of $D.A()$ whenever $C$ is a subcategory of $D$.

As usual in such closure computations, the result does not depend on the order of execution. Furthermore, given that adding an axiom is an idempotent and regressive operation, the process is guaranteed to stop in a number of steps which is bounded by the number of super categories of $J$. In particular, it is a finite process.
Todo: Detail this a bit. What could typically go wrong is a situation where, for some category $C_1$, $C_1.A()$ specifies a category $C_2$ as super category such that $C_2.A()$ specifies $C_3$ as super category such that …; this would clearly cause an infinite execution. Note that this situation violates the specifications since $C_1.A()$ is supposed to be a subcategory of $C_2.A()$, … so we would have an infinite increasing chain of constructible categories.

It's reasonable to assume that there is a finite number of axioms defined in the code. There remains to use this assumption to argue that any infinite execution of the algorithm would give rise to such an infinite sequence.

Adding an axiom

Let $C_s$ be a category and $A$ an axiom defined for this category. To compute $C_s().A()$, there are two cases.

Adding an axiom $A$ to a category $C_s()$ not implementing it

In this case, $C_s().A()$ returns the join of:

- $C_s()$
- $B_s().A()$ for every direct super category $B_s()$ of $C_s()$
- the categories appearing in $C_s().A_{extra\_super\_categories}()$

This is a highly recursive process. In fact, as such, it would run right away into an infinite loop! Indeed, the join of $C_s()$ with $B_s().A()$ would trigger the construction of $C_s().A()$ and reciprocally. To avoid this, the `Category.join()` method itself does not use `Category._with_axiom()` to add axioms, but its sister `Category._with_axiom_as_tuple()`; the latter builds a tuple of categories that should be joined together but leaves the computation of the join to its caller, the master join calculation.

Adding an axiom $A$ to a category $C_s()$ implementing it

In this case $C_s().A()$ simply constructs an instance $D$ of $C_s.A$ which models the desired category. The non trivial part is the construction of the super categories of $D$. Very much like above, this includes:

- $C_s()$
- $B_s().A()$ for every super category $B_s()$ of $C_s()$
- the categories appearing in $D_{extra\_super\_categories}()$

This by itself may not be sufficient, due in particular to deduction rules. On may for example discover a new axiom $A_1$ satisfied by $D$, imposing to add $A_1$ to all of the above categories. Therefore the super categories are computed as the join of the above categories. Up to one twist: as is, the computation of this join would trigger recursively a recalculation of $C_s().A()$. To avoid this, `Category.join()` is given an optional argument to specify that the axiom $A$ should not be applied to $C_s()$. 

Sketch of proof of correctness and evaluation of complexity

As we have seen, this is a highly recursive process! In particular, one needs to argue that, as long as the specifications are satisfied, the algorithm won’t run in an infinite recursion, in particular in case of deduction rule.

**Theorem**

Consider the construction of a category $C$ by adding an axiom to a category (or computing of a join). Let $H$ be the hierarchy of implemented categories above $C$. Let $n$ and $m$ be respectively the number of categories and the number of inheritance edges in $H$.

Assuming that the specifications are satisfied, the construction of $C$ involves constructing the categories in $H$ exactly once (and no other category), and at most $n$ join calculations. In particular, the time complexity should be, roughly speaking, bounded by $n^2$. In particular, it's finite.

**Remark**

It's actually to be expected that the complexity is more of the order of magnitude of $na + m$, where $a$ is the number of axioms satisfied by $C$. But this is to be checked in detail, in particular due to the many category inclusion tests involved.

The key argument is that `Category.join` cannot call itself recursively without going through the construction of some implemented category. In turn, the construction of some implemented category $C$ only involves constructing strictly smaller categories, and possibly a direct join calculation whose result is strictly smaller than $C$. This statement is obvious if $C$ implements the `super_categories` method directly, and easy to check for functorial construction categories. It requires a proof for categories with axioms since there is a recursive join involved.

**Lemma**

Let $C$ be a category implementing an axiom $A$. Recall that the construction of $C.A()$ involves a single direct join calculation for computing the super categories. No other direct join calculation occur, and the calculation involves only implemented categories that are strictly smaller than $C.A()$.

**Proof**

Let $D$ be a category involved in the join calculation for the super categories of $C.A()$, and assume by induction that $D$ is strictly smaller than $C.A()$. A category $E$ newly constructed from $D$ can come from:

- $D.(\text{extra}_\_\text{super_categories}())$
  
  In this case, the specifications impose that $E$ should be strictly smaller than $D$ and therefore strictly smaller than $C$.

- $D.\text{with}_{\text{axiom}}.\text{as}_\text{tuple}(\text{"B"})$ or $D.\text{B}_\text{extra}_\text{super_categories}()$ for some axiom $B$

  In this case, the axiom $B$ is satisfied by some subcategory of $C.A()$, and therefore must be satisfied by $C.A()$ itself. Since adding an axiom is a regressive construction, $E$ must be a subcategory of $C.A()$. If there is equality, then $E$ and $C.A()$ must have the same class, and therefore, $E$ must be directly constructed as $C.A()$. However the join construction explicitly prevents this call.

2.3. Axioms
Note that a call to \( D \text{.with_axiom_as_tuple('B')} \) does not trigger a direct join calculation; but of course, if \( D \) implements \( B \), the construction of the implemented category \( E = D.B() \) will involve a strictly smaller join calculation.

### 2.3.6 Conclusion

This is the end of the axioms documentation. Congratulations on having read that far!

### 2.3.7 Tests

**Note:** Quite a few categories with axioms are constructed early on during Sage's startup. Therefore, when playing around with the implementation of the axiom infrastructure, it is easy to break Sage. The following sequence of tests is designed to test the infrastructure from the ground up even in a partially broken Sage. Please don't remove the imports!

```python
class sage.categories.category_with_axiom.Bars:
    Bases: Category_singleton

    A toy singleton category, for testing purposes.

    See also:

    Blahs

    Unital_extra_super_categories()

    Return extraneous super categories for the unital objects of self.

    This method specifies that a unital bar is a test object. Thus, the categories of unital bars and of unital test objects coincide.

    EXAMPLES:

    ```sage```
    from sage.categories.category_with_axiom import Bars, TestObjects
    sage: Bars().Unital_extra_super_categories()
    [Category of test objects]
    sage: Bars().Unital()
    Category of unital test objects
    sage: TestObjects().Unital().all_super_categories()
    [Category of unital test objects,
     Category of unital blahs,
     Category of test objects,
     Category of bars,
     Category of blahs,
     Category of sets,
     Category of sets with partial maps,
     Category of objects]
    >>> from sage.all import *
    >>> from sage.categories.category_with_axiom import Bars, TestObjects
    >>> Bars().Unital_extra_super_categories()
    [Category of test objects]
    >>> Bars().Unital()
    Category of unital test objects
    >>> TestObjects().Unital().all_super_categories()
    ```
```
super_categories()

class sage.categories.category_with_axiom.Blahs

Bases: sage.categories.category_singleton

A toy singleton category, for testing purposes.

This is the root of a hierarchy of mathematically meaningless categories, used for testing Sage's category framework:

- Bars
- TestObjects
- TestObjectsOverBaseRing

Blue_extra_super_categories()

Illustrates a current limitation in the way to have an axiom imply another one.

Here, we would want Blue to imply Unital, and to put the class for the category of unital blue blahs in Blahs.Unital.Blue rather than Blahs.Blue.

This currently fails because Blahs is the category where the axiom Blue is defined, and the specifications currently impose that a category defining an axiom should also implement it (here in a category with axiom Blahs.Blue). In practice, due to this violation of the specifications, the axiom is lost during the join calculation.

Todo: Decide whether we care about this feature. In such a situation, we are not really defining a new axiom, but just defining an axiom as an alias for a couple others, which might not be that useful.

Todo: Improve the infrastructure to detect and report this violation of the specifications, if this is easy. Otherwise, it's not so bad: when defining an axiom A in a category Cs the first thing one is supposed to doctest is that Cs().A() works. So the problem should not go unnoticed.

class Commutative(base_category)

Bases: CategoryWithAxiom

class Connected(base_category)

Bases: CategoryWithAxiom

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom

class Flying(base_category)

Bases: CategoryWithAxiom
This illustrates a way to have an axiom imply another one.

Here, we want `Flying` to imply `Unital`, and to put the class for the category of unital flying blahs in `Blahs.Flying` rather than `Blahs.Unital.Flying`.

class SubcategoryMethods
    Bases: object
    Blue()
    Commutative()
    Connected()
    FiniteDimensional()
    Flying()
    Unital()

class Unital(base_category)
    Bases: CategoryWithAxiom

class Blue(base_category)
    Bases: CategoryWithAxiom

super_categories()

class sage.categories.category_with_axiom.CategoryWithAxiom(base_category)
    Bases: Category

An abstract class for categories obtained by adding an axiom to a base category.

See the category primer, and in particular its section about axioms for an introduction to axioms, and CategoryWithAxiom for how to implement axioms and the documentation of the axiom infrastructure.

static __classcall__(*args, **options)
    Make `FoosBar(**)` an alias for `Foos(**)._with_axiom("Bar")`.

EXAMPLES:

```
sage: FiniteGroups()
Category of finite groups
sage: ModulesWithBasis(ZZ)
Category of modules with basis over Integer Ring
sage: AlgebrasWithBasis(QQ)
Category of algebras with basis over Rational Field
```

```
>>> from sage.all import *
>>> FiniteGroups()
Category of finite groups
>>> ModulesWithBasis(ZZ)
Category of modules with basis over Integer Ring
>>> AlgebrasWithBasis(QQ)
Category of algebras with basis over Rational Field
```

This is relevant when e.g. `Foos(**)` does some non trivial transformations:
\texttt{sage}: Modules(QQ) \texttt{is} VectorSpaces(QQ) \\
True \\
\texttt{sage}: type(Modules(QQ)) \\
<class 'sage.categories.vector_spaces.VectorSpaces_with_category'> \\
\texttt{sage}: ModulesWithBasis(QQ) \texttt{is} VectorSpaces(QQ).WithBasis() \\
True \\
\texttt{sage}: type(ModulesWithBasis(QQ)) \\
<class 'sage.categories.vector_spaces.VectorSpaces.WithBasis_with_category'> \\

\begin{lstlisting}
>>> from sage.all import * 
>>> Modules(QQ) \texttt{is} VectorSpaces(QQ) 
True \\\n>>> ModulesWithBasis(QQ) \texttt{is} VectorSpaces(QQ).WithBasis() 
True \\
>>> ModulesWithBasis(QQ)) \\
\end{lstlisting}

\texttt{static \_classget\_} (\texttt{base\_category, base\_category\_class}) 
Implement the binding behavior for categories with axioms. 

This method implements a binding behavior on category with axioms so that, when a category \texttt{Cs} implements an axiom \texttt{A} with a nested class \texttt{Cs.A}, the expression \texttt{Cs().A} evaluates to the method defining the axiom \texttt{A} and not the nested class. See those design notes for the rationale behind this behavior.

\textbf{EXAMPLES:}

\begin{lstlisting}
\texttt{sage}: Sets().Infinite() 
Category of infinite sets 
\texttt{sage}: Sets().Infinite 
Cached version of <function ...Infinite at ...> 
\texttt{sage}: Sets().Infinite.f == Sets.SubcategoryMethods.Infinite.f 
True \\

>>> from sage.all import * 
>>> Sets().Infinite() 
Category of infinite sets 
>>> Sets().Infinite 
Cached version of <function ...Infinite at ...> 
>>> Sets().Infinite.f == Sets.SubcategoryMethods.Infinite.f 
True \\

We check that this also works when the class is implemented in a separate file, and lazy imported:

\texttt{sage}: Sets().Finite 
Cached version of <function ...Finite at ...> \\

\begin{lstlisting}
>>> from sage.all import * 
>>> Sets().Finite 
Cached version of <function ...Finite at ...> \\
\end{lstlisting}

There is no binding behavior when accessing \texttt{Finite} or \texttt{Infinite} from the class of the category instead of the category itself:
This method also initializes the attribute `_base_category_class_and_axiom` if not already set:

```python
>>> from sage.all import *
>>> Sets.Infinite._base_category_class_and_axiom
<class 'sage.categories.sets_cat.Sets'>, 'Infinite')
>>> Sets.Infinite._base_category_class_and_axiom_origin
'set by __classget__'
```

---

`__init__` *(base_category)*

`_repr_object_names()`

The names of the objects of this category, as used by `_repr_`.

See also:

`Category._repr_object_names()`

EXAMPLES:

```python
sage: FiniteSets()._repr_object_names()
'finite sets'
sage: AlgebrasWithBasis(QQ).FiniteDimensional()._repr_object_names()
'finite dimensional algebras with basis over Rational Field'
sage: Monoids()._repr_object_names()
'monoids'
sage: Semigroups().Unital().Finite()._repr_object_names()
'finite monoids'
sage: Algebras(QQ).Commutative()._repr_object_names()
'commutative algebras over Rational Field'
```
Note: This is implemented by taking _repr_object_names from self._without_axioms(named=True), and adding the names of the relevant axioms in appropriate order.

```
static _repr_object_names_static(category, axioms)
INPUT:
  • base_category – a category
  • axioms – a list or iterable of strings
EXAMPLES:

sage: from sage.categories.category_with_axiom import CategoryWithAxiom
sage: CategoryWithAxiom._repr_object_names_static(Semigroups(), ["Flying", "Blue"])  
'flying blue semigroups'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ), ["Flying", "WithBasis", "Blue"])  
'flying blue algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ), ["WithBasis"])
  
'algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ), ["WithBasis"])  
'algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object_names_static(Sets().Finite().Subquotients(), ["Finite"])  
'subquotients of finite sets'
sage: CategoryWithAxiom._repr_object_names_static(Monoids(), ["Unital"])  
'monoids'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ['x']['y']), ["Flying", "WithBasis", "Blue"])
  
'flying blue algebras with basis over Univariate Polynomial Ring in y over␣  
  Univariate Polynomial Ring in x over Rational Field'
```

If the axioms is a set or frozen set, then they are first sorted using canonicalize_axioms():

```
sage: CategoryWithAxiom._repr_object_names_static(Semigroups(), set(["Finite", "Commutative", "Facade"]))
'facade finite commutative semigroups'
```
See also:

_repr_object_names()

Note: The logic here is shared between _repr_object_names() and category.

_join_category_with_axiom(**options)
Run generic tests on this category with axioms.

See also:

TestSuite.

This check that an axiom category of a Category_singleton is a singleton category, and similarwise for Category_over_base_ring.

EXAMPLES:

sage: Sets().Finite()._test_category_with_axiom()
sage: Modules(ZZ).FiniteDimensional()._test_category_with_axiom()

>>> from sage.all import *
>>> Sets().Finite()._test_category_with_axiom()
>>> Modules(ZZ).FiniteDimensional()._test_category_with_axiom()

_without_axioms (named=False)
Return the category without the axioms that have been added to create it.

EXAMPLES:

sage: Sets().Finite()._without_axioms()
Category of sets
sage: Monoids().Finite()._without_axioms()
Category of magmas

This is because:

sage: Semigroups().Unital() is Monoids()
True

>>> from sage.all import *
>>> Semigroups().Unital() is Monoids()
True
If `named` is `True`, then `_without_axioms` stops at the first category that has an explicit name of its own:

```python
sage: Sets().Finite()._without_axioms(named=True)
Category of sets
sage: Monoids().Finite()._without_axioms(named=True)
Category of monoids
```

Technically we test this by checking if the class specifies explicitly the attribute `_base_category_class_and_axiom` by looking up `_base_category_class_and_axiom_origin`.

Some more examples:

```python
sage: Algebras(QQ).Commutative()._without_axioms()
Category of magmatic algebras over Rational Field
sage: Algebras(QQ).Commutative()._without_axioms(named=True)
Category of algebras over Rational Field
```

### additional_structure()

Return the additional structure defined by `self`.

**OUTPUT:** None

By default, a category with axiom defines no additional structure.

See also:

`Category.additional_structure()`.

**EXAMPLES:**

```python
sage: Sets().Finite().additional_structure()
sage: Monoids().additional_structure()
```

### axioms()

Return the axioms known to be satisfied by all the objects of `self`.

See also:

`Category.axioms()`

**EXAMPLES:**

```python
>>> from sage.all import *
>>> Sets().Finite().additional_structure()
>>> Monoids().additional_structure()
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: C.axioms()
frozenset({'Finite'})

sage: C = Modules(GF(5)).FiniteDimensional(); C
Category of finite dimensional vector spaces over Finite Field of size 5
sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Finite', 'FiniteDimensional']

sage: sorted(FiniteMonoids().Algebras(QQ).axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Associative', 'Distributive', 'FiniteDimensional', 'Unital', 'WithBasis']

sage: sorted(FiniteMonoids().Algebras(GF(Integer(3))).axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Associative', 'Distributive', 'Finite', 'FiniteDimensional', 'Unital', 'WithBasis']

sage: from sage.categories.magmas_and_additive_magmas import *

sage: MagmasAndAdditiveMagmas().Distributive().Unital().axioms()
frozenset({Distributive, Unital})

sage: D = MagmasAndAdditiveMagmas().Distributive()
sage: X = D.AdditiveAssociative().AdditiveCommutative(). Associative()
sage: X.Unital().super_categories()[1]
Category of monoids
sage: X.Unital().super_categories()[1] is Monoids()
True

(continues on next page)
base_category()

Return the base category of self.

EXAMPLES:

```python
sage: C = Sets.Finite(); C
Category of finite sets
sage: C.base_category()
Category of sets
sage: C._without_axioms()
Category of sets
```

extra_super_categories()

Return the extra super categories of a category with axiom.

Default implementation which returns [].

EXAMPLES:

```python
sage: FiniteSets().extra_super_categories()
[]
```

super_categories()

Return a list of the (immediate) super categories of self, as per `Category.super_categories()`.

This implements the property that if As is a subcategory of Bs, then the intersection of As with FiniteSets() is a subcategory of As and of the intersection of Bs with FiniteSets().

EXAMPLES:

A finite magma is both a magma and a finite set:

```python
sage: Magmas().Finite().super_categories()
[Category of magmas, Category of finite sets]
```
>>> from sage.all import *
>>> Magmas().Finite().super_categories()
[Category of magmas, Category of finite sets]

Variants:

sage: Sets().Finite().super_categories()
[Category of sets]
sage: Monoids().Finite().super_categories()
[Category of monoids, Category of finite semigroups]

>>> from sage.all import *
>>> Sets().Finite().super_categories()
[Category of sets]

>>> Monoids().Finite().super_categories()
[Category of monoids, Category of finite semigroups]

EXAMPLES:

class sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring(base_category)

Bases: CategoryWithAxiom, Category_over_base_ring

class sage.categories.category_with_axiom.CategoryWithAxiom_singleton(base_category)

Bases: Category_singleton, CategoryWithAxiom

class sage.categories.category_with_axiom.TestObjects

Bases: Category_singleton

A toy singleton category, for testing purposes.

See also:

Blahs

class Commutative(base_category)

Bases: CategoryWithAxiom

class Facade(base_category)

Bases: CategoryWithAxiom

class Finite(base_category)

Bases: CategoryWithAxiom

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom

class Finite(base_category)

Bases: CategoryWithAxiom
class Unital(base_category)
    Bases: CategoryWithAxiom

class Commutative(base_category)
    Bases: CategoryWithAxiom

class Unital(base_category)
    Bases: CategoryWithAxiom

sage.categories.category_with_axiom.TestObjectsOverBaseRing(base,
    name=None)

Bases: Category_over_base_ring
A toy singleton category, for testing purposes.
See also:
Blahs

class Commutative(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Facade(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Finite(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class FiniteDimensional(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class FiniteDimensional(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Finite(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Unital(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Commutative(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class Unital(base_category)
    Bases: CategoryWithAxiom_over_base_ring

sage.categories.category_with_axiom.axiom(axiom)

Return a function/method self -> self._with_axiom(axiom).
This can used as a shorthand to define axioms, in particular in the tests below. Usually one will want to attach
documentation to an axiom, so the need for such a shorthand in real life might not be that clear, unless we start
creating lots of axioms.
In the long run maybe this could evolve into an @axiom decorator.

EXAMPLES:
sage: from sage.categories.category_with_axiom import axiom
sage: axiom("Finite") (Semigroups())
Category of finite semigroups

>>> from sage.all import *
>>> from sage.categories.category_with_axiom import axiom
>>> axiom("Finite") (Semigroups())
Category of finite semigroups

Upon assigning the result to a class this becomes a method:

sage: class As:
....:     def _with_axiom(self, axiom):
....:         return self, axiom
sage: As().Finite()
(<__main__.As ... at ...>, 'Finite')

>>> from sage.all import *
>>> class As:
...   def _with_axiom(self, axiom):
...     return self, axiom
... >>> As().Finite()
... (<__main__.As ... at ...>, 'Finite')

sage.categories.category_with_axiom.axiom_of_nested_class(nested_cls)

Given a class and a nested axiom class, return the axiom.

EXAMPLES:

This uses some heuristics like checking if the nested_cls carries the name of the axiom, or is built by appending or
prepending the name of the axiom to that of the class:

sage: from sage.categories.category_with_axiom import TestObjects, axiom_of_nested_class
sage: axiom_of_nested_class(TestObjects, TestObjects.FiniteDimensional)
'FiniteDimensional'
sage: axiom_of_nested_class( TestObjects.FiniteDimensional, TestObjects.FiniteDimensional.Finite)
'Finite'
sage: axiom_of_nested_class( Sets, FiniteSets)
'Finite'
sage: axiom_of_nested_class( Algebras, AlgebrasWithBasis)
'WithBasis'

>>> from sage.all import *
>>> from sage.categories.category_with_axiom import TestObjects, axiom_of_nested_class
>>> axiom_of_nested_class(TestObjects, TestObjects.FiniteDimensional)
'FiniteDimensional'
>>> axiom_of_nested_class(TestObjects.FiniteDimensional, TestObjects.FiniteDimensional.Finite)
'Finite'
>>> axiom_of_nested_class(Sets, FiniteSets)
'Finite'
>>> axiom_of_nested_class(Algebras, AlgebrasWithBasis)
'WithBasis'
In all other cases, the nested class should provide an attribute \_base_category_class_and_axiom:

```python
sage: Semigroups._base_category_class_and_axiom
(<class 'sage.categories.magas.Magmas'>, 'Associative')

sage: axiom_of_nested_class(Magmas, Semigroups)
'Associative'
```

```python
from sage.all import *

>> from sage.all import *
>> Semigroups._base_category_class_and_axiom
(<class 'sage.categories.magas.Magmas'>, 'Associative')
>> axiom_of_nested_class(Magmas, Semigroups)
'Associative'
```

Try to deduce the base category and the axiom from the name of cls.

The heuristic is to try to decompose the name as the concatenation of the name of a category and the name of an axiom, and looking up that category in the standard location (i.e. in `sage.categories.hopf_algebras` for HopfAlgebras, and in `sage.categories.sets_cat` as a special case for Sets).

If the heuristic succeeds, the result is guaranteed to be correct. Otherwise, an error is raised.

**EXAMPLES:**

```python
sage: from sage.categories.category_with_axiom import base_category_class_and_axiom
sage: base_category_class_and_axiom(FiniteSets)
(<class sage.categories.sets_cat.Sets>, Finite)

sage: Sets.Finite
<class sage.categories.finite_sets.FiniteSets>

sage: base_category_class_and_axiom(Sets.Finite)
(<class 'sage.categories.sets_cat.Sets'>, 'Finite')

sage: base_category_class_and_axiom(FiniteDimensionalHopfAlgebrasWithBasis)
(<class 'sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis'>, 'FiniteDimensional')

sage: base_category_class_and_axiom(HopfAlgebrasWithBasis)
(<class sage.categories.hopf_algebras.HopfAlgebras>, 'WithBasis')
```

```python
from sage.all import *

>> from sage.all import *
>> base_category_class_and_axiom(FiniteSets)
(<class 'sage.categories.sets_cat.Sets'>, 'Finite')
>> Sets.Finite
<class 'sage.categories.finite_sets.FiniteSets'>
>> base_category_class_and_axiom(Sets.Finite)
(<class 'sage.categories.sets_cat.Sets'>, 'Finite')

>> base_category_class_and_axiom(FiniteDimensionalHopfAlgebrasWithBasis)
(<class 'sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis'>, 'FiniteDimensional')

>> base_category_class_and_axiom(HopfAlgebrasWithBasis)
(<class 'sage.categories.hopf_algebras.HopfAlgebras'>, 'WithBasis')
```

Along the way, this does some sanity checks:
```python
sage: class FacadeSemigroups(CategoryWithAxiom):
....:    pass
sage: base_category_class_and_axiom(FacadeSemigroups)
Traceback (most recent call last):
..._
AssertionError: Missing (lazy import) link
for <class 'sage.categories.semigroups.Semigroups'>
to <class '__main__.FacadeSemigroups'> for axiom Facade?

sage: Semigroups.Facade = FacadeSemigroups
sage: base_category_class_and_axiom(FacadeSemigroups)
..._
AssertionError: Missing (lazy import) link
for <class 'sage.categories.semigroups.Semigroups'>
to <class '__main__.FacadeSemigroups'> for axiom Facade?

>>> from sage.all import *
>>> class FacadeSemigroups(CategoryWithAxiom):
...    pass
>>> base_category_class_and_axiom(FacadeSemigroups)
Traceback (most recent call last):
..._
AssertionError: Missing (lazy import) link
for <class 'sage.categories.semigroups.Semigroups'>
to <class '__main__.FacadeSemigroups'> for axiom Facade?

>>> Semigroups.Facade = FacadeSemigroups
>>> base_category_class_and_axiom(FacadeSemigroups)
..._

Note: In the following example, we could possibly retrieve Sets from the class name. However this cannot be implemented robustly until Issue #9107 is fixed. Anyway this feature has not been needed so far:

```
2.4 Functors

AUTHORS:

- David Kohel and William Stein
- David Joyner (2005-12-17): examples
- Simon King (2010-04-30): more examples, several bug fixes, re-implementation of the default call method, making functors applicable to morphisms (not only to objects)
- Simon King (2010-12): Pickling of functors without losing domain and codomain

```
sage.categories.category_with_axiom.uncamelcase("JTrivialMonoids")
\n"j trivial monoids"
```

```
sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras", \n\n→ "_")
\n"finite dimensional algebras"
```

```
>>> from sage.all import *
```

```
>>> sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras")
\n'finite dimensional algebras'
```

```
>>> sage.categories.category_with_axiom.uncamelcase("JTrivialMonoids")
\n'j trivial monoids'
```

```
>>> sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras", \n\n→ "_")
\n'finite dimensional algebras'
```

sage.categories.functor.ForgetfulFunctor(domain, codomain)

Construct the forgetful function from one category to another.

INPUT:

C, D – two categories

OUTPUT:

A functor that returns the corresponding object of D for any element of C, by forgetting the extra structure.

ASSUMPTION:

The category C must be a sub-category of D.

EXAMPLES:

```
sage: rings = Rings()
sage: abgrps = CommutativeAdditiveGroups()
sage: F = ForgetfulFunctor(rings, abgrps)
sage: F
```

The forgetful functor from Category of rings to Category of commutative additive groups

```
>>> from sage.all import *
```

```
>>> rings = Rings()
```

(continues on next page)
>>> abgrps = CommutativeAdditiveGroups()
>>> F = ForgetfulFunctor(rings, abgrps)
>>> F
The forgetful functor
from Category of rings
to Category of commutative additive groups

It would be a mistake to call it in opposite order:

```
sage: F = ForgetfulFunctor(abgrps, rings)
Traceback (most recent call last):
  ...  
ValueError: Forgetful functor not supported for domain
Category of commutative additive groups
```

```
>>> from sage.all import *
```

```
>>> F = ForgetfulFunctor(abgrps, rings)
Traceback (most recent call last):
  ...  
ValueError: Forgetful functor not supported for domain
Category of commutative additive groups
```

If both categories are equal, the forgetful functor is the same as the identity functor:

```
sage: ForgetfulFunctor(abgrps, abgrps) == IdentityFunctor(abgrps)
True
```

```
>>> from sage.all import *
```

```
>>> ForgetfulFunctor(abgrps, abgrps) == IdentityFunctor(abgrps)
True
```

```
class sage.categories.functor.ForgetfulFunctor_generic

Bases: Functor

The forgetful functor, i.e., embedding of a subcategory.

NOTE:

Forgetful functors should be created using ForgetfulFunctor(), since the init method of this class does not check whether the domain is a subcategory of the codomain.

EXAMPLES:

```
sage: F = ForgetfulFunctor(FiniteFields(), Fields())  # indirect doctest
sage: F
The forgetful functor
from Category of finite enumerated fields
to Category of fields
sage: F(GF(3))
Finite Field of size 3
```

```
>>> from sage.all import *
```

```
>>> F = ForgetfulFunctor(FiniteFields(), Fields())  # indirect doctest
```
class sage.categories.functor.Functor

Bases: SageObject

A class for functors between two categories

NOTE:

• In the first place, a functor is given by its domain and codomain, which are both categories.
• When defining a sub-class, the user should not implement a call method. Instead, one should implement three methods, which are composed in the default call method:
  - `_coerce_into_domain(self, x)`: Return an object of self’s domain, corresponding to x, or raise a `TypeError`.
    * Default: Raise `TypeError` if x is not in self’s domain.
  - `_apply_functor(self, x)`: Apply self to an object x of self’s domain.
    * Default: Conversion into self’s codomain.
  - `_apply_functor_to_morphism(self, f)`: Apply self to a morphism f in self’s domain.
    - Default: Return self(f.domain()).hom(f, self(f.codomain())).

EXAMPLES:

```python
sage: rings = Rings()
sage: abgrps = CommutativeAdditiveGroups()
sage: F = ForgetfulFunctor(rings, abgrps)
sage: F.domain()
Category of rings
sage: F.codomain()
Category of commutative additive groups
sage: from sage.categories.functor import Functor
sage: isinstance(F, Functor)
True
sage: I = IdentityFunctor(abgrps)
sage: I
The identity functor on Category of commutative additive groups
sage: I.domain()
Category of commutative additive groups
sage: isinstance(I, Functor)
True
```

```python
>>> from sage.all import *
>>> rings = Rings()
>>> abgrps = CommutativeAdditiveGroups()
>>> F = ForgetfulFunctor(rings, abgrps)
>>> F.domain()
Category of rings
>>> F.codomain()
Category of commutative additive groups
>>> from sage.categories.functor import Functor
>>> isinstance(F, Functor)
True
```
I = IdentityFunctor(abgrps)
I
The identity functor on Category of commutative additive groups
I.domain()
Category of commutative additive groups
isinstance(I, Functor)
True

Note that by default, an instance of the class Functor is coercion from the domain into the codomain. The above subclasses overloaded this behaviour. Here we illustrate the default:

>>> from sage.categories.functor import Functor
>>> F = Functor(Rings(), Fields())
>>> F
Functor from Category of rings to Category of fields
>>> F(ZZ)
Rational Field
>>> F(GF(Integer(2)))
Finite Field of size 2

Functors are not only about the objects of a category, but also about their morphisms. We illustrate it, again, with the coercion functor from rings to fields.

>>> from sage.all import *
>>> from sage.categories.functor import Functor
>>> F = Functor(Rings(), Fields())
>>> F
Functor from Category of rings to Category of fields
>>> F(ZZ)
Rational Field
>>> F(GF(Integer(2)))
Finite Field of size 2

sage: R1.<x> = ZZ[]
sage: R2.<a,b> = QQ[]
sage: f = R1.hom([a + b], R2)
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Multivariate Polynomial Ring in a, b over Rational Field
  Defn: x |--> a + b
sage: F(f)
Ring morphism:
  From: Fraction Field of Univariate Polynomial Ring in x over Integer Ring
  To:   Fraction Field of Multivariate Polynomial Ring in a, b over Rational Field
  Defn: x |--> a + b
sage: F(f)(1/x)
1/(a + b)

>>> from sage.all import *
>>> R1 = ZZ['x']; (x,) = R1._first_ngens(1)
>>> R2 = QQ['a, b']; (a, b,) = R2._first_ngens(2)
>>> f = R1.hom([a + b], R2)
>>> f
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Multivariate Polynomial Ring in a, b over Rational Field
  Defn: x |--> a + b

(continues on next page)
To: Multivariate Polynomial Ring in a, b over Rational Field
Defn: x |--> a + b

>>> F(f)

Ring morphism:
From: Fraction Field of Univariate Polynomial Ring in x over Integer Ring
To: Fraction Field of Multivariate Polynomial Ring in a, b over Rational Field
Defn: x |--> a + b

>>> F(f)(Integer(1)/x)

1/(a + b)

We can also apply a polynomial ring construction functor to our homomorphism. The result is a homomorphism that is defined on the base ring:

```
sage: F = QQ['t'].construction()[0]
sage: F
Poly[t]
sage: F(f)
Ring morphism:
From: Univariate Polynomial Ring in t
over Univariate Polynomial Ring in x over Integer Ring
To: Univariate Polynomial Ring in t
over Multivariate Polynomial Ring in a, b over Rational Field
Defn: Induced from base ring by
Ring morphism:
From: Univariate Polynomial Ring in x over Integer Ring
To: Multivariate Polynomial Ring in a, b over Rational Field
Defn: x |--> a + b

sage: p = R1['t']('(-x^2 + x)*t^2 + (x^2 - x)*t - 4*x^2 - x + 1')
sage: F(f)(p)

(-a^2 - 2*a*b - b^2 + a + b)*t^2 + (a^2 + 2*a*b + b^2 - a - b)*t
- 4*a^2 - 8*a*b - 4*b^2 - a - b + 1
```

```
>>> from sage.all import *
   >>> F = QQ['t'].construction()[Integer(0)]
   >>> F
Poly[t]
   >>> F(f)
   Ring morphism:
   From: Univariate Polynomial Ring in t
over Univariate Polynomial Ring in x over Integer Ring
To: Univariate Polynomial Ring in t
over Multivariate Polynomial Ring in a, b over Rational Field
Defn: Induced from base ring by
   Ring morphism:
   From: Univariate Polynomial Ring in x over Integer Ring
   To: Multivariate Polynomial Ring in a, b over Rational Field
   Defn: x |--> a + b

   >>> p = R1['t']('(-x^2 + x)*t^2 + (x^2 - x)*t - 4*x^2 - x + 1')
   >>> F(f)(p)

   (-a^2 - 2*a*b - b^2 + a + b)*t^2 + (a^2 + 2*a*b + b^2 - a - b)*t
   - 4*a^2 - 8*a*b - 4*b^2 - a - b + 1
```

codomain()

The codomain of self

EXAMPLES:
sage: F = ForgetfulFunctor(FiniteFields(), Fields())
sage: F.codomain()
Category of fields

>>> from sage.all import *
>>> F = ForgetfulFunctor(FiniteFields(), Fields())
>>> F.codomain()
Category of fields

domain()
The domain of self

EXAMPLES:

sage: F = ForgetfulFunctor(FiniteFields(), Fields())
sage: F.domain()
Category of finite enumerated fields

>>> from sage.all import *
>>> F = ForgetfulFunctor(FiniteFields(), Fields())
>>> F.domain()
Category of finite enumerated fields

sage.categories.functor.IdentityFunctor(C)
Construct the identity functor of the given category.

INPUT:
A category, C.

OUTPUT:
The identity functor in C.

EXAMPLES:

sage: rings = Rings()
sage: F = IdentityFunctor(rings)
sage: F(ZZ['x','y']) is ZZ['x','y']
True

>>> from sage.all import *
>>> rings = Rings()
>>> F = IdentityFunctor(rings)
>>> F(ZZ['x','y']) is ZZ['x','y']
True

class sage.categories.functor.IdentityFunctor_generic(C)
Bases: ForgetfulFunctor_generic
Generic identity functor on any category

NOTE:
This usually is created using IdentityFunctor().

EXAMPLES:
sage: F = IdentityFunctor(Fields())  # indirect doctest
sage: F
The identity functor on Category of fields
sage: F(RR) is RR
True
sage: F(ZZ)
Traceback (most recent call last):
... ...
            TypeError: x (=Integer Ring) is not in Category of fields

>>> from sage.all import *
>>> F = IdentityFunctor(Fields())  # indirect doctest
>>> F(RR) is RR
True
>>> F(ZZ)
Traceback (most recent call last):
... ...
            TypeError: x (=Integer Ring) is not in Category of fields

sage.categories.functor.is_Functor(x)

Test whether the argument is a functor.

This function is deprecated.

EXAMPLES:

sage: from sage.categories.functor import is_Functor
sage: F1 = QQ.construction()[0]
sage: F1
FractionField
sage: is_Functor(F1)
doctest:warning...
DeprecationWarning: The function is_Functor is deprecated; use 'isinstance(..., Functor)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
sage: is_Functor(FractionField)
False
sage: F2 = ForgetfulFunctor(Fields(), Rings())
sage: F2
The forgetful functor from Category of fields to Category of rings
sage: is_Functor(F2)
True

>>> from sage.all import *
>>> from sage.categories.functor import is_Functor
>>> F1 = QQ.construction()[Integer(0)]
>>> F1
FractionField
>>> is_Functor(F1)
doctest:warning...
DeprecationWarning: The function is_Functor is deprecated; use 'isinstance(..., Functor)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
2.5 Implementing a new parent: a tutorial

The easiest approach for implementing a new parent is to start from a close example in `sage.categories.examples`. Here, we will get through the process of implementing a new finite semigroup, taking as starting point the provided example:

```python
sage: S = FiniteSemigroups().example()
sage: S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
```

You may lookup the implementation of this example with:

```python
sage: S ? # not tested
```

Or by browsing the source code of `sage.categories.examples.finite_semigroups.LeftRegularBand`.

Copy-paste this code into, say, a cell of the notebook, and replace every occurrence of `FiniteSemigroups().example(...)` in the documentation by `LeftRegularBand`. This will be equivalent to:

```python
sage: from sage.categories.examples.finite_semigroups import LeftRegularBand
```

Now, try:

```python
sage: S = LeftRegularBand(); S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
```

```python
>>> from sage.all import *
```
and play around with the examples in the documentation of $S$ and of \textit{FiniteSemigroups}.

Rename the class to \textit{ShiftSemigroup}, and modify the product to implement the semigroup generated by the given alphabet such that $au = u$ for any $u$ of length 3.

Use \textit{TestSuite} to test the newly implemented semigroup; draw its Cayley graph.

Add another option to the constructor to generalize the construction to any $u$ of length $k$.

Lookup the Sloane for the sequence of the sizes of those semigroups.

Now implement the commutative monoid of subsets of $\{1, \ldots, n\}$ endowed with union as product. What is its category? What are the extra functionalities available there? Implement iteration and cardinality.

\textbf{Todo:} The tutorial should explain there how to reuse the enumerated set of subsets, and endow it with more structure.
3.1 Base class for maps

AUTHORS:
- Robert Bradshaw: initial implementation

```
class sage.categories.map.FormalCompositeMap
    Bases: Map

    Formal composite maps.

    A formal composite map is formed by two maps, so that the codomain of the first map is contained in the domain of the second map.

    Note: When calling a composite with additional arguments, these arguments are only passed to the second underlying map.

EXAMPLES:
```
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: from sage.categories.morphism import SetMorphism
sage: f = SetMorphism(Hom(R, S, Rings()), lambda p: p[0]*a^p.degree())
sage: g = S.hom([2*x])
sage: f*g
Composite map:
    From: Univariate Polynomial Ring in a over Rational Field
    To:   Univariate Polynomial Ring in a over Rational Field
    Defn: Ring morphism:
        From: Univariate Polynomial Ring in a over Rational Field
        To:   Univariate Polynomial Ring in x over Rational Field
        Defn: a |--> 2*x
        then
        Generic morphism:
            From: Univariate Polynomial Ring in x over Rational Field
            To:   Univariate Polynomial Ring in a over Rational Field
sage: g*f
Composite map:
    From: Univariate Polynomial Ring in x over Rational Field
    To:   Univariate Polynomial Ring in x over Rational Field
    Defn: Generic morphism:
        From: Univariate Polynomial Ring in x over Rational Field
        To:   Univariate Polynomial Ring in x over Rational Field
```

(continues on next page)
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Univariate Polynomial Ring in \( x \) over Rational Field
Defn:

\[
\text{Generic morphism:}
\]
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Univariate Polynomial Ring in \( a \) over Rational Field
then

\[
\text{Ring morphism:}
\]
From: Univariate Polynomial Ring in \( a \) over Rational Field
To: Univariate Polynomial Ring in \( x \) over Rational Field
Defn: \( a \mapsto 2x \)

sage: \((f*g)(2*a^2+5)\)
\[5*a^2\]
sage: \((g*f)(2*x^2+5)\)
\[20*x^2\]

```python
>>> from sage.all import *

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> S = QQ['a']; (a,) = S._first_ngens(1)

>>> from sage.categories.morphism import SetMorphism

>>> f = SetMorphism(Hom(R, S, Rings()), lambda p: p[Integer(0)]*a**p.degree())

>>> g = S.hom([Integer(2)*x])

>>> f*g

Composite map:
From: Univariate Polynomial Ring in \( a \) over Rational Field
To: Univariate Polynomial Ring in \( a \) over Rational Field
Defn:

\[
\text{Ring morphism:}
\]
From: Univariate Polynomial Ring in \( a \) over Rational Field
To: Univariate Polynomial Ring in \( x \) over Rational Field
Defn: \( a \mapsto 2x \)
then

\[
\text{Generic morphism:}
\]
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Univariate Polynomial Ring in \( a \) over Rational Field

>>> g*f

Composite map:
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Univariate Polynomial Ring in \( x \) over Rational Field
Defn:

\[
\text{Generic morphism:}
\]
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Univariate Polynomial Ring in \( a \) over Rational Field
then

\[
\text{Ring morphism:}
\]
From: Univariate Polynomial Ring in \( a \) over Rational Field
To: Univariate Polynomial Ring in \( x \) over Rational Field
Defn: \( a \mapsto 2x \)
```

```python
>>> (f*g)(Integer(2)*a**Integer(2)+Integer(5))
\[5*a^2\]

>>> (g*f)(Integer(2)*x**Integer(2)+Integer(5))
\[20*x^2\]
```

domains()
Iterate over the domains of the factors of this map.
(This is useful in particular to check for loops in coercion maps.)

See also:

Map.domains()
EXAMPLES:

```python
sage: f = QQ.coerce_map_from(ZZ)
sage: g = MatrixSpace(QQ, 2, 2).coerce_map_from(QQ)  # needs sage.modules
sage: list((g * f).domains())  # needs sage.modules
[Integer Ring, Rational Field]
```

```python
>>> from sage.all import *
>>> f = QQ.coerce_map_from(ZZ)
>>> g = MatrixSpace(QQ, Integer(2), Integer(2)).coerce_map_from(QQ)  # needs sage.modules
...>>> list((g * f).domains())  # needs sage.modules
...[Integer Ring, Rational Field]
```

```python
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: from sage.categories.morphism import SetMorphism
def f = SetMorphism(Hom(R, S, Rings()), lambda p: p[Integer(0)]*a**p.degree())
sage: g = S.hom([2*x])
sage: fg = f * g
sage: fg.first() == g
True
sage: fg == fg.then() * fg.first()
True
```

```python
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> S = QQ['a']; (a,) = S._first_ngens(1)
>>> from sage.categories.morphism import SetMorphism
def f = SetMorphism(Hom(R, S, Rings()), lambda p: p[Integer(0)]*a**p.degree())
def g = S.hom([Integer(2)*x])
>>> fg = f * g
>>> fg.first() == g
True
>>> fg == fg.then() * fg.first()
True
```

```python
sage: V1 = QQ^2
sage: V2 = QQ^3
```

first()  
Return the first map in the formal composition.  
If self represents $f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0$, then self.first() returns $f_0$. We have self == self.then() * self.first().

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: from sage.categories.morphism import SetMorphism
def f = SetMorphism(Hom(R, S, Rings()), lambda p: p[0]*a**p.degree())
sage: g = S.hom([2*x])
sage: fg = f * g
sage: fg.first() == g
True
sage: fg == fg.then() * fg.first()
True
```

is_injective()  
Tell whether self is injective.  
It raises Not Implemented Error if it cannot be determined.

EXAMPLES:

```python
sage: V1 = QQ^2
sage: V2 = QQ^3
```

(continues on next page)
If both constituents are injective, the composition is injective:

```python
>>> from sage.categories.map import FormalCompositeMap
>>> c1 = FormalCompositeMap(Hom(QQ^1, V2, phi1.category_for()), phi1, phi2)
>>> c1.is_injective()
True
```

If it cannot be determined whether the composition is injective, an error is raised:

```python
>>> ps11 = V2.hom(Matrix([[1, 2], [3, 4], [5, 6]]), V1)
>>> c2 = FormalCompositeMap(Hom(V1, V1, phi2.category_for()), phi2, ps11)
>>> c2.is_injective()
Traceback (most recent call last):
... NotImplementedError: not enough information to deduce injectivity
```

If the first map is surjective and the second map is not injective, then the composition is not injective:
is_surjective()

Tell whether self is surjective.

It raises `NotImplementedError` if it cannot be determined.

EXAMPLES:

```python
sage: from sage.categories.map import FormalCompositeMap
sage: V3 = QQ^3 # needs sage.modules
sage: V2 = QQ^2 # needs sage.modules
sage: V1 = QQ^1 # needs sage.modules
>>> from sage.all import *
>>> from sage.categories.map import FormalCompositeMap
>>> V3 = QQ**Integer(3) # needs sage.modules
>>> V2 = QQ**Integer(2) # needs sage.modules
>>> V1 = QQ**Integer(1) # needs sage.modules
>>> if both maps are surjective, the composition is surjective:
```
If the second map is not surjective, the composition is not surjective:

```python
sage: FormalCompositeMap(Hom(V3, V1, phi32.category_for()),
            # needs sage.modules
            ....: phi32,
            ....: V2.hom(Matrix([[0], [0]]), V1)).is_surjective()
False
```  

```python
>>> from sage.all import *

>>> FormalCompositeMap(Hom(V3, V1, phi32.category_for()),
            # needs sage.modules
            ....: phi32,
            ....: V2.hom(Matrix([[Integer(0)], [Integer(0)]]), V1)).is_surjective()
False
```  

If the second map is an isomorphism and the first map is not surjective, then the composition is not surjective:

```python
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()),
            # needs sage.modules
            ....: V2.hom(Matrix([[0], [0]]), V1),
            ....: V1.hom(Matrix([[1]], V1)).is_surjective()
False
```  

```python
>>> from sage.all import *

>>> FormalCompositeMap(Hom(V2, V1, phi32.category_for()),
            # needs sage.modules
            ....: V2.hom(Matrix([[Integer(0)], [Integer(0)]]), V1),
            ....: V1.hom(Matrix([[Integer(1)]], V1)).is_surjective()
False
```  

Otherwise, surjectivity of the composition cannot be determined:

```python
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()),
            # needs sage.modules
            ....: V2.hom(Matrix([[1, 1], [1, 1]]), V2),
            ....: V2.hom(Matrix([[1], [1]]), V1)).is_surjective()
Traceback (most recent call last):
  ... Not Implemented: not enough information to deduce surjectivity
```  

```python
>>> from sage.all import *

>>> FormalCompositeMap(Hom(V2, V1, phi32.category_for()),
            # needs sage.modules
            ....: V2.hom(Matrix([[Integer(1), Integer(1)], [Integer(1), Integer(1)]]),
            ....: V2),
            ....: V2.hom(Matrix([[Integer(1)], [Integer(1)]]), V1)).is_surjective()
Traceback (most recent call last):
```
... NotImplementedError: not enough information to deduce surjectivity

section()

Compute a section map from sections of the factors of self if they have been implemented.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: incl = P.coerce_map_from(ZZ)
sage: sect = incl.section(); sect
Composite map:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Integer Ring
  Defn: Generic map:
        From: Univariate Polynomial Ring in x over Rational Field
        To:   Rational Field
        then
        Generic map:
        From: Rational Field
        To:   Integer Ring
sage: p = x + 5; q = x + 2
sage: sect(p-q)
3
```

```
>>> from sage.all import *
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> incl = P.coerce_map_from(ZZ)
>>> sect = incl.section(); sect
Composite map:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Integer Ring
  Defn: Generic map:
        From: Univariate Polynomial Ring in x over Rational Field
        To:   Rational Field
        then
        Generic map:
        From: Rational Field
        To:   Integer Ring
>>> p = x + Integer(5); q = x + Integer(2)
>>> sect(p-q)
3
```

the following example has been attached to _integer_() of sage.rings.polynomial.polynomial_element.Polynomial before (see comment there):

```
sage: k = GF(47)
sage: R.<x> = PolynomialRing(k)
sage: R.coerce_map_from(ZZ).section()
Composite map:
  From: Univariate Polynomial Ring in x over Finite Field of size 47
  To:   Integer Ring
  Defn: Generic map:
        From: Univariate Polynomial Ring in x over Finite Field of size 47
        To:   Finite Field of size 47
        then
```

(continues on next page)
Lifting map:
From: Finite Field of size 47
To: Integer Ring

```
sage: ZZ(R(45))        # indirect doctest
45
```

```
sage: ZZ(3*x + 45)     # indirect doctest
Traceback (most recent call last):
... TypeError: 3*x + 45 is not a constant polynomial
```

then ()

Return the tail of the list of maps.

If `self` represents \( f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0 \), then `self.first()` returns \( f_n \circ f_{n-1} \circ \cdots \circ f_1 \). We have `self == self.then() * self.first()`.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

```
sage: f = QQ.coerce_map_from(ZZ)  
```

""" (continues on next page)"""
```python
>>> from sage.all import *

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> S = QQ['a']; (a,) = S._first_ngens(1)

>>> from sage.categories.morphism import SetMorphism

>>> f = SetMorphism(Hom(R, S, Rings()), lambda p: p[Integer(0)]*a**p.degree())
>>> g = S.hom([Integer(2)*x])

>>> (f*g).then() == f
True

>>> f = QQ.coerce_map_from(ZZ)

>>> f = f.extend_domain(ZZ).extend_codomain(QQ)

>>> f.then()
Composite map:
From: Integer Ring
To: Rational Field
Defn: Natural morphism:
From: Integer Ring
To: Rational Field
then
Identity endomorphism of Rational Field
```

```python
class sage.categories.map.Map

Bases: `Element`

Basic class for all maps.

Note: The call method is of course not implemented in this base class. This must be done in the sub classes, by overloading `_call_` and possibly also `_call_with_args_`.

EXAMPLES:

Usually, instances of this class will not be constructed directly, but for example like this:

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]

sage: Y = ZZ

sage: phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])

sage: phi(x**Integer(2)+2*x-1)
-1

sage: R.<x,y> = QQ[]

sage: f = R.hom([x+y, x-y], R)

sage: f(x**2+2*x*y + y^2 + 2*x + 2*y - 1)
```

```python
>>> from sage.all import *

>>> from sage.categories.morphism import SetMorphism

>>> X = ZZ['x']; (x,) = X._first_ngens(1)

>>> Y = ZZ

>>> phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[Integer(0)])

>>> phi(x**Integer(2)+Integer(2)*x-Integer(1))
-1

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
```

(continues on next page)
\begin{verbatim}
>>> f = R.hom([x+y, x-y], R)
>>> f(x**Integer(2)+Integer(2)*x-Integer(1))
x^2 + 2*x*y + y^2 + 2*x + 2*y - 1
\end{verbatim}

\textbf{category_for()}\textbf{ }

Returns the category self is a morphism for.

\textbf{Note: } This is different from the category of maps to which this map belongs \textit{as an object}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
sage: Y = ZZ
sage: phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])
sage: phi.category_for()
Category of rings
sage: phi.category()
Category of homsets of unital magmas and additive unital additive magmas
sage: R.<x,y> = QQ[]
sage: f = R.hom([x+y, x-y], R)
sage: f.category_for()
Join of Category of unique factorization domains and Category of commutative algebras over (number fields and quotient fields and metric spaces) and Category of infinite sets
sage: f.category()
Category of endsets of unital magmas and right modules over (number fields and quotient fields and metric spaces) and left modules over (number fields and quotient fields and metric spaces)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.categories.morphism import SetMorphism
>>> X = ZZ['x']; (x,) = X._first_ngens(1)
>>> Y = ZZ
>>> phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[Integer(0)])
>>> phi.category_for()
Category of rings
>>> phi.category()
Category of homsets of unital magmas and additive unital additive magmas
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> f = R.hom([x+y, x-y], R)
>>> f.category_for()
Join of Category of unique factorization domains and Category of commutative algebras over (number fields and quotient fields and metric spaces) and Category of infinite sets
>>> f.category()
Category of endsets of unital magmas and right modules over (number fields and quotient fields and metric spaces) and left modules over (number fields and quotient fields and metric spaces)
\end{verbatim}

\textbf{FIXME: } find a better name for this method

\textbf{codomain}
domain

domains()

Iterate over the domains of the factors of a (composite) map.

This default implementation simply yields the domain of this map.

See also:

FormalCompositeMap.domains()

EXAMPLES:

```
sage: list(QQ.coerce_map_from(ZZ).domains())
[Integer Ring]
```

```
>>> from sage.all import *

>>> list(QQ.coerce_map_from(ZZ).domains())
[Integer Ring]
```

extend_codomain(new_codomain)

INPUT:

• self—a member of Hom(X, Y)

• new_codomain—an object Z such that there is a canonical coercion \( \phi \) in Hom(Y, Z)

OUTPUT:

An element of Hom(X, Z) obtained by composing self with \( \phi \). If no canonical \( \phi \) exists, a TypeError is raised.

EXAMPLES:

```
sage: mor = QQ.coerce_map_from(ZZ)
sage: mor.extend_codomain(RDF)
Composite map:
  From: Integer Ring
  To:  Real Double Field
Defn:  Natural morphism:
    From: Integer Ring
    To:  Rational Field
    then
    Native morphism:
      From: Rational Field
      To:  Real Double Field

sage: mor.extend_codomain(GF(7))
Traceback (most recent call last):
  ...
TypeError: No coercion from Rational Field to Finite Field of size 7
```

```
>>> from sage.all import *

>>> mor = QQ.coerce_map_from(ZZ)

>>> mor.extend_codomain(RDF)
Composite map:
  From: Integer Ring
  To:  Real Double Field
Defn:  Natural morphism:
    From: Integer Ring
    To:  Rational Field
```

(continues on next page)
then
    Native morphism:
    From: Rational Field
    To: Real Double Field

>>> mor.extend_codomain(GF(Integer(7)))
Traceback (most recent call last):
...
TypeError: No coercion from Rational Field to Finite Field of size 7

**extend_domain** *(new_domain)*

**INPUT:**

- self – a member of Hom(Y, Z)
- new_codomain – an object X such that there is a canonical coercion \( \phi \) in Hom(X, Y)

**OUTPUT:**

An element of Hom(X, Z) obtained by composing self with \( \phi \). If no canonical \( \phi \) exists, a *TypeError* is raised.

**EXAMPLES:**

```
sage: # needs sage.rings.complex_double
sage: mor = CDF.coerce_map_from(RDF)
sage: mor.extend_domain(QQ)
Composite map:
    From: Rational Field
    To: Complex Double Field
    Defn: Native morphism:
        From: Rational Field
        To: Real Double Field
        then
    Native morphism:
        From: Real Double Field
        To: Complex Double Field
```

```
sage: mor.extend_domain(ZZ['x'])
Traceback (most recent call last):
...
TypeError: No coercion from Univariate Polynomial Ring in x over Integer Ring to Real Double Field
```

```
>>> from sage.all import *
```
... "TypeError: No coercion from Univariate Polynomial Ring in x over Integer Ring to Real Double Field"

is_surjective()

Tells whether the map is surjective (not implemented in the base class).

parent()

Return the homset containing this map.

Note: The method _make_weak_references(), that is used for the maps found by the coercion system, needs to remove the usual strong reference from the coercion map to the homset containing it. As long as the user keeps strong references to domain and codomain of the map, we will be able to reconstruct the homset. However, a strong reference to the coercion map does not prevent the domain from garbage collection!

EXAMPLES:

```
sage: Q = QuadraticField(-5)  # needs sage.rings.number_field
sage: phi = CDF._internal_convert_map_from(Q)  # needs sage.rings.number_field
sage: print(phi.parent())  # needs sage.rings.number_field
Set of field embeddings from Number Field in a with defining polynomial x^2 + 5
with a = 2.236067977499790?*I to Complex Double Field
```

```>>> from sage.all import *
>>> Q = QuadraticField(-Integer(5))  # needs sage.rings.number_field
>>> phi = CDF._internal_convert_map_from(Q)  # needs sage.rings.number_field
>>> print(phi.parent())  # needs sage.rings.number_field
Set of field embeddings from Number Field in a with defining polynomial x^2 + 5
with a = 2.236067977499790?*I to Complex Double Field
```

We now demonstrate that the reference to the coercion map \( \phi \) does not prevent \( Q \) from being garbage collected:

```
sage: import gc
sage: del Q  # needs sage.rings.number_field
sage: _ = gc.collect()
```

```
sage: phi.parent()  # needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: This map is in an invalid state, the domain has been garbage collected
```

3.1. Base class for maps
You can still obtain copies of the maps used by the coercion system with strong references:

```python
sage: # needs sage.rings.number_field
sage: Q = QuadraticField(-5)
sage: phi = CDF.convert_map_from(Q)
sage: print(phi.parent())
Set of field embeddings
from Number Field in a with defining polynomial x^2 + 5
with a = 2.236067977499790?*I
to Complex Double Field
sage: import gc
sage: del Q
sage: _ = gc.collect()
sage: phi.parent()  # needs sage.rings.number_field
Set of field embeddings
from Number Field in a with defining polynomial x^2 + 5
with a = 2.236067977499790?*I
to Complex Double Field
```

```python
>>> from sage.all import *
>>> import gc
>>> del Q  # needs sage.rings.number_field
>>> _ = gc.collect()
>>> phi.parent()  # needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: This map is in an invalid state,
the domain has been garbage collected
```

```python
post_compose(left)

INPUT:

- `self` - a Map in some $\text{Hom}(X, Y, \text{category} \_\text{right})$
- `left` - a Map in some $\text{Hom}(Y, Z, \text{category} \_\text{left})$

Returns the composition of `self` followed by `left` as a morphism in $\text{Hom}(X, Z, \text{category})$ where `category` is the meet of `category_left` and `category_right`.
```
Caveat: see the current restrictions on `Category.meet()`

EXAMPLES:

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
sage: Y = ZZ
sage: Z = QQ
sage: phi_xy = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])
sage: phi_yz = SetMorphism(Hom(Y, Z, Monoids()), lambda y: QQ(y**2))
sage: phi_xz = phi_xy.post_compose(phi_yz); phi_xz
Composite map:
  From: Univariate Polynomial Ring in x over Integer Ring
  To: Rational Field
  Defn: Generic morphism:
    From: Univariate Polynomial Ring in x over Integer Ring
    To: Integer Ring
    then
    Generic morphism:
    From: Integer Ring
    To: Rational Field
sage: phi_xz.category_for()
Category of monoids
```

```python
>>> from sage.all import *
>>> from sage.categories.morphism import SetMorphism
>>> X = ZZ['x']; (x,) = X._first_ngens(1)
>>> Y = ZZ
>>> Z = QQ
>>> phi_xy = SetMorphism(Hom(X, Y, Rings()), lambda p: p[Integer(0)])
>>> phi_yz = SetMorphism(Hom(Y, Z, Monoids()), lambda y: QQ(y**Integer(2)))
>>> phi_xz = phi_xy.post_compose(phi_yz); phi_xz
Composite map:
  From: Univariate Polynomial Ring in x over Integer Ring
  To: Rational Field
  Defn: Generic morphism:
    From: Univariate Polynomial Ring in x over Integer Ring
    To: Integer Ring
    then
    Generic morphism:
    From: Integer Ring
    To: Rational Field
>>> phi_xz.category_for()
Category of monoids
```

`pre_compose(right)`

**INPUT:**

- `self` — a Map in some `Hom(Y, Z, category_left)`
- `left` — a Map in some `Hom(X, Y, category_right)`

Returns the composition of `right` followed by `self` as a morphism in `Hom(X, Z, category)` where `category` is the meet of `category_left` and `category_right`.

**EXAMPLES:**

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
```

(continues on next page)
sage: Y = ZZ
sage: Z = QQ
sage: phi_xy = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])

sage: phi_yz = SetMorphism(Hom(Y, Z, Monoids()), lambda y: QQ(y**2))

sage: phi_xz = phi_yz.pre_compose(phi_xy); phi_xz

Composite map:
   From: Univariate Polynomial Ring in x over Integer Ring
      To: Rational Field
   Defn: Generic morphism:
          From: Univariate Polynomial Ring in x over Integer Ring
          To: Integer Ring
          then
          Generic morphism:
          From: Integer Ring
          To: Rational Field

sage: phi_xz.category_for()

Category of monoids

>>> from sage.all import *
>>> from sage.categories.morphism import SetMorphism
>>> X = ZZ['x']; (x,) = X._first_ngens(1)
>>> Y = ZZ
>>> Z = QQ
>>> phi_xy = SetMorphism(Hom(X, Y, Rings()), lambda p: p[Integer(0)])
>>> phi_yz = SetMorphism(Hom(Y, Z, Monoids()), lambda y: QQ(y**Integer(2)))
>>> phi_xz = phi_yz.pre_compose(phi_xy); phi_xz

Composite map:
   From: Univariate Polynomial Ring in x over Integer Ring
      To: Rational Field
   Defn: Generic morphism:
          From: Univariate Polynomial Ring in x over Integer Ring
          To: Integer Ring
          then
          Generic morphism:
          From: Integer Ring
          To: Rational Field

>>> phi_xz.category_for()

Category of monoids

section()

Return a section of self.

Note: By default, it returns None. You may override it in subclasses.

class sage.categories.map.Section

Bases: Map

A formal section of a map.

Note: Call methods are not implemented for the base class Section.

EXAMPLES:
sage: from sage.categories.map import Section
sage: R.<x,y> = ZZ[]
sage: S.<a,b> = QQ[]
sage: f = R.hom([a+b, a-b])
sage: sf = Section(f); sf
Section map:
    From: Multivariate Polynomial Ring in a, b over Rational Field
    To: Multivariate Polynomial Ring in x, y over Integer Ring
sage: sf(a)
Traceback (most recent call last):
  ... not implemented
NotImplementedError: <class 'sage.categories.map.Section'>

>>> from sage.all import *
>>> from sage.categories.map import Section
>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> S = QQ['a', 'b']; (a, b,) = S._first_ngens(2)
>>> f = R.hom([a+b, a-b])
>>> sf = Section(f); sf
Section map:
    From: Multivariate Polynomial Ring in a, b over Rational Field
    To: Multivariate Polynomial Ring in x, y over Integer Ring
>>> sf(a)
Traceback (most recent call last):
  ... not implemented
NotImplementedError: <class 'sage.categories.map.Section'>

inverse()

    Return inverse of self.

sage.categories.map.is_Map(x)

    Auxiliary function: Is the argument a map?

    EXAMPLES:

sage: R.<x,y> = QQ[]

sage: f = R.hom([x+y, x-y], R)

sage: from sage.categories.map import is_Map

sage: is_Map(f)

DeprecationWarning: The function is_Map is deprecated; use 'isinstance(..., Map)'...
←instead.
See https://github.com/sagemath/sage/issues/38103 for details.

True

sage.categories.map.unpickle_map(_class, parent, _dict, _slots)
3.2 Homsets

The class \texttt{Hom} is the base class used to represent sets of morphisms between objects of a given category. \texttt{Hom} objects are usually “weakly” cached upon creation so that they don’t have to be generated over and over but can be garbage collected together with the corresponding objects when these are not strongly referred anymore.

EXAMPLES:

In the following, the \texttt{Hom} object is indeed cached:

\begin{verbatim}
sage: K = GF(17)
sage: H = Hom(ZZ, K)
sage: H
Set of Homomorphisms from Integer Ring to Finite Field of size 17
sage: H is Hom(ZZ, K) True
\end{verbatim}

Nonetheless, garbage collection occurs when the original references are overwritten:

\begin{verbatim}
sage: from sage.all import *
>>> K = GF(Integer(17))
>>> H = Hom(ZZ, K)
>>> H
Set of Homomorphisms from Integer Ring to Finite Field of size 17
>>> H is Hom(ZZ, K) True

sage: # needs sage.libs.pari
sage: for p in prime_range(200):
    ....:     K = GF(p)
    ....:     H = Hom(ZZ, K)
sage: import gc
sage: _ = gc.collect()
sage: from sage.rings.finite_rings.finite_field_prime_modn import FiniteField_prime_modn as FF
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]
sage: len(L)
1
sage: L
[Finite Field of size 199]
\end{verbatim}

(continues on next page)
AUTHORS:

- David Kohel and William Stein
- David Joyner (2005-12-17): added examples
- Nicolas M. Thiery (2008-12-): Updated for the new category framework
- Simon King (2011-12): Use a weak cache for homsets
- Simon King (2013-02): added examples

```
sage.categories.homset.End(X, category=None)
```
Create the set of endomorphisms of \(X\) in the category category.

**INPUT:**

- \(X\) – anything
- \(\text{category}\) – (optional) category in which to coerce \(X\)

**OUTPUT:**

A set of endomorphisms in category

**EXAMPLES:**

```
sage: V = VectorSpace(QQ, 3)  # needs sage.modules
sage: End(V)  # needs sage.modules
Set of Morphisms (Linear Transformations)
from Vector space of dimension 3 over Rational Field
to Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))  # needs sage.modules
>>> End(V)  # needs sage.modules
Set of Morphisms (Linear Transformations)
from Vector space of dimension 3 over Rational Field
to Vector space of dimension 3 over Rational Field
```

```
sage: # needs sage.groups
sage: G = AlternatingGroup(3)
sage: S = End(G); S
Set of Morphisms
from Alternating group of order 3!/2 as a permutation group
to Alternating group of order 3!/2 as a permutation group
in Category of finite enumerated permutation groups
sage: S.domain()
Alternating group of order 3!/2 as a permutation group
```

3.2. Homsets
To avoid creating superfluous categories, a homset in a category $Cs()$ is in the homset category of the lowest full super category $Bs()$ of $Cs()$ that implements $Bs$.Homsets (or the join thereof if there are several). For example, finite groups form a full subcategory of unital magmas: any unital magma morphism between two finite groups is a finite group morphism. Since finite groups currently implement nothing more than unital magmas about their homsets, we have:

Similarly, a ring morphism just needs to preserve addition, multiplication, zero, and one. Accordingly, and since the category of rings implements nothing specific about its homsets, a ring homset is currently constructed in the category of homsets of unital magmas and unital additive magmas:

```
sage: H = Hom(ZZ, ZZ, Rings())
sage: H.category()
Category of endsets of unital magmas and additive unital additive magmas
```

```
sage.categories.homset.Hom (X, Y, category=None, check=True)
Create the space of homomorphisms from X to Y in the category category.

INPUT:

- X – an object of a category
```
• Y – an object of a category
• category – a category in which the morphisms must be. (default: the meet of the categories of X and Y) Both X and Y must belong to that category.
• check – a boolean (default: True): whether to check the input, and in particular that X and Y belong to category.

OUTPUT: a homset in category

EXAMPLES:

```sage
V = VectorSpace(QQ, 3)  #...
Hom(V, V)  #...

G = AlternatingGroup(3)  #...
Hom(G, G)  #...

Hom(FreeModule(ZZ, 1), FreeModule(QQ, 1))  #...
Hom(FreeModule(QQ, 1), FreeModule(ZZ, 1))  #...
```

```python
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> Hom(V, V)  # needs sage.modules
>>> G = AlternatingGroup(Integer(3))
>>> Hom(G, G)  # needs sage.groups
```

(continues on next page)
Here, we test against a memory leak that has been fixed at Issue #11521 by using a weak cache:

```sage
sage: # needs sage.libs.pari
sage: for p in prime_range(10^3):
....:     K = GF(p)
....:     a = K(0)
sage: import gc
sage: gc.collect()  # random
624
sage: from sage.rings.finite_rings.finite_field_prime_modn import FiniteField_prime_modn as FF
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]
```

To illustrate the choice of the category, we consider the following parents as running examples:

```sage
sage: X = ZZ; X
Integer Ring
sage: Y = SymmetricGroup(3); Y
Symmetric group of order 3! as a permutation group
```
Y = SymmetricGroup(Integer(3)); Y
˓→ # needs sage.groups
Symmetric group of order 3! as a permutation group

By default, the smallest category containing both X and Y, is used:

```sage```
Hom(X, Y)
˓→ needs sage.groups
Set of Morphisms from Integer Ring
to Symmetric group of order 3! as a permutation group
in Category of enumerated monoids
```

```sage```
from sage.all import *
```sage```
Hom(X, Y)
˓→ needs sage.groups
Set of Morphisms from Integer Ring
to Symmetric group of order 3! as a permutation group
in Category of enumerated monoids
```

Otherwise, if category is specified, then category is used, after checking that X and Y are indeed in category:

```sage```
Hom(X, Y, Magmas())
˓→ needs sage.groups
Set of Morphisms
from Integer Ring
to Symmetric group of order 3! as a permutation group
in Category of magmas
```

```sage```
Hom(X, Y, Groups())
˓→ needs sage.groups
Traceback (most recent call last):
...
ValueError: Integer Ring is not in Category of groups
```

```sage```
from sage.all import *
```sage```
Hom(X, Y, Magmas())
˓→ needs sage.groups
Set of Morphisms
from Integer Ring
to Symmetric group of order 3! as a permutation group
in Category of magmas

```sage```
Hom(X, Y, Groups())
˓→ needs sage.groups
Traceback (most recent call last):
...
ValueError: Integer Ring is not in Category of groups
```

A parent (or a parent class of a category) may specify how to construct certain homsets by implementing a method
_Hom_(self, codomain, category). This method should either construct the requested homset or raise
a TypeError. This hook is currently mostly used to create homsets in some specific subclass of Homset (e.g.
sage.rings.homset.RingHomset):

```sage```
Hom(QQ, QQ).__class__
<class 'sage.rings.homset.RingHomset_generic_with_category'>

3.2. Homsets
Todo:

- Design decision: how much of the homset comes from the category of \( X \) and \( Y \), and how much from the specific \( X \) and \( Y \). In particular, do we need several parent classes depending on \( X \) and \( Y \), or does the difference only lie in the elements (i.e. the morphism), and of course how the parent calls their constructors.

- Specify the protocol for the \_Hom\_ hook in case of ambiguity (e.g. if both a parent and some category thereof provide one).

\[
\text{class} \quad \text{sage.categories.homset.Homset}(X, Y, \text{category}=\text{None}, \text{base}=\text{None}, \text{check}=\text{True})
\]

\text{Bases:} \text{Set\_generic}

The class for collections of morphisms in a category.

EXAMPLES:

\[
\begin{align*}
\text{sage:} \quad & H = \text{Hom}(\text{QQ}^2, \text{QQ}^3) \quad \#\_\_\_ \\
\quad & \text{\_\_\_needs sage.modules} \\
\text{sage:} \quad & \text{loads(H.dumps()) is H} \quad \#\_\_\_ \\
\quad & \text{\_\_\_needs sage.modules} \\
\quad & \text{True}
\end{align*}
\]

\[
\begin{align*}
\text{from sage.all import *} \quad & \text{\_\_\_} \\
\text{Hom(QQ,QQ)} \quad & \text{\_\_\_class\_} \\
\text{\_\_\_class\_} \quad & \text{\_\_\_class\_} \\
\text{<class 'sage.rings.homset.RingHomset\_generic\_with\_category'>} \quad & \text{\_\_\_class\_}
\end{align*}
\]

Do not call this hook directly to create homsets, as it does not handle unique representation:

\[
\begin{align*}
\text{sage:} \quad & \text{Hom(QQ,QQ)} == \text{QQ}._\text{Hom}_(\text{QQ}, \text{category}=\text{QQ}._\text{category}()) \\
\quad & \text{True} \\
\text{sage:} \quad & \text{Hom(QQ,QQ)} \text{ is QQ}._\text{Hom}_(\text{QQ}, \text{category}=\text{QQ}._\text{category}()) \\
\quad & \text{False}
\end{align*}
\]

\[
\begin{align*}
\text{from sage.all import *} \quad & \text{\_\_\_} \\
\text{Hom(QQ,QQ)} \quad & \text{\_\_\_class\_} \\
\text{\_\_\_class\_} \quad & \text{\_\_\_class\_} \\
\text{<class 'sage.rings.homset.RingHomset\_generic\_with\_category'>} \quad & \text{\_\_\_class\_}
\end{align*}
\]

Homsets of unique parents are unique as well:

\[
\begin{align*}
\text{sage:} \quad & H = \text{End(AffineSpace(2, names='x,y'))} \\
\text{sage:} \quad & \text{loads(dumps(AffineSpace(2, names='x,y')))} \text{ is AffineSpace(2, names='x,y')} \\
\quad & \text{True} \\
\text{sage:} \quad & \text{loads(dumps(H)) is H} \\
\quad & \text{True}
\end{align*}
\]
Conversely, homsets of non-unique parents are non-unique:

```python
sage: P11 = ProductProjectiveSpaces(QQ, [1, 1])
sage: H = End(P11)
sage: loads(dumps(P11)) is ProductProjectiveSpaces(QQ, [1, 1])
False
sage: loads(dumps(P11)) == ProductProjectiveSpaces(QQ, [1, 1])
True
sage: loads(dumps(H)) is H
False
sage: loads(dumps(H)) == H
True
```

```python
>> from sage.all import *
>> P11 = ProductProjectiveSpaces(QQ, [Integer(1), Integer(1)])
>> H = End(P11)
>> loads(dumps(P11)) is ProductProjectiveSpaces(QQ, [Integer(1), Integer(1)])
False
>> loads(dumps(P11)) == ProductProjectiveSpaces(QQ, [Integer(1), Integer(1)])
True
>> loads(dumps(H)) is H
False
>> loads(dumps(H)) == H
True
```

**codomain()**

Return the codomain of this homset.

**EXAMPLES:**

```python
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().codomain()
Univariate Polynomial Ring in t over Rational Field
sage: f.codomain() is f.parent().codomain()
True
```

```python
>> from sage.all import *
>> P = ZZ['t']; (t,) = P._first_ngens(1)
>> f = P.hom([Integer(1)/Integer(2)*t])
>> f.parent().codomain()
Univariate Polynomial Ring in t over Rational Field
>> f.codomain() is f.parent().codomain()
True
```

domain()

Return the domain of this homset.

**EXAMPLES:**
Sage: P.<t> = ZZ[]
Sage: f = P.hom([1/2*t])
Sage: f.parent().domain()
Univariate Polynomial Ring in t over Integer Ring
Sage: f.domain() is f.parent().domain()
True

>>> from sage.all import *
>>> P = ZZ['t']; (t,) = P._first_ngens(1)
>>> f = P.hom([Integer(1)/Integer(2)*t])
>>> f.parent().domain()
Univariate Polynomial Ring in t over Integer Ring
>>> f.domain() is f.parent().domain()
True

element_class_set_morphism()
A base class for elements of this homset which are also SetMorphism, i.e., implemented by mean of a Python function.
This is currently plain SetMorphism, without inheritance from categories.

Todo: Refactor during the upcoming homset cleanup.

EXAMPLES:

Sage: H = Hom(ZZ, ZZ)
Sage: H.element_class_set_morphism
<class 'sage.categories.morphism.SetMorphism'>

>>> from sage.all import *
>>> H = Hom(ZZ, ZZ)
>>> H.element_class_set_morphism
<class 'sage.categories.morphism.SetMorphism'>

homset_category()
Return the category that this is a Hom in, i.e., this is typically the category of the domain or codomain object.

EXAMPLES:

Sage: H = Hom(AlternatingGroup(4), AlternatingGroup(7))  # needs sage.groups
Sage: H.homset_category()  # needs sage.groups
Category of finite enumerated permutation groups

>>> from sage.all import *
>>> H = Hom(AlternatingGroup(Integer(4)), AlternatingGroup(Integer(7)))  # needs sage.groups
>>> H.homset_category()  # needs sage.groups
Category of finite enumerated permutation groups

identity()
The identity map of this homset.
Note: Of course, this only exists for sets of endomorphisms.

EXAMPLES:

```
sage: H = Hom(QQ,QQ)
sage: H.identity()
Identity endomorphism of Rational Field
sage: H = Hom(ZZ,QQ)
sage: H.identity()
Traceback (most recent call last):
...
TypeError: identity map only defined for endomorphisms; try natural_map() instead

sage: H.natural_map()
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

```
>>> from sage.all import *
>>> H = Hom(QQ,QQ)
>>> H.identity()
Identity endomorphism of Rational Field
>>> H = Hom(ZZ,QQ)
>>> H.identity()
Traceback (most recent call last):
...
TypeError: identity map only defined for endomorphisms; try natural_map() instead

>>> H.natural_map()
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

natural_map()  
Return the “natural map” of this homset.

Note: By default, a formal coercion morphism is returned.

EXAMPLES:

```
sage: H = Hom(ZZ['t'],QQ['t'], CommutativeAdditiveGroups())
sage: H.natural_map()
Coercion morphism:
  From: Univariate Polynomial Ring in t over Integer Ring
  To:   Univariate Polynomial Ring in t over Rational Field
sage: H = Hom(QQ['t'], GF(3)['t'])
sage: H.natural_map()
Traceback (most recent call last):
...
TypeError: natural coercion morphism from Univariate Polynomial Ring in t over Rational Field to Univariate Polynomial Ring in t over Finite Field of size 3 not defined
```

3.2. Homsets 183
one()  
The identity map of this homset.

Note: Of course, this only exists for sets of endomorphisms.

EXAMPLES:

```
sage: K = GaussianIntegers()  
needs sage.rings.number_field
sage: End(K).one()  
needs sage.rings.number_field
Identity endomorphism of Gaussian Integers generated by I  
in Number Field in I with defining polynomial x^2 + 1 with I = 1*I
```

reversed()  
Return the corresponding homset, but with the domain and codomain reversed.

EXAMPLES:

```
sage: # needs sage.modules
sage: H = Hom(ZZ^2, ZZ^3); H  
Set of Morphisms from Ambient free module of rank 2 over  
the principal ideal domain Integer Ring to Ambient free module  
of rank 3 over the principal ideal domain Integer Ring in  
Category of finite dimensional modules with basis over (Dedekind  
domains and euclidean domains and noetherian rings  
and infinite enumerated sets and metric spaces)
sage: type(H)  
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: H.reversed()  
Set of Morphisms from Ambient free module of rank 3 over  
the principal ideal domain Integer Ring to Ambient free module  
of rank 2 over the principal ideal domain Integer Ring in
```
(continues on next page)
Category of finite dimensional modules with basis over (Dedekind domains and euclidean domains and noetherian rings and infinite enumerated sets and metric spaces)

```python
sage: type(H.reversed())
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
```

```python
>>> from sage.all import *

>>> H = Hom(ZZ**Integer(2), ZZ**Integer(3)); H
Set of Morphisms from Ambient free module of rank 2 over the principal ideal domain Integer Ring to Ambient free module of rank 3 over the principal ideal domain Integer Ring in Category of finite dimensional modules with basis over (Dedekind domains and euclidean domains and noetherian rings and infinite enumerated sets and metric spaces)
```

```python
>>> type(H)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
```

```python
>>> H.reversed()
Set of Morphisms from Ambient free module of rank 3 over the principal ideal domain Integer Ring to Ambient free module of rank 2 over the principal ideal domain Integer Ring in Category of finite dimensional modules with basis over (Dedekind domains and euclidean domains and noetherian rings and infinite enumerated sets and metric spaces)
```

```python
>>> type(H.reversed())
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
```

```python
class sage.categories.homset.HomsetWithBase(X, Y, category=None, check=True, base=None)

Bases: Homset

sage.categories.homset.end(X,f)

Return End(X) (f), where f is data that defines an element of End(X).

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: phi = end(R, [x + 1])
sage: phi
Ring endomorphism of Univariate Polynomial Ring in x over Rational Field
  Defn: x |--> x + 1
sage: phi(x^2 + 5)
x^2 + 2*x + 6
```

```python
>>> from sage.all import *

>>> R = QQ['x']; (x,) = R._first_ngens(1)

```python
>>> phi = end(R, [x + Integer(1)])

```python
>>> phi
Ring endomorphism of Univariate Polynomial Ring in x over Rational Field
  Defn: x |--> x + 1
```

```python
>>> phi(x**Integer(2) + Integer(5))
x^2 + 2*x + 6
```

```
sage.categories.homset.hom(X, Y,f)

Return Hom(X, Y) (f), where f is data that defines an element of Hom(X, Y).

EXAMPLES:
```
sage: R.<x> = QQ[]
sage: phi = hom(R, QQ, [2])
sage: phi(x^2 + 3)
7

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> phi = hom(R, QQ, [Integer(2)])
>>> phi(x^Integer(2) + Integer(3))
7

sage.categories.homset.is_Endset(x)

Return True if \( x \) is a set of endomorphisms in a category.

EXAMPLES:

sage: from sage.categories.homset import is_Endset
sage: P.<t> = ZZ[]

sage: f = P.hom([1/2*t])

sage: is_Endset(f)
False

sage: g = P.hom([2*t])

sage: is_Endset(g)
True

sage.categories.homset.is_Homset(x)

Return True if \( x \) is a set of homomorphisms in a category.

EXAMPLES:

sage: from sage.categories.homset import is_Homset
sage: P.<t> = ZZ[]

sage: f = P.hom([1/2*t])

sage: is_Homset(f)
False

(continues on next page)
3.3 Morphisms

This module defines the base classes of morphisms between objects of a given category.

EXAMPLES:

Typically, a morphism is defined by the images of the generators of the domain.

\[
\begin{align*}
\text{sage: } & X.<a, b> = ZZ[] \\
\text{sage: } & Y.<c> = ZZ[] \\
\text{sage: } & X.hom([c, c^2])
\end{align*}
\]

Ring morphism:
From: Multivariate Polynomial Ring in a, b over Integer Ring
To: Univariate Polynomial Ring in c over Integer Ring
Defn: a |--> c
\hspace{1cm} b |--> c^2

\[
\begin{align*}
\text{AUTHORS:} & \\
\text{• William Stein (2005): initial version} \\
\text{• David Joyner (2005-12-17): added examples} \\
\text{• Robert Bradshaw (2007-06-25): Pyrexification}
\end{align*}
\]
class sage.categories.morphism.CallMorphism
    Bases: Morphism

class sage.categories.morphism.FormalCoercionMorphism
    Bases: Morphism

class sage.categories.morphism.IdentityMorphism
    Bases: Morphism

    is_identity()
    Return True if this morphism is the identity morphism.

    EXAMPLES:

    sage: E = End(Partitions(5))
    # needs sage.combinat
    sage: E.identity().is_identity()
    # needs sage.combinat
    True

    >>> from sage.all import *
    >>> E = End(Partitions(Integer(5)))
    # needs sage.combinat
    >>> E.identity().is_identity()
    needs sage.combinat
    True

    Check that Issue #15478 is fixed:

    sage: # needs sage.rings.finite_rings
    sage: K.<z> = GF(4)
    sage: phi = End(K)([z^2])
    sage: R.<t> = K[]
    sage: psi = End(R)(phi)
    sage: psi.is_identity()
    False

    >>> from sage.all import *
    >>> # needs sage.rings.finite_rings
    >>> K = GF(Integer(4), names=(z,)); (z,) = K._first_ngens(1)
    >>> phi = End(K)([z**Integer(2)])
    >>> R = K['t']; (t,) = R._first_ngens(1)
    >>> psi = End(R)(phi)
    >>> psi.is_identity()
    False

    is_injective()
    Return whether this morphism is injective.

    EXAMPLES:

    sage: Hom(ZZ, ZZ).identity().is_injective()
    True

    >>> from sage.all import *
    >>> Hom(ZZ, ZZ).identity().is_injective()
    True
**is_surjective()**

Return whether this morphism is surjective.

**EXAMPLES:**

```python
sage: Hom(ZZ, ZZ).identity().is_surjective()
True

>>> from sage.all import *

>>> Hom(ZZ, ZZ).identity().is_surjective()
True
```

**section()**

Return a section of this morphism.

**EXAMPLES:**

```python
sage: T = Hom(ZZ, ZZ).identity()
sage: T.section() is T
True

>>> from sage.all import *

>>> T = Hom(ZZ, ZZ).identity()

>>> T.section() is T
True
```

**class** `sage.categories.morphism.Morphism`

**Bases:** `Map`

**category()**

Return the category of the parent of this morphism.

**EXAMPLES:**

```python
sage: R.<t> = ZZ[]
sage: f = R.hom([t**2])
sage: f.category()
Category of endsets of unital magmas and right modules over
   (Dedekind domains and euclidean domains
   and noetherian rings
   and infinite enumerated sets and metric spaces)
   and left modules over
   (Dedekind domains and euclidean domains
   and noetherian rings
   and infinite enumerated sets and metric spaces)

sage: # needs sage.rings.number_field
sage: K = CyclotomicField(12)
sage: L = CyclotomicField(132)
sage: phi = L._internal_coerce_map_from(K)
sage: phi.category()
Category of homsets of number fields

>>> from sage.all import *

>>> R = ZZ['t']; (t,) = R._first_ngens(1)

>>> f = R.hom([t**Integer(2)])

>>> f.category()
```
is_endomorphism()
Return True if this morphism is an endomorphism.

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: f = R.hom([t])
sage: f.is_endomorphism()
True

sage: # needs sage.rings.number_field
sage: K = CyclotomicField(12)
sage: L = CyclotomicField(132)
sage: phi = L._internal_coerce_map_from(K)
sage: phi.is_endomorphism()
False
```

is_identity()
Return True if this morphism is the identity morphism.

Note: Implemented only when the domain has a method gens()
A morphism between two different spaces cannot be the identity:

```
sage: R2.<t2> = QQ[]
sage: h = R2.hom([t2])
sage: h.is_identity()  
False
```

```
>>> from sage.all import *
>>> R2 = QQ['t2']; (t2,) = R2._first_ngens(1)
>>> h = R2.hom([t2])
>>> h.is_identity()
False
```

pushforward()  

register_as_coercion()  

Register this morphism as a coercion to Sage's coercion model (see `sage.structure.coerce`).  

EXAMPLES:  

By default, adding polynomials over different variables triggers an error:

```
sage: X.<x> = ZZ[]
sage: Y.<y> = ZZ[]
sage: x**Integer(2) + y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Univariate Polynomial Ring in x over Integer Ring' and
'Univariate Polynomial Ring in y over Integer Ring'
```

```
>>> from sage.all import *
>>> X = ZZ['x']; (x,) = X._first_ngens(1)
>>> Y = ZZ['y']; (y,) = Y._first_ngens(1)
>>> x**Integer(2) + y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Univariate Polynomial Ring in x over Integer Ring' and
'Univariate Polynomial Ring in y over Integer Ring'
```

Let us declare a coercion from \(\mathbb{Z}[x]\) to \(\mathbb{Z}[z]\):
sage: Z.<z> = ZZ[]
sage: phi = Hom(X, Z)(z)
sage: phi(x^2+1)
z^2 + 1
sage: phi.register_as_coercion()

```python
>>> from sage.all import *
>>>
Z = ZZ['z']; (z,) = Z._first_ngens(1)
>>>
phi = Hom(X, Z)(z)
>>>
phi(x**Integer(2)+Integer(1))
z^2 + 1
>>>
phi.register_as_coercion()
```

Now we can add elements from \( \mathbb{Z}[x] \) and \( \mathbb{Z}[z] \), because the elements of the former are allowed to be implicitly coerced into the later:

sage: x^2 + z
z^2 + z

```python
>>> from sage.all import *
>>>
x**Integer(2) + z
z^2 + z
```

Caveat: the registration of the coercion must be done before any other coercion is registered or discovered:

```python
sage: phi = Hom(X, Z)(z^2)
sage: phi.register_as_coercion()
Traceback (most recent call last):
  ...  
AssertionError: coercion from Univariate Polynomial Ring in x over Integer Ring to Univariate Polynomial Ring in z over Integer Ring already registered or discovered
```

```python
>>> from sage.all import *
>>>
phi = Hom(X, Z)(z**Integer(2))
>>>
phi.register_as_coercion()
Traceback (most recent call last):
  ...  
AssertionError: coercion from Univariate Polynomial Ring in x over Integer Ring to Univariate Polynomial Ring in z over Integer Ring already registered or discovered
```

**register_as_conversion()**

Register this morphism as a conversion to Sage’s coercion model

(see `sage.structure.coerce`).

**EXAMPLES:**

Let us declare a conversion from the symmetric group to \( \mathbb{Z} \) through the sign map:

```python
sage: # needs sage.groups
sage: S = SymmetricGroup(4)
sage: phi = Hom(S, ZZ)(lambda x: ZZ(x.sign()))
sage: x = S.an_element(); x
```

(continues on next page)
sage: phi(x)
1
sage: phi.register_as_conversion()

sage: ZZ(x)
1

>>> from sage.all import *
>>> # needs sage.groups
>>> S = SymmetricGroup(Integer(4))
>>> phi = Hom(S, ZZ)(lambda x: ZZ(x.sign()))
>>> x = S.an_element(); x
(2,3,4)
>>> phi(x)
1
>>> phi.register_as_conversion()
>>> ZZ(x)
1

== class sage.categories.morphism.SetIsomorphism ==

Bases: SetMorphism

An isomorphism of sets.

INPUT:

- `parent` -- a Homset

- `function` -- a Python function that takes elements of the domain as input and returns elements of the codomain.

EXAMPLES:

\[
\text{sage: } f = \text{sage.categories.morphism.SetIsomorphism}(\text{Hom(ZZ, ZZ, Sets())}, \text{operator.__neg__}); f
\]

Generic endomorphism of Integer Ring
\[
\text{sage: } f._set_inverse(f)
\]

\[
\text{sage: } \neg f \text{ is } f
\]

True

\[
\text{>>> from sage.all import *}
\text{>>> } f = \text{sage.categories.morphism.SetIsomorphism}(\text{Hom(ZZ, ZZ, Sets())}, \text{operator.__neg__}); f
\]

Generic endomorphism of Integer Ring
\[
\text{>>> } \neg f \text{ is } f
\]

True

\[
is\_injective()
\]

Return whether this morphism is injective.

EXAMPLES:

\[
\text{sage: } f = \text{sage.categories.morphism.SetIsomorphism}(\text{Hom(ZZ, ZZ, Sets())}, \text{operator.__neg__})
\]

\[
\text{sage: } f.is\_injective()
\]

True

3.3. Morphisms
>>> from sage.all import *
>>> f = sage.categories.morphism.SetIsomorphism(Hom(ZZ, ZZ, Sets()),
       operator.__neg__)
>>> f.is_injective()
True

\textbf{is_surjective}()

Return whether this morphism is surjective.

\textbf{EXAMPLES}:

sage: f = sage.categories.morphism.SetIsomorphism(Hom(ZZ, ZZ, Sets()),
       operator.__neg__)
sage: f.is_surjective()
True

\textbf{section}()

Return a section of this morphism.

\textbf{EXAMPLES}:

sage: f = sage.categories.morphism.SetIsomorphism(Hom(ZZ, ZZ, Sets()),
       operator.__neg__)
sage: f._set_inverse(f)
sage: f.section() \text{ is } f
True

\textbf{class} sage.categories.morphism.SetMorphism

\textbf{Bases:} Morphism

\textbf{INPUT}:

\begin{itemize}
  \item parent – a Homset
  \item function – a Python function that takes elements of the domain as input and returns elements of the codomain.
\end{itemize}

\textbf{EXAMPLES}:

sage: from sage.categories.morphism import SetMorphism
sage: f = SetMorphism(Hom(QQ, ZZ, Sets()), numerator)
sage: f.parent()
Set of Morphisms from Rational Field to Integer Ring in Category of sets
sage: f.domain()
Rational Field

```python
sage: f.codomain()
Integer Ring

sage: TestSuite(f).run()
```

```python
>>> from sage.all import *
>>> from sage.categories.morphism import SetMorphism
>>> f = SetMorphism(Hom(QQ, ZZ, Sets()), numerator)
>>> f.parent()
Set of Morphisms from Rational Field to Integer Ring in Category of sets
>>> f.domain()
Rational Field
>>> f.codomain()
Integer Ring
>>> TestSuite(f).run()
```

```python
sage.categories.morphism.is_Morphism(x)
```

### 3.4 Coercion via construction functors

**class** sage.categories.pushout.AlgebraicClosureFunctor

**Bases:** ConstructionFunctor

Algebraic Closure.

**EXAMPLES:**

```python
sage: # needs sage.rings.complex_double sage.rings.number_field
sage: F = CDF.construction()[0]
sage: F(QQ)
Algebraic Field

sage: F(RR)  # needs sage.rings.real_mpfr
Complex Field with 53 bits of precision

sage: F(F(QQ)) is F(QQ)
True
```

```python
>>> from sage.all import *

>>> # needs sage.rings.complex_double sage.rings.number_field
>>> F = CDF.construction()[Integer(0)]

>>> F(QQ)
Algebraic Field

>>> F(RR)  # needs sage.rings.real_mpfr
Complex Field with 53 bits of precision

>>> F(F(QQ)) is F(QQ)
True
```

**merge**(other)

Mathematically, Algebraic Closure subsumes Algebraic Extension. However, it seems that people do want to work with algebraic extensions of RR. Therefore, we do not merge with algebraic extension.

**rank** = 3
class sage.categories.pushout.AlgebraicExtensionFunctor(polys, names, embeddings=None, structures=None, cyclotomic=None, prec=None, implementations=None, *, residue=None, latex_names=None, **kwds)

Bases: ConstructionFunctor

Algebraic extension (univariate polynomial ring modulo principal ideal).

EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x^3 + x^2 + 1)  # needs sage.rings.number_field
sage: F = K.construction()[0]  # needs sage.rings.number_field
sage: F(ZZ['t'])
Univariate Quotient Polynomial Ring in a
over Univariate Polynomial Ring in t over Integer Ring
with modulus a^3 + a^2 + 1
```

Note that, even if a field is algebraically closed, the algebraic extension will be constructed as the quotient of a univariate polynomial ring:

```
sage: F(CC)  # needs sage.rings.number_field
Univariate Quotient Polynomial Ring in a
over Complex Field with 53 bits of precision
with modulus a^3 + a^2 + 1.00000000000000
```

(continues on next page)
Univariate Quotient Polynomial Ring in a
over Real Field with 53 bits of precision
with modulus a^3 + a^2 + 1.00000000000000

Note that the construction functor of a number field applied to the integers returns an order (not necessarily maximal) of that field, similar to the behaviour of \texttt{ZZ.extension(...)}:

```
sage: F(ZZ)  # needs sage.rings.number_field
Order generated by a in Number Field in a with defining polynomial x^3 + x^2 + 1
```

```
>>> from sage.all import *
>>> F(ZZ)  # needs sage.rings.number_field
Order generated by a in Number Field in a with defining polynomial x^3 + x^2 + 1
```

This also holds for non-absolute number fields:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: K.<a,b> = NumberField([x^3 + x^2 + 1, x^2 + x + 1])
sage: F = K.construction()[0]
sage: O = F(ZZ); O
Relative Order
  generated by [(b - 2)*a^2 + (3*b - 1)*a + 3*b + 4, a - b]
  in Number Field in a with defining polynomial x^3 + x^2 + 1
  over its base field
sage: O.ambient() is K
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(QQ, 'x')
>>> K = NumberField([x**Integer(3) + x**Integer(2) + Integer(1), x**Integer(2) + ...]
  + x + Integer(1)], names=('a', 'b',)); (a, b,) = K._first_ngens(2)
>>> F = K.construction()[Integer(0)]
>>> O = F(ZZ); O
Relative Order
  generated by [(b - 2)*a^2 + (3*b - 1)*a + 3*b + 4, a - b]
  in Number Field in a with defining polynomial x^3 + x^2 + 1
  over its base field
>>> O.ambient() is K
True
```

Special cases are made for cyclotomic fields and residue fields:

```
sage: # needs sage.rings.number_field
sage: C = CyclotomicField(8)
sage: F, R = C.construction()
sage: F
AlgebraicExtensionFunctor
sage: R
Rational Field
sage: F(R)
Cyclotomic Field of order 8 and degree 4
```

(continues on next page)
Decompose the functor $F$ into sub-functors, whose product returns $F$.

**EXAMPLES:**

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K.<z> = CyclotomicField(7)
>>> P = K.factor(17)[0][0]
>>> k = K.residue_field(P)
>>> F, R = k.construction()
>>> F
AlgebraicExtensionFunctor
>>> R
Cyclotomic Field of order 7 and degree 6
>>> F(R) is k
True
>>> F(ZZ)
Residue field of Integers modulo 17
>>> F(CyclotomicField(Integer(49)))
Residue field in zbar of Fractional ideal (17)
```
merge \(\text{(other)}\)

Merging with another \texttt{AlgebraicExtensionFunctor}.

**INPUT:**

- \texttt{other} – Construction Functor.

**OUTPUT:**

- If \texttt{self}==\texttt{other}, \texttt{self} is returned.
- If \texttt{self} and \texttt{other} are simple extensions and both provide an embedding, then it is tested whether one of the number fields provided by the functors coerces into the other; the functor associated with the target of the coercion is returned. Otherwise, the construction functor associated with the pushout of the codomains of the two embeddings is returned, provided that it is a number field.
- If these two extensions are defined by Conway polynomials over finite fields, merges them into a single extension of degree the lcm of the two degrees.
- Otherwise, None is returned.

**REMARK:**

Algebraic extension with embeddings currently only works when applied to the rational field. This is why we use the admittedly strange rule above for merging.

**EXAMPLES:**

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> K = NumberField(x**Integer(3) - Integer(5), embedding=Integer(0), names=(
 \texttt{a}',)); (a,) = K._first_ngens(1)
>>> L = K.extension(x**Integer(2) + a, names=('b',)); (b,) = L._first_ngens(1)
>>> F, R = L.construction()
>>> prod(F.expand())(R) == L
True
>>> K = NumberField([x**Integer(2) - Integer(2), x**Integer(2) - Integer(3)],
 \texttt{a})
>>> F, R = K.construction()
>>> F
\texttt{AlgebraicExtensionFunctor}
>>> L = F.expand(); L
[\texttt{AlgebraicExtensionFunctor}, \texttt{AlgebraicExtensionFunctor}]
>>> L[-Integer(1)](QQ)
Number Field in a1 with defining polynomial x^2 - 3
```
The following demonstrate coercions for finite fields using Conway or pseudo-Conway polynomials:

```python
sage: k = GF(3^2, prefix='z'); a = k.gen()  # needs sage.rings.finite_rings
sage: l = GF(3^3, prefix='z'); b = l.gen()  # needs sage.rings.finite_rings
sage: a + b  # indirect doctest  # needs sage.rings.finite_rings
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
```

Note that embeddings are compatible in lattices of such finite fields:

```python
sage: # needs sage.rings.finite_rings
sage: m = GF(3^5, prefix='z'); c = m.gen()
sage: (a + b) + c == a + (b + c)  # indirect doctest
True
```

Coercion is also available for number fields:

```python
sage: # needs sage.rings.number_field
sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^8 - x^4 + 1, embedding=CDF.0)
sage: M1.<c1> = NumberField(x^2 + x + 1, embedding=b^4 - 1)
sage: M2.<c2> = NumberField(x^2 + 1, embedding=-b^6)
sage: M1.coerce_map_from(M2)
sage: M2.coerce_map_from(M1)
sage: c1 + c2; parent(c1 + c2)  # indirect doctest
-b^6 + b^4 - 1
```

(continues on next page)
sage: pushout(M1['x'], M2['x'])  #...  
Univariate Polynomial Ring in x  
over Number Field in b with defining polynomial x^8 - x^4 + 1  
with b = -0.2588190451025208? + 0.9659258262890683?*I

>>> from sage.all import *  
>>> # needs sage.rings.number_field  
>>> P = QQ['x']; (x,) = P._first_ngens(1)  
>>> L = NumberField(x**Integer(8) - x**Integer(4) + Integer(1), embedding=CDF.˓→gen(0), names=('b',)); (b,) = L._first_ngens(1)  
>>> M1 = NumberField(x**Integer(2) + x + Integer(1), embedding=b**Integer(4) - ˓→Integer(1), names=('c1',)); (c1,) = M1._first_ngens(1)  
>>> M2 = NumberField(x**Integer(2) + Integer(1), embedding=-b**Integer(6),˓→names=('c2',)); (c2,) = M2._first_ngens(1)  
>>> M1.coerce_map_from(M2)  
>>> M2.coerce_map_from(M1)  
>>> c1 + c2; parent(c1 + c2)  
# indirect doctest  
-b^6 + b^4 - 1  
Number Field in b with defining polynomial x^8 - x^4 + 1  
with b = -0.2588190451025208? + 0.9659258262890683?*I

In the previous example, the number field L becomes the pushout of M1 and M2 since both are provided with an embedding into L, and since L is a number field. If two number fields are embedded into a field that is not a numberfield, no merging occurs:

sage: # needs sage.rings.complex_double sage.rings.number_field  
sage: cbrt2 = CDF(Integer(2))**(Integer(1)/Integer(3))  
sage: zeta3 = CDF.zeta(Integer(3))  
sage: K.<a> = NumberField(x**Integer(3) - Integer(2), embedding=cbrt2 * zeta3)  
sage: L.<b> = NumberField(x**Integer(6) - Integer(2), embedding=RealNumber('1.1'))  
sage: L.coerce_map_from(K)  
sage: K.coerce_map_from(L)  
sage: pushout(K, L)  
#...  
Traceback (most recent call last):  
  ...  
CoercionException: ('Ambiguous Base Extension', Number Field in a with  
defining polynomial x^3 - 2 with a = -0.6299605249474365? + 1.091123635971722?*I,  
Number Field in b with defining polynomial x^6 - 2 with b = 1.122462048309373?)

>>> from sage.all import *  
>>> # needs sage.rings.complex_double sage.rings.number_field  
>>> cbrt2 = CDF(Integer(2))**(Integer(1)/Integer(3))  
>>> zeta3 = CDF.zeta(Integer(3))  
>>> K.<a> = NumberField(x**Integer(3) - Integer(2), embedding=cbrt2 * zeta3,˓→names=('a',)); (a,) = K._first_ngens(1)  
>>> L.<b> = NumberField(x**Integer(6) - Integer(2), embedding=RealNumber('1.1'),˓→names=('b',)); (b,) = L._first_ngens(1)
rank = 3

class sage.categories.pushout.BlackBoxConstructionFunctor(box)

Bases: ConstructionFunctor

Construction functor obtained from any callable object.

EXAMPLES:

sage: from sage.categories.pushout import BlackBoxConstructionFunctor

sage: # needs sage.libs.gap

sage: from sage.interfaces.gap import gap

sage: FG = BlackBoxConstructionFunctor(gap)

sage: FG

BlackBoxConstructionFunctor

sage: FG(ZZ)

Integers

sage: FG(ZZ).parent()

Gap

sage: FG == loads(dumps(FG))

True

sage: FS = BlackBoxConstructionFunctor(singular)

sage: FS(QQ['t'])

# needs sage.libs.singular

polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 1
// 1: ordering lp
// names     t
// 2: ordering C

sage: FG == FS

# needs sage.libs.gap sage.libs.singular

False
Integers

```python
>>> FG(ZZ).parent()
Gap
>>> FG == loads(dumps(FG))
True
```

```python
>>> FS = BlackBoxConstructionFunctor(singular)
... FS(QQ['t'])
... #... needs sage.libs.singular
```

```python
... polynomial ring, over a field, global ordering
... // coefficients: QQ
... // number of vars : 1
... //   block 1 : ordering lp
... //     : names t
... //   block 2 : ordering C
... FG == FS
... #... needs sage.libs.gap sage.libs.singular
False
```

```python
rank = 100
```

**class** sage.categories.pushout.CompletionFunctor *(p, prec, extras=None)*

**Bases:** ConstructionFunctor

Completion of a ring with respect to a given prime (including infinity).

**EXAMPLES:**

```python
sage: # needs sage.rings.padics
sage: R = Zp(5)
sage: R
5-adic Ring with capped relative precision 20
sage: F1 = R.construction()[0]
sage: F1
Completion[5, prec=20]
sage: F1(ZZ)
is R
True
sage: F1(QQ)
5-adic Field with capped relative precision 20
sage: F2 = RR.construction()[0]
sage: F2
Completion[+Infinity, prec=53]
sage: F2(QQ)
is RR
True
sage: P.<x> = ZZ[]
sage: Px = P.completion(x) # currently the only implemented completion of P
sage: Px
Power Series Ring in x over Integer Ring
sage: F3 = Px.construction()[0]
sage: F3(GF(3)['x'])
Power Series Ring in x over Finite Field of size 3
```

```python
>>> from sage.all import *
... # needs sage.rings.padics
>>> R = Zp(Integer(5))
```

---

3.4. Coercion via construction functors
>>> R
5-adic Ring with capped relative precision 20
>>> F1 = R.construction()[Integer(0)]
>>> F1
Completion[5, prec=20]
>>> F1(ZZ) is R
True
>>> F1(QQ)
5-adic Field with capped relative precision 20
>>> F2 = RR.construction()[Integer(0)]
>>> F2
Completion[+Infinity, prec=53]
>>> F2(QQ) is RR
True

>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> Px = P.completion(x)  # currently the only implemented completion of P
>>> Px
Power Series Ring in x over Integer Ring
>>> F3 = Px.construction()[Integer(0)]
>>> F3(GF(Integer(3))['x'])
Power Series Ring in x over Finite Field of size 3

**commutes** *(other)*

Completion commutes with fraction fields.

**EXAMPLES:**

```python
sage: F1 = Zp(5).construction()[0]  # needs sage.rings.padics
sage: F2 = QQ.construction()[0]
sage: F1.commutes(F2)  # needs sage.rings.padics
True
```

```python
>>> from sage.all import *
>>> F1 = Zp(Integer(5)).construction()[Integer(0)]  # needs sage.rings.padics
>>> F2 = QQ.construction()[Integer(0)]
>>> F1.commutes(F2)  # needs sage.rings.padics
True
```

**merge** *(other)*

Two Completion functors are merged, if they are equal. If the precisions of both functors coincide, then a Completion functor is returned that results from updating the *extras* dictionary of *self* by *other* *extras*. Otherwise, if the completion is at infinity then merging does not increase the set precision, and if the completion is at a finite prime, merging does not decrease the capped precision.

**EXAMPLES:**

```python
sage: # needs sage.rings.padics
sage: R1.<a> = Zp(5, prec=20)[]
sage: R2 = Qp(5, prec=40)
sage: R2(1) + a  # indirect doctest
```

(continues on next page)
(1 + O(5^20))*a + 1 + O(5^40)
sage: R3 = RealField(30)
sage: R4 = RealField(50)
sage: R3(1) + R4(1)  # indirect doctest
2.0000000
sage: (R3(1) + R4(1)).parent()
Real Field with 30 bits of precision

>>> from sage.all import *
>>> # needs sage.rings.padics
>>> R1 = Zp(Integer(5), prec=Integer(20))[a]; (a,) = R1._first_ngens(1)
>>> R2 = Qp(Integer(5), prec=Integer(40))
>>> R2(Integer(1)) + a  # indirect doctest
(1 + O(5^20))*a + 1 + O(5^40)
>>> R3 = RealField(Integer(30))
>>> R4 = RealField(Integer(50))
>>> R3(Integer(1)) + R4(Integer(1))  # indirect doctest
2.0000000
>>> (R3(Integer(1)) + R4(Integer(1))).parent()
Real Field with 30 bits of precision

rank = 4

class sage.categories.pushout.CompositeConstructionFunctor(*args)

Bases: ConstructionFunctor

A Construction Functor composed by other Construction Functors.

INPUT:

F1, F2,...: A list of Construction Functors. The result is the composition F1 followed by F2 followed by ...

EXAMPLES:

sage: from sage.categories.pushout import CompositeConstructionFunctor
sage: F = CompositeConstructionFunctor(QQ.construction()[0], ZZ['x'].˓→construction()[0], QQ.construction()[0], ZZ['y'].˓→construction()[0])

sage: F == loads(dumps(F))
True
sage: F == CompositeConstructionFunctor(*F.all)
True
sage: F(GF(2)['t'])  # needs sage.libsntl
Univariate Polynomial Ring in y
over Fraction Field of Univariate Polynomial Ring in x
over Fraction Field of Univariate Polynomial Ring in t
over Finite Field of size 2 (using GF2X)

>>> from sage.all import *
>>> from sage.categories.pushout import CompositeConstructionFunctor
>>> F = CompositeConstructionFunctor(QQ.construction()[Integer(0)], ZZ['x'].˓→construction()[Integer(0)], QQ.construction()[Integer(0)], ZZ['y'].˓→construction()[Integer(0)], ZZ['y'].˓→needs sage.libsntl
Univariate Polynomial Ring in y
over Fraction Field of Univariate Polynomial Ring in x
over Fraction Field of Univariate Polynomial Ring in t
over Finite Field of size 2 (using GF2X)

(continues on next page)
expand()

Return expansion of a CompositeConstructionFunctor.

**Note:** The product over the list of components, as returned by the expand() method, is equal to self.

**EXAMPLES:**

```sage
def from sage.categories.pushout import CompositeConstructionFunctor
def F = CompositeConstructionFunctor(QQ.construction()[0],
    ....: ZZ['x'].construction()[0],
    ....: QQ.construction()[0],
    ....: ZZ['y'].construction()[0])
def F
Poly[y](FractionField(Poly[x](FractionField(...))))
def prod(F.expand()) == F
True
```

```sage
from sage.all import *
from sage.categories.pushout import CompositeConstructionFunctor
F = CompositeConstructionFunctor(QQ.construction()[Integer(0)],
    ... Z['x'].construction()[Integer(0)],
    ... ZZ['y'].construction()[Integer(0)])
F
Poly[y]((FractionField(Poly[x](FractionField(...))))
prod(F.expand()) == F
True
```

class sage.categories.pushout.ConstructionFunctor

**Bases:** `Functor`

Base class for construction functors.

A construction functor is a functorial algebraic construction, such as the construction of a matrix ring over a given ring or the fraction field of a given ring.

In addition to the class `Functor`, construction functors provide rules for combining and merging constructions. This is an important part of Sage's coercion model, namely the pushout of two constructions: When a polynomial \( p \) in a variable \( x \) with integer coefficients is added to a rational number \( q \), then Sage finds that the parents \( \ZZ['x'] \) and \( \QQ \) are obtained from \( \ZZ \) by applying a polynomial ring construction respectively the fraction field construction. Each construction functor has an attribute `rank`, and the rank of the polynomial ring construction is higher than
the rank of the fraction field construction. This means that the pushout of \( \mathbb{Q} \) and \( \mathbb{Z}[x] \), and thus a common
parent in which \( p \) and \( q \) can be added, is \( \mathbb{Q}[x] \), since the construction functor with a lower rank is applied
first.

```
sage: F1, R = QQ.construction()
sage: F1
FractionField
sage: R
Integer Ring
sage: F2, R = (ZZ['x']).construction()
sage: F2
Poly[x]
sage: R
Integer Ring
sage: F3 = F2.pushout(F1)
sage: F3
Poly[x](FractionField(...))
sage: F3(R)
Univariate Polynomial Ring in x over Rational Field
```

When composing two construction functors, they are sometimes merged into one, as is the case in the Quotient
construction:

```
sage: Q15, R = (ZZ.quo(15*ZZ)).construction()
sage: Q15
QuotientFunctor
sage: Q35, R = (ZZ.quo(35*ZZ)).construction()
sage: Q35
```

(continues on next page)
Functors can not only be applied to objects, but also to morphisms in the respective categories. For example:

```python
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]
sage: f = A.hom([a + b, a - b], A)
sage: F(A)
Multivariate Polynomial Ring in x, y
   over Multivariate Polynomial Ring in a, b over Finite Field of size 5
sage: F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
   over Multivariate Polynomial Ring in a, b over Finite Field of size 5
   Defn: Induced from base ring by
       Ring endomorphism of Multivariate Polynomial Ring in a, b
           over Finite Field of size 5
           Defn: a |--> a + b
                b |--> a - b
sage: F(f)(F(A)(x)*a)
(a + b)*x
```

```python
>>> from sage.all import *
>>> P = ZZ['x', 'y']; (x, y,) = P._first_ngens(2)
>>> F = P.construction()[Integer(0)]; F
MPoly[x,y]
>>> A = GF(Integer(5)]['a', 'b']; (a, b,) = A._first_ngens(2)
>>> f = A.hom([a + b, a - b], A)
>>> F(A)
Multivariate Polynomial Ring in x, y
   over Multivariate Polynomial Ring in a, b over Finite Field of size 5
>>> F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y
   over Multivariate Polynomial Ring in a, b over Finite Field of size 5
   Defn: Induced from base ring by
       Ring endomorphism of Multivariate Polynomial Ring in a, b
           over Finite Field of size 5
           Defn: a |--> a + b
                b |--> a - b
```

(continues on next page)
coercion_reversed = False

common_base (other_functor, self_bases, other_bases)
This function is called by pushout() when no common parent is found in the construction tower.

**Note:** The main use is for multivariate construction functors, which use this function to implement recursion for pushout().

**INPUT:**
- other_functor – a construction functor.
- self_bases – the arguments passed to this functor.
- other_bases – the arguments passed to the functor other_functor.

**OUTPUT:**
Nothing, since a CoercionException is raised.

**Note:** Overload this function in derived class, see e.e. MultivariateConstructionFunctor.

commutes (other)
Determine whether self commutes with another construction functor.

**Note:** By default, False is returned in all cases (even if the two functors are the same, since in this case merge() will apply anyway). So far there is no construction functor that overloads this method. Anyway, this method only becomes relevant if two construction functors have the same rank.

**EXAMPLES:**
```python
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.commutes(P)
False
sage: P.commutes(F)
False
sage: F.commutes(F)
False
```

```python
>>> from sage.all import *
>>> F = QQ.construction()[Integer(0)]
>>> P = ZZ['t'].construction()[Integer(0)]
>>> F.commutes(P)
False
>>> P.commutes(F)
False
>>> F.commutes(F)
False
```
**expand()**

Decompose `self` into a list of construction functors.

**Note:** The default is to return the list only containing `self`.

**EXAMPLES:**

```sage
define F = QQ.construction()[0]
define P = ZZ['t'].construction()[0]
F.expand()
P.expand()
```

```python
define F = QQ.construction()[Integer(0)]
define P = ZZ['t'].construction()[Integer(0)]
F.merge(P)
P.merge(F)
```

---

**merge(other)**

Merge `self` with another construction functor, or return `None`.

**Note:** The default is to merge only if the two functors coincide. But this may be overloaded for subclasses, such as the quotient functor.

**EXAMPLES:**

```sage
define F = QQ.construction()[0]
define P = ZZ['t'].construction()[0]
F.merge(F)
P.merge(P)
```

```python
define F = QQ.construction()[Integer(0)]
define P = ZZ['t'].construction()[Integer(0)]
F.merge(F)
P.merge(P)
```

(continues on next page)
pushout (other)
Composition of two construction functors, ordered by their ranks.

Note:

• This method seems not to be used in the coercion model.
• By default, the functor with smaller rank is applied first.

class sage.categories.pushout.EquivariantSubobjectConstructionFunctor(S, action=<built-in function mul>, side='left', other_action=None, other_side='left')

Bases: ConstructionFunctor

Constructor for subobjects invariant or equivariant under given semigroup actions.

Let $S$ be a semigroup that - acts on a parent $X$ as $s \cdot x$ (action, side='left') or - acts on $X$ as $x \cdot s$ (action, side='right'), and (possibly trivially) - acts on $X$ as $s \ast x$ (other_action, other_side='left') or - acts on $X$ as $x \ast s$ (other_action, other_side='right').

The $S$-equivariant subobject is the subobject

$$X^S := \{ x \in X : s \cdot x = s \ast x, \forall s \in S \}$$

when side = other_side = 'left' and mutatis mutandis for the other values of side and other_side.

When other_action is trivial, $X^S$ is called the $S$-invariant subobject.

EXAMPLES:

Monoterm symmetries of a tensor, here only for matrices: row (index 0), column (index 1); the order of the extra element 2 in a permutation determines whether it is a symmetry or an antisymmetry:
.....:            return -x.transpose()
.....:        else:
.....:            return x
.....:        raise NotImplementedError
sage: M = matrix([[1, 2], [3, 4]]); M
[1 2]
[3 4]
sage: GSym01_action = TensorIndexAction(GSym01, M.parent())
sage: GASym01_action = TensorIndexAction(GASym01, M.parent())
sage: GSym01_action.act(GSym01.0, M)
[1 3]
[2 4]
sage: GASym01_action.act(GASym01.0, M)
[-1 -3]
[-2 -4]
sage: Sym01 = M.parent().invariant_module(GSym01, action=GSym01_action); Sym01
(Permutation Group with generators [(0,1)])-invariant submodule
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: list(Sym01.basis())
[B[0], B[1], B[2]]
sage: list(Sym01.basis().map(Sym01.lift))
[
[ 1 0]
[0 1]
]
sage: ASym01 = M.parent().invariant_module(GASym01, action=GASym01_action)
sage: ASym01
(Permutation Group with generators [(0,1)(2,3)])-invariant submodule
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: list(ASym01.basis())
[B[0]]
sage: list(ASym01.basis().map(ASym01.lift))
[
[ 0 1]
[-1 0]
]
sage: from sage.categories.pushout import pushout
sage: pushout(Sym01, QQ)
(Permutation Group with generators [(0,1)])-invariant submodule
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> GSym01 = PermutationGroup([(Integer(0),Integer(1)),(Integer(2),), (Integer(3), )]); GSym01
Permutation Group with generators [(0,1)]
>>> GASym01 = PermutationGroup([(Integer(0),Integer(1)),(Integer(2),
→(Integer(3)))]); GASym01
Permutation Group with generators [(0,1)(2,3)]
>>> from sage.categories.action import Action
>>> from sage.structure.element import Matrix
>>> class TensorIndexAction(Action):
...     def _act_(self, g, x):
...         if isinstance(x, Matrix):
...             if g(Integer(0)) == Integer(1):
...                 if g(Integer(2)) == Integer(2): (continues on next page)
... return x.transpose()
... else:
... return -x.transpose()
... else:
... raise NotImplementedError

>>> M = matrix([[Integer(1), Integer(2)], [Integer(3), Integer(4)]]); M
[1 2]
[3 4]

>>> GSym01_action = TensorIndexAction(GSym01, M.parent())
>>> GASym01_action = TensorIndexAction(GASym01, M.parent())
>>> GSym01_action.act(GSym01.gen(0), M)

[[1 3]
[2 4]]

>>> GASym01_action.act(GASym01.gen(0), M)

[[-1 -3]
[-2 -4]]

>>> Sym01 = M.parent().invariant_module(GSym01, action=GSym01_action); Sym01
(Permutation Group with generators [(0,1)])-invariant submodule of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

>>> list(Sym01.basis())
[B[0], B[1], B[2]]

>>> list(Sym01.basis().map(Sym01.lift))

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

>>> ASym01 = M.parent().invariant_module(GASym01, action=GASym01_action)
>>> ASym01
(Permutation Group with generators [(0,1)(2,3)])-invariant submodule of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

>>> list(ASym01.basis())
[B[0]]

>>> list(ASym01.basis().map(ASym01.lift))

[[0 1]
[-1 0]]

>>> from sage.categories.pushout import pushout

pushout(Sym01, QQ)

(Permutation Group with generators [(0,1)])-invariant submodule of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

class sage.categories.pushout.FractionField

Bases: ConstructionFunctor

Construction functor for fraction fields.

EXAMPLES:

sage: F = QQ.construction()[0]
sage: F
FractionField
sage: F.domain()
Category of integral domains
sage: F.codomain()
Category of fields

(continues on next page)
sage: F(GF(5)) is GF(5)
True
sage: F(ZZ['t'])
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
sage: P.<x,y> = QQ[]
\texttt{sage: f = P.hom([x+2*y,3*x-y],P)}
\texttt{sage: F(f)}
Ring endomorphism of
\texttt{Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field}
\texttt{Defn: x |--> x + 2*y}
\texttt{y |--> 3*x - y}
\texttt{sage: F(f)(1/x)}
\texttt{1/(x + 2*y)}
\texttt{sage: F == loads(dumps(F))}
True

>>> \texttt{from sage.all import *}
>>> \texttt{F = QQ.construction()[Integer(0)]}
>>> \texttt{F}
\texttt{FractionField}
>>> \texttt{F.domain()}
\texttt{Category of integral domains}
>>> \texttt{F.codomain()}
\texttt{Category of fields}
>>> \texttt{F(GF(Integer(5))) is GF(Integer(5))}
\texttt{True}
>>> \texttt{F(ZZ['t'])}
\texttt{Fraction Field of Univariate Polynomial Ring in t over Integer Ring}
>>> \texttt{P = QQ['x', 'y']; (x, y,) = P._first_ngens(2)}
>>> \texttt{f = P.hom([x+Integer(2)*y,Integer(3)*x-y],P)}
>>> \texttt{F(f)}
Ring endomorphism of
\texttt{Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field}
\texttt{Defn: x |--> x + 2*y}
\texttt{y |--> 3*x - y}
>>> \texttt{F(f)(Integer(1)/x)}
\texttt{1/(x + 2*y)}
>>> \texttt{F == loads(dumps(F))}
\texttt{True}

\texttt{rank = 5}

\texttt{class sage.categories.pushout.IdentityConstructionFunctor}
\texttt{Bases: ConstructionFunctor}
A construction functor that is the identity functor.
\texttt{rank = -100}

\texttt{class sage.categories.pushout.InfinitePolynomialFunctor(gens, order, implementation)}
\texttt{Bases: ConstructionFunctor}
A Construction Functor for Infinite Polynomial Rings (see infinite_polynomial_ring).
\texttt{AUTHOR:}
– Simon King
This construction functor is used to provide uniqueness of infinite polynomial rings as parent structures. As usual, the construction functor allows for constructing pushouts.

Another purpose is to avoid name conflicts of variables of the to-be-constructed infinite polynomial ring with variables of the base ring, and moreover to keep the internal structure of an Infinite Polynomial Ring as simple as possible: If variables \( v_1, \ldots, v_n \) of the given base ring generate an ordered sub-monoid of the monomials of the ambient Infinite Polynomial Ring, then they are removed from the base ring and merged with the generators of the ambient ring. However, if the orders don’t match, an error is raised, since there was a name conflict without merging.

**EXAMPLES:**

```python
sage: A.<a,b> = InfinitePolynomialRing(ZZ['t'])
sage: A.construction()
[InfPoly{[a,b], "lex", "dense"}, Univariate Polynomial Ring in t over Integer Ring]
sage: type(_[0])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
sage: B.<x,y,a_3,a_1> = PolynomialRing(QQ, order='lex')
sage: B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
sage: A.construction()[0] * B.construction()[0]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

```python
>>> from sage.all import *
>>> A = InfinitePolynomialRing(ZZ['t'], names=('a', 'b',)); (a, b,) = A._first_ngens(2)
>>> A.construction()
[InfPoly{[a,b], "lex", "dense"}, Univariate Polynomial Ring in t over Integer Ring]
>>> type(_[Integer(0)])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
>>> B = PolynomialRing(QQ, order='lex', names=('x', 'y', 'a_3', 'a_1',)); (x, y,
                        a_3, a_1,) = B._first_ngens(4)
>>> B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
>>> A.construction()[Integer(0)] * B.construction()[Integer(0)]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

Apparently the variables \( a_1, a_3 \) of the polynomial ring are merged with the variables \( a_0, a_1, a_2, \ldots \) of the infinite polynomial ring; indeed, they form an ordered sub-structure. However, if the polynomial ring was given a different ordering, merging would not be allowed, resulting in a name conflict:

```python
sage: R = PolynomialRing(QQ, names=['x','y','a_3','a_1'])
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
...
CoercionException: Incompatible term orders lex, degrevlex
```

```python
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=['x','y','a_3','a_1'])
>>> A.construction()[Integer(0)] * R.construction()[Integer(0)]
Traceback (most recent call last):
...
CoercionException: Incompatible term orders lex, degrevlex
```

In an infinite polynomial ring with generator \( a_* \), the variable \( a_3 \) will always be greater than the variable \( a_1 \). Hence, the orders are incompatible in the next example as well:

3.4. Coercion via construction functors
Another requirement is that after merging the order of the remaining variables must be unique. This is not the case in the following example, since it is not clear whether the variables \(x, y\) should be greater or smaller than the variables \(b\):

```
sage: R = PolynomialRing(QQ, names=['a_3', 'a_1', 'x', 'y'], order='lex')
sage: A.construction()[Integer(0)] * R.construction()[Integer(0)]
Traceback (most recent call last):
...  
CoercionException: Overlapping variables (('a', 'b'), ['a_3', 'a_1']) are incompatible
```

Since the construction functors are actually used to construct infinite polynomial rings, the following result is no surprise:

```
sage: C.<a,b> = InfinitePolynomialRing(B); C
Infinite polynomial ring in a, b
  over Multivariate Polynomial Ring in x, y over Rational Field
```

There is also an overlap in the next example:

```
sage: X.<w,x,y> = InfinitePolynomialRing(ZZ)
sage: Y.<x,y,z> = InfinitePolynomialRing(QQ)
```

(continues on next page)
X and Y have an overlapping generators $x_*, y_*$. Since the default lexicographic order is used in both rings, it gives rise to isomorphic sub-monoids in both X and Y. They are merged in the pushout, which also yields a common parent for doing arithmetic:

```
>>> Y = InfinitePolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z) = Y._first_ngens(3)
>>> X and Y have an overlapping generators x_, y_. Since the default lexicographic order is used in both rings, it gives rise to isomorphic sub-monoids in both X and Y. They are merged in the pushout, which also yields a common parent for doing arithmetic:

sage: P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
w_2 + z_3
sage: _.parent() is P
True

>>> from sage.all import *
from sage.all import *
>>> P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
>>> w[Integer(2)]+z[Integer(3)]
w_2 + z_3
>>> _.parent() is P
True

expand()

Decompose the functor $F$ into sub-functors, whose product returns $F$.

EXAMPLES:

```
>>> from sage.all import *
>>> A = InfinitePolynomialRing(QQ, ['x','y'], order='degrevlex')
>>> F = A.construction()[0]; F
InfPoly([y], "degrevlex", "dense")
>>> F.expand()
[InfPoly([y], "degrevlex", "dense"), InfPoly([x], "degrevlex", "dense")]
>>> A = InfinitePolynomialRing(QQ, ['x','y','z'], order='degrevlex')
>>> F = A.construction()[0]; F
InfPoly([x,y,z], "degrevlex", "dense")
>>> F.expand()
[InfPoly([z], "degrevlex", "dense"), InfPoly([y], "degrevlex", "dense"), InfPoly([x], "degrevlex", "dense")]
>>> prod(F.expand())==F
True

>>> from sage.all import *
>>> A = InfinitePolynomialRing(QQ, ['x','y'], order='degrevlex')
>>> F = A.construction()[Integer(0)]; F
InfPoly([y], "degrevlex", "dense")
>>> F.expand()
[InfPoly([y], "degrevlex", "dense"), InfPoly([x], "degrevlex", "dense")]
>>> A = InfinitePolynomialRing(QQ, ['x','y','z'], order='degrevlex')
>>> F = A.construction()[Integer(0)]; F
InfPoly([x,y,z], "degrevlex", "dense")
>>> F.expand()
[InfPoly([z], "degrevlex", "dense"), InfPoly([y], "degrevlex", "dense"), InfPoly([x], "degrevlex", "dense")]
```
merge (other)

Merge two construction functors of infinite polynomial rings, regardless of monomial order and implementation.

The purpose is to have a pushout (and thus, arithmetic) even in cases when the parents are isomorphic as rings, but not as ordered rings.

EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, order='degrevlex')
sage: X.construction()  
[InfPoly[[x,y], "lex", "sparse"], Rational Field]
sage: Y.construction()  
[InfPoly[[x,y], "degrevlex", "dense"], Rational Field]
```

```python
>>> from sage.all import *
```

```python
X = InfinitePolynomialRing(QQ, implementation='sparse', names=('x', 'y',)); (x, y,) = X._first_ngens(2)
Y = InfinitePolynomialRing(QQ, order='degrevlex', names=('x', 'y',)); (x, y,)= Y._first_ngens(2)
X.construction()  
[InfPoly[[x,y], "lex", "sparse"], Rational Field]
```

```python
>>> Y.construction()  
[InfPoly[[x,y], "degrevlex", "dense"], Rational Field]
```

```python
>>> Y.construction()[0].merge(Y.construction()[0])
```

```python
InfPoly[[x,y], "degrevlex", "dense"]
```

```python
>>> y[Integer(3)] + X(x[Integer(2)])
```

```python
x_2 + y_3
```

```python
>>> _.parent().construction()  
[InfPoly[[x,y], "degrevlex", "dense"], Rational Field]
```

```
(continued from previous page)
```
**sage:** `F(K)`
Univariate Laurent Polynomial Ring in t
over Univariate Laurent Polynomial Ring in x over Integer Ring

```python
sage: P.<x,y> = ZZ[]
sage: f = P.hom([x + 2*y, 3*x - y],P)
sage: F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t
over Multivariate Polynomial Ring in x, y over Integer Ring
    Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer...
        Defn: x |--> x + 2*y
        y |--> 3*x - y
```

```python
sage: F(f)(x*F(P).gen()^-2 + y*F(P).gen()^3)
(x + 2*y)*t^-2 + (3*x - y)*t^3
```

**merge** *(other)*

Two Laurent polynomial construction functors merge if the variable names coincide.

The result is multivariate if one of the arguments is multivariate.

**EXAMPLES:**

```python
>>> from sage.categories.pushout import *
>>> L = LaurentPolynomialRing(ZZ, names=('t',)); (t,) = L._first_ngens(1)
>>> F = L.construction()[Integer(0)]
>>> F
LaurentPolynomialFunctor
>>> F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
>>> K = LaurentPolynomialRing(ZZ, names=('x',)); (x,) = K._first_ngens(1)
>>> F(K)
Univariate Laurent Polynomial Ring in t
over Univariate Laurent Polynomial Ring in x over Integer Ring
>>> P = ZZ['x, y']; (x, y,) = P._first_ngens(2)
>>> f = P.hom([x + Integer(2)*y, Integer(3)*x - y],P)
>>> F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t
over Multivariate Polynomial Ring in x, y over Integer Ring
    Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer...
        Defn: x |--> x + 2*y
        y |--> 3*x - y

>>> F(f)(x*F(P).gen()**Integer(2) + y*F(P).gen()**Integer(3))
(x + 2*y)*t^-2 + (3*x - y)*t^3
```
Univariate Laurent Polynomial Ring in a over Finite Field of size 2

```python
>>> from sage.all import *
>>> from sage.categories.pushout import LaurentPolynomialFunctor
>>>
F1 = LaurentPolynomialFunctor('t')
>>> F2 = LaurentPolynomialFunctor('t', multi_variate=True)
>>> F1.merge(F2)
LaurentPolynomialFunctor
>>> F1.merge(F2)(LaurentPolynomialRing(GF(Integer(2)), 'a'))  # needs sage.modules
Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2
>>> F1.merge(F1)(LaurentPolynomialRing(GF(Integer(2)), 'a'))
Univariate Laurent Polynomial Ring in t over Univariate Laurent Polynomial Ring in a over Finite Field of size 2
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
>>> from sage.all import *  
>>> # needs sage.modules
>>> MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>> F = MS.construction()[Integer(0)]; F
MatrixFunctor
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, 2)
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```

```python
rank = 9

class sage.categories.pushout.MatrixFunctor (nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ, Integer(2), Integer(3))
>>>
```
MatrixFunctor

```python
>>> P = QQ['x', 'y']; (x, y,) = P._first_ngens(2)
>>> R = P(R); R
Full MatrixSpace of 2 by 2 dense matrices
over Multivariate Polynomial Ring in x, y over Rational Field
>>> f = P.hom([x+y, x-y], P); R(f)
Ring endomorphism
of Full MatrixSpace of 2 by 2 dense matrices
over Multivariate Polynomial Ring in x, y over Rational Field
Defn: Induced from base ring by
   Ring endomorphism
   of Multivariate Polynomial Ring in x, y over Rational Field
   Defn: x |--> x + y
   y |--> x - y
>>> M = R([x, y, x*y, x + y])
>>> R(f)(M)
[   x + y   x - y]
[ x^2 - y^2    2*x]
```

merge (other)
Merging is only happening if both functors are matrix functors of the same dimension.

The result is sparse if and only if both given functors are sparse.

EXAMPLES:

```python
sage: # needs sage.modules
sage: F1 = MatrixSpace(ZZ, 2, 2).construction()[0]
sage: F2 = MatrixSpace(ZZ, 2, 3).construction()[0]
sage: F3 = MatrixSpace(ZZ, 2, 2, sparse=True).construction()[0]
sage: F1.merge(F2)
MatrixFunctor
sage: F1.merge(F3)
MatrixFunctor
sage: F13 = F1.merge(F3)
MatrixFunctor
sage: F13.is_sparse
False
sage: F1.is_sparse
False
sage: F3.is_sparse
True
sage: F3.merge(F3).is_sparse
True
```

```python
>>> from sage.all import *
```

```python
>>> F1 = MatrixSpace(ZZ, Integer(2), Integer(2)).construction()[0]
>>> F2 = MatrixSpace(ZZ, Integer(2), Integer(3)).construction()[0]
>>> F3 = MatrixSpace(ZZ, Integer(2), Integer(2), sparse=True).construction()[0]
```

```python
>>> F1.merge(F2)
MatrixFunctor
>>> F1.merge(F3)
MatrixFunctor
>>> F13 = F1.merge(F3)
>>> F13.is_sparse
False
>>> F1.is_sparse
False
>>> F3.is_sparse
True
```

(continues on next page)
```python
>>> F3.is_sparse
True
>>> F3.merge(F3).is_sparse
True
```

```python
rank = 10
class sage.categories.pushout.MultiPolynomialFunctor(vars, term_order):
    Bases: ConstructionFunctor

    A constructor for multivariate polynomial rings.

    EXAMPLES:

    sage: P.<x,y> = ZZ[]
    sage: F = P.construction()[0]; F
    MPoly[x,y]
    sage: A.<a,b> = GF(5)[]
    sage: F(A)
    Multivariate Polynomial Ring in x, y
    over Multivariate Polynomial Ring in a, b over Finite Field of size 5
    sage: f = A.hom([a+b, a-b], A)
    sage: F(f)
    Ring endomorphism of Multivariate Polynomial Ring in x, y
    over Multivariate Polynomial Ring in a, b over Finite Field of size 5
    Defn: Induced from base ring by
         Ring endomorphism of Multivariate Polynomial Ring in a, b over Finite Field of size 5
         Defn: a |--> a + b
             b |--> a - b
    sage: F(f)(F(A)(x)*a)
    (a + b)*x

    >>> from sage.all import *
    >>> P = ZZ['x', y]; (x, y,) = P._first_ngens(2)
    >>> F = P.construction()[Integer(0)]; F
    MPoly[x,y]
    >>> A = GF(Integer(5))['a', 'b']; (a, b,) = A._first_ngens(2)
    >>> F(A)
    Multivariate Polynomial Ring in x, y
    over Multivariate Polynomial Ring in a, b over Finite Field of size 5
    >>> f = A.hom([a+b, a-b], A)
    >>> F(f)
    Ring endomorphism of Multivariate Polynomial Ring in x, y
    over Multivariate Polynomial Ring in a, b over Finite Field of size 5
    Defn: Induced from base ring by
         Ring endomorphism of Multivariate Polynomial Ring in a, b over Finite Field of size 5
         Defn: a |--> a + b
             b |--> a - b
    >>> F(f)(F(A)(x)*a)
    (a + b)*x
```

```python
expand()

    Decompose self into a list of construction functors.

    EXAMPLES:

222 Chapter 3. Maps and Morphisms
sage: F = QQ['x,y,z,t'].construction()[0]; F
MPoly[x,y,z,t]
sage: F.expand()
[MPoly[t], MPoly[z], MPoly[y], MPoly[x]]

>>> from sage.all import *
>>> F = QQ['x,y,z,t'].construction()[Integer(0)]; F
MPoly[x,y,z,t]
>>> F.expand()
[MPoly[t], MPoly[z], MPoly[y], MPoly[x]]

Now an actual use case:

sage: R.<x,y,z> = ZZ[]
sage: S.<z,t> = QQ[]
sage: x+t
x + t
sage: parent(x+t)
Multivariate Polynomial Ring in x, y, z over Integer Ring
sage: T.<y,s> = QQ[]
sage: x+s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Multivariate Polynomial Ring in x, y, z over Integer Ring' and
'Multivariate Polynomial Ring in y, s over Rational Field'
sage: R = PolynomialRing(ZZ, 'x', 50)
sage: S = PolynomialRing(GF(5), 'x', 20)
sage: R.gen(0) + S.gen(0)
2*x0

>>> from sage.all import *
>>> R = ZZ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> S = QQ['z, t']; (z, t,) = S._first_ngens(2)
>>> x+t
x + t
>>> parent(x+t)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
>>> T = QQ['y, s']; (y, s,) = T._first_ngens(2)
>>> x + s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Multivariate Polynomial Ring in x, y, z over Integer Ring' and
'Multivariate Polynomial Ring in y, s over Rational Field'

merge (other)

Merge self with another construction functor, or return None.

EXAMPLES:

sage: F = sage.categories.pushout.MultiPolynomialFunctor(['x', 'y'], None)
sage: G = sage.categories.pushout.MultiPolynomialFunctor(['t'], None)

(continues on next page)
sage: F.merge(G) is None
True
sage: F.merge(F)
MPoly[x,y]

>>> from sage.all import *
>>> F = sage.categories.pushout.MultiPolynomialFunctor([x,y], None)
>>> G = sage.categories.pushout.MultiPolynomialFunctor([t], None)
>>> F.merge(G) is None
True
>>> F.merge(F)
MPoly[x,y]

rank = 9

class sage.categories.pushout.MultivariateConstructionFunctor

Bases: ConstructionFunctor

An abstract base class for functors that take multiple inputs (e.g. Cartesian products).

common_base (other_functor, self_bases, other_bases)

This function is called by pushout() when no common parent is found in the construction tower.

INPUT:

• other_functor – a construction functor.

• self_bases – the arguments passed to this functor.

• other_bases – the arguments passed to the functor other_functor.

OUTPUT:

A parent.

If no common base is found a sage.structure.coerce_exceptions.CoercionException is raised.

Note: Overload this function in derived class, see e.g. MultivariateConstructionFunctor.

class sage.categories.pushout.PermutationGroupFunctor (gens, domain)

Bases: ConstructionFunctor

EXAMPLES:

sage: from sage.categories.pushout import PermutationGroupFunctor
sage: PF = PermutationGroupFunctor([PermutationGroupElement([(1,2)])], # needs sage.groups
....: [1,2]); PF
PermutationGroupFunctor([(1,2)]

>>> from sage.all import *
>>> from sage.categories.pushout import PermutationGroupFunctor
>>> PF = PermutationGroupFunctor([PermutationGroupElement([(1,1), # needs sage.groups
... [Integer(1),Integer(2)]]); PF
PermutationGroupFunctor([(1,2)])
**gens()**

**EXAMPLES:**

```python
sage: P1 = PermutationGroup([[1, 2]])  # needs sage.groups
sage: PF, P = P1.construction()  # needs sage.groups
sage: PF.gens()  # needs sage.groups
((1, 2),)
```

```python
>>> from sage.all import *
>>> P1 = PermutationGroup([(Integer(1), Integer(2))])  # needs sage.groups
>>> PF, P = P1.construction()
>>> PF.gens()  # needs sage.groups
((1, 2),)
```

**merge**(other)

Merge self with another construction functor, or return None.

**EXAMPLES:**

```python
sage: # needs sage.groups
sage: P1 = PermutationGroup([[1, 2]])
sage: PF1, P = P1.construction()
sage: P2 = PermutationGroup([[1, 3]])
sage: PF2, P = P2.construction()
sage: PF1.merge(PF2)  # needs sage.groups
PermutationGroupFunctor[(1, 2), (1, 3)]
```

```python
>>> from sage.all import *
>>> # needs sage.groups
>>> P1 = PermutationGroup([[(Integer(1), Integer(2))]])
>>> PF1, P = P1.construction()
>>> P2 = PermutationGroup([[(Integer(1), Integer(3))]])
>>> PF2, P = P2.construction()
>>> PF1.merge(PF2)  # needs sage.groups
PermutationGroupFunctor[(1, 2), (1, 3)]
```

**rank = 10**

```python
class sage.categories.pushout.PolynomialFunctor(var, multi_variate=False, sparse=False, implementation=None)
```

**Bases:** ConstructionFunctor

Construction functor for univariate polynomial rings.

**EXAMPLES:**

```python
sage: P = ZZ['t'].construction()[0]
sage: P(GF(3))
Univariate Polynomial Ring in t over Finite Field of size 3
sage: P == loads(dumps(P))
True
```
By Issue #9944, the construction functor distinguishes sparse and dense polynomial rings. Before, the following example failed:

```
sage: R.<x> = PolynomialRing(GF(5), sparse=True)
sage: F, B = R.construction()
sage: F(B) is R
True
sage: S.<x> = PolynomialRing(ZZ)
sage: R.has_coerce_map_from(S)
False
sage: S.0 + R.0
2*x
sage: (S.0 + R.0).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
sage: (S.0 + R.0).parent().is_sparse()
False
```

```
>>> from sage.all import *
>>> R = PolynomialRing(GF(Integer(5)), sparse=True, names=('x',)); (x,) = R._first_ngens(1)
>>> F, B = R.construction()
>>> F(B) is R
True
>>> S = PolynomialRing(ZZ, names=('x',)); (x,) = S._first_ngens(1)
>>> R.has_coerce_map_from(S)
False
>>> S.gen(0) + R.gen(0)
2*x
>>> (S.gen(0) + R.gen(0)).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
>>> (S.gen(0) + R.gen(0)).parent().is_sparse()
False
```

**merge**(other)

Merge self with another construction functor, or return None.
Note: Internally, the merging is delegated to the merging of multipolynomial construction functors. But in effect, this does the same as the default implementation, that returns `None` unless the to-be-merged functors coincide.

EXAMPLES:

```python
sage: P = ZZ['x'].construction()[0]
sage: Q = ZZ['y','x'].construction()[0]
sage: P.merge(Q)
sage: P.merge(P)  # P
True
```

```python
>>> from sage.all import *

P = ZZ['x'].construction()[Integer(0)]

Q = ZZ['y','x'].construction()[Integer(0)]

P.merge(Q)

class sage.categories.pushout.QuotientFunctor (I, names=None, as_field=False, domain=None, codomain=None, **kwds):

Bases: ConstructionFunctor

Construction functor for quotient rings.

Note: The functor keeps track of variable names. Optionally, it may keep track of additional properties of the quotient, such as its category or its implementation.

EXAMPLES:

```python
sage: P.<x,y> = ZZ[]
sage: Q = P.quo([x^2 + y^2] * P)
sage: F = Q.construction()[0]
sage: F(QQ['x,y'])
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)

sage: F(QQ['x','y']) == QQ['x','y'].quo([x^2 + y^2] * QQ['x','y'])
True

sage: F(QQ['x','y','z'])
Traceback (most recent call last):
...
CoercionException: Cannot apply this quotient functor to Multivariate Polynomial Ring in x, y, z over Rational Field
```

```python
>>> from sage.all import *

P = ZZ['x', 'y']; (x, y,) = P._first_ngens(2)

Q = P.quo([x**Integer(2) + y**Integer(2)] * P)
```

(continues on next page)
category: merge (other)

Two quotient functors with coinciding names are merged by taking the gcd of their moduli, the meet of their domains, and the join of their codomains.

In particular, if one of the functors being merged knows that the quotient is going to be a field, then the merged functor will return fields as well.

EXAMPLES:

```python
sage: # needs sage.libs.pari
sage: P.⟨x⟩ = QQ[]
```

```python
sage: Q1 = P.quo([(x**Integer(2)+Integer(1))**Integer(2)*(x**Integer(2)-Integer(3))])
```

```python
sage: from sage.categories.pushout import pushout
```

```python
sage: P = QQ['x']; (x,) = P._first_ngens(1)
```

```python
sage: Q2 = P.quo([(x**Integer(2)+Integer(1))**Integer(2)*(x**Integer(5)+Integer(3))])
```

The following was fixed in Issue #8800:

```python
sage: pushout(GF(5), Integers(5))  # needs sage.libs.pari
```

```python
Finite Field of size 5
```

(continues on next page)
class sage.categories.pushout.SubspaceFunctor(basis)
Bases: ConstructionFunctor

Constructing a subspace of an ambient free module, given by a basis.

Note: This construction functor keeps track of the basis. It can only be applied to free modules into which this basis coerces.

EXAMPLES:

```
sage: # needs sage.modules
sage: M = ZZ^3
sage: S = M.submodule([(1,2,3), (4,5,6)]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: F = S.construction()[0]
sage: F(GF(2)^3)
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> M = ZZ**Integer(3)
>>> S = M.submodule([(Integer(1),Integer(2),Integer(3)), (Integer(4),Integer(5),
˓→Integer(6))]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
>>> F = S.construction()[Integer(0)]
>>> F(GF(Integer(2))**Integer(3))
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

cocercion_reversed = True

merge(other)

Two Subspace Functors are merged into a construction functor of the sum of two subspaces.

EXAMPLES:

```
sage: # needs sage.modules
sage: M = GF(5)^3
sage: S1 = M.submodule([\(1,2,3\), (4,5,6)])
```
```python
sage: S2 = M.submodule([(2,2,3)])
sage: F1 = S1.construction()[0]
sage: F2 = S2.construction()[0]
sage: F1.merge(F2)
SubspaceFunctor
sage: F1.merge(F2)(GF(5)^3) == S1 + S2
True
sage: F1.merge(F2)(GF(5)[t]^3)
Free module of degree 3 and rank 3
over Univariate Polynomial Ring in t over Finite Field of size 5
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> M = GF(Integer(5))**Integer(3)
>>> S1 = M.submodule([(Integer(1),Integer(2),Integer(3)),(Integer(4),...
>>> S2 = M.submodule([(Integer(2),Integer(2),Integer(3))])
>>> F1 = S1.construction()[Integer(0)]
>>> F2 = S2.construction()[Integer(0)]
>>> F1.merge(F2)
SubspaceFunctor
>>> F1.merge(F2)(GF(Integer(5))**Integer(3)) == S1 + S2
True
>>> F1.merge(F2)(GF(Integer(5))[t]**Integer(3))
Free module of degree 3 and rank 3
over Univariate Polynomial Ring in t over Finite Field of size 5
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

```
rank = 11

class sage.categories.pushout.VectorFunctor (n=None, is_sparse=False,
inner_product_matrix=None, *
with_basis='standard', basis_keys=None,
name_mapping=None,
latex_name_mapping=None)

Bases: ConstructionFunctor

A construction functor for free modules over commutative rings.

EXAMPLES:
```
```
>>> from sage.all import *
>>> # needs sage.modules
>>> F = (ZZ**Integer(3)).construction()[Integer(0)]
>>> F
VectorFunctor

Ambient free module of rank 3
over the principal ideal domain Univariate Polynomial Ring in t
over Finite Field of size 2 (using GF2X)

merge(other)

Two constructors of free modules merge, if the module ranks and the inner products coincide. If both have explicitly given inner product matrices, they must coincide as well.

EXAMPLES:

Two modules without explicitly given inner product allow coercion:

```
sage: M1 = QQ^3
→ needs sage.modules
sage: P.<t> = ZZ[]
sage: M2 = FreeModule(P, 3)
→ needs sage.modules
sage: M1([1,1/2,1/3]) + M2([t,t^2+t,3])
→ needs sage.modules
(t + 1, t^2 + t + 1/2, 10/3)
```

If only one summand has an explicit inner product, the result will be provided with it:

```
sage: M3 = FreeModule(P, 3, inner_product_matrix=Matrix(3, 3, range(9)))
→ needs sage.modules
sage: M1([1,1/2,1/3]) + M3([t,t^2+t,3])
→ needs sage.modules
(t + 1, t^2 + t + 1/2, 10/3)
sage: (M1([1,1/2,1/3]) + M3([t,t^2+t,3])).parent().inner_product_matrix()
→ needs sage.modules
[0 1 2]
[3 4 5]
[6 7 8]
```

(continues on next page)
If both summands have an explicit inner product (even if it is the standard inner product), then the products must coincide. The only difference between $M_1$ and $M_4$ in the following example is the fact that the default inner product was \textit{explicitly} requested for $M_4$. It is therefore not possible to coerce with a different inner product:

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M4 = FreeModule(QQ, Integer(3), inner_product_matrix=Matrix(Integer(3),
-> Integer(3), Integer(1)))
>>> M4 == M1
True
>>> M4.inner_product_matrix() == M1.inner_product_matrix()
True
>>> M4([[1,1/2,1/3]] + M3([t,*t^2+t,3])) # indirect doctest
Traceback (most recent call last):
  ...TypeError: unsupported operand parent(s) for +:
'Ambient quadratic space of dimension 3 over Rational Field
Inner product matrix:
[1 0 0]
[0 1 0]
[0 0 1]' and
'Ambient free quadratic module of rank 3 over the integral domain
Univariate Polynomial Ring in t over Integer Ring
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

(continues on next page)
Inner product matrix:
\[
\begin{bmatrix}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{bmatrix}
\]

Names are removed when they conflict:

```python
sage: # needs sage.modules
sage: from sage.categories.pushout import VectorFunctor, pushout
sage: M_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='M_ZZx')
sage: N_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='N_ZZx')
sage: pushout (M_ZZx, QQ)
Rank-4 free module M_ZZx_base_ext
over the Univariate Polynomial Ring in x over Rational Field
sage: pushout (M_ZZx, N_ZZx)
Rank-4 free module
over the Univariate Polynomial Ring in x over Integer Ring
sage: pushout (pushout (M_ZZx, N_ZZx), QQ)
Rank-4 free module
over the Univariate Polynomial Ring in x over Rational Field
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.categories.pushout import VectorFunctor, pushout

>>> M_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='M_ZZx')
>>> N_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='N_ZZx')

>>> pushout (M_ZZx, QQ)
Rank-4 free module M_ZZx_base_ext
over the Univariate Polynomial Ring in x over Rational Field

>>> pushout (M_ZZx, N_ZZx)
Rank-4 free module
over the Univariate Polynomial Ring in x over Integer Ring

>>> pushout (pushout (M_ZZx, N_ZZx), QQ)
Rank-4 free module
over the Univariate Polynomial Ring in x over Rational Field
```

```python
>>> from sage.categories.pushout import construction_tower
```

```python
sage: # needs sage.modules
sage: from sage.categories.pushout import VectorFunctor, pushout

sage: M_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='M_ZZx')

sage: N_ZZx = FreeModule(ZZ['x'], Integer(4), with_basis=None, name='N_ZZx')

sage: pushout (M_ZZx, QQ)
Rank-4 free module M_ZZx_base_ext
over the Univariate Polynomial Ring in x over Rational Field

sage: pushout (M_ZZx, N_ZZx)
Rank-4 free module
over the Univariate Polynomial Ring in x over Integer Ring

sage: pushout (pushout (M_ZZx, N_ZZx), QQ)
Rank-4 free module
over the Univariate Polynomial Ring in x over Rational Field
```

```
```

```
```

```python
rank = 10
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
of Univariate Polynomial Ring in t over Rational Field),
(Poly[t], Rational Field), (FractionField, Integer Ring)]

>>> from sage.all import *
>>> from sage.categories.pushout import construction_tower
>>> construction_tower(MatrixSpace(FractionField(QQ['t']), Integer(2)))  
˓→ # needs sage.modules
[(None, Full MatrixSpace of 2 by 2 dense matrices over Fraction Field
of Univariate Polynomial Ring in t over Rational Field),
(MatrixFunctor, Fraction Field
of Univariate Polynomial Ring in t over Rational Field),
(FractionField, Univariate Polynomial Ring in t over Rational Field),
(Poly[t], Rational Field), (FractionField, Integer Ring)]

sage.categories.pushout.expand_tower(tower)
An auxiliary function that is used in pushout().

INPUT:

A construction tower as returned by construction_tower().

OUTPUT:

A new construction tower with all the construction functors expanded.

EXAMPLES:

sage: from sage.categories.pushout import construction_tower, expand_tower
sage: construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
(MPoly[x,y,z], Rational Field),
(FractionField, Integer Ring)]

sage: expand_tower(construction_tower(QQ['x,y,z']))
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
(MPoly[z], Univariate Polynomial Ring in y
over Univariate Polynomial Ring in x over Rational Field),
(MPoly[y], Univariate Polynomial Ring in x over Rational Field),
(MPoly[x], Rational Field),
(FractionField, Integer Ring)]

sage.categories.pushout.pushout(R, S)
Given a pair of objects $R$ and $S$, try to construct a reasonable object $Y$ and return maps such that canonically $R \leftarrow Y \rightarrow S$. 

234 Chapter 3. Maps and Morphisms
**ALGORITHM:**

This incorporates the idea of functors discussed at Sage Days 4. Every object $R$ can be viewed as an initial object and a series of functors (e.g. polynomial, quotient, extension, completion, vector/matrix, etc.). Call the series of increasingly simple objects (with the associated functors) the “tower” of $R$. The construction method is used to create the tower.

Given two objects $R$ and $S$, try to find a common initial object $Z$. If the towers of $R$ and $S$ meet, let $Z$ be their join. Otherwise, see if the top of one coalesces naturally into the other.

Now we have an initial object and two ordered lists of functors to apply. We wish to merge these in an unambiguous order, popping elements off the top of one or the other tower as we apply them to $Z$.

- If the functors are of distinct types, there is an absolute ordering given by the rank attribute. Use this.
- Otherwise:
  - If the tops are equal, we (try to) merge them.
  - If exactly one occurs lower in the other tower, we may unambiguously apply the other (hoping for a later merge).
  - If the tops commute, we can apply either first.
  - Otherwise fail due to ambiguity.

The algorithm assumes by default that when a construction $F$ is applied to an object $X$, the object $F(X)$ admits a coercion map from $X$. However, the algorithm can also handle the case where $F(X)$ has a coercion map to $X$ instead. In this case, the attribute coercion_reversed of the class implementing $F$ should be set to True.

**EXAMPLES:**

Here our “towers” are $R = \text{Complete}_7(\text{Frac}(\mathbb{Z}))$ and $\text{Frac}(\text{Poly}_x(\mathbb{Z}))$, which give us $\text{Frac}(\text{Poly}_x(\text{Complete}_7(\text{Frac}(\mathbb{Z}))))$:

```sage
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7), Frac(ZZ['x'])) #...
˓→ needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
over 7-adic Field with capped relative precision 20
```

```sage
>>> from sage.all import *
>>> from sage.categories.pushout import pushout
>>> pushout(Qp(Integer(7)), Frac(ZZ['x'])) #...
˓→ # needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
over 7-adic Field with capped relative precision 20
```

Note we get the same thing with

```sage
sage: pushout(Zp(7), Frac(QQ['x'])) #...
˓→ needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
over 7-adic Field with capped relative precision 20
sage: pushout(Zp(7)['x'], Frac(QQ['x'])) #...
˓→ needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
over 7-adic Field with capped relative precision 20
```
Note that polynomial variable ordering must be unambiguously determined.

```python
sage: pushout(ZZ['x,y,z'], QQ['w,z,t'])
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension',
Multivariate Polynomial Ring in x, y, z over Integer Ring,
Multivariate Polynomial Ring in w, z, t over Rational Field)
```

Some other examples:

```python
sage: pushout(Zp(7)['y'], Frac(QQ['t'])['x,y,z'])
# needs sage.rings.padics
Multivariate Polynomial Ring in x, y, z over Fraction Field of Univariate Polynomial Ring in t
over 7-adic Field with capped relative precision 20
```
A construction with coercion\_reversed=True (currently only the SubspaceFunctor construction) is only applied if it leads to a valid coercion:

```python
sage: # needs sage.modules
sage: A = ZZ^2
sage: V = span([[1, 2]], QQ)
sage: P = sage.categories.pushout.pushout(A, V)
sage: P
Vector space of dimension 2 over Rational Field
sage: P.has_coerce_map_from(A)
True
```

```python
sage: # needs sage.modules
sage: V = (QQ^3).span([[1, 2, 3/4]])
```

```python
sage: A = ZZ^3
```

```python
sage: pushout(A, V)
Vector space of dimension 3 over Rational Field
```

```python
sage: B = A.span([[0, 0, 2/3]])
```

```python
sage: pushout(B, V)
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.modules
```

```python
>>> A = ZZ**Integer(2)
```

```python
>>> V = span([[Integer(1), Integer(2)]], QQ)
```

```python
>>> P = sage.categories.pushout.pushout(A, V)
```

```python
>>> P
Vector space of dimension 2 over Rational Field
>>> P.has_coerce_map_from(A)
True
```
```python
>>> V = (QQ**Integer(3)).span([[Integer(1), Integer(2), Integer(3)/Integer(4)]]))
>>> A = ZZ**Integer(3)
>>> pushout(A, V)
Vector space of dimension 3 over Rational Field
>>> B = A.span([[Integer(0), Integer(0), Integer(2)/Integer(3)]]))
>>> pushout(B, V)
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

Some more tests with `coercion_reversed=True`:

```python
sage: from sage.categories.pushout import ConstructionFunctor
sage: class EvenPolynomialRing(type(QQ['x'])):
    ....:     def __init__(self, base, var):
    ....:         super().__init__(base, var)
    ....:         self.register_embedding(base[var])
    ....:     def __repr__(self):
    ....:         return "Even Power " + super().__repr__()
    ....:     def construction(self):
    ....:         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
    ....:     def _coerce_map_from_(self, R):
    ....:         return self.base().has_coerce_map_from(R)

sage: class EvenPolynomialFunctor(ConstructionFunctor):
    ....:     rank = 10
    ....:     coercion_reversed = True
    ....:     def __init__(self):
    ....:         ConstructionFunctor.__init__(self, Rings(), Rings())
    ....:     def _apply_functor(self, R):
    ....:         return EvenPolynomialRing(R.base(), R.variable_name())

sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR)
# needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of...
# precision
sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR['x'])
# needs sage.rings.real_mpfr
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(QQ, 'x'))
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(RR, 'x'))
# needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of...
# precision
```
Ambient free module of rank 2
over the principal ideal domain Even Power Univariate Polynomial Ring in x
over Real Field with 53 bits of precision

```
sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR['x']^2)  # needs sage.modules sage.rings.real_mpfr
```

```
Ambient free module of rank 2
over the principal ideal domain Univariate Polynomial Ring in x
over Real Field with 53 bits of precision
```

```
... def __init__(self, base, var):
...     super().__init__(base, var)
...     self.register_embedding(base[var])
... def __repr__(self):
...     return "Even Power " + super().__repr__()
... def construction(self):
...     return EvenPolynomialFunctor(), self.base()[self.variable_name()]
... def _coerce_map_from_(self, R):
...     return self.base().has_coerce_map_from(R)
```

```
>>> from sage.all import *
>>> from sage.categories.pushout import ConstructionFunctor
>>> class EvenPolynomialRing(type(QQ['x'])):
...     def __init__(self, base, var):
...         super().__init__(base, var)
...         self.register_embedding(base[var])
...     def __repr__(self):
...         return "Even Power " + super().__repr__()
...     def construction(self):
...         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
...     def _coerce_map_from_(self, R):
...         return self.base().has_coerce_map_from(R)
```

```
...
>>> class EvenPolynomialFunctor(ConstructionFunctor):
...     rank = Integer(10)
...     coercion_reversed = True
...     def __init__(self):
...         ConstructionFunctor.__init__(self, Rings(), Rings())
...     def _apply_functor(self, R):
...         return EvenPolynomialRing(R.base(), R.variable_name())
```

```
>>> pushout(EvenPolynomialRing(QQ['x'], ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ['x'], QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ['x'], RR)  # needs sage.rings.real_mpfr
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of...
```

```
>>> pushout(EvenPolynomialRing(QQ['x'], ZZ['x'])
Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ['x'], QQ['x'])
Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ['x'], RR['x'])  # needs sage.rings.real_mpfr
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
```

```
>>> pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(QQ, 'x'))
Even Power Univariate Polynomial Ring in x over Rational Field
>>> pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(RR, 'x'))  # needs sage.rings.real_mpfr
```

```
>>> pushout(EvenPolynomialRing(QQ, 'x')**Integer(2), RR**Integer(2))  # needs sage.modules sage.rings.real_mpfr
Ambient free module of rank 2
over the principal ideal domain Even Power Univariate Polynomial Ring in x
```
Some more tests related to univariate/multivariate constructions. We consider a generalization of polynomial rings, where in addition to the coefficient ring $C$ we also specify an additive monoid $E$ for the exponents of the indeterminate. In particular, the elements of such a parent are given by

$$
\sum_{i=0}^{l} c_i X^{e_i}
$$

with $c_i \in C$ and $e_i \in E$. We define

```python
sage: class GPolynomialRing(Parent):
    ....: def __init__(self, coefficients, var, exponents):
    ....:     self.coefficients = coefficients
    ....:     self.var = var
    ....:     self.exponents = exponents
    ....:     super().__init__(category=Rings())
    ....: def __repr__(self):
    ....:     return 'Generalized Polynomial Ring in %s(%s) over %s' % (self.var, self.exponents, self.coefficients)
    ....: def construction(self):
    ....:     return GPolynomialFunctor(self.var, self.exponents), self.coefficients
    ....: def _coerce_map_from_(self, R):
    ....:     return self.coefficients.has_coerce_map_from(R)
```

and

```python
sage: class GPolynomialFunctor(ConstructionFunctor):
    ....: rank = 10
    ....: def __init__(self, var, exponents):
    ....:     self.var = var
    ....:     self.exponents = exponents
    ....:     ConstructionFunctor.__init__(self, Rings(), Rings())
    ....: def __repr__(self):
    ....:     return 'GPoly[%s(%s)]' % (self.var, self.exponents)
```

(continues on next page)
def _apply_functor(self, coefficients):
    return GPolynomialRing(coefficients, self.var, self.exponents)

def merge(self, other):
    if isinstance(other, GPolynomialFunctor) and self.var == other.var:
        exponents = pushout(self.exponents, other.exponents)
        return GPolynomialFunctor(self.var, exponents)

We can construct a parent now in two different ways:

sage: GPolynomialRing(QQ, 'X', ZZ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: GP_ZZ = GPolynomialFunctor('X', ZZ); GP_ZZ
GPoly[X^(Integer Ring)]
sage: GP_ZZ(QQ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field

Since the construction

sage: GP_ZZ(QQ).construction()
(GPoly[X^(Integer Ring)], Rational Field)

uses the coefficient ring, we have the usual coercion with respect to this parameter:

sage: pushout(GP_ZZ(ZZ), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: pushout(GP_ZZ(ZZ['t']), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring)

3.4. Coercion via construction functors
over Univariate Polynomial Ring in t over Rational Field

```sage```
pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['b,c']))
```
Generalized Polynomial Ring in X^(Integer Ring)
over Multivariate Polynomial Ring in a, b, c over Integer Ring

```sage```
pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(QQ['b,c']))
```
Generalized Polynomial Ring in X^(Integer Ring)
over Multivariate Polynomial Ring in a, b, c over Rational Field

```sage```
pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
```
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)

```>> from sage.all import *
>> pushout(GP_ZZ(ZZ), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
>> pushout(GP_ZZ(ZZ['t']), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring)
over Univariate Polynomial Ring in t over Rational Field
>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
over Multivariate Polynomial Ring in a, b, c over Integer Ring
>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(QQ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
over Multivariate Polynomial Ring in a, b, c over Rational Field
>> pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

```sage```
GP_QQ = GPolynomialFunctor('X', QQ)
sage
```pushout(GP_ZZ(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Integer Ring) over Integer Ring
```
```sage```
pushout(GP_QQ(ZZ), GP_ZZ(ZZ))
```
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring

```>> from sage.all import *
>> GP_QQ = GPolynomialFunctor('X', QQ)
>> pushout(GP_ZZ(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
>> pushout(GP_QQ(ZZ), GP_ZZ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
```

```sage```
GP_ZZt = GPolynomialFunctor('X', ZZ['t'])
sage
```pushout(GP_ZZt(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t over Rational Field) over Integer Ring
```

```>> from sage.all import *
>> GP_ZZt = GPolynomialFunctor('X', ZZ['t'])
>> pushout(GP_ZZt(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t over Rational Field) over Integer Ring
```

```sage```
pushout(GP_ZZ(ZZ), GP_QQ(QQ))
```
(continues on next page)
Generalized Polynomial Ring in X^`(Rational Field) over Rational Field
\texttt{sage: pushout(GP\_ZZ(QQ), GP\_QQ(ZZ))}

Generalized Polynomial Ring in X^`(Rational Field) over Rational Field
\texttt{sage: pushout(GP\_ZZt(QQ), GP\_QQ(ZZ))}

Generalized Polynomial Ring in X^`(Univariate Polynomial Ring in t over Rational Field) over Rational Field
\texttt{sage: pushout(GP\_ZZt(ZZ), GP\_QQ(QQ))}

Generalized Polynomial Ring in X^`(Univariate Polynomial Ring in t over Rational Field) over Rational Field
\texttt{sage: pushout(GP\_ZZt(ZZ\([a,b]\)), GP\_QQ(ZZ\([c,d]\)))}

Traceback (most recent call last):
... CoercionException: ('Ambiguous Base Extension', ...)
\texttt{sage: pushout(GP\_ZZt(ZZ\([a,b]\)), GP\_QQ(ZZ\([b,c]\)))}

Generalized Polynomial Ring in X^`(Univariate Polynomial Ring in t over Rational Field) over Multivariate Polynomial Ring in a, b, c over Integer Ring

```python
>>> from sage.all import *
>>> pushout(GP\_ZZ(ZZ), GP\_QQ(QQ))
Generalized Polynomial Ring in X^`(Rational Field) over Rational Field
>>> pushout(GP\_ZZ(QQ), GP\_QQ(ZZ))
Generalized Polynomial Ring in X^`(Rational Field) over Rational Field
>>> pushout(GP\_ZZt(QQ), GP\_QQ(ZZ))
Generalized Polynomial Ring in X^`(Univariate Polynomial Ring in t over Rational Field) over Rational Field
>>> pushout(GP\_ZZt(ZZ), GP\_QQ(QQ))
Generalized Polynomial Ring in X^`(Univariate Polynomial Ring in t over Rational Field) over Rational Field
>>> pushout(GP\_ZZt(ZZ\([a,b]\)), GP\_QQ(ZZ\([c,d]\)))
Traceback (most recent call last):
... CoercionException: ('Ambiguous Base Extension', ...)
>>> pushout(GP\_ZZt(ZZ\([a,b]\)), GP\_QQ(ZZ\([b,c]\)))
```

Some tests with Cartesian products:

```python
\texttt{sage: from sage.sets.cartesian_product import CartesianProduct}
\texttt{sage: A = CartesianProduct((ZZ\('[x]\), QQ\('[y]\), QQ\('[z]\)), ...:
\quad Sets().CartesianProducts())
\texttt{sage: B = CartesianProduct((ZZ\('[x]\), ZZ\('[y]\), ZZ\('[t]\), ZZ\('[z]\)), ...:
\quad Sets().CartesianProducts())
\texttt{sage: A.construction()}
(\texttt{The cartesian_product functorial construction,}
 (Univariate Polynomial Ring in x over Integer Ring,}
 Univariate Polynomial Ring in y over Rational Field,}
 Univariate Polynomial Ring in z over Rational Field))
\texttt{sage: pushout(A, B)}
The Cartesian product of
(Union of Polynomial Ring in x over Integer Ring,}
Univariate Polynomial Ring in y over Rational Field,}
Univariate Polynomial Ring in z over}
Univariate Polynomial Ring in t over Rational Field)
```

(continues on next page)
sage: pushout(ZZ, cartesian_product([ZZ, QQ]))
Traceback (most recent call last):
... 
CoercionException: 'NoneType' object is not iterable

>>> from sage.all import *
>>> from sage.sets.cartesian_product import CartesianProduct
>>> A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z'])),
... Sets().CartesianProducts()
>>> B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z'])),
... Sets().CartesianProducts()
>>> A.construction()
(The cartesian_product functorial construction,
(Univariate Polynomial Ring in x over Integer Ring,
Univariate Polynomial Ring in y over Rational Field,
Univariate Polynomial Ring in z over Rational Field))
>>> pushout(A, B)
The Cartesian product of
(Univariate Polynomial Ring in x over Integer Ring,
Univariate Polynomial Ring in y over Rational Field,
Univariate Polynomial Ring in z over Rational Field)
>>> pushout(ZZ, cartesian_product([ZZ, QQ]))
Traceback (most recent call last):
... 
CoercionException: 'NoneType' object is not iterable

sage: from sage.categories.pushout import PolynomialFunctor
sage: from sage.sets.cartesian_product import CartesianProduct
sage: class CartesianProductPoly(CartesianProduct):
.....:     def __init__(self, polynomial_rings):
.....:         self.polynomial_rings = polynomial_rings,
.....:         super().__init__(sort, Sets().CartesianProducts())
.....:     def vars(self):
.....:         return tuple(P.variable_name() for P in self.cartesian_factors())
.....:     def _pushout_(self, other):
.....:         if isinstance(other, CartesianProductPoly):
.....:             s_vars = self.vars()
.....:             o_vars = other.vars()
.....:             if s_vars == o_vars:
.....:                 return pushout(CartesianProductPoly(
.....:                     self.cartesian_factors() +
.....:                     tuple(f for f in other.cartesian_factors()
.....:                         if f.variable_name() not in s_vars)),
.....:                     CartesianProductPoly(
.....:                     other.cartesian_factors() +
.....:                     tuple(f for f in self.cartesian_factors()
.....:                         if f.variable_name() not in o_vars))
.....:             C = other.construction()
.....:             if C is None:
.....:                 return
.....:             elif isinstance(C[0], PolynomialFunctor):
.....:                 return pushout(self, CartesianProductPoly((other,)))

...
>>> from sage.all import *
>>> from sage.categories.pushout import PolynomialFunctor
>>> from sage.sets.cartesian_product import CartesianProduct

>>> class CartesianProductPoly(CartesianProduct):
...     def __init__(self, polynomial_rings):
...         sort = sorted(polynomial_rings,
...             key=lambda P: P.variable_name())
...         super().__init__(sort, Sets().CartesianProducts())
...     def vars(self):
...         return tuple(P.variable_name()
...             for P in self.cartesian_factors())
...     def _pushout_(self, other):
...         if isinstance(other, CartesianProductPoly):
...             s_vars = self.vars()
...             o_vars = other.vars()
...             if s_vars == o_vars:
...                 return
...             return
...             pushout(CartesianProductPoly(
...                 self.cartesian_factors() +
...                 tuple(f for f in other.cartesian_factors()
...                     if f.variable_name() not in s_vars)),
...                 CartesianProductPoly(
...                 other.cartesian_factors() +
...                 tuple(f for f in self.cartesian_factors()
...                     if f.variable_name() not in o_vars))
...     C = other.construction()
...     if C is None:
...         return
...     elif isinstance(C[Integer(0)], PolynomialFunctor):
...         return
...         pushout(self, CartesianProductPoly((other,)))

sage: pushout(CartesianProductPoly((ZZ['x'],)),
           CartesianProductPoly((ZZ['y'],)))
The Cartesian product of
(Univariate Polynomial Ring in x over Integer Ring,
 Univariate Polynomial Ring in y over Integer Ring)
sage: pushout(CartesianProductPoly((ZZ['x'], ZZ['y'])),
           CartesianProductPoly((ZZ['x'], ZZ['z'])))
The Cartesian product of
(Univariate Polynomial Ring in x over Integer Ring,
 Univariate Polynomial Ring in y over Integer Ring,
 Univariate Polynomial Ring in z over Integer Ring)
sage: pushout(CartesianProductPoly((QQ['a,b']()['x'], QQ['y'])),
           CartesianProductPoly((ZZ['b,c']()['x'], SR['z'])))
The Cartesian product of
(Univariate Polynomial Ring in x over
 Multivariate Polynomial Ring in a, b, c over Rational Field,
 Univariate Polynomial Ring in y over Rational Field,
 Univariate Polynomial Ring in z over Symbolic Ring)

>>> from sage.all import *
>>> pushout(CartesianProductPoly((ZZ['x'][],)),
           CartesianProductPoly((ZZ['y'][],)))
The Cartesian product of
(Univariate Polynomial Ring in x over Integer Ring,
 Univariate Polynomial Ring in y over Integer Ring)

3.4. Coercion via construction functors 245
The Cartesian product of\(\text{Univariate Polynomial Ring in } x \text{ over Integer Ring},\)\(\text{Univariate Polynomial Ring in } y \text{ over Integer Ring},\)
\(\text{Univariate Polynomial Ring in } z \text{ over Integer Ring})\)

\[\text{CartesianProductPolys}((\text{QQ}[a,b][x], \text{QQ}[y])),\]
\needs sage.symbolic
\[\text{CartesianProductPolys}((\text{ZZ}[b,c][x], \text{SR}[z]))\)

The Cartesian product of\(\text{Univariate Polynomial Ring in } x \text{ over }\)
\(\text{Multivariate Polynomial Ring in } a, b, c \text{ over Rational Field},\)
\(\text{Univariate Polynomial Ring in } y \text{ over Rational Field},\)
\(\text{Multivariate Polynomial Ring in } z \text{ over Symbolic Ring})\)

sage: pushout(CartesianProductPolys((ZZ['x'],)), ZZ['y'])
The Cartesian product of\(\text{Univariate Polynomial Ring in } x \text{ over Integer Ring},\)
\(\text{Univariate Polynomial Ring in } y \text{ over Integer Ring})\)

sage: pushout(QQ['b,c'][y], CartesianProductPolys((ZZ['a,b'][x],)))
The Cartesian product of\(\text{Univariate Polynomial Ring in } x \text{ over }\)
\(\text{Multivariate Polynomial Ring in } a, b \text{ over Integer Ring},\)
\(\text{Univariate Polynomial Ring in } y \text{ over }\)
\(\text{Multivariate Polynomial Ring in } b, c \text{ over Rational Field})\)

sage: pushout(CartesianProductPolys((ZZ['x'],)), ZZ)
Traceback (most recent call last):
... CoercionException: No common base ("join") found for
The cartesian_product functorial construction(...) and None(Integer Ring):
(Multivariate) functors are incompatible.

sage: pushout(CartesianProductPolys((ZZ['x'],)), ZZ)
Traceback (most recent call last):
... CoercionException: No common base ("join") found for
The cartesian_product functorial construction(...) and None(Integer Ring):
(Multivariate) functors are incompatible.

AUTHORS:

- Robert Bradshaw
sage.categories.pushout.pushout_lattice(R, S)

Given a pair of objects \( R \) and \( S \), try to construct a reasonable object \( Y \) and return maps such that canonically \( R \leftarrow Y \rightarrow S \).

**ALGORITHM:**

This is based on the model that arose from much discussion at Sage Days 4. Going up the tower of constructions of \( R \) and \( S \) (e.g. the reals come from the rationals come from the integers), try to find a common parent, and then try to fill in a lattice with these two towers as sides with the top as the common ancestor and the bottom will be the desired ring.

See the code for a specific worked-out example.

**EXAMPLES:**

```python
sage: from sage.categories.pushout import pushout_lattice
sage: A, B = pushout_lattice(Qp(7), Frac(ZZ['x']))
needs sage.rings.padics
sage: A.codomain()
needs sage.rings.padics
Fraction Field of Univariate Polynomial Ring in x
over 7-adic Field with capped relative precision 20
sage: A.codomain() is B.codomain()
needs sage.rings.padics
True
sage: A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], 3, 3))
needs sage.modules
sage: B
Identity endomorphism of Full MatrixSpace of 3 by 3 dense matrices
over Power Series Ring in x over Integer Ring
```

**AUTHOR:**

- Robert Bradshaw
sage.categories.pushout.type_to_parent\((P)\)

An auxiliary function that is used in \texttt{pushout()}.

INPUT:

A type

OUTPUT:

A Sage parent structure corresponding to the given type
4.1 Group, ring, etc. actions on objects

The terminology and notation used is suggestive of groups acting on sets, but this framework can be used for modules, algebras, etc.

A group action $G \times S \to S$ is a functor from $G$ to $\text{Sets}$.

**Warning:** An $\text{Action}$ object only keeps a weak reference to the underlying set which is acted upon. This decision was made in Issue #715 in order to allow garbage collection within the coercion framework (this is where actions are mainly used) and avoid memory leaks.

```sage
sage: from sage.categories.action import Action
gsage: class P:
    pass
gsage: A = Action(P(),P())
gsage: import gc
gsage: _ = gc.collect()
gsage: A
<repr(<sage.categories.action.Action at 0x...>) failed: RuntimeError: This action acted on a set that became garbage collected>
```

To avoid garbage collection of the underlying set, it is sufficient to create a strong reference to it before the action is created.

```sage
sage: _ = gc.collect()
sage: from sage.categories.action import Action
sage: class P: pass
sage: q = P()
sage: A = Action(P(),q)
sage: gc.collect()
0
sage: A
Left action by <__main__.P ... at ...> on __main__.P ... at ...
```

```sage
sage: from sage.all import *
sage: from sage.categories.action import Action
sage: class P:
    pass
sage: A = Action(P(),P())
sage: import gc
sage: _ = gc.collect()
sage: A
<repr(<sage.categories.action.Action at 0x...>) failed: RuntimeError: This action acted on a set that became garbage collected>
```
```
>>> from sage.all import *
>>> _ = gc.collect()
>>> from sage.categories.action import Action
>>> class P: pass
>>> q = P()
>>> A = Action(P(),q)
>>> gc.collect()
0
>>> A
Left action by <__main__.P ... at ...> on <__main__.P ... at ...>
```

**AUTHOR:**

- Robert Bradshaw: initial version

**class sage.categories.action.Action**

**Bases:** Functo

The action of $G$ on $S$.

**INPUT:**

- $G$ – a parent or Python type
- $S$ – a parent or Python type
- `is_left` – (boolean, default: True) whether elements of $G$ are on the left
- `op` – (default: None) operation. This is not used by `Action` itself, but other classes may use it

$G$

$\text{act}(g, x)$

This is a consistent interface for acting on $x$ by $g$, regardless of whether it's a left or right action.

If needed, $g$ and $x$ are converted to the correct parent.

**EXAMPLES:**

```
sage: R.<x> = ZZ []
sage: from sage.structure.coerce_actions import IntegerMulAction
sage: A = IntegerMulAction(ZZ, R, True)  # Left action
sage: A.act(5, x)
5\times
sage: A.act(int(5), x)
5\times
```

```
>>> from sage.all import *
>>> R = ZZ ['x']; (x,) = R._first_ngens(1)
>>> from sage.structure.coerce_actions import IntegerMulAction
>>> A = IntegerMulAction(ZZ, R, True)  # Left action
>>> A.act(Integer(5), x)
5\times
>>> A.act(int(Integer(5)), x)
```

(continues on next page)
A = IntegerMulAction(ZZ, R, False)  # Right action
A.act(Integer(5), x)
5*x
A.act(int(Integer(5)), x)
5*x

actor()
codomain()
domain()
is_left()
left_domain()
op
operation()
right_domain()

class sage.categories.action.ActionEndomorphism

Bases: Morphism

The endomorphism defined by the action of one element.

EXAMPLES:

sage: A = ZZ['x'].get_action(QQ, self_on_left=False, op=operator.mul)
sage: A
Left scalar multiplication by Rational Field
on Univariate Polynomial Ring in x over Integer Ring
sage: A(1/2)
Action of 1/2 on Univariate Polynomial Ring in x over Integer Ring
under Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Ring.

class sage.categories.action.InverseAction

Bases: Action

An action that acts as the inverse of the given action.

EXAMPLES:
sage: V = QQ^3
˓→# needs sage.modules
sage: v = V((1, 2, 3))
˓→# needs sage.modules
sage: cm = get_coercion_model()

sage: # needs sage.modules
sage: a = cm.get_action(V, QQ, operator.mul)
sage: a
Right scalar multiplication by Rational Field
  on Vector space of dimension 3 over Rational Field
sage: ~a
Right inverse action by Rational Field
  on Vector space of dimension 3 over Rational Field
sage: (~a)(v, 1/3)
  (3, 6, 9)

sage: # needs sage.modules
sage: b = cm.get_action(QQ, V, operator.mul)
sage: b
Left scalar multiplication by Rational Field
  on Vector space of dimension 3 over Rational Field
sage: ~b
Left inverse action by Rational Field
  on Vector space of dimension 3 over Rational Field
sage: (~b)(1/3, v)
  (3, 6, 9)

sage: c = cm.get_action(ZZ, list, operator.mul)
sage: c
Left action by Integer Ring on <... 'list'>
sage: ~c
Traceback (most recent call last):
  ...
TypeError: no inverse defined for Left action by Integer Ring on <... 'list'>

>>> from sage.all import *
>>> V = QQ**Integer(3) ˓→# needs sage.modules
>>> v = V((Integer(1), Integer(2), Integer(3))) ˓→# needs sage.modules
>>> cm = get_coercion_model()

>>> # needs sage.modules
>>> a = cm.get_action(V, QQ, operator.mul)
>>> a
Right scalar multiplication by Rational Field
  on Vector space of dimension 3 over Rational Field
>>> ~a
Right inverse action by Rational Field
  on Vector space of dimension 3 over Rational Field
>>> (~a)(v, Integer(1)/Integer(3))
  (3, 6, 9)

>>> # needs sage.modules
>>> b = cm.get_action(QQ, V, operator.mul)
>>> b
(continues on next page)
Left scalar multiplication by Rational Field
on Vector space of dimension 3 over Rational Field
>>> ~b
Left inverse action by Rational Field
on Vector space of dimension 3 over Rational Field
>>> (~b)(Integer(1)/Integer(3), v)
(3, 6, 9)
>>> c = cm.get_action(ZZ, list, operator.mul)
>>> c
Left action by Integer Ring on <... 'list'>
>>> ~c
Traceback (most recent call last):
... TypeError: no inverse defined for Left action by Integer Ring on <... 'list'>
codomain()

class sage.categories.action.PrecomposedAction
Bases: Action

A precomposed action first applies given maps, and then applying an action to the return values of the maps.

EXAMPLES:

We demonstrate that an example discussed on Issue #14711 did not become a problem:

```python
sage: # needs sage.libs.flint sage.modular
sage: E = ModularSymbols(11).2
sage: s = E.modular_symbol_rep()
sage: del E, s
sage: import gc
sage: _ = gc.collect()
sage: E = ModularSymbols(11).2
sage: v = E.manin_symbol_rep()
sage: c,x = v[0]
sage: y = x.modular_symbol_rep()
sage: coercion_model.get_action(QQ, parent(y), op=operator.mul)
Left scalar multiplication by Rational Field
on Abelian Group of all Formal Finite Sums over Rational Field
with precomposition on right by Coercion map:
  From: Abelian Group of all Formal Finite Sums over Integer Ring
  To:   Abelian Group of all Formal Finite Sums over Rational Field
```

```python
>>> from sage.all import *
>>> # needs sage.libs.flint sage.modular
>>> E = ModularSymbols(Integer(11)).gen(2)
>>> s = E.modular_symbol_rep()
>>> del E, s
>>> import gc
>>> _ = gc.collect()
>>> E = ModularSymbols(Integer(11)).gen(2)
>>> v = E.manin_symbol_rep()
>>> c,x = v[Integer(0)]
>>> y = x.modular_symbol_rep()
>>> coercion_model.get_action(QQ, parent(y), op=operator.mul)
Left scalar multiplication by Rational Field
```

(continues on next page)
on Abelian Group of all Formal Finite Sums over Rational Field
with precomposition on right by Coercion map:
From: Abelian Group of all Formal Finite Sums over Integer Ring
To: Abelian Group of all Formal Finite Sums over Rational Field

codomain()
domain()

left_precomposition
   The left map to precompose with, or None if there is no left precomposition map.

right_precomposition
   The right map to precompose with, or None if there is no right precomposition map.

4.2 Additive groups

class sage.categories.additive_groups.AdditiveGroups(base_category)
   Bases: CategoryWithAxiom_singleton
   The category of additive groups.
   An additive group is a set with an internal binary operation + which is associative, admits a zero, and where every
element can be negated.

   EXAMPLES:

   sage: from sage.categories.additive_groups import AdditiveGroups
   sage: from sage.categories.additive_monoids import AdditiveMonoids
   sage: AdditiveGroups()
   Category of additive groups
   sage: AdditiveGroups().super_categories()
   [Category of additive inverse additive unital additive magmas,
   Category of additive monoids]
   sage: AdditiveGroups().all_super_categories()
   [Category of additive groups,
   Category of additive inverse additive unital additive magmas,
   Category of additive monoids,
   Category of additive unital additive magmas,
   Category of additive semigroups,
   Category of additive magmas,
   Category of sets,
   Category of sets with partial maps,
   Category of objects]
   sage: AdditiveGroups().axioms()
   frozenset({'AdditiveAssociative', 'AdditiveInverse', 'AdditiveUnital'})
   sage: AdditiveGroups() is AdditiveMonoids().AdditiveInverse()
   True

   >>> from sage.all import *
   >>> from sage.categories.additive_groups import AdditiveGroups
   >>> from sage.categories.additive_monoids import AdditiveMonoids
   >>> AdditiveGroups()
Category of additive groups

```python
>>> AdditiveGroups().super_categories()
[Category of additive inverse additive unital additive magmas,
 Category of additive monoids]
```

```python
>>> AdditiveGroups().all_super_categories()
[Category of additive groups,
 Category of additive inverse additive unital additive magmas,
 Category of additive monoids,
 Category of additive unital additive magmas,
 Category of additive semigroups,
 Category of additive magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

```python
>>> AdditiveGroups().axioms()
frozenset({'AdditiveAssociative', 'AdditiveInverse', 'AdditiveUnital'})
```

```python
>>> AdditiveGroups() is AdditiveMonoids().AdditiveInverse()
True
```

AdditiveCommutative

alias of `CommutativeAdditiveGroups`

class `Algebras`(*category, *args)

    Bases: `AlgebrasCategory`

class `ParentMethods`

    Bases: `object`

    `group()`

    Return the underlying group of the group algebra.

    EXAMPLES:

    ```python
    sage: GroupAlgebras(QQ).example(GL(3, GF(11))).group()  # needs sage.groups sage.modules
    General Linear Group of degree 3 over Finite Field of size 11
    sage: SymmetricGroup(10).algebra(QQ).group()  # needs sage.combinat sage.groups sage.modules
    Symmetric group of order 10! as a permutation group
    ```

```python
>>> from sage.all import *
```  
```python
>>> GroupAlgebras(QQ).example(GL(Integer(3), GF(Integer(11)))).group()  # needs sage.groups sage.modules
General Linear Group of degree 3 over Finite Field of size 11
>>> SymmetricGroup(Integer(10)).algebra(QQ).group()  # needs sage.combinat sage.groups sage.modules
Symmetric group of order 10! as a permutation group
```  

class `Finite`(*`base_category`)

    Bases: `CategoryWithAxiomSingleton`

class `Algebras`(*`category`, *`args`)

    Bases: `AlgebrasCategory`

class `ParentMethods`

    Bases: `object`

4.2. Additive groups 255
extra_super_categories()

Implement Maschke's theorem.

In characteristic 0 all finite group algebras are semisimple.

EXAMPLES:

```
sage: FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
sage: FiniteGroups().Algebras(FiniteField(7)).is_subcategory(Algebras(FiniteField(7)).Semisimple())
False
sage: FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False

sage: Cat = CommutativeAdditiveGroups().Finite()
sage: Cat.Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
sage: Cat.Algebras(GF(7)).is_subcategory(Algebras(GF(7)).Semisimple())
False
sage: Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False
```

>>> from sage.all import *

>>> FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
>>> FiniteGroups().Algebras(FiniteField(Integer(7))).is_subcategory(Algebras(FiniteField(Integer(7))).Semisimple())
False
>>> FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
>>> FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False

>>> Cat = CommutativeAdditiveGroups().Finite()

>>> Cat.Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
>>> Cat.Algebras(GF(Integer(7))).is_subcategory(Algebras(GF(Integer(7))).Semisimple())
False
>>> Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
>>> Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False
4.3 Additive magmas

class sage.categories.additive_magmas.AdditiveMagmas
Bases: Category_singleton

The category of additive magmas.

An additive magma is a set endowed with a binary operation +.

EXAMPLES:

```python
sage: AdditiveMagmas()
Category of additive magmas
sage: AdditiveMagmas().super_categories()
[Category of sets]
sage: AdditiveMagmas().all_super_categories()
[Category of additive magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

>>> from sage.all import *
>>> AdditiveMagmas()
Category of additive magmas
>>> AdditiveMagmas().super_categories()
[Category of sets]
>>> AdditiveMagmas().all_super_categories()
[Category of additive magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

The following axioms are defined by this category:

```python
sage: AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
sage: AdditiveMagmas().AdditiveUnital()
Category of additive unital additive magmas
sage: AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
sage: C = AdditiveMagmas().AdditiveAssociative().AdditiveCommutative(); C
Category of commutative additive semigroups
sage: C.AdditiveUnital()
Category of commutative additive monoids
sage: C.AdditiveUnital().AdditiveInverse()
Category of commutative additive groups

>>> from sage.all import *
>>> AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
>>> AdditiveMagmas().AdditiveUnital()
Category of additive unital additive magmas
>>> AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
>>> AdditiveMagmas().AdditiveUnital().AdditiveInverse()
```

(continues on next page)
Category of additive inverse additive unital additive magmas

```python
>>> C = AdditiveMagmas().AdditiveAssociative().AdditiveCommutative(); C
```

Category of commutative additive semigroups

```python
>>> C.AdditiveUnital()
```

Category of commutative additive monoids

```python
>>> C.AdditiveUnital().AdditiveInverse()
```

Category of commutative additive groups

AdditiveAssociative

alias of AdditiveSemigroups

class AdditiveCommutative(base_category)

    Bases: CategoryWithAxiom_singleton

class Algebras(category, *args)

    Bases: AlgebrasCategory

    extra_super_categories()

    Implement the fact that the algebra of a commutative additive magmas is commutative.

    EXAMPLES:

    ```python
    sage: C = AdditiveMagmas().AdditiveCommutative().Algebras(QQ)
    sage: C.extra_super_categories()
    [Category of commutative magmas]
    sage: C.super_categories()
    [Category of additive magma algebras over Rational Field, Category of commutative magmas]
    ```

>>> from sage.all import *

```python
>>> C = AdditiveMagmas().AdditiveCommutative().Algebras(QQ)
>>> C.extra_super_categories()
[Category of commutative magmas]
>>> C.super_categories()
[Category of additive magma algebras over Rational Field, Category of commutative magmas]
```
class AdditiveUnital (base_category)
    Bases: CategoryWithAxiom_singleton

class AdditiveInverse (base_category)
    Bases: CategoryWithAxiom_singleton

class CartesianProducts (category, *args)
    Bases: CartesianProductsCategory

class ElementMethods
    Bases: object

    extra_super_categories()
    Implement the fact that a Cartesian product of additive magmas with inverses is an additive magma with inverse.

    EXAMPLES:

    sage: C = AdditiveMagmas().AdditiveUnital().AdditiveInverse().CartesianProducts()
    sage: C.extra_super_categories()
    [Category of additive inverse additive unital additive magmas]
    sage: sorted(C.axioms())
    ['AdditiveInverse', 'AdditiveUnital']

    >>> from sage.all import *
    >>> C = AdditiveMagmas().AdditiveUnital().AdditiveInverse().CartesianProducts()
    >>> C.extra_super_categories()
    [Category of additive inverse additive unital additive magmas]
    >>> sorted(C.axioms())
    ['AdditiveInverse', 'AdditiveUnital']

class Algebras (category, *args)
    Bases: AlgebrasCategory

class ParentMethods
    Bases: object

    one_basis()
    Return the zero of this additive magma, which index the one of this algebra, as per AlgebrasWithBasis.ParentMethods.one_basis().

    EXAMPLES:

    sage: # needs sage.modules
    sage: S = CommutativeAdditiveMonoids().example(); S
    An example of a commutative monoid:
    the free commutative monoid generated by ('a', 'b', 'c', 'd')
    sage: A = S.algebra(ZZ)
    sage: A.one_basis()
sage: A.one()
B[0]
sage: A(3)
3*B[0]

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid:
the free commutative monoid generated by ('a', 'b', 'c', 'd')
>>> A = S.algebra(ZZ)
>>> A.one_basis()
0
>>> A.one()
B[0]
>>> A(Integer(3))
3*B[0]
```

**extra_super_categories()**

**EXAMPLES:**

```python
sage: C = AdditiveMagmas().AdditiveUnital().Algebras(QQ)
sage: C.extra_super_categories()
[Category of unital magmas]
sage: C.super_categories()
[Category of unital algebras with basis over Rational Field,
 Category of additive magma algebras over Rational Field]
```

```python
>>> from sage.all import *
>>> C = AdditiveMagmas().AdditiveUnital().Algebras(QQ)
>>> C.extra_super_categories()
[Category of unital magmas]
>>> C.super_categories()
[Category of unital algebras with basis over Rational Field,
 Category of additive magma algebras over Rational Field]
```

**class CartesianProducts**

**Bases:** *CartesianProductsCategory*

**class ParentMethods**

**Bases:** object

**zero()**

Returns the zero of this group

**EXAMPLES:**

```python
sage: GF(8, 'x').cartesian_product(GF(5)).zero()  # needs sage.rings.finite_rings
(0, 0)
```
extra_super_categories()
Implement the fact that a Cartesian product of unital additive magmas is a unital additive magma.

EXAMPLES:

```python
sage: C = AdditiveMagmas().AdditiveUnital().CartesianProducts()
sage: C.extra_super_categories()
[Category of additive unital additive magmas]
sage: C.axioms()
frozenset({'AdditiveUnital'})
```

class ElementMethods
Bases: object

class Homsets (category, *args)
Bases: HomsetsCategory

class ParentMethods
Bases: object

```python
zero()
```

EXAMPLES:

```python
sage: R = QQ['x']
sage: H = Hom(ZZ, R, AdditiveMagmas().AdditiveUnital())
sage: f = H.zero()
sage: f
Generic morphism:
    From: Integer Ring
    To:    Univariate Polynomial Ring in x over Rational Field
sage: f(3)
0
sage: f(3) is R.zero()
True
```

```python
>>> from sage.all import *
>>> GF(Integer(8), 'x').cartesian_product(GF(Integer(5))).zero() → # needs sage.rings.finite_rings
(0, 0)
```
extra_super_categories()

Implement the fact that a homset between two unital additive magmas is a unital additive magma.

EXAMPLES:

```
sage: AdditiveMagmas().AdditiveUnital().Homsets().extra_super_categories()
[Category of additive unital additive magmas]
sage: AdditiveMagmas().AdditiveUnital().Homsets().super_categories()
[Category of additive unital additive magmas, Category of homsets]
```

class ParentMethods

Bases: object

**is_empty()**

Return whether this set is empty.

Since this set is an additive magma it has a zero element and hence is not empty. This method thus always returns `False`.

EXAMPLES:

```
sage: # needs sage.modules
sage: A = AdditiveAbelianGroup([3, 3])
sage: A in AdditiveMagmas()
True
sage: A.is_empty()
False
sage: B = CommutativeAdditiveMonoids().example()
sage: B.is_empty()
False
```

```
>>> from sage.all import *

>>> AdditiveMagmas().AdditiveUnital().Homsets().extra_super_categories()
[Category of additive unital additive magmas]
```

```
>>> AdditiveMagmas().AdditiveUnital().Homsets().super_categories()
[Category of additive unital additive magmas, Category of homsets]
```

```
>>> from sage.all import *

>>> A = AdditiveAbelianGroup([Integer(3), Integer(3)])

>>> A in AdditiveMagmas()
True

>>> A.is_empty()
False

>>> B = CommutativeAdditiveMonoids().example()

>>> B.is_empty()
False
```
zero()

Return the zero of this additive magma, that is the unique neutral element for +.

The default implementation is to coerce 0 into self.

It is recommended to override this method because the coercion from the integers:
• is not always meaningful (except for 0), and
• often uses self.zero() otherwise.

EXAMPLES:

```
sage: S = CommutativeAdditiveMonoids().example()
sage: S.zero()
0
```

```
>>> from sage.all import *

>>> S = CommutativeAdditiveMonoids().example()
>>> S.zero()
0
```

class SubcategoryMethods

Bases: object

AdditiveInverse()

Return the full subcategory of the additive inverse objects of self.

An inverse additive magma is a unital additive magma such that every element admits both an additive inverse on the left and on the right. Such an additive magma is also called an additive loop.

See also:

Wikipedia article Inverse_element, Wikipedia article Quasigroup

EXAMPLES:

```
sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
sage: from sage.categories.additive_monoids import AdditiveMonoids
sage: AdditiveMonoids().AdditiveInverse()
Category of additive groups
```

```
>>> from sage.all import *

>>> AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
>>> from sage.categories.additive_monoids import AdditiveMonoids
>>> AdditiveMonoids().AdditiveInverse()
Category of additive groups
```

class WithRealizations (category, *args)

Bases: WithRealizationsCategory

class ParentMethods

Bases: object

zero()

Return the zero of this unital additive magma.

This default implementation returns the zero of the realization of self given by a_realization().

4.3. Additive magmas
EXAMPLES:

```python
sage: A = Sets().WithRealizations().example(); A
#---
needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
sage: A.zero.__module__
#---
needs sage.modules
'sage.categories.additive_magmas'
sage: A.zero()
#---
needs sage.modules
0
```

```python
>>> from sage.all import *

>>> A = Sets().WithRealizations().example(); A
#---
needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
>>> A.zero.__module__
#---
needs sage.modules
'sage.categories.additive_magmas'
>>> A.zero()
#---
needs sage.modules
0
```

**additional_structure()**

Return whether `self` is a structure category.

See also:

`Category.additional_structure()`

The category of unital additive magmas defines the zero as additional structure, and this zero shall be preserved by morphisms.

EXAMPLES:

```python
sage: AdditiveMagmas().AdditiveUnital().additional_structure()
Category of additive unital additive magmas
```

```python
>>> from sage.all import *

>>> AdditiveMagmas().AdditiveUnital().additional_structure()
Category of additive unital additive magmas
```

**class Algebras(category, *args)**

Bases: `AlgebrasCategory`

**class ParentMethods**

Bases: `object`

**algebra_generators()**

The generators of this algebra, as per `MagmaicAlgebras.PARENTMethods.algebra_generators()`.

They correspond to the generators of the additive semigroup.

EXAMPLES:

```python
sage: S = CommutativeAdditiveSemigroups().example(); S
An example of a commutative semigroup:
(continues on next page)
```

(continues on next page)
the free commutative semigroup generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(QQ)
# needs sage.modules
sage: A.algebra_generators()
# needs sage.modules
Family (B[a], B[b], B[c], B[d])

Todo: This doctest does not actually test this method, but rather the method of the same name for AdditiveSemigroups. Find a better doctest!

product_on_basis\((g_1, g_2)\)

Product, on basis elements, as per MagmaticAlgebras.WithBasis.ParentMethods.product_on_basis().

The product of two basis elements is induced by the addition of the corresponding elements of the group.

EXAMPLES:

Todo: This doctest does not actually test this method, but rather the method of the same name for
**AdditiveSemigroups. Find a better doctest!**

### extra_super_categories()

**EXAMPLES:**

```text
sage: AdditiveMagmas().Algebras(QQ).extra_super_categories()
[Category of magmatic algebras with basis over Rational Field]

sage: AdditiveMagmas().Algebras(QQ).super_categories()
[Category of magmatic algebras with basis over Rational Field,
 Category of set algebras over Rational Field]
```

```python
>>> from sage.all import *

>>> AdditiveMagmas().Algebras(QQ).extra_super_categories()
[Category of magmatic algebras with basis over Rational Field]

>>> AdditiveMagmas().Algebras(QQ).super_categories()
[Category of magmatic algebras with basis over Rational Field,
 Category of set algebras over Rational Field]
```

### class CartesianProducts (category, *args)

**Bases:** `CartesianProductsCategory`

**class ElementMethods**

**Bases:** `object`

**extra_super_categories()**

Implement the fact that a Cartesian product of additive magmas is an additive magma.

**EXAMPLES:**

```text
sage: C = AdditiveMagmas().CartesianProducts()
sage: C.extra_super_categories()
[Category of additive magmas]

sage: C.super_categories()
[Category of additive magmas, Category of Cartesian products of sets]

sage: C.axioms()
frozenset()
```

```python
>>> from sage.all import *

>>> C = AdditiveMagmas().CartesianProducts()

>>> C.extra_super_categories()
[Category of additive magmas]

>>> C.super_categories()
[Category of additive magmas, Category of Cartesian products of sets]

>>> C.axioms()
frozenset()
```

### class ElementMethods

**Bases:** `object`

### class Homsets (category, *args)

**Bases:** `HomsetsCategory`

**extra_super_categories()**

Implement the fact that a homset between two magmas is a magma.
EXAMPLES:

```python
sage: AdditiveMagmas().Homsets().extra_super_categories()
[Category of additive magmas]
sage: AdditiveMagmas().Homsets().super_categories()
[Category of additive magmas, Category of homsets]
```

```
>>> from sage.all import *

AdditiveMagmas().Homsets().extra_super_categories()
[Category of additive magmas]
AdditiveMagmas().Homsets().super_categories()
[Category of additive magmas, Category of homsets]
```

class ParentMethods

Bases: object

```
adition_table (names='letters', elements=None)
```

Return a table describing the addition operation.

**Note:** The order of the elements in the row and column headings is equal to the order given by the table's `column_keys()` method. The association can also be retrieved with the `translation()` method.

**INPUT:**

- `names` — the type of names used:
  - 'letters' — lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by `column_keys()`, padded to a common width with leading 'a's.
  - 'digits' — base 10 representation of the elements' positions in the list given by `column_keys()`, padded to a common width with leading zeros.
  - 'elements' — the string representations of the elements themselves.
  - a list — a list of strings, where the length of the list equals the number of elements.
- `elements` — (default: None) A list of elements of the additive magma, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering the `S.list` method returns. Or the `elements` can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.

**OUTPUT:**

The addition table as an object of the class `OperationTable` which defines several methods for manipulating and displaying the table. See the documentation there for full details to supplement the documentation here.

**EXAMPLES:**

All that is required is that an algebraic structure has an addition defined. The default is to represent elements as lowercase ASCII letters.

```python
sage: R = IntegerModRing(5)
sage: R.addition_table()

a b c d e
+-----------
a| a b c d e
b| b c d e a
----------
c| c d e a b
(continues on next page)
```

4.3. Additive magmas
The `names` argument allows displaying the elements in different ways. Requesting `elements` will use the representation of the elements of the set. Requesting `digits` will include leading zeros as padding.

```python
sage: P = R.addition_table(names='elements'); P
+-------------------
<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```

```python
sage: T = R.addition_table(names='digits'); T
+-------------------
<table>
<thead>
<tr>
<th>00 01 02 03 04 05 06 07 08 09 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>02</td>
</tr>
<tr>
<td>03</td>
</tr>
<tr>
<td>04</td>
</tr>
<tr>
<td>05</td>
</tr>
<tr>
<td>06</td>
</tr>
<tr>
<td>07</td>
</tr>
<tr>
<td>08</td>
</tr>
<tr>
<td>09</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```
### Specifying the elements in an alternative order can provide more insight into how the operation behaves.

```python
sage: S = IntegerModRing(7)
sage: elts = [0, 3, 6, 2, 5, 1, 4]
sage: S.addition_table(elements=elts)                      #...
+---------------------
<table>
<thead>
<tr>
<th>a  b  c  d  e  f  g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a  c  d  e  f  g  b</td>
</tr>
<tr>
<td>b  d  e  f  g  a  c</td>
</tr>
<tr>
<td>c  e  f  g  a  b  d</td>
</tr>
<tr>
<td>d  f  g  a  b  c  e</td>
</tr>
<tr>
<td>e  g  a  b  c  d  f</td>
</tr>
<tr>
<td>g  a  b  c  d  e  f</td>
</tr>
</tbody>
</table>
```

```python
>>> from sage.all import *
>>>
>>> S = IntegerModRing(Integer(7))
>>> elts = [Integer(0), Integer(3), Integer(6), Integer(2), Integer(5), Integer(1), Integer(4)]
>>> S.addition_table(elements=elts)                      #...
+---------------------
<table>
<thead>
<tr>
<th>a  b  c  d  e  f  g</th>
</tr>
</thead>
<tbody>
<tr>
<td>a  c  d  e  f  g  b</td>
</tr>
<tr>
<td>b  d  e  f  g  a  c</td>
</tr>
<tr>
<td>c  e  f  g  a  b  d</td>
</tr>
<tr>
<td>d  f  g  a  b  c  e</td>
</tr>
<tr>
<td>e  g  a  b  c  d  f</td>
</tr>
<tr>
<td>g  a  b  c  d  e  f</td>
</tr>
</tbody>
</table>
```

---

### 4.3. Additive magmas

(continued from previous page)
The `elements` argument can be used to provide a subset of the elements of the structure. The subset must be closed under the operation. Elements need only be in a form that can be coerced into the set. The `names` argument can also be used to request that the elements be represented with their usual string representation.

```sage
sage: T = IntegerModRing(12)
sage: elts = [0, 3, 6, 9]
sage: T.addition_table(names='elements', elements=elts) # needs sage.modules
+ 0 3 6 9
+----------
0| 0 3 6 9
3| 3 6 9 0
6| 6 9 0 3
9| 9 0 3 6
```

The table returned can be manipulated in various ways. See the documentation for `OperationTable` for more comprehensive documentation.

```sage
sage: # needs sage.modules
elements = [Integer(0), Integer(3), Integer(6), Integer(9)]
sage: T.addition_table(names='elements', elements=elts) # needs sage.modules
+ 0 3 6 9
+----------
0| 0 3 6 9
3| 3 6 9 0
6| 6 9 0 3
9| 9 0 3 6
```

```python
>>> from sage.all import *
>>> from sage.all import *
>>> T = IntegerModRing(Integer(12))
>>> elts = [0, 3, 6, 9]
>>> T.addition_table(names='elements', elements=elts)
+ 0 3 6 9
+----------
0| 0 3 6 9
3| 3 6 9 0
6| 6 9 0 3
9| 9 0 3 6
```
summation \( (x, y) \)

Return the sum of \( x \) and \( y \).

The binary addition operator of this additive magma.

**INPUT:**

- \( x, y \) – elements of this additive magma

**EXAMPLES:**

```
sage: S = CommutativeAdditiveSemigroups().example()
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: S.summation(a, b)
a + b
```

```
>>> from sage.all import *
>>> S = CommutativeAdditiveSemigroups().example()
>>> (a,b,c,d) = S.additive_semigroup_generators()
>>> S.summation(a, b)
a + b
```

A parent in `AdditiveMagmas()` must either implement `summation()` in the parent class or `_add_` in the element class. By default, the addition method on elements \( x._add_(y) \) calls \( S.\text{summation}(x,y) \), and reciprocally.

As a bonus effect, \( S.\text{summation} \) by itself models the binary function from \( S \) to \( S \):

```
sage: bin = S.summation
sage: bin(a,b)
a + b
```

```
>>> from sage.all import *
>>> bin = S.summation
>>> bin(a,b)
a + b
```

Here, \( S.\text{summation} \) is just a bound method. Whenever possible, it is recommended to enrich \( S.\text{summation} \) with extra mathematical structure. Lazy attributes can come handy for this.

**Todo:** Add an example.
summation_from_element_class_add \((x, y)\)

Return the sum of \(x\) and \(y\).

The binary addition operator of this additive magma.

**INPUT:**

- \(x, y\) – elements of this additive magma

**EXAMPLES:**

```python
sage: S = CommutativeAdditiveSemigroups().example()

sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: Ssummation(a, b)
a + b
```

```python
>>> from sage.all import *

>>> S = CommutativeAdditiveSemigroups().example()

>>> (a,b,c,d) = S.additive_semigroup_generators()

>>> Ssummation(a, b)
a + b
```

A parent in `AdditiveMagmas()` must either implement `summation()` in the parent class or `_add_` in the element class. By default, the addition method on elements \(x._add_(y)\) calls \(S.\) `summation(x,y)` and reciprocally.

As a bonus effect, \(S.\) `summation` by itself models the binary function from \(S\) to \(S\):

```python
sage: bin = Ssummation

sage: bin(a,b)
a + b
```

```python
>>> from sage.all import *

>>> bin = Ssummation

>>> bin(a,b)
a + b
```

Here, \(S.\) `summation` is just a bound method. Whenever possible, it is recommended to enrich \(S.\) `summation` with extra mathematical structure. Lazy attributes can come handy for this.

**Todo:** Add an example.

class SubcategoryMethods
    Bases: object

AdditiveAssociative()

Return the full subcategory of the additive associative objects of \(self\).

An additive magma \(M\) is associative if, for all \(x, y, z \in M\),

\[
x + (y + z) = (x + y) + z
\]

See also:

Wikipedia article Associative_property

**EXAMPLES:**
AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups

AdditiveCommutative()

Return the full subcategory of the commutative objects of self.
An additive magma $M$ is commutative if, for all $x, y \in M$,
\[ x + y = y + x \]

See also:
Wikipedia article Commutative_property

EXAMPLES:

```sage
sage: AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
sage: C = AdditiveMagmas().AdditiveAssociative().AdditiveUnital()
sage: C.AdditiveCommutative()
Category of commutative additive monoids
sage: C.AdditiveCommutative() is CommutativeAdditiveMonoids()
True
```

AdditiveUnital()

Return the subcategory of the unital objects of self.
An additive magma $M$ is unital if it admits an element 0, called neutral element, such that for all $x \in M$,
\[ 0 + x = x + 0 = x \]

This element is necessarily unique, and should be provided as $M.zero()$.

See also:
Wikipedia article Unital_magma#unital

EXAMPLES:

```sage
sage: AdditiveMagmas().AdditiveUnital()
Category of additive unital additive magmas
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: AdditiveSemigroups().AdditiveUnital()
```
(continues on next page)
Category Framework, Release 10.4

### 4.4 Additive monoids

**class** `sage.categories.additive_monoids.AdditiveMonoids`(base_category)

**Bases:** `CategoryWithAxiom_singleton`

The category of additive monoids.

An additive monoid is a unital additive semigroup, that is a set endowed with a binary operation $+$ which is associative and admits a zero (see Wikipedia article Monoid).

**EXAMPLES:**

```python
sage: from sage.categories.additive_monoids import AdditiveMonoids
sage: C = AdditiveMonoids(); C
Category of additive monoids
sage: C.super_categories()
[Category of additive unital additive magmas, Category of additive semigroups]
sage: sorted(C.axioms())
[AdditiveAssociative, AdditiveUnital]
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: C is AdditiveSemigroups().AdditiveUnital()
True
```
AdditiveCommutative
alias of CommutativeAdditiveMonoids

AdditiveInverse
alias of AdditiveGroups

class Homsets(category, *args)
Bases: HomsetsCategory

extra_super_categories()
Implement the fact that a homset between two monoids is associative.

EXAMPLES:

```python
sage: from sage.categories.additive_monoids import AdditiveMonoids
sage: AdditiveMonoids().Homsets().extra_super_categories()
[Category of additive semigroups]
sage: AdditiveMonoids().Homsets().super_categories()
[Category of homsets of additive unital additive magmas, Category of...
  → additive monoids]
```

Todo: This could be deduced from AdditiveSemigroups.Homsets.
extra_super_categories(). See comment in Objects.SubcategoryMethods.
Homsets().

class ParentMethods
Bases: object

sum(args)
Return the sum of the elements in args, as an element of self.

INPUT:
• args – a list (or iterable) of elements of self

EXAMPLES:

```python
sage: S = CommutativeAdditiveMonoids().example()
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: S.sum((a,b,a,c,a,b))
3*a + 2*b + c
sage: S.sum({})
0
```
4.5 Additive semigroups

class sage.categories.additive_semigroups.AdditiveSemigroups(base_category)

Bases: CategoryWithAxiom_singleton

The category of additive semigroups.

An additive semigroup is an associative additive magma, that is a set endowed with an operation + which is associative.

EXAMPLES:

\[
\begin{align*}
\text{sage: from sage.categories.additive_semigroups import AdditiveSemigroups} & \quad \text{AdditiveSemigroups} \\
\text{sage: C = AdditiveSemigroups(); C} & \quad \text{Category of additive semigroups} \\
\text{sage: C.super_categories()} & \quad \text{[Category of additive magmas]} \\
\text{sage: C.all_super_categories()} & \quad \text{[Category of additive semigroups,} \\
& \quad \text{Category of additive magmas,} \\
& \quad \text{Category of sets,} \\
& \quad \text{Category of sets with partial maps,} \\
& \quad \text{Category of objects]} \\
\text{sage: C.axioms()} & \quad \text{frozenset({}'AdditiveAssociative'})} \\
\text{sage: C is AdditiveMagmas().AdditiveAssociative()} & \quad \text{True}
\end{align*}
\]

(continues on next page)
AdditiveCommutative
    alias of CommutativeAdditiveSemigroups

AdditiveUnital
    alias of AdditiveMonoids

class Algebras(category, *args)
    Bases: AlgebrasCategory

class ParentMethods
    Bases: object

    algebra_generators()
        Return the generators of this algebra, as per MagmaticAlgebras.ParentMethods.
        algebra_generators().

        They correspond to the generators of the additive semigroup.

        EXAMPLES:

        sage: S = CommutativeAdditiveSemigroups().example(); S
        An example of a commutative semigroup:
        the free commutative semigroup generated by (a, b, c, d)
        sage: A = S.algebra(QQ)
        # needs sage.modules
        sage: A.algebra_generators()
        # needs sage.modules
        Family (B[a], B[b], B[c], B[d])

    product_on_basis(g1, g2)
        Product, on basis elements, as per MagmaticAlgebras.WithBasis.ParentMethods.
        product_on_basis().

        The product of two basis elements is induced by the addition of the corresponding elements of the group.

        EXAMPLES:

        sage: S = CommutativeAdditiveSemigroups().example(); S
        An example of a commutative semigroup:
        the free commutative semigroup generated by (a, b, c, d)
.. _additive_semigroups.CartesianProducts:

Cartesian Products

.. currentmodule:: sage.categories.additive_semigroups

.. autoclass:: CartesianProducts
   :module: sage.categories.additive_semigroups

extra_super_categories()

EXAMPLES:

```python
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: AdditiveSemigroups().CartesianProducts().extra_super_categories()
[Category of additive semigroups]
```

```python
sage: C = AdditiveSemigroups().CartesianProducts()
```

class CartesianProducts (category, *args)

Bases: CartesianProductsCategory

extra_super_categories()

Implement the fact that a Cartesian product of additive semigroups is an additive semigroup.

EXAMPLES:

```python
sage: C.extra_super_categories()
[Category of additive semigroups]
```

```
```
```python
>>> from sage.all import *
>>> from sage.categories.additive_semigroups import AdditiveSemigroups
>>> C = AdditiveSemigroups().CartesianProducts()
>>> C.extra_super_categories()
[Category of additive semigroups]
>>> C.axioms()
{'AdditiveAssociative'}
```

```python
class Homsets(category, *args):
    Bases: HomsetsCategory

    extra_super_categories()

    Implement the fact that a homset between two semigroups is a semigroup.

    EXAMPLES:

    sage: from sage.categories.additive_semigroups import AdditiveSemigroups
    sage: AdditiveSemigroups().Homsets().extra_super_categories()
    [Category of additive semigroups]
    sage: AdditiveSemigroups().Homsets().super_categories()
    [Category of homsets of additive magmas, Category of additive semigroups]
```

```python
>>> from sage.all import *
>>> from sage.categories.additive_semigroups import AdditiveSemigroups
>>> AdditiveSemigroups().Homsets().extra_super_categories()
[Category of additive semigroups]
>>> AdditiveSemigroups().Homsets().super_categories()
[Category of homsets of additive magmas, Category of additive semigroups]
```

```python
class ParentMethods:
    Bases: object
```

### 4.6 Affine Weyl groups

```python
class sage.categories.affine_weyl_groups.AffineWeylGroups:
    Bases: Category_singleton

    The category of affine Weyl groups

    Todo: add a description of this category

    See also:

    • Wikipedia article Affine_weyl_group
    • WeylGroups, WeylGroup

    EXAMPLES:

    sage: C = AffineWeylGroups(); C
    Category of affine Weyl groups
    sage: C.super_categories()
    [Category of infinite Weyl groups]
```

(continues on next page)
>>> from sage.all import *
>>> C = AffineWeylGroups(); C
Category of affine Weyl groups
>>> C.super_categories()
[Category of infinite Weyl groups]

class ElementMethods
    Bases: object

    affine_grassmannian_to_core()

    Bijection between affine Grassmannian elements of type \( A_{k}^{(1)} \) and \((k + 1)\)-cores.

    INPUT:
    • self – an affine Grassmannian element of some affine Weyl group of type \( A_{k}^{(1)} \)
    
    Recall that an element \( w \) of an affine Weyl group is affine Grassmannian if all its all reduced words end in 0, see \is_affine_grassmannian()\.

    OUTPUT:
    • a \((k + 1)\)-core

    See also:

    \affine_grassmannian_to_partition()

EXAMPLES:

sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 2, 1])
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)
sage: W.category()
Category of irreducible affine Weyl groups

sage: W = WeylGroup(['A', Integer(4), Integer(1)]); W
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
sage: W.category()
Category of irreducible affine Weyl groups
affine_grassmannian_to_partition()

Bijection between affine Grassmannian elements of type $A^{(1)}_k$ and $k$-bounded partitions.

INPUT:
• self is affine Grassmannian element of the affine Weyl group of type $A^{(1)}_k$ (i.e. all reduced words end in 0)

OUTPUT:
• $k$-bounded partition

See also:

affine_grassmannian_to_core()

EXAMPLES:

sage: # needs sage.combinat sage.groups
sage: k = 2
sage: W = WeylGroup(['A', k, 1])
sage: w = W.from_reduced_word([0, 2, 1, 0])
sage: la = w.affine_grassmannian_to_partition(); la
[2, 2]
sage: la.from_kbounded_to_grassmannian(k) == w
True

>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(2), Integer(1)])
>>> w = W.from_reduced_word([Integer(0), Integer(2), Integer(1), Integer(0)])
>>> la = w.affine_grassmannian_to_partition(); la
[4, 2]
>>> type(la)
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
>>> la.to_grassmannian() == w
True

>>> w = W.affine_grassmannian_to_partition()  # needs sage.combinat sage.groups
Traceback (most recent call last):
...
ValueError: this only works on type 'A' affine Grassmannian elements
**is_affine_grassmannian()**

Test whether self is affine Grassmannian.

An element of an affine Weyl group is *affine Grassmannian* if any of the following equivalent properties holds:

- All reduced words for `self` end with 0.
- `self` is the identity, or 0 is its single right descent.
- `self` is a minimal coset representative for \( W / cl W \).

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3, 1])
sage: w = W.from_reduced_word([2,1,0])
sage: w.is_affine_grassmannian()  # True
sage: w = W.from_reduced_word([2,0])
sage: w.is_affine_grassmannian()  # False
sage: W.one().is_affine_grassmannian()  # True
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(3), Integer(1)])
>>> w = W.from_reduced_word([Integer(2),Integer(1),Integer(0)])
>>> w.is_affine_grassmannian()  # True
>>> w = W.from_reduced_word([Integer(2),Integer(0)])
>>> w.is_affine_grassmannian()  # False
>>> W.one().is_affine_grassmannian()  # True
```

**class ParentMethods**

**affine_grassmannian_elements_of_given_length(k)**

Return the affine Grassmannian elements of length \( k \).

This is returned as a finite enumerated set.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A', 3, 1])
... # needs sage.combinat sage.groups
sage: [x.reduced_word() for x in W.affine_grassmannian_elements_of_given_length(3)]
[[2, 1, 0], [3, 1, 0], [2, 3, 0]]
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(3), Integer(1)])
... # needs sage.combinat sage.groups
>>> [x.reduced_word() for x in W.affine_grassmannian_elements_of_given_length(Integer(3))]
[[2, 1, 0], [3, 1, 0], [2, 3, 0]]
```
See also:

`AffineWeylGroups.ElementMethods.is_affine_grassmannian()`

**special_node()**

Return the distinguished special node of the underlying Dynkin diagram.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A', 3, 1])
# needs sage.combinat sage.groups
sage: W.special_node() # needs sage.combinat sage.groups
0
```

```python
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(3), Integer(1)]) # needs sage.combinat sage.groups
>>> W.special_node() # needs sage.combinat sage.groups
0
```

**additional_structure()**

Return None.

Indeed, the category of affine Weyl groups defines no additional structure: affine Weyl groups are a special class of Weyl groups.

**See also:**

`Category.additional_structure()`

**Todo:** Should this category be a `CategoryWithAxiom`?

**EXAMPLES:**

```python
sage: AffineWeylGroups().additional_structure()
```

```python
>>> from sage.all import *
>>> AffineWeylGroups().additional_structure()
```

**super_categories()**

**EXAMPLES:**

```python
sage: AffineWeylGroups().super_categories()
[Category of infinite Weyl groups]
```

```python
>>> from sage.all import *
>>> AffineWeylGroups().super_categories()
[Category of infinite Weyl groups]
```
4.7 Algebra ideals

class sage.categories.algebra_ideals.AlgebraIdeals(A)
    Bases: Category_ideal

The category of two-sided ideals in a fixed algebra $A$.

EXAMPLES:

```python
sage: AlgebraIdeals(QQ['a'])
Category of algebra ideals in Univariate Polynomial Ring in a over Rational Field
```

Todo:

- Add support for non commutative rings (this is currently not supported by the subcategory `AlgebraModules`).
- Make `AlgebraIdeals(R).return CommutativeAlgebraIdeals(R)` when $R$ is commutative.
- If useful, implement `AlgebraLeftIdeals` and `AlgebraRightIdeals` of which `AlgebraIdeals` would be a subcategory.

algebra()

EXAMPLES:

```python
sage: AlgebraIdeals(QQ['x']).algebra()
Univariate Polynomial Ring in x over Rational Field
```

super_categories()

The category of algebra modules should be a super category of this category.

However, since algebra modules are currently only available over commutative rings, we have to omit it if our ring is non-commutative.

EXAMPLES:

```python
sage: AlgebraIdeals(QQ['x']).super_categories()
[Category of algebra modules
  over Univariate Polynomial Ring in x over Rational Field]
```
4.8 Algebra modules

class sage.categories.algebra_modules.AlgebraModules(A)

Bases: Category_module

The category of modules over a fixed algebra $A$.

EXAMPLES:

```python
sage: AlgebraModules(QQ['a'])
Category of algebra modules over Univariate Polynomial Ring in a over Rational Field
```

Note: as of now, $A$ is required to be commutative, ensuring that the categories of left and right modules are isomorphic. Feedback and use cases for potential generalizations to the non commutative case are welcome.

algebra()

EXAMPLES:

```python
sage: AlgebraModules(QQ['x']).algebra()
Univariate Polynomial Ring in x over Rational Field
```

classmethod an_instance()

Returns an instance of this class

EXAMPLES:

```python
sage: AlgebraModules.an_instance()
Category of algebra modules over Univariate Polynomial Ring in x over Rational Field
```
4.9 Algebras

AUTHORS:

• David Kohel & William Stein (2005): initial revision
• Nicolas M. Thiery (2008-2011): rewrote for the category framework

class sage.categories.algebras.Algebras(base_category)

The category of associative and unital algebras over a given base ring.

An associative and unital algebra over a ring $R$ is a module over $R$ which is itself a ring.

Warning: *Algebras* will be eventually be replaced by *magmatic_algebras.MagmaticAlgebras* for consistency with e.g. Wikipedia article *Algebras* which assumes neither associativity nor the existence of a unit (see Issue #15043).

Todo: Should $R$ be a commutative ring?

EXAMPLES:
class CartesianProducts (category, *args)

Bases: CartesianProductsCategory

The category of algebras constructed as Cartesian products of algebras

This construction gives the direct product of algebras. See discussion on:

- http://groups.google.fr/group/sage-devel/browse_thread/thread/35a72b1d0a2fc77a/348f42ae77a66d16#348f42ae77a66d16
- Wikipedia article Direct_product

extra_super_categories()

A Cartesian product of algebras is endowed with a natural algebra structure.

EXAMPLES:

```python
sage: C = Algebras(QQ).CartesianProducts()
sage: C.extra_super_categories()
[Category of algebras over Rational Field]
sage: sorted(C.super_categories(), key=str)
[Category of Cartesian products of monoids, Category of Cartesian products of unital algebras over Rational Field, Category of algebras over Rational Field]
```

Commutative

alias of CommutativeAlgebras

class DualObjects (category, *args)

Bases: DualObjectsCategory

extra_super_categories()

Return the dual category

EXAMPLES:

The category of algebras over the Rational Field is dual to the category of coalgebras over the same field:

```python
sage: C = Algebras(QQ)
sage: C.dual()
Category of duals of algebras over Rational Field
sage: C.dual().extra_super_categories()
[Category of coalgebras over Rational Field]
```

```python
>>> from sage.all import *
>>> C = Algebras(QQ)
>>> C.dual()
Category of duals of algebras over Rational Field
>>> C.dual().extra_super_categories()
[Category of coalgebras over Rational Field]
```
Warning: This is only correct in certain cases (finite dimension, ...). See Issue #15647.

class ElementMethods
    Bases: object

Filtered
    alias of FilteredAlgebras

Graded
    alias of GradedAlgebras

class ParentMethods
    Bases: object

    characteristic()

    Return the characteristic of this algebra, which is the same as the characteristic of its base ring.

    EXAMPLES:

    sage: # needs sage.modules
    sage: ZZ.characteristic()
    0

    sage: A = GF(7^3, 'a')
    # needs sage.rings.finite_rings
    sage: A.characteristic()
    7

    >>> from sage.all import *
    >>> # needs sage.modules
    >>> ZZ.characteristic()
    0
    >>> A = GF(Integer(7)**Integer(3), 'a')
    # needs sage.rings.finite_rings
    >>> A.characteristic()
    7

    has_standard_involution()

    Return True if the algebra has a standard involution and False otherwise.

    This algorithm follows Algorithm 2.10 from John Voight’s Identifying the Matrix Ring. Currently the only type of algebra this will work for is a quaternion algebra. Though this function seems redundant, once algebras have more functionality, in particular have a method to construct a basis, this algorithm will have more general purpose.

    EXAMPLES:

    sage: # needs sage.combinat sage.modules
    sage: B = QuaternionAlgebra(2)
    sage: B.has_standard_involution()
    True

    sage: R.<x> = PolynomialRing(QQ)
    sage: K.<u> = NumberField(x**2 - 2)
    # needs sage.rings.number_field
    sage: A = QuaternionAlgebra(K, -2, 5)

    (continues on next page)
... # needs sage.rings.number_field
sage: A.has_standard_involution() → # needs sage.rings.number_field
True
sage: L.<a,b> = FreeAlgebra(QQ, 2)
sage: L.has_standard_involution()
Traceback (most recent call last):
... NotImplementedError: has_standard_involution is not implemented for this algebra

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> B = QuaternionAlgebra(Integer(2))
>>> B.has_standard_involution()
True
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(2) - Integer(2), names=('u',)); (u,) = K._
first_ngens(1)# needs sage.rings.number_field
>>> A = QuaternionAlgebra(K, -Integer(2), Integer(5)) # needs sage.rings.number_field
>>> A.has_standard_involution()
# needs sage.rings.number_field
True
>>> L = FreeAlgebra(QQ, Integer(2), names=('a', 'b',)); (a, b,) = L._
first_ngens(2)
>>> L.has_standard_involution()
Traceback (most recent call last):
... NotImplementedError: has_standard_involution is not implemented for this algebra

class Quotients (category, *args)
Bases: QuotientsCategory
class ParentMethods
Bases: object

algebra_generators()
Return algebra generators for self.
This implementation retracts the algebra generators from the ambient algebra.

EXAMPLES:

sage: # needs sage.graphs sage.modules
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x→y and b:x→y) over Rational Field
sage: S = A.semisimple_quotient()
sage: S.algebra_generators()
Finite family {'x': B['x'], 'y': B['y'], 'a': 0, 'b': 0}

>>> from sage.all import *
>>> # needs sage.graphs sage.modules

(continues on next page)
A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
S = A.semisimple_quotient()
S.algebra_generators()
Finite family {'x': B['x'], 'y': B['y'], 'a': 0, 'b': 0}

Todo: this could possibly remove the elements that retract to zero.

Semisimple
alias of SemisimpleAlgebras
class SubcategoryMethods
    Bases: object
    Semisimple()
        Return the subcategory of semisimple objects of self.

Note: This mimics the syntax of axioms for a smooth transition if Semisimple becomes one.

EXAMPLES:

sage: Algebras(QQ).Semisimple()
Category of semisimple algebras over Rational Field
sage: Algebras(QQ).WithBasis().FiniteDimensional().Semisimple()
Category of finite dimensional semisimple algebras with basis over Rational Field

Supercommutative()
Return the full subcategory of the supercommutative objects of self.
This is shorthand for creating the corresponding super category.

EXAMPLES:

sage: Algebras(ZZ).Supercommutative()
Category of supercommutative algebras over Integer Ring
sage: Algebras(ZZ).WithBasis().Supercommutative()
Category of supercommutative super algebras with basis over Integer Ring
sage: Cat = Algebras(ZZ).Supercommutative()
sage: Cat is Algebras(ZZ).Super().Supercommutative()
True
>>> from sage.all import *
>>> Algebras(ZZ).Supercommutative()
Category of supercommutative algebras over Integer Ring
>>> Algebras(ZZ).WithBasis().Supercommutative()
Category of supercommutative super algebras with basis over Integer Ring

>>> Cat = Algebras(ZZ).Supercommutative()
>>> Cat is Algebras(ZZ).Super().Supercommutative()
True

```
Super
alias of SuperAlgebras

class TensorProducts(category, *args)
    Bases: TensorProductsCategory

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

extra_super_categories()

EXAMPLES:

sage: Algebras(QQ).TensorProducts().extra_super_categories()
[Category of algebras over Rational Field]
sage: Algebras(QQ).TensorProducts().super_categories()
[Category of algebras over Rational Field,
 Category of tensor products of vector spaces over Rational Field]

 Meaning: a tensor product of algebras is an algebra

WithBasis
alias of AlgebrasWithBasis

4.10 Algebras With Basis

```
class sage.categories.algebras_with_basis.AlgebrasWithBasis(base_category)
    Bases: CategoryWithAxiom_over_base_ring

The category of algebras with a distinguished basis.

EXAMPLES:

sage: C = AlgebrasWithBasis(QQ); C
Category of algebras with basis over Rational Field
```
(continues on next page)
sage: sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field,
Category of unital algebras with basis over Rational Field]

>>> from sage.all import *

>>> C = AlgebrasWithBasis(QQ); C
Category of algebras with basis over Rational Field

>>> sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field,
Category of unital algebras with basis over Rational Field]

We construct a typical parent in this category, and do some computations with it:

sage: # needs sage.combinat sage.modules
sage: A = C.example(); A
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field
sage: A.category()
Category of algebras with basis over Rational Field
sage: A.one_basis()
word:

sage: A.one()
B[word: ]
sage: A.base_ring()
Rational Field

sage: A.basis().keys()
Finite words over ('a', 'b', 'c')
sage: (a,b,c) = A.algebra_generators()
sage: a^3, b^2
B[word: aaa], B[word: bb]

sage: a * c * b
B[word: acb]
sage: A.product
<bound method MagmaticAlgebras.WithBasis.ParentMethods._product_from_product_on_˓
→basis_multiply of An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field>

sage: A.product(a * b, b)
B[word: abb]

sage: TestSuite(A).run(verbose=True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_construction() . . . pass
running ._test_distributivity() . . . pass
running ._test_elements() . . .
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_monomial_coefficients() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
(continues on next page)
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = C.example(); A
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field

>>> A.category()
Category of algebras with basis over Rational Field

>>> A.one_basis()
word:

>>> A.one()
B[word: ]

>>> A.base_ring()
Rational Field

>>> A.basis().keys()
Finite words over {'a', 'b', 'c'}

>>> (a,b,c) = A.algebra_generators()

>>> a**Integer(3), b**Integer(2)
(B[word: aaa], B[word: bb])

>>> a * c * b
B[word: acb]

>>> A.product
<bound method MagmaticAlgebras.WithBasis.ParentMethods._product_from_product_on_→basis_multiply of
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field>

>>> A.product(a * b, b)
B[word: abb]

>>> TestSuite(A).run(verbos=True)
running _test_additive_associativity() ... pass
running _test_an_element() ... pass
running _test_associativity() ... pass
running _test_cardinality() ... pass
running _test_category() ... pass
running _test_characteristic() ... pass
running _test_construction() ... pass
running _test_distributivity() ... pass

(continues on next page)
Please see the source code of \(A\) (with \(A??\)) for how to implement other algebras with basis.

**class CartesianProducts (category, *args)**

Bases: *CartesianProductsCategory*

The category of algebras with basis, constructed as Cartesian products of algebras with basis.

Note: this construction give the direct products of algebras with basis. See comment in *Algebras*. *CartesianProducts*

**class ParentMethods**

Bases: *object*

**one ()**

**one_from_cartesian_product_of_one_basis ()**

Return the one of this Cartesian product of algebras, as per *Monoids.ParentMethods.one*

It is constructed as the Cartesian product of the ones of the summands, using their one_basis() methods.

This implementation does not require multiplication by scalars nor calling cartesian_product. This might help keeping things as lazy as possible upon initialization.

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over Rational Field
```
sage: A.one_basis()
word:
sage: B = cartesian_product((A, A, A))
sage: B.one_from_cartesian_product_of_one_basis()
B[(0, word: )] + B[(1, word: )] + B[(2, word: )]
sage: B.one()
B[(0, word: )] + B[(1, word: )] + B[(2, word: )]

sage: cartesian_product([SymmetricGroupAlgebra(QQ, 3), SymmetricGroupAlgebra(QQ, 4)]).one()
B[(0, [1, 2, 3])] + B[(1, [1, 2, 3, 4])]

extra_super_categories()
A Cartesian product of algebras with basis is endowed with a natural algebra with basis structure.

EXAMPLES:

```python
class ElementMethods
    Bases: object

sage: AlgebrasWithBasis(QQ).CartesianProducts().extra_super_categories()
[Category of algebras with basis over Rational Field]
sage: AlgebrasWithBasis(QQ).CartesianProducts().super_categories()
[Category of algebras with basis over Rational Field, Category of Cartesian products of algebras over Rational Field, Category of Cartesian products of vector spaces with basis over Rational Field]
```
Filtered
  alias of FilteredAlgebrasWithBasis

FiniteDimensional
  alias of FiniteDimensionalAlgebrasWithBasis

Graded
  alias of GradedAlgebrasWithBasis

class ParentMethods
  Bases: object

hochschild_complex(M)
  Return the Hochschild complex of self with coefficients in M.

See also:
  HochschildComplex

EXAMPLES:

sage: R.<x> = QQ[]
sage: A = algebras.DifferentialWeyl(R)  # needs sage.modules
sage: H = A.hochschild_complex(A)  # needs sage.modules
sage: # needs sage.combinat sage.groups sage.modules
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)

>>> from sage.all import *

one()
  Return the multiplicative unit element.

EXAMPLES:

sage: A = AlgebrasWithBasis(QQ).example()  # needs sage.combinat sage.modules
sage: A.one_basis()  # needs sage.combinat sage.modules
word:
sage: A.one()  # needs sage.combinat sage.modules
B[word: ]
Super

alias of SuperAlgebrasWithBasis

class TensorProducts(category, *args)

Bases: TensorProductsCategory

The category of algebras with basis constructed by tensor product of algebras with basis

class ElementMethods

Bases: object

Implements operations on elements of tensor products of algebras with basis

class ParentMethods

Bases: object

implements operations on tensor products of algebras with basis

one_basis()

Returns the index of the one of this tensor product of algebras, as per AlgebrasWithBasis.

ParentMethods.one_basis

It is the tuple whose operands are the indices of the ones of the operands, as returned by their

one_basis() methods.

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra
on the generators ('a', 'b', 'c') over Rational Field
sage: A.one_basis()
word:

sage: B = tensor((A, A, A))

sage: B.one()

(continues on next page)
product_on_basis()(t₁,t₂)
The product of the algebra on the basis, as per `AlgebrasWithBasis.ParentMethods.product_on_basis`.

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra
   on the generators ('a', 'b', 'c') over Rational Field
  sage: (a,b,c) = A.algebra_generators()
sage: x = tensor((a, b, c)); x
  B[a] # B[b] # B[c]
sage: y = tensor((c, b, a)); y
  B[c] # B[b] # B[a]
sage: x * y
  B[ac] # B[bb] # B[ca]
sage: x = tensor(((a + 2*b), c)); x
sage: y = tensor((c, a)) + 1; y
sage: x * y
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra
   on the generators ('a', 'b', 'c') over Rational Field
>>> (a,b,c) = A.algebra_generators()
>>> x = tensor((a, b, c)); x
  B[a] # B[b] # B[c]
>>> y = tensor((c, b, a)); y
  B[c] # B[b] # B[a]
>>> x * y
```

TODO: optimize this implementation!

extra_super_categories()

**EXAMPLES:**

```
sage: AlgebrasWithBasis(QQ).TensorProducts().extra_super_categories()
[Category of algebras with basis over Rational Field]
```
sage: AlgebrasWithBasis(QQ).TensorProducts().super_categories()
[Category of algebras with basis over Rational Field,
Category of tensor products of algebras over Rational Field,
Category of tensor products of vector spaces with basis over Rational Field]

>>> from sage.all import *

>>> AlgebrasWithBasis(QQ).TensorProducts().extra_super_categories()
[Category of algebras with basis over Rational Field]

>>> AlgebrasWithBasis(QQ).TensorProducts().super_categories()
[Category of algebras with basis over Rational Field,
Category of tensor products of algebras over Rational Field,
Category of tensor products of vector spaces with basis over Rational Field]

example (alphabet=('a', 'b', 'c'))

Return an example of algebra with basis.

EXAMPLES:

An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field

An other set of generators can be specified as optional argument:

An example of an algebra with basis:
the free algebra on the generators (1, 2, 3) over Rational Field

4.11 Aperiodic semigroups

class sage.categories.aperiodic_semigroups.AperiodicSemigroups (base_category)
Bases: CategoryWithAxiom

eextra_super_categories ()

Implement the fact that an aperiodic semigroup is $H$-trivial.

EXAMPLES:
4.12 Associative algebras

class sage.categories.associative_algebras.AssociativeAlgebras(base_category):
    Bases: CategoryWithAxiom_over_base_ring

    The category of associative algebras over a given base ring.
    An associative algebra over a ring $R$ is a module over $R$ which is also a not necessarily unital ring.

    Warning: Until Issue #15043 is implemented, Algebras is the category of associative unital algebras; thus, unlike the name suggests, AssociativeAlgebras is not a subcategory of Algebras but of MagmaticAlgebras.

    EXAMPLES:

    sage: from sage.categories.associative_algebras import AssociativeAlgebras
    sage: C = AssociativeAlgebras(ZZ); C
    Category of associative algebras over Integer Ring

    >>> from sage.all import *
    >>> from sage.categories.associative_algebras import AssociativeAlgebras
    >>> C = AssociativeAlgebras(ZZ); C
    Category of associative algebras over Integer Ring

    Unital
    alias of Algebras

4.13 Bialgebras

class sage.categories.bialgebras.Bialgebras(base, name=None):
    Bases: Category_over_base_ring

    The category of bialgebras

    EXAMPLES:

    sage: Bialgebras(ZZ)
    Category of bialgebras over Integer Ring
    sage: Bialgebras(ZZ).super_categories()
    [Category of algebras over Integer Ring, Category of coalgebras over Integer Ring]
```python
>>> from sage.all import *
>>> Bialgebras(ZZ)
Category of bialgebras over Integer Ring
>>> Bialgebras(ZZ).super_categories()
[Category of algebras over Integer Ring, Category of coalgebras over Integer Ring]

class ElementMethods
    Bases: object

    is_grouplike()
    Return whether self is a grouplike element.

    EXAMPLES:
    sage: s = SymmetricFunctions(QQ).schur() # needs sage.modules
    sage: s([5]).is_grouplike()  # needs lrcalc_python sage.modules
    False
    sage: s([]).is_grouplike()  # needs lrcalc_python sage.modules
    True

    isPrimitive()
    Return whether self is a primitive element.

    EXAMPLES:
    sage: # needs sage.modules
    sage: s = SymmetricFunctions(QQ).schur()
    sage: s([5]).isPrimitive()  # needs lrcalc_python
    False
    sage: p = SymmetricFunctions(QQ).powersum()
    sage: p([5]).isPrimitive()
    True
```

4.13. Bialgebras
class Super(base_category)
    Bases: SuperModulesCategory
WithBasis
    alias of BialgebrasWithBasis
additional_structure()
    Return None.
    Indeed, the category of bialgebras defines no additional structure: a morphism of coalgebras and of algebras between two bialgebras is a bialgebra morphism.
See also:
    Category.additional_structure()
Todo: This category should be a CategoryWithAxiom.

EXAMPLES:
sage: Bialgebras(QQ).additional_structure()

>>> from sage.all import *
>>> Bialgebras(QQ).additional_structure()

super_categories()
    EXAMPLES:
sage: Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]

>>> from sage.all import *
>>> Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]

4.14 Bialgebras with basis

class sage.categories.bialgebras_with_basis.BialgebrasWithBasis(base_category)
    Bases: CategoryWithAxiom_over_base_ring
    The category of bialgebras with a distinguished basis.

EXAMPLES:
sage: C = BialgebrasWithBasis(QQ); C
Category of bialgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]
class ElementMethods
Bases: object

adams_operator(*args, **kwds)

Deprecated: Use convolution_power_of_id() instead. See Issue #36396 for details.

convolution_power_of_id(n)

Compute the $n$-th convolution power of the identity morphism $Id$ on $self$.

INPUT:
• $n$ – a nonnegative integer

OUTPUT:
• the image of $self$ under the convolution power $Id^n$

Note: In the literature, this is also called a Hopf power or Sweedler power, cf. [AL2015].

See also:
sage.categories.bialgebras.ElementMethods.convolution_product()

Todo: Remove dependency on modules_with_basis methods.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: h = SymmetricFunctions(QQ).h()

sage: h[5].convolution_power_of_id(2)

sage: h[5].plethysm(2*h[1])

sage: h([]).convolution_power_of_id(0)

h[]

sage: h([]).convolution_power_of_id(1)

h[]

sage: h[3,2].convolution_power_of_id(0)
0

sage: h[3,2].convolution_power_of_id(1)

h[3, 2]
```

(continues on next page)
Return the image of self under the convolution product (map) of the maps.

Let $A$ and $B$ be bialgebras over a commutative ring $R$. Given maps $f_i : A \to B$ for $1 \leq i < n$, define the convolution product

$$(f_1 \circ f_2 \circ \cdots \circ f_n) := \mu^{(n-1)} \circ (f_1 \otimes f_2 \otimes \cdots \otimes f_n) \circ \Delta^{(n-1)},$$

where $\Delta^{(k)} := (\Delta \otimes \text{Id}^{(k-1)}) \circ \Delta^{(k-1)}$, with $\Delta^{(1)} = \Delta$ (the ordinary coproduct in $A$) and $\Delta^{(0)} = \text{Id}$; and with $\mu^{(k)} := \mu \circ (\mu^{(k-1)} \otimes \text{Id})$ and $\mu^{(1)} = \mu$ (the ordinary product in $B$). See [Swe1969].

(In the literature, one finds, e.g., $\Delta^{(2)}$ for what we denote above as $\Delta^{(1)}$. See [KMN2012].)

**INPUT:**
- maps — any number $n \geq 0$ of linear maps $f_1, f_2, \ldots, f_n$ on self.parent(); or a single list or tuple of such maps

**OUTPUT:**
- the convolution product of maps applied to self
AUTHORS:
• Amy Pang - 12 June 2015 - Sage Days 65

Todo: Remove dependency on modules_with_basis methods.

EXAMPLES:

We compute convolution products of the identity and antipode maps on Schur functions:

```python
sage: # needs sage.combinat sage.modules
sage: Id = lambda x: x
sage: Antipode = lambda x: x.antipode()

sage: s[3].convolution_product(Id, Id)  # needs lrcalc_python
2*s[2, 1] + 4*s[3]

sage: s[3,2].convolution_product(Id) == s[3,2]
True

>>> from sage.all import *

>>> # needs sage.combinat sage.modules

>>> Id = lambda x: x

>>> Antipode = lambda x: x.antipode()

>>> s = SymmetricFunctions(QQ).schur()

>>> s[Integer(3)].convolution_product(Id, Id)  # needs lrcalc_python
2*s[2, 1] + 4*s[3]

>>> s[Integer(3),Integer(2)].convolution_product(Id, Id) == s[Integer(3), Integer(2)]
True
```

The method accepts multiple arguments, or a single argument consisting of a list of maps:

```python
sage: s[3,2].convolution_product(Id, Id)  # needs lrcalc_python sage.combinat sage.modules

sage: s[3,2].convolution_product([Id, Id])  # needs lrcalc_python sage.combinat sage.modules

>>> from sage.all import *

>>> s[Integer(3),Integer(2)].convolution_product(Id, Id)  # needs lrcalc_python sage.combinat sage.modules

>>> s[Integer(3),Integer(2)].convolution_product([Id, Id])  # needs lrcalc_python sage.combinat sage.modules
```

We test the defining property of the antipode morphism; namely, that the antipode is the inverse of the identity map in the convolution algebra whose identity element is the composition of the counit and unit:

```python
sage: (s[3,2].convolution_product())  # needs lrcalc_python sage.combinat sage.modules
(continues on next page)
```

4.14. Bialgebras with basis 305
...:
  == s[3,2].convolution_product(Antipode, Id)
  == s[3,2].convolution_product(Id, Antipode))
True

>>> from sage.all import *
>>> (s[Integer(3),Integer(2)].convolution_product() → # needs lrcalc_python sage.combinat sage.modules...
... == s[Integer(3),Integer(2)].convolution_product(Antipode, Id)
... == s[Integer(3),Integer(2)].convolution_product(Id, Antipode))
True

sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
# needs sage.combinat sage.modules
sage: Psi[2,1].convolution_product(Id, Id, Id) # needs sage.combinat sage.modules
3*Psi[1, 2] + 6*Psi[2, 1]
sage: (Psi[5,1] - Psi[1,5]).convolution_product(Id, Id, Id) # needs sage.combinat sage.modules
-3*Psi[1, 5] + 3*Psi[5, 1]

>>> from sage.all import *
>>> Psi = NonCommutativeSymmetricFunctions(QQ).Psi() #...
# needs sage.combinat sage.modules
>>> Psi[Integer(2),Integer(1)].convolution_product(Id, Id, Id) #...
# needs sage.combinat sage.modules
3*Psi[1, 2] + 6*Psi[2, 1]
>>> (Psi[Integer(5),Integer(1)] - Psi[Integer(1),Integer(5)]).convolution_
... product(Id, Id, Id) # needs sage.combinat sage.modules
-3*Psi[1, 5] + 3*Psi[5, 1]

sage: G = SymmetricGroup(3)
sage: QG = GroupAlgebra(G, QQ)
sage: x = QG.sum_of_terms([[p, p.length()] ...
....:     for p in Permutations(3)]); x
[1, 3, 2] + [2, 1, 3] + 2*[2, 3, 1] + 2*[3, 1, 2] + 3*[3, 2, 1]
sage: x.convolution_product(Id, Id)
5*[1, 2, 3] + 2*[2, 3, 1] + 2*[3, 1, 2]
sage: x.convolution_product(Id, Id, Id)
4*[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + 3*[3, 2, 1]
sage: x.convolution_product([Id] * 6)
9*[1, 2, 3]

>>> from sage.all import *
>>> # needs sage.combinat sage.groups sage.modules
>>> G = SymmetricGroup(Integer(3))
>>> QG = GroupAlgebra(G, QQ)
>>> x = QG.sum_of_terms([[p, p.length()] ...
....:     for p in Permutations(Integer(3))]); x
[1, 3, 2] + [2, 1, 3] + 2*[2, 3, 1] + 2*[3, 1, 2] + 3*[3, 2, 1]
>>> x.convolution_product(Id, Id)
5*[1, 2, 3] + 2*[2, 3, 1] + 2*[3, 1, 2]
>>> x.convolution_product(Id, Id, Id)
4*[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + 3*[3, 2, 1]
>>> x.convolution_product([Id] * Integer(6))
9*[1, 2, 3]

class ParentMethods
Bases: object

convolution_product(*maps)

Return the convolution product (a map) of the given maps.

Let $A$ and $B$ be bialgebras over a commutative ring $R$. Given maps $f_i : A \to B$ for $1 \leq i < n$, define the convolution product

$$(f_1 * f_2 * \cdots * f_n) := \mu^{(n-1)} \circ (f_1 \otimes f_2 \otimes \cdots \otimes f_n) \circ \Delta^{(n-1)},$$

where $\Delta^{(k)} := (\Delta \otimes \text{Id}^{(k-1)}) \circ \Delta^{(k-1)}$, with $\Delta^{(1)} = \Delta$ (the ordinary coproduct in $A$) and $\Delta^{(0)} = \text{Id}$; and with $\mu^{(k)} := \mu \circ (\mu^{(k-1)} \otimes \text{Id})$ and $\mu^{(1)} = \mu$ (the ordinary product in $B$). See [Swe1969].

(In the literature, one finds, e.g., $\Delta^{(2)}$ for what we denote above as $\Delta^{(1)}$. See [KMN2012].)

INPUT:
• maps – any number $n \geq 0$ of linear maps $f_1, f_2, \ldots, f_n$ on self; or a single list or tuple of such maps

OUTPUT:
• the new map $f_1 * f_2 * \cdots * f_2$ representing their convolution product

See also:
sage.categories.bialgebras.ElementMethods.convolution_product()

AUTHORS:
• Aaron Lauve - 12 June 2015 - Sage Days 65

Todo: Remove dependency on modules_with_basis methods.

EXCEPTIONS:
We construct some maps: the identity, the antipode and projection onto the homogeneous component of degree 2:

```
sage: Id = lambda x: x
sage: Antipode = lambda x: x.antipode()
sage: Proj2 = lambda x: x.parent().sum_of_terms([(m, c) for (m, c) in x if m.size() == 2])
```

>>> from sage.all import *
>>> Id = lambda x: x
>>> Antipode = lambda x: x.antipode()
>>> Proj2 = lambda x: x.parent().sum_of_terms([(m, c) for (m, c) in x if m.size() == Integer(2)])

Compute the convolution product of the identity with itself and with the projection Proj2 on the Hopf algebra of non-commutative symmetric functions:

```
sage: # needs sage.combinat sage.modules
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()
sage: T = R.convolution_product([Id, Id])
sage: [T(R(comp)) for comp in Compositions(3)]
[4*R[1, 1, 1] + R[1, 2] + R[2, 1],
```

(continues on next page)
Compute the convolution product of no maps on the Hopf algebra of symmetric functions in non-commuting variables. This is the composition of the counit with the unit:

```python
sage: # needs sage.combinat sage.graphs sage.modules
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: T = m.convolution_product()
```

```python
sage: [T(m(lam)) for lam in SetPartitions(0).list() + SetPartitions(2).list()]
[0, 0]
```

Compute the convolution product of the projection $\text{Proj2}$ with the identity on the Hopf algebra of symmetric functions in non-commuting variables:

```python
sage: T = m.convolution_product(Proj2, Id)  # Needs sage.combinat sage.graphs sage.modules
sage: [T(m(lam)) for lam in SetPartitions(3)]  # Needs sage.combinat sage.graphs sage.modules
[0,
m\{{1, 2}, \{3\}\} + m\{{1, 2, 3}\},
m\{{1, 2}, \{3\}\} + m\{{1, 2, 3}\},
m\{{1, 2}, \{3\}\} + m\{{1, 2, 3}\},
3*m\{{1, \{2\}, \{3\}\}} + 3*m\{{1, \{2, 3\}\}} + 3*m\{{1, \{1, 3\}, \{2\}\}}]
```

```python
>>> from sage.all import *  # Needs sage.combinat sage.graphs sage.modules
>>> m = SymmetricFunctionsNonCommutingVariables(QQ).m()
>>> T = m.convolution_product()
```
Compute the convolution product of the antipode with itself and the identity map on group algebra of the symmetric group:

```
sage: G = SymmetricGroup(3)
sage: QG = GroupAlgebra(G, QQ)
sage: x = QG.sum_of_terms(
    ...:     (p, p.number_of_peaks() + p.number_of_inversions())
    ...:     for p in Permutations(3))
    ...:
    ); x
2*[1, 3, 2] + [2, 1, 3] + 3*[2, 3, 1] + 2*[3, 1, 2] + 3*[3, 2, 1]
```

```
sage: T = QG.convolution_product(Antipode, Antipode, Id)
sage: T(x)
2*[1, 3, 2] + [2, 1, 3] + 2*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2, 1]
```

### 4.15 Bimodules

**class** `sage.categories.bimodules.Bimodules`(left_base, right_base, name=None)

**Bases:** `CategoryWithParameters`

The category of \((R, S)\)-bimodules

For \(R\) and \(S\) rings, a \((R, S)\)-bimodule \(X\) is a left \(R\)-module and right \(S\)-module such that the left and right actions commute: \(r \cdot (x \cdot s) = (r \cdot x) \cdot s\).

**EXAMPLES:**

```
sage: Bimodules(QQ, ZZ)
Category of bimodules over Rational Field on the left and Integer Ring on the right
```

```
sage: Bimodules(QQ, ZZ).super_categories()
[Category of left modules over Rational Field, Category of right modules over...
```
class ElementMethods
   Bases: object

class ParentMethods
   Bases: object

additional_structure()
   Return None.

   Indeed, the category of bimodules defines no additional structure: a left and right module morphism between two bimodules is a bimodule morphism.

   See also:
   Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

sage: Bimodules(QQ, ZZ).additional_structure()

classmethod an_instance()
   Return an instance of this class.

EXAMPLES:

sage: Bimodules.an_instance()  # needs sage.rings.real_mpfr
Category of bimodules over Rational Field on the left and Real Field with 53 bits of precision on the right

sage: Bimodules.an_instance()  # needs sage.rings.real_mpfr
Category of bimodules over Rational Field on the left and Real Field with 53 bits of precision on the right

left_base_ring()
   Return the left base ring over which elements of this category are defined.

EXAMPLES:

sage: Bimodules(QQ, ZZ).left_base_ring()
Rational Field
>>> from sage.all import *
>>> Bimodules(QQ, ZZ).left_base_ring()
Rational Field

right_base_ring()

Return the right base ring over which elements of this category are defined.

EXAMPLES:

sage: Bimodules(QQ, ZZ).right_base_ring()
Integer Ring

super_categories()

EXAMPLES:

sage: Bimodules(QQ, ZZ).super_categories()

4.16 Classical Crystals

class sage.categories.classical_crystals.ClassicalCrystals

Bases: Category_singleton

The category of classical crystals, that is crystals of finite Cartan type.

EXAMPLES:

sage: C = ClassicalCrystals()
sage: C
Category of classical crystals
sage: C.super_categories()
[Category of regular crystals, Category of finite crystals, Category of highest weight crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1

(continues on next page)
class ElementMethods

    Bases: object

    lusztig_involution()

    Return the Lusztig involution on the classical highest weight crystal self.

    The Lusztig involution on a finite-dimensional highest weight crystal \( B(\lambda) \) of highest weight \( \lambda \) maps the highest weight vector to the lowest weight vector and the Kashiwara operator \( f_i \) to \( e_i^* \), where \( i^* \) is defined as \( \alpha_i^* = -w_0(\alpha_i) \). Here \( w_0 \) is the longest element of the Weyl group acting on the \( i \)-th simple root \( \alpha_i \).

    EXAMPLES:

        sage: B = crystals.Tableaux(['A',3],shape=[2,1])
        sage: b = B(rows=[[1,2],[4]])
        sage: b.lusztig_involution()
        [[1, 4], [3]]
        sage: b.to_tableau().schuetzenberger_involution(n=4)
        [[1, 4], [3]]
        sage: all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involution(n=4) for b in B)
        True

        sage: B = crystals.Tableaux(['D',4],shape=[1])
        sage: [(b,b.lusztig_involution()) for b in B]
        [[[[1]], [[-1]], [[2]], [[-2]], [[3]], [[-3]], [[4]], [[-4]], [[-4]]],
         [[[3]], [[-3]], [[2]], [[-2]], [[1]], [[-1]], [[1]]]]

        sage: B = crystals.Tableaux(['D',3],shape=[1])
        sage: [(b,b.lusztig_involution()) for b in B]
        [[[[1]], [[-1]], [[2]], [[-2]], [[3]], [[-3]], [[-3]], [[-3]], [[1]]],
         [[[3]], [[3]], [[-3]], [[2]], [[-2]], [[1]], [[-1]], [[1]]]]

        sage: C = CartanType(['E',6])
        sage: La = C.root_system().weight_lattice().fundamental_weights()
        sage: T = crystals.HighestWeight(La[1])
        sage: t = T[3]; t
        (-4, 2, 5)
        sage: t.lusztig_involution()
        (-2, -3, 4)

        >>> from sage.all import *
        >>> B = crystals.Tableaux(['A',Integer(3)],shape=[Integer(2),Integer(1)])
        >>> b = B(rows=[[Integer(1),Integer(2)],[Integer(4)]])
        >>> b.lusztig_involution()
        [[1, 4], [3]]
        >>> b.to_tableau().schuetzenberger_involution(n=Integer(4))
        [[1, 4], [3]]
        >>> all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involution(n=Integer(4)) for b in B)
>>> B = crystals.Tableaux(['D',Integer(4)],shape=[Integer(1)])
>>> [[b,b.lusztig_involution()] for b in B]
[[[1]], [[-1]], [[2]], [[-2]], [[3]], [[-3]], [[4]], [[-4]],
 [[4]], [[-3]], [[3]], [[-2]], [[2]], [[-1]], [[1]]]

>>> B = crystals.Tableaux(['D',Integer(3)],shape=[Integer(1)])
>>> [[b,b.lusztig_involution()] for b in B]
[[[1]], [[-1]], [[2]], [[-2]], [[3]], [[3]], [[-3]], [[-3]],
 [[-2]], [[2]], [[-1]], [[1]]]

>>> C = CartanType(['E',Integer(6)])
>>> La = C.root_system().weight_lattice().fundamental_weights()
>>> T = crystals.HighestWeight(La[Integer(1)])
>>> t = T[Integer(3)]; t
(-4, 2, 5)
>>> t.lusztig_involution()
(-2, -3, 4)

class ParentMethods
Bases: object
cardinality()

Returns the number of elements of the crystal, using Weyl's dimension formula on each connected component.

EXAMPLES:
sage: C = ClassicalCrystals().example(5)
sage: C.cardinality()
6

>>> from sage.all import *
>>> C = ClassicalCrystals().example(Integer(5))
>>> C.cardinality()
6

character (R=None)

Returns the character of this crystal.

INPUT:
• R—a WeylCharacterRing (default: the default WeylCharacterRing for this Cartan type)

Returns the character of self as an element of R.

EXAMPLES:
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: chi = C.character(); chi
A2(2,1,0)
sage: T = crystals.TensorProduct(C,C)
sage: chiT = T.character(); chiT
A2(2,2,2) + 2*A2(3,2,1) + A2(3,3,0) + A2(4,1,1) + A2(4,2,0)
sage: chiT == chi^2
True
```python
>>> from sage.all import *
>>> C = crystals.Tableaux("A2", shape=[Integer(2),Integer(1)])
>>> chi = C.character(); chi
A2(2,1,0)

>>> T = crystals.TensorProduct(C,C)
>>> chiT = T.character(); chiT
A2(2,2,2) + 2*A2(3,2,1) + A2(3,3,0) + A2(4,1,1) + A2(4,2,0)
>>> chiT == chi**Integer(2)
True
```

One may specify an alternate `WeylCharacterRing`:

```python
sage: R = WeylCharacterRing("A2", style="coroots")
sage: chiT = T.character(R); chiT
A2(0,0) + 2*A2(1,1) + A2(0,3) + A2(3,0) + A2(2,2)
sage: chiT in R
True
```

```python
>>> from sage.all import *
>>> R = WeylCharacterRing("A2", style="coroots")
>>> chiT = T.character(R); chiT
A2(0,0) + 2*A2(1,1) + A2(0,3) + A2(3,0) + A2(2,2)
>>> chiT in R
True
```

It should have the same Cartan type and use the same realization of the weight lattice as `self`:

```python
sage: R = WeylCharacterRing("A3", style="coroots")
sage: T.character(R)
Traceback (most recent call last):
...
ValueError: Weyl character ring does not have the right Cartan type
```

```python
>>> from sage.all import *
>>> R = WeylCharacterRing("A3", style="coroots")
>>> T.character(R)
Traceback (most recent call last):
...
ValueError: Weyl character ring does not have the right Cartan type
```

### demazure_character(w, f=None)

Return the Demazure character associated to `w`.

**INPUT:**
- `w` – an element of the ambient weight lattice realization of the crystal, or a reduced word, or an element in the associated Weyl group

**OPTIONAL:**
- `f` – a function from the crystal to a module

This is currently only supported for crystals whose underlying weight space is the ambient space.

The Demazure character is obtained by applying the Demazure operator $D_w$ (see `sage.categories.regular_crystals.RegularCrystals.ParentMethods.demazure_operator()`) to the highest weight element of the classical crystal. The simple Demazure operators $D_i$ (see `sage.categories.regular_crystals.RegularCrystals.ElementMethods.demazure_operator_simple()`) do not braid on the level of crystals,
but on the level of characters they do. That is why it makes sense to input \( w \) either as a weight, a reduced word, or as an element of the underlying Weyl group.

EXAMPLES:

```
sage: T = crystals.Tableaux(['A',2], shape = [2,1])
sage: e = T.weight_lattice_realization().basis()
sage: weight = e[0] + 2*e[2]
sage: weight.reduced_word()
[2, 1]
sage: T.demazure_character(weight)
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x1*x3^2

sage: T = crystals.Tableaux(['A',3], shape=[2,1])
sage: T.demazure_character([1,2,3])
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x2^2*x3
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([1,2,3])
sage: T.demazure_character(w)
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x2^2*x3

sage: T = crystals.Tableaux(['B',2], shape = [2,1])
sage: e = T.weight_lattice_realization().basis()
sage: weight = -2*e[1]
sage: T.demazure_character(weight)
x1^2 + x1*x2 + x2^2 + x1 + x2 + x1/x2 + 1/x2 + 1/x2^2 + 1
```
REFERENCES:
• [De1974]
• [Ma2009]

class TensorProducts (category, *args)

Bases: TensorProductsCategory

The category of classical crystals constructed by tensor product of classical crystals.

extra_super_categories ()

EXAMPLES:

sage: ClassicalCrystals().TensorProducts().extra_super_categories()
[Category of classical crystals]

additional_structure ()

Return None.

Indeed, the category of classical crystals defines no additional structure: it only states that its objects are $U_q(g)$-crystals, where $g$ is of finite type.

See also:
Category.additional_structure ()

EXAMPLES:

sage: ClassicalCrystals().additional_structure()

example (n=3)

Returns an example of highest weight crystals, as per Category.example ().

EXAMPLES:

sage: B = ClassicalCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1

super_categories ()

EXAMPLES:
4.17 Coalgebras

class sage.categories.coalgebras.Coalgebras(base, name=None)
    Bases: Category_over_base_ring

The category of coalgebras

EXAMPLES:

sage: Coalgebras(QQ)
Category of coalgebras over Rational Field
sage: Coalgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

>>> from sage.all import *
>>> Coalgebras(QQ)
Category of coalgebras over Rational Field
>>> Coalgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

class Cocommutative(base_category)
    Bases: CategoryWithAxiom_over_base_ring

Category of cocommutative coalgebras.

class DualObjects(category, *args)
    Bases: DualObjectsCategory

extra_super_categories()
Return the dual category.

EXAMPLES:

The category of coalgebras over the Rational Field is dual to the category of algebras over the same field:

sage: C = Coalgebras(QQ)
sage: C.dual()
Category of duals of coalgebras over Rational Field
sage: C.dual().super_categories() # indirect doctest
[Category of algebras over Rational Field,
 Category of duals of vector spaces over Rational Field]
>>> from sage.all import *
>>> C = Coalgebras(QQ)
>>> C.dual()
Category of duals of coalgebras over Rational Field
>>> C.dual().super_categories() # indirect doctest
[Category of algebras over Rational Field,
 Category of duals of vector spaces over Rational Field]

Warning: This is only correct in certain cases (finite dimension, ...). See Issue #15647.

class ElementMethods
    Bases: object

coproduct ()
    Return the coproduct of self.

    EXAMPLES:

    sage: # needs sage.groups sage.modules
    sage: A = HopfAlgebrasWithBasis(QQ).example(); A
    An example of Hopf algebra with basis:
    the group algebra of the Dihedral group of order 6 as a permutation group
    over Rational Field
    sage: [a,b] = A.algebra_generators()
    sage: a, a.coproduct()
    (B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
    sage: b, b.coproduct()
    (B[(1,3)], B[(1,3)] # B[(1,3)])

    counit ()
    Return the counit of self.

    EXAMPLES:

    sage: # needs sage.groups sage.modules
    sage: A = HopfAlgebrasWithBasis(QQ).example(); A
    An example of Hopf algebra with basis:
    the group algebra of the Dihedral group of order 6 as a permutation group
    over Rational Field
    sage: [a,b] = A.algebra_generators()
    sage: a, a.counit()
    (B[(1,2,3)], 1)

    (continues on next page)
class Filtered(base_category)

Bases: FilteredModulesCategory

Category of filtered coalgebras.

Graded

alias of GradedCoalgebras

class ParentMethods

Bases: object

coproduct(x)

Return the coproduct of x.

Eventually, there will be a default implementation, delegating to the overloading mechanism and forcing
the conversion back

EXAMPLES:

sage: # needs sage.groups sage.modules
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6 as a permutation group
over Rational Field
sage: [a,b] = A.algebra_generators()
sage: a, A.coproduct(a)
(B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
sage: b, A.coproduct(b)
(B[(1,3)], B[(1,3)] # B[(1,3)])

4.17. Coalgebras 319
counit(x)

Return the counit of x.

Eventually, there will be a default implementation, delegating to the overloading mechanism and forcing the conversion back.

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6 as a permutation group
over Rational Field
sage: [a,b] = A.algebra_generators()
sage: a, A.counit(a)
(B[(1,2,3)], 1)
sage: b, A.counit(b)
(B[(1,3)], 1)
```

TODO: implement some tests of the axioms of coalgebras, bialgebras and Hopf algebras using the counit.

class Realizations(category, *args)

Bases: RealizationsCategory

class ParentMethods

Bases: object

coproduct_by_coercion(x)

Return the coproduct by coercion if coproduct_by_basis is not implemented.

EXAMPLES:

```python
sage: # needs sage.modules
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = m[2,1]
sage: f.coproduct.__module__
'sage.categories.coalgebras'
sage: m.coprod_on_basis
NotImplemented
sage: m.coprod == m.coprod_by_coercion
True
sage: f.coprod()
# needs lrcalc_python
```
>>> from sage.all import *
>>> # needs sage.modules
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> f = m[Integer(2), Integer(1)]
>>> f.coproduct.__module__
'sage.categories.coalgebras'
>>> m.coproduct_on_basis
NotImplemented
>>> m.coproduct == m.coproduct_by_coercion
True
>>> f.coproduct()
˓→ needs lrcalc_python

sage: # needs sage.modules
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: R = N.ribbon()
sage: R.coproduct_by_coercion.__module__
'sage.categories.coalgebras'
sage: R.coproduct_on_basis
NotImplemented
sage: R.coproduct == R.coproduct_by_coercion
True
sage: R[1].coproduct()

>>> from sage.all import *
>>> # needs sage.modules
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> R = N.ribbon()
>>> R.coproduct_by_coercion.__module__
'sage.categories.coalgebras'
>>> R.coproduct_on_basis
NotImplemented
>>> R.coproduct == R.coproduct_by_coercion
True
>>> R[Integer(1)].coproduct()

counit_by_coercion(x)

Return the counit of $x$ if $\text{counit\_by\_basis}$ is not implemented.

EXAMPLES:

sage: # needs lrcalc_python sage.modules
sage: sp = SymmetricFunctions(QQ).sp()

sage: sp.an_element()
2*sp[] + 2*sp[1] + 3*sp[2]

sage: sp.counit(sp.an_element())
2

sage: o = SymmetricFunctions(QQ).o()

sage: o.an_element()
2*o[] + 2*o[1] + 3*o[2]

sage: o.counit(o.an_element())
-1

4.17. Coalgebras
>>> from sage.all import *
>>> # needs lrcalc_python sage.modules
>>> sp = SymmetricFunctions(QQ).sp()
>>> sp.an_element()
2*sp[]} + 2*sp[1] + 3*sp[2]
>>> sp.counit(sp.an_element())
2
>>> o = SymmetricFunctions(QQ).o()
>>> o.an_element()
2*o[]} + 2*o[1] + 3*o[2]
>>> o.counit(o.an_element())
-1

class SubcategoryMethods
    Bases: object
    
    Cocommutative()
    
        Return the full subcategory of the cocommutative objects of self.
        
        A coalgebra $C$ is said to be cocommutative if
        
        $$\Delta(c) = \sum_{(c)} c^{(1)} \otimes c^{(2)} = \sum_{(c)} c^{(2)} \otimes c^{(1)}$$
        
        in Sweedler's notation for all $c \in C$.
        
        EXAMPLES:
        
        sage: C1 = Coalgebras(ZZ).Cocommutative().WithBasis(); C1
        Category of cocommutative coalgebras with basis over Integer Ring
        sage: C2 = Coalgebras(ZZ).WithBasis().Cocommutative()
        sage: C1 is C2
        True
        sage: BialgebrasWithBasis(QQ).Cocommutative()
        Category of cocommutative bialgebras with basis over Rational Field

    >>> from sage.all import *
    >>> C1 = Coalgebras(ZZ).Cocommutative().WithBasis(); C1
    Category of cocommutative coalgebras with basis over Integer Ring
    >>> C2 = Coalgebras(ZZ).WithBasis().Cocommutative()
    >>> C1 is C2
    True
    >>> BialgebrasWithBasis(QQ).Cocommutative()
    Category of cocommutative bialgebras with basis over Rational Field

class Super (base_category)
    Bases: SuperModulesCategory
    
    class SubcategoryMethods
        Bases: object
        
        Supercocommutative()
        
            Return the full subcategory of the supercocommutative objects of self.
            
            EXAMPLES:
sage: Coalgebras(ZZ).WithBasis().Super().Supercocommutative()
Category of supercocommutative super coalgebras with basis over...
→ Integer Ring
sage: BialgebrasWithBasis(QQ).Super().Supercocommutative()
Join of Category of super algebras with basis over Rational Field
and Category of super bialgebras over Rational Field
and Category of super coalgebras with basis over Rational Field
and Category of supercocommutative super coalgebras over Rational...
→ Field

>>> from sage.all import *

>>> Coalgebras(ZZ).WithBasis().Super().Supercocommutative()
Category of supercocommutative super coalgebras with basis over...
→ Integer Ring
>>> BialgebrasWithBasis(QQ).Super().Supercocommutative()
Join of Category of super algebras with basis over Rational Field
and Category of super bialgebras over Rational Field
and Category of super coalgebras with basis over Rational Field
and Category of supercocommutative super coalgebras over Rational...
→ Field

class Supercocommutative(base_category)

  Bases: CategoryWithAxiom_over_base_ring

  Category of supercocommutative coalgebras.

  extra_super_categories()

  EXAMPLES:

sage: Coalgebras(ZZ).Super().extra_super_categories()
[Category of graded coalgebras over Integer Ring]
sage: Coalgebras(ZZ).Super().super_categories()
[Category of graded coalgebras over Integer Ring,
  Category of super modules over Integer Ring]

>>> from sage.all import *

>>> Coalgebras(ZZ).Super().extra_super_categories()
[Category of graded coalgebras over Integer Ring]
>>> Coalgebras(ZZ).Super().super_categories()
[Category of graded coalgebras over Integer Ring,
  Category of super modules over Integer Ring]

Compare this with the situation for bialgebras:

sage: Bialgebras(ZZ).Super().extra_super_categories()
[]
sage: Bialgebras(ZZ).Super().super_categories()
[Category of super algebras over Integer Ring,
  Category of super coalgebras over Integer Ring]

>>> from sage.all import *

>>> Bialgebras(ZZ).Super().extra_super_categories()
[]
>>> Bialgebras(ZZ).Super().super_categories()
[Category of super algebras over Integer Ring,
  Category of super coalgebras over Integer Ring]
The category of bialgebras does not occur in these results, since super bialgebras are not bialgebras.

```python
class TensorProducts (category, *args)
    Bases: TensorProductsCategory

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

extra_super_categories()

EXAMPLES:

sage: Coalgebras(QQ).TensorProducts().extra_super_categories()
[Category of coalgebras over Rational Field]

sage: Coalgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of vector spaces over Rational Field,
  Category of coalgebras over Rational Field]
```

Meaning: a tensor product of coalgebras is a coalgebra

```python
WithBasis
    alias of CoalgebrasWithBasis

class WithRealizations (category, *args)
    Bases: WithRealizationsCategory

class ParentMethods
    Bases: object

coproduct (x)
    Return the coproduct of x.

EXAMPLES:

sage: # needs sage.modules
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: N.coproduct.__module__
'sage.categories.coalgebras'
sage: N.coproduct(S[2])
```

```python
>>> from sage.all import *
>>> Coalgebras(QQ).TensorProducts().extra_super_categories()
[Category of coalgebras over Rational Field]
>>> Coalgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of vector spaces over Rational Field,
  Category of coalgebras over Rational Field]
```

```python
>>> from sage.all import *
>>> NonCommutativeSymmetricFunctions(QQ)
>>> S = N.complete()
>>> N.coproduct.__module__
'sage.categories.coalgebras'
>>> N.coproduct(S[Integer(2)])
```
\textbf{counit}(x)

Return the counit of $x$.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: # needs sage.modules
c Sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: f = s[2,1]
sage: f.counit.__module__
'sage.categories.coalgebras'
sage: f.counit()
0

>>> from sage.all import *
>>> # needs sage.modules
>>> Sym = SymmetricFunctions(QQ)
>>> s = Sym.schur()
>>> f = s[Integer(2),Integer(1)]
>>> f.counit.__module__
'sage.categories.coalgebras'
>>> f.counit()
0

sage: # needs sage.modules
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.counit.__module__
'sage.categories.coalgebras'
sage: N.counit(N.one())
1
sage: x = N.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
sage: N.counit(x)
2

>>> from sage.all import *
>>> # needs sage.modules
>>> N = NonCommutativeSymmetricFunctions(QQ)
>>> N.counit.__module__
'sage.categories.coalgebras'
>>> N.counit(N.one())
1
>>> x = N.an_element(); x
2*S[] + 2*S[1] + 3*S[1, 1]
>>> N.counit(x)
2
\end{verbatim}

\textbf{super_categories}()

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Coalgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

>>> from sage.all import *
>>> Coalgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]
\end{verbatim}

\section{Coalgebras}
4.18 Coalgebras with basis

class sage.categories.coalgebras_with_basis.CoalgebrasWithBasis(base_category):
    Bases: CategoryWithAxiom_over_base_ring

    The category of coalgebras with a distinguished basis.

    EXAMPLES:

    sage: CoalgebrasWithBasis(ZZ)
    Category of coalgebras with basis over Integer Ring
    sage: sorted(CoalgebrasWithBasis(ZZ).super_categories(), key=str)
    [Category of coalgebras over Integer Ring,
     Category of modules with basis over Integer Ring]

    >>> from sage.all import *
    >>> CoalgebrasWithBasis(ZZ)
    Category of coalgebras with basis over Integer Ring
    >>> sorted(CoalgebrasWithBasis(ZZ).super_categories(), key=str)
    [Category of coalgebras over Integer Ring,
     Category of modules with basis over Integer Ring]

    class ElementMethods
        Bases: object

        coproduct_iterated(n=1)
            Apply n coproducts to self.

            Todo: Remove dependency on modules_with_basis methods.

        EXAMPLES:

        sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
        # needs sage.combinat sage.modules
        sage: Psi[2,2].coproduct_iterated(0)
        # needs sage.combinat sage.modules
        Psi[2, 2]
        sage: Psi[2,2].coproduct_iterated(2)
        # needs sage.combinat sage.modules

        >>> from sage.all import *
        >>> Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
        # needs sage.combinat sage.modules
        >>> Psi[Integer(2),Integer(2)].coproduct_iterated(Integer(0))
        # needs sage.combinat sage.modules
        Psi[2, 2]
        >>> Psi[Integer(2),Integer(2)].coproduct_iterated(Integer(2))
        # needs sage.combinat sage.modules

326 Chapter 4. Individual Categories
class Filtered(base_category)
    Bases: FilteredModulesCategory
    Category of filtered coalgebras.

Graded
    alias of GradedCoalgebrasWithBasis

class ParentMethods
    Bases: object
    coproduct()
        If coproduct_on_basis() is available, construct the coproduct morphism from self to self ⊗ self by extending it by linearity. Otherwise, use coproduct_by_coercion(), if available.

EXAMPLES:

sage: # needs sage.groups sage.modules
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis:
    the group algebra of the Dihedral group of order 6
    as a permutation group over Rational Field
sage: a, b = A.algebra_generators()
sage: a, A.coproduct(a)
(B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
sage: b, A.coproduct(b)
(B[(1,3)], B[(1,3)] # B[(1,3)])

>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis:
    the group algebra of the Dihedral group of order 6
    as a permutation group over Rational Field
>>> a, b = A.algebra_generators()
>>> a, A.coproduct(a)
>>> a, A.coproduct(a)
(B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
>>> b, A.coproduct(b)
>>> b, A.coproduct(b)
(B[(1,3)], B[(1,3)] # B[(1,3)])

coproduct_on_basis(i)
    The coproduct of the algebra on the basis (optional).

    INPUT:
    • i -- the indices of an element of the basis of self
    
    Returns the coproduct of the corresponding basis elements If implemented, the coproduct of the algebra is defined from it by linearity.

    EXAMPLES:
B[(1,2,3)] # B[(1,2,3)]

```python
from sage.all import *

A = HopfAlgebrasWithBasis(QQ).example(); A
# needs sage.groups sage.modules
```

An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a permutation group over Rational Field

```python
(a, b) = A._group.gens()
# needs sage.groups sage.modules

A.coproduct_on_basis(a)
# needs sage.groups sage.modules

B[(1,2,3)] # B[(1,2,3)]
```

counit()

If `counit_on_basis()` is available, construct the counit morphism from `self` to `self ⊗ self` by extending it by linearity.

**EXAMPLES:**

```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A

sage: a, b = A.algebra_generators()
sage: a, A.counit(a)
(B[(1,2,3)], 1)
sage: b, A.counit(b)
(B[(1,3)], 1)
```

counit_on_basis(i)

The counit of the algebra on the basis (optional).

**INPUT:**

• `i` – the indices of an element of the basis of `self`

Returns the counit of the corresponding basis elements If implemented, the counit of the algebra is defined from it by linearity.

**EXAMPLES:**

```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
# needs sage.groups sage.modules
```
(continues on next page)
An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6
as a permutation group over Rational Field

sage: (a, b) = A._group.gens()  
# needs sage.groups sage.modules
sage: A.counit_on_basis(a)  
# needs sage.groups sage.modules

```python
>>> from sage.all import *

>>> A = HopfAlgebrasWithBasis(QQ).example(); A
```

```
An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6
as a permutation group over Rational Field

```python
>>> (a, b) = A._group.gens()  
# needs sage.groups sage.modules
>>> A.counit_on_basis(a)  
# needs sage.groups sage.modules
```

```python
class Super(base_category)

Bases: SuperModulesCategory

extra_super_categories()

EXAMPLES:

```python
sage: C = Coalgebras(ZZ).WithBasis().Super()

```

```python
sage: sorted(C.super_categories(), key=str)  
# indirect doctest
```

```
4.19 Commutative additive groups

```python
class sage.categories.commutative_additive_groups.CombinativeAdditiveGroups(base_category)

Bases: CategoryWithAxiom_singleton, AbelianCategory

The category of abelian groups, i.e. additive abelian monoids where each element has an inverse.

EXAMPLES:

```python
sage: C = CombinativeAdditiveGroups(); C
```

Category of commutative additive groups

```python
(continues on next page)```
sage: C.super_categories()
[Category of additive groups, Category of commutative additive monoids]
sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital →']
sage: C is CommutativeAdditiveMonoids().AdditiveInverse()
True
sage: from sage.categories.additive_groups import AdditiveGroups
sage: C is AdditiveGroups().AdditiveCommutative()
True

Note: This category is currently empty. It's left there for backward compatibility and because it is likely to grow in the future.

class Algebras(category, *args)
    Bases: AlgebrasCategory

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

class ElementMethods
    Bases: object

    additive_order()
    Return the additive order of this element.

    EXAMPLES:

sage: G = cartesian_product([Zmod(3), Zmod(6), Zmod(5)])
sage: G((1,1,1)).additive_order()
30
sage: any((i * G((1,1,1))).is_zero() for i in range(1,30))
False
sage: 30 * G((1,1,1))
(0, 0, 0)

sage: G = cartesian_product([ZZ, ZZ])
sage: G((0,0)).additive_order()
1
sage: G((0,1)).additive_order()
sage: # needs sage.rings.finite_rings
sage: K = GF(9)
sage: H = cartesian_product([cartesian_product([Zmod(2), Zmod(9)]), K])
sage: z = H(((1,2), K.gen()))
sage: z.additive_order()
18

>>> from sage.all import *

>>> G = cartesian_product([Zmod(Integer(3)), Zmod(Integer(6)),
→Zmod(Integer(5))])

>>> G((Integer(1),Integer(1),Integer(1))).additive_order()
30

>>> any((i * G((Integer(1),Integer(1),Integer(1)))).is_zero() for i in
→range(Integer(1),Integer(30)))
False

>>> Integer(30) * G((Integer(1),Integer(1),Integer(1)))
(0, 0, 0)

>>> G = cartesian_product([ZZ, ZZ])

>>> G((Integer(0),Integer(0))).additive_order()
1

>>> G((Integer(0),Integer(1))).additive_order()
+Infinity

>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(9))

>>> H = cartesian_product([cartesian_product([Zmod(Integer(2)), Zmod(Integer(9))]), K])

>>> z = H(((Integer(1),Integer(2)), K.gen()))

>>> z.additive_order()
18

4.20 Commutative additive monoids

class sage.categories.commutative_additive_monoids.CommutativeAdditiveMonoids(base_category)

Bases: CategoryWithAxiom_singleton

The category of commutative additive monoids, that is abelian additive semigroups with a unit

EXAMPLES:

sage: C = CommutativeAdditiveMonoids(); C
Category of commutative additive monoids

sage: C.super_categories()

[Category of additive monoids, Category of commutative additive semigroups]

sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveUnital']

sage: C is AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().

(continues on next page)
\begin{verbatim}
>>> from sage.all import *
>>> C = CommutativeAdditiveMonoids(); C
Category of commutative additive monoids
>>> C.super_categories()
[Category of additive monoids, Category of commutative additive semigroups]
>>> sorted(C.axioms())
[AdditiveAssociative, AdditiveCommutative, AdditiveUnital]
>>> C is AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().AdditiveUnital()
True
\end{verbatim}

\textbf{Note}: This category is currently empty and only serves as a place holder to make \texttt{C.example()} work.

\section*{4.21 Commutative additive semigroups}

\begin{verbatim}
class sage.categories.commutative_additive_semigroups.CommutativeAdditiveSemigroups

Bases: CategoryWithAxiom_singleton

The category of additive abelian semigroups, i.e. sets with an associative and abelian operation +.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: C = CommutativeAdditiveSemigroups(); C
Category of commutative additive semigroups
sage: C.example()
An example of a commutative semigroup: the free commutative semigroup generated by ('a', 'b', 'c', 'd')
sage: sorted(C.super_categories(), key=str)
[Category of additive commutative additive magmas, Category of additive semigroups]
sage: C is AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().AdditiveUnital()
True
\end{verbatim}
\end{verbatim}
sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative']
C is AdditiveMagmas().AdditiveAssociative().AdditiveCommutative()
True

Note: This category is currently empty and only serves as a placeholder to make C.example() work.

4.22 Commutative algebra ideals

class sage.categories.commutative_algebra_ideals.CommutativeAlgebraIdeals(A):
   Bases: Category_ideal

   The category of ideals in a fixed commutative algebra A.

   EXAMPLES:

   sage: C = CommutativeAlgebraIdeals(QQ['x'])
   sage: C
   Category of commutative algebra ideals in
   Univariate Polynomial Ring in x over Rational Field

   sage: from sage.all import *
   >>> C = CommutativeAlgebraIdeals(QQ['x'])
   >>> C
   Category of commutative algebra ideals in
   Univariate Polynomial Ring in x over Rational Field

   algebra()  

   EXAMPLES:

   sage: CommutativeAlgebraIdeals(QQ['x']).algebra()  
   Univariate Polynomial Ring in x over Rational Field

   sage: from sage.all import *
   >>> CommutativeAlgebraIdeals(QQ['x']).algebra()  
   Univariate Polynomial Ring in x over Rational Field

   super_categories()  

   EXAMPLES:

   sage: CommutativeAlgebraIdeals(QQ['x']).super_categories()  
   [Category of algebra ideals in Univariate Polynomial Ring in x over Rational...
   Field]

   sage: from sage.all import *
   >>> CommutativeAlgebraIdeals(QQ['x']).super_categories()  
   [Category of algebra ideals in Univariate Polynomial Ring in x over Rational...
   Field]
4.23 Commutative algebras

class sage.categories.commutative_algebras.CommutativeAlgebras(base_category):

    Bases: CategoryWithAxiom_over_base_ring

    The category of commutative algebras with unit over a given base ring.

    EXAMPLES:

    sage: M = CommutativeAlgebras(GF(19))
    sage: M
    Category of commutative algebras over Finite Field of size 19
    sage: CommutativeAlgebras(QQ).super_categories()
    [Category of algebras over Rational Field, Category of commutative rings]

    >>> from sage.all import *
    >>> M = CommutativeAlgebras(GF(Integer(19)))
    >>> M
    Category of commutative algebras over Finite Field of size 19
    >>> CommutativeAlgebras(QQ).super_categories()
    [Category of algebras over Rational Field, Category of commutative rings]

    This is just a shortcut for:

    sage: Algebras(QQ).Commutative() # Category of commutative algebras over Rational Field

    >>> from sage.all import *
    >>> Algebras(QQ).Commutative() # Category of commutative algebras over Rational Field

    Todo:
    • product (= Cartesian product)
    • coproduct (= tensor product over base ring)

    class TensorProducts(category, *args):

        Bases: TensorProductsCategory

        The category of commutative algebras constructed by tensor product of commutative algebras.

        extra_super_categories()

        EXAMPLES:

        sage: Algebras(QQ).Commutative().TensorProducts().extra_super_categories()
        [Category of commutative rings]
        sage: Algebras(QQ).Commutative().TensorProducts().super_categories()
        [Category of tensor products of algebras over Rational Field, 
        Category of commutative algebras over Rational Field]

        >>> from sage.all import *
        >>> Algebras(QQ).Commutative().TensorProducts().extra_super_categories()
        [Category of commutative rings]
        >>> Algebras(QQ).Commutative().TensorProducts().super_categories()
4.24 Commutative ring ideals

class sage.categories.commutative_ring_ideals.CombinativeRingIdeals(R)

Bases: Category_ideal

The category of ideals in a fixed commutative ring.

EXAMPLES:

```
sage: C = CommutativeRingIdeals(IntegerRing())
sage: C
Category of commutative ring ideals in Integer Ring
```

```
>>> from sage.all import *
>>> C = CommutativeRingIdeals(IntegerRing())
>>> C
Category of commutative ring ideals in Integer Ring
```

```
super_categories()

EXAMPLES:

```
sage: CommutativeRingIdeals(ZZ).super_categories()
[Category of ring ideals in Integer Ring]
```

```
>>> from sage.all import *
>>> CommutativeRingIdeals(IZZ).super_categories()
[Category of ring ideals in Integer Ring]
```

4.25 Commutative rings

class sage.categories.commutative_rings.CombinativeRings(base_category)

Bases: CategoryWithAxiom_singleton

The category of commutative rings

commutative rings with unity, i.e. rings with commutative * and a multiplicative identity

EXAMPLES:

```
sage: C = CommutativeRings(); C
Category of commutative rings
sage: C.super_categories()
[Category of rings, Category of commutative monoids]
```

```
>>> from sage.all import *
>>> C = CommutativeRings(); C
Category of commutative rings
```
class CartesianProducts (category, *args)
    Bases: CartesianProductsCategory

    extra_super_categories()
    Let Sage knows that Cartesian products of commutative rings is a commutative ring.
    EXAMPLES:

    sage: CommutativeRings().Commutative().CartesianProducts().extra_super_categories()
    [Category of commutative rings]
    sage: cartesian_product([ZZ, Zmod(34), ....: QQ, GF(5)]) in CommutativeRings()
    True

    >>> from sage.all import *
    ... CommutativeRings().Commutative().CartesianProducts().extra_super_categories()
    [Category of commutative rings]
    >>> cartesian_product([ZZ, Zmod(Integer(34)), ....: QQ, GF(Integer(5))]) in CommutativeRings()
    True

class ElementMethods
    Bases: object

class Finite (base_category)
    Bases: CategoryWithAxiom_singleton

    Check that Sage knows that Cartesian products of finite commutative rings is a finite commutative ring.
    EXAMPLES:

    sage: cartesian_product([Zmod(34), ....: GF(5)]) in Rings().Commutative().Finite()
    True

    >>> from sage.all import *
    ... cartesian_product([Zmod(Integer(34)), ....: GF(Integer(5))]) in Rings().Commutative().Finite()
    True

class ParentMethods
    Bases: object

    cyclotomic_cosets (q, cosets=None)
    Return the (multiplicative) orbits of $q$ in the ring.
    Let $R$ be a finite commutative ring. The group of invertible elements $R^*$ in $R$ gives rise to a group action on $R$ by multiplication. An orbit of the subgroup generated by an invertible element $q$ is called a $q$-cyclotomic coset (since in a finite ring, each invertible element is a root of unity).

    These cosets arise in the theory of minimal polynomials of finite fields, duadic codes and combinatorial designs. Fix a primitive element $z$ of $GF(q^k)$. The minimal polynomial of $z^a$ over $GF(q)$ is.
given by

\[ M_s(x) = \prod_{i \in C_s} (x - z^i), \]

where \( C_s \) is the \( q \)-cyclotomic coset mod \( n \) containing \( s, n = q^k - 1 \).

**Note:** When \( R = \mathbb{Z}/n\mathbb{Z} \) the smallest element of each coset is sometimes called a **coset leader**. This function returns sorted lists so that the coset leader will always be the first element of the coset.

**INPUT:**
- \( q \) – an invertible element of the ring
- \( \text{cosets} \) – an optional list of elements of \( \text{self} \). If provided, the function only return the list of cosets that contain some element from \( \text{cosets} \).

**OUTPUT:**
A list of lists.

**EXAMPLES:**

```
sage: Zmod(11).cyclotomic_cosets(2)
[[0], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
sage: Zmod(15).cyclotomic_cosets(2)
[[0], [1, 2, 4, 8], [3, 6, 9, 12], [5, 10], [7, 11, 13, 14]]
```

Since the group of invertible elements of a finite field is cyclic, the set of squares is a particular case of cyclotomic coset:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(25, 'z')
sage: a = K.multiplicative_generator()
sage: K.cyclotomic_cosets(a**2, cosets=[1])
[[1, 2, 3, 4, z + 1, z + 3, 2*z + 1, 2*z + 2, 3*z + 3, 3*z + 4, 4*z + 2, 4*z + 4]]
sage: sorted(b for b in K if not b.is_zero() and b.is_square())
[1, 2, 3, 4, z + 1, z + 3, 2*z + 1, 2*z + 2, 3*z + 3, 3*z + 4, 4*z + 2, 4*z + 4]
```

(continues on next page)
We compute some examples of minimal polynomials:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(27, 'z')
sage: a = K.multiplicative_generator()
sage: R.<X> = PolynomialRing(K, 'X')
sage: a.minimal_polynomial('X')
X^3 + 2*X + 1

sage: cyc3 = Zmod(26).cyclotomic_cosets(3, cosets=[1]); cyc3
[[1, 3, 9]]
sage: prod(X - a**i for i in cyc3[0])  # needs sage.rings.finite_rings
X^3 + 2*X + 1

sage: (a**7).minimal_polynomial('X')  # needs sage.rings.finite_rings
X^3 + X^2 + 2*X + 1

sage: cyc7 = Zmod(26).cyclotomic_cosets(3, cosets=[7]); cyc7
[[7, 11, 21]]
sage: prod(X - a**i for i in cyc7[0])  # needs sage.rings.finite_rings
X^3 + X^2 + 2*X + 1
```

Cyclotomic cosets of fields are useful in combinatorial design theory to provide so called difference families (see Wikipedia article Difference_set and difference_family). This is illustrated on the following examples:

```
sage: K = GF(5)
sage: a = K.multiplicative_generator()  # needs sage.rings.finite_rings
```
needs sage.libs.pari

sage: H = K.cyclotomic_cosets(a**2, cosets=[1, 2]); H
[[1, 4], [2, 3]]

sage: sorted(x - y for D in H for x in D for y in D if x != y)
[1, 2, 3, 4]

sage: K = GF(37)
sage: a = K.multiplicative_generator()

sage: H = K.cyclotomic_cosets(a**4, cosets=[1]); H
[[1, 7, 9, 10, 12, 16, 26, 33, 34]]

sage: sorted(x - y for D in H for x in D for y in D if x != y)
[1, 2, 3, 4]

The method `cyclotomic_cosets` works on any finite commutative ring:

```
>>> from sage.all import *
>>> K = GF(Integer(5))
>>> a = K.multiplicative_generator()

>>> H = K.cyclotomic_cosets(a**Integer(2), cosets=[Integer(1),
             Integer(2)]); H
[[1, 4], [2, 3]]

>>> sorted(x - y for D in H for x in D for y in D if x != y)
[1, 2, 3, 4]
```

```
>>> K = GF(Integer(37))
>>> a = K.multiplicative_generator()

>>> H = K.cyclotomic_cosets(a**Integer(4), cosets=[Integer(1)]); H
[[1, 7, 9, 10, 12, 16, 26, 33, 34]]

>>> sorted(x - y for D in H for x in D for y in D if x != y)
[1, 2, 3, 4]
```

extra_super_categories()

Let Sage know that finite commutative rings are Noetherian.

EXAMPLES:
class ParentMethods
Bases: object
derivation (arg=\texttt{None}, twist=\texttt{None})

Return the twisted or untwisted derivation over this ring specified by \texttt{arg}.

\textbf{Note:} A twisted derivation with respect to $\theta$ (or a $\theta$-derivation for short) is an additive map $d$ satisfying the following axiom for all $x, y$ in the domain:

\[d(xy) = \theta(x)d(y) + d(x)y.\]

\textbf{INPUT:}
\begin{itemize}
  \item \texttt{arg} – (optional) a generator or a list of coefficients that defines the derivation
  \item \texttt{twist} – (optional) the twisting homomorphism
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}sage: R.<x,y,z> = QQ[]
sage: R.derivation()
˓→ # needs sage.modules
d/dx:
\end{verbatim}

\begin{verbatim}>>> from sage.all import *
   >>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
   >>> R.derivation()
˓→ # needs sage.modules
d/dx
\end{verbatim}

In that case, \texttt{arg} could be a generator:

\begin{verbatim}sage: R.derivation(y)
˓→ # needs sage.modules
d/dy
\end{verbatim}

\begin{verbatim}>>> from sage.all import *
   >>> R.derivation(y)
˓→ # needs sage.modules
d/dy
\end{verbatim}

or a list of coefficients:

\begin{verbatim}sage: R.derivation([1,2,3])
˓→ # needs sage.modules
d/dx + 2*d/dy + 3*d/dz
\end{verbatim}
It is not possible to define derivations with respect to a polynomial which is not a variable:

```python
sage: R.derivation(x^2)  # needs sage.modules
Traceback (most recent call last):
...
ValueError: unable to create the derivation
```

Here is an example with twisted derivations:

```python
sage: theta = R.hom([x^2, y^2, z^2])
sage: f = R.derivation(twist=theta); f  # needs sage.modules
0
```

Specifying a scalar, the returned twisted derivation is the corresponding multiple of $\theta - \text{id}$:

```python
sage: R.derivation(1, twist=theta)  # needs sage.modules
[x |---> x^2, y |---> y^2, z |---> z^2] - id
```

(continues on next page)
### derivation_module (codomain=None, twist=None)

Return the module of derivations over this ring.

**INPUT:**

- **codomain** — an algebra over this ring or a ring homomorphism whose domain is this ring or None (default: None); if it is a morphism, the codomain of derivations will be the codomain of the morphism viewed as an algebra over `self` through the given morphism; if None, the codomain will be this ring

- **twist** — a morphism from this ring to codomain or None (default: None); if None, the coercion map from this ring to codomain will be used

**Note:** A twisted derivation with respect to \(\theta\) (or a \(\theta\)-derivation for short) is an additive map \(d\) satisfying the following axiom for all \(x, y\) in the domain:

\[
d(xy) = \theta(x)d(y) + d(x)y.
\]

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y, z over Rational Field
sage: M.gens()
(d/dx, d/dy, d/dz)
```

We can specify a different codomain:

```python
sage: K = R.fraction_field()
sage: M = R.derivation_module(K); M
Module of derivations from Multivariate Polynomial Ring in x, y, z over Rational Field to Fraction Field of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: M.gen() / x
1/x*d/dx
```
Here is an example with a non-canonical defining morphism:

```python
sage: ev = R.hom([QQ(0), QQ(1), QQ(2)])
sage: ev
Ring morphism:

From: Multivariate Polynomial Ring in x, y, z over Rational Field
To:   Rational Field
Defn: x |--> 0
       y |--> 1
       z |--> 2

sage: M = R.derivation_module(ev)

```

Elements in $M$ acts as derivations at $(0, 1, 2)$:

```python
sage: Dx = M.gen(0); Dx
d/dx
sage: Dy = M.gen(1); Dy
d/dy
sage: Dz = M.gen(2); Dz
d/dz
sage: f = x^2 + y^2 + z^2
```

(continues on next page)
An example with a twisting homomorphism:

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> Dx = M.gen(Integer(0)); Dx
d/dx
>>> Dy = M.gen(Integer(1)); Dy
d/dy
>>> Dz = M.gen(Integer(2)); Dz
d/dz
>>> f = x**Integer(2) + y**Integer(2) + z**Integer(2)
>>> Dx(f)  # = 2*x evaluated at (0,1,2)
0
>>> Dy(f)  # = 2*y evaluated at (0,1,2)
2
>>> Dz(f)  # = 2*z evaluated at (0,1,2)
4
```

See also:

derivation()

\texttt{frobenius_endomorphism}(n=1)

Return the Frobenius endomorphism.

\begin{itemize}
\item \texttt{n} – a nonnegative integer (default: 1)
\end{itemize}

\textbf{OUTPUT}:

The $n$-th power of the absolute arithmetic Frobenius endomorphism on this commutative ring.

\textbf{EXAMPLES}:

```python
sage: K.<u> = PowerSeriesRing(GF(5))
sage: Frob = K.frobenius_endomorphism(); Frob
```
Frobenius endomorphism \( x \mapsto x^5 \) of Power Series Ring in \( u \) over Finite Field of size 5

```
sage: Frob(u)
u^5
```

We can specify a power:

```
sage: f = K.frobenius_endomorphism(2); f
Frobenius endomorphism \( x \mapsto x^{(5^2)} \) of Power Series Ring in \( u \) over Finite Field of size 5

sage: f(1+u)
1 + u^25
```

### is_commutative()

Return whether the ring is commutative.

The answer is `True` only if the category is a sub-category of `CommutativeRings`.

It is recommended to use instead `R in Rings().Commutative()`.

#### EXAMPLES:

```
sage: QQ.is_commutative()
True
sage: QQ['x,y,z'].is_commutative()
True
```

```
>>> from sage.all import *
>>> QQ.is_commutative()
True
>>> QQ['x,y,z'].is_commutative()
True
```

### over \((\text{base}=\text{None}, \text{gen}=\text{None}, \text{gens}=\text{None}, \text{name}=\text{None}, \text{names}=\text{None})\)

Return this ring, considered as an extension of \( \text{base} \).

#### INPUT:

- \text{base} – a commutative ring or a morphism or \text{None} (default: \text{None}); the base of this extension or its defining morphism
- \text{gen} – a generator of this extension (over its base) or \text{None} (default: \text{None});
• gens – a list of generators of this extension (over its base) or None (default: None);
• name – a variable name or None (default: None)
• names – a list or a tuple of variable names or None (default: None)

EXAMPLES:

We construct an extension of finite fields:

```python
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2)
sage: k = GF(5^4)
sage: z4 = k.gen()
sage: K = k.over(F); K
Field in z4 with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base
```

If not explicitly given, the default generator of the top ring (here k) is used and the same name is kept:

```python
sage: K.gen()
needs sage.modules sage.rings.finite_rings
z4
sage: K(z4)
needs sage.modules sage.rings.finite_rings
z4
```

However, it is possible to specify another generator and/or another name. For example:

```python
sage: # needs sage.modules sage.rings.finite_rings
sage: Ka = k.over(F, name='a'); Ka
Field in a with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base
sage: Ka.gen()
a
sage: Ka(z4)
a
```

(continues on next page)
Field in \(b\) with defining polynomial \(x^2 + z^2 + 4\) over its base

```python
sage: Kb.gen()
\texttt{b}
```

```python
sage: Kb(-z4+1)
\texttt{b}
```

```python
>>> from sage.all import *
>>> # needs sage.modules sage.rings.finite_rings
>>> Ka = k.over(F, name='a'); Ka
Field in \(a\) with defining polynomial
\(x^2 + (4*z^2 + 3)*x + z^2\) over its base

>>> Ka.gen()
a
>>> Ka(z4)
a

>>> # needs sage.modules sage.rings.finite_rings
>>> Kb = k.over(F, gen=-z4+Integer(1), name='b')
>>> Kb
Field in \(b\) with defining polynomial \(x^2 + z^2 + 4\) over its base

>>> Kb.gen()
b
>>> Kb(-z4+Integer(1))
b
```

Note that the shortcut \(K.<a>\) is also available:

```python
sage: KKa.<a> = k.over(F)  # needs sage.modules sage.rings.finite_rings
```

```python
sage: KKa
\texttt{K}\texttt{a}

>>> from sage.all import *
>>> # needs sage.modules sage.rings.finite_rings
>>> (a,) = KKa._first_ngens(1)
```

Building an extension on top of another extension is allowed:

```python
sage: L = GF(5^12).over(K); L  # needs sage.modules sage.rings.finite_rings
Field in \(z_{12}\) with defining polynomial
\(x^3 + (1 + (4*z^2 + 2)*z^4)*x^2 + (2 + 2*z^4)*x - z^4\) over its base

sage: L.base_ring()  # needs sage.modules sage.rings.finite_rings
Field in \(z^4\) with defining polynomial
\(x^2 + (4*z^2 + 3)*x + z^2\) over its base
```

```python
>>> from sage.all import *
>>> L = GF(Integer(5)**Integer(12)).over(K); L  # needs sage.modules sage.rings.finite_rings
```

(continues on next page)
x^3 + (1 + (4*z2 + 2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base

```python
>>> L.base_ring()

Field in z4 with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base
```

The successive bases of an extension are accessible via the method `sage.rings.
ring_extension.RingExtension_generic.bases()`:

```python
sage: L.bases()  #...

[Field in z12 with defining polynomial
x^3 + (1 + (4*z2 + 2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base,
Field in z4 with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base,
Finite Field in z2 of size 5^2]
```

```python
>>> from sage.all import *

>>> L.bases()  #...

[Field in z12 with defining polynomial
x^3 + (1 + (4*z2 + 2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base,
Field in z4 with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base,
Finite Field in z2 of size 5^2]
```

When `base` is omitted, the canonical base of the ring is used:

```python
sage: S.<x> = QQ[]
sage: E = S.over(); E  #...

Univariate Polynomial Ring in x over Rational Field over its base
sage: E.base_ring()  #...

Rational Field
```

```python
>>> from sage.all import *

>>> S = QQ['x']; (x,) = S._first_ngens(1)

>>> E = S.over(); E  #...

Univariate Polynomial Ring in x over Rational Field over its base
```

Rational Field

Here is an example where `base` is a defining morphism:

```python
sage: # needs sage.modules sage.rings.number_field
sage: k.<a> = QQ.extension(x^2 - 2)
sage: l.<b> = QQ.extension(x^4 - 2)
sage: f = k.hom([b^2])
sage: L = l.over(f)
sage: L
Field in b with defining polynomial x^2 - a over its base
sage: L.base_ring()
Number Field in a with defining polynomial x^2 - 2
```
Similarly, one can create a tower of extensions:

```python
sage: # needs sage.modules sage.rings.number_field
sage: K = k.over()
sage: L = l.over(Hom(K, l)(f)); L
Field in b with defining polynomial $x^2 - a$ over its base
sage: L.base_ring()
Field in a with defining polynomial $x^2 - 2$ over its base
sage: L.bases()
[Field in b with defining polynomial $x^2 - a$ over its base,
  Field in a with defining polynomial $x^2 - 2$ over its base,
  Rational Field]
```

## 4.26 Complete Discrete Valuation Rings (CDVR) and Fields (CDVF)

class
sage.categories.complete_discrete_valuation.CompleteDiscreteValuationFields

Bases: Category_singleton

The category of complete discrete valuation fields

EXAMPLES:

```python
sage: Zp(7) in CompleteDiscreteValuationFields()  # needs sage.rings.padics
False
sage: QQ in CompleteDiscreteValuationFields()
False
sage: LaurentSeriesRing(QQ, 'u') in CompleteDiscreteValuationFields()
True
```
```python
sage: Qp(7) in CompleteDiscreteValuationFields()  # needs sage.rings.padics
True
sage: TestSuite(CompleteDiscreteValuationFields()).run()

>>> from sage.all import *
>>> Zp(Integer(7)) in CompleteDiscreteValuationFields()  # needs sage.rings.padics
False
>>> QQ in CompleteDiscreteValuationFields()
False
>>> LaurentSeriesRing(QQ, 'u') in CompleteDiscreteValuationFields()
True
>>> Qp(Integer(7)) in CompleteDiscreteValuationFields()  # needs sage.rings.padics
True
>>> TestSuite(CompleteDiscreteValuationFields()).run()

class ElementMethods
    Bases: object
denominator()

    Return the denominator of this element normalized as a power of the uniformizer

    EXAMPLES:

sage: # needs sage.rings.padics
sage: K = Qp(7)
sage: x = K(1/21)
sage: x.denominator()
7 + O(7^21)
sage: x = K(7)
sage: x.denominator()
1 + O(7^20)

>>> from sage.all import *
>>> # needs sage.rings.padics
>>> K = Qp(Integer(7))
>>> x = K(Integer(1)/Integer(21))
>>> x.denominator()
7 + O(7^21)
>>> x = K(Integer(7))
>>> x.denominator()
1 + O(7^20)

Note that the denominator lives in the ring of integers:

sage: x.denominator().parent()  # needs sage.rings.padics
7-adic Ring with capped relative precision 20

>>> from sage.all import *
>>> x.denominator().parent()  # needs sage.rings.padics
7-adic Ring with capped relative precision 20
```
When the denominator is indistinguishable from 0 and the precision on the input is $O(p^n)$, the return value is 1 if $n$ is nonnegative and $p^{k-n}$ otherwise:

```sage
# needs sage.rings.padics
sage: x = K(0, 5); x
O(7^5)
sage: x.denominator()
1 + O(7^20)
sage: x = K(0, -5); x
O(7^-5)
sage: x.denominator()
7^5 + O(7^25)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> x = K(Integer(0), Integer(5)); x
O(7^5)
>>> x.denominator()
1 + O(7^20)
>>> x = K(Integer(0), -Integer(5)); x
O(7^-5)
>>> x.denominator()
7^5 + O(7^25)
```

### numerator()

Return the numerator of this element, normalized in such a way that $x = x.numerator() / x.denominator()$ always holds true.

**EXAMPLES:**

```sage
# needs sage.rings.padics
sage: K = Qp(7, 5)
sage: x = K(1/21)
sage: x.numerator()
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + O(7^5)
sage: x == x.numerator() / x.denominator()
True
```

```python
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> K = Qp(Integer(7), Integer(5))
>>> x = K(Integer(1)/Integer(21))
>>> x.numerator()
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + O(7^5)
>>> x == x.numerator() / x.denominator()
True
```

Note that the numerator lives in the ring of integers:

```sage
sage: x.numerator().parent()
# needs sage.rings.padics
7-adic Ring with capped relative precision 5
```

```python
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> x.numerator().parent()
# needs sage.rings.padics
7-adic Ring with capped relative precision 5
```
valuation()

Return the valuation of this element.

EXAMPLES:

```
sage: K = Qp(7)  # needs sage.rings.padics
sage: x = K(7); x  # needs sage.rings.padics
7 + O(7^21)
sage: x.valuation()  # needs sage.rings.padics
1
```

super_categories()

EXAMPLES:

```
sage: CompleteDiscreteValuationFields().super_categories()
[Category of discrete valuation fields]
```

class

sage.categories.complete_discretevaluation.CompleteDiscreteValuationRings

Bases: Category_singleton

The category of complete discrete valuation rings

EXAMPLES:

```
sage: Zp(7) in CompleteDiscreteValuationRings()  # needs sage.rings.padics
True
sage: QQ in CompleteDiscreteValuationRings()
False
sage: QQ[['u']] in CompleteDiscreteValuationRings()
True
sage: Qp(7) in CompleteDiscreteValuationRings()  # needs sage.rings.padics
False
sage: TestSuite(CompleteDiscreteValuationRings()).run()
```
class ElementMethods

   Bases: object

   denominator()

   Return the denominator of this element normalized as a power of the uniformizer

   EXAMPLES:

```
sage: # needs sage.rings.padics
sage: K = Qp(7)
sage: x = K(1/21)
sage: x.denominator()
7 + O(7^21)
sage: x = K(7)
sage: x.denominator()
1 + O(7^20)
```

Note that the denominator lives in the ring of integers:

```
sage: x.denominator().parent()  # needs sage.rings.padics
7-adic Ring with capped relative precision 20
```

When the denominator is indistinguishable from 0 and the precision on the input is $O(p^n)$, the return value is 1 if $n$ is nonnegative and $p^(-n)$ otherwise:

```
sage: # needs sage.rings.padics
sage: x = K(0, 5); x
0(7^5)
sage: x.denominator()
1 + O(7^20)
```
lift_to_precision (absprec=None)

Return another element of the same parent with absolute precision at least absprec, congruent to this element modulo the precision of this element.

INPUT:
• absprec—an integer or None (default: None), the absolute precision of the result. If None, lifts to the maximum precision allowed.

Note: If setting absprec that high would violate the precision cap, raises a precision error. Note that the new digits will not necessarily be zero.

EXAMPLES:

sage: # needs sage.rings.padics
sage: R = ZpCA(17)
sage: R(-1, 2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1, 15).lift_to_precision(10)
1 + O(17^15)
sage: R(1, 15).lift_to_precision(30)
Traceback (most recent call last):
... PrecisionError: precision higher than allowed by the precision cap
sage: (R(-1, 2).lift_to_precision().precision_absolute())
....:   == R.precision_cap()
True

sage: R = Zp(5); c = R(17, 3); c.lift_to_precision(8)  # needs sage.rings.padics
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()  # needs sage.rings.padics
True

>>> from sage.all import *
>>> # needs sage.rings.padics
>>> import sage.all as all
...
 numerator()  
Return the numerator of this element, normalized in such a way that \( x = x.\text{numerator}() / x.\text{denominator}() \) always holds true.

EXAMPLES:

```
sage: # needs sage.rings.padics
e sage: K = Qp(7, 5)
e sage: x = K(1/21)
e sage: x.numerator()  
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + O(7^5)  
sage: x == x.numerator() / x.denominator()  
True
```

Note that the numerator lives in the ring of integers:

```
sage: x.numerator().parent()  
7-adic Ring with capped relative precision 5
```

valuation()  
Return the valuation of this element.

EXAMPLES:
sage: R = Zp(7)  # needs sage.rings.padics
sage: x = R(7); x  # needs sage.rings.padics
7 + O(7^21)
sage: x.valuation()  # needs sage.rings.padics
1

>>> from sage.all import *

>>> R = Zp(Integer(7))  # needs sage.rings.padics

>>> x = R(Integer(7)); x  # needs sage.rings.padics
7 + O(7^21)

>>> x.valuation()  # needs sage.rings.padics
1

super_categories()

EXAMPLES:

sage: CompleteDiscreteValuationRings().super_categories()
[Category of discrete valuation rings]

4.27 Complex reflection groups

class sage.categories.complex_reflection_groups.ComplexReflectionGroups

Bases: Category_singleton

The category of complex reflection groups.

Let $V$ be a complex vector space. A complex reflection is an element of $\text{GL}(V)$ fixing a hyperplane pointwise and acting by multiplication by a root of unity on a complementary line.

A complex reflection group is a group $W$ that is (isomorphic to) a subgroup of some general linear group $\text{GL}(V)$ generated by a distinguished set of complex reflections.

The dimension of $V$ is the rank of $W$.

For a comprehensive treatment of complex reflection groups and many definitions and theorems used here, we refer to [LT2009]. See also Wikipedia article Reflection_group.

See also:

ReflectionGroup() for usage examples of this category.

EXAMPLES:

sage: from sage.categories.complex_reflection_groups import _

sage: ComplexReflectionGroups()

(continues on next page)
Category of complex reflection groups

sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized Coxeter groups]

sage: ComplexReflectionGroups().all_super_categories()
[Category of complex reflection groups,
 Category of complex reflection or generalized Coxeter groups,
 Category of groups,
 Category of monoids,
 Category of finitely generated semigroups,
 Category of semigroups,
 Category of finitely generated magmas,
 Category of inverse unital magmas,
 Category of unital magmas,
 Category of magmas,
 Category of enumerated sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

>>> from sage.all import *
>>> from sage.categories.complex_reflection_groups import ComplexReflectionGroups
>>> ComplexReflectionGroups()
Category of complex reflection groups

Category of complex reflection or generalized Coxeter groups

complex reflection groups

An example of a reflection group:

sage: W = ComplexReflectionGroups().example(); W
#...
˓
needs sage.combinat
5-colored permutations of size 3

>>> from sage.all import *
>>> W = ComplexReflectionGroups().example(); W
˓
needs sage.combinat
5-colored permutations of size 3

W is in the category of complex reflection groups:

sage: W in ComplexReflectionGroups()
˓
needs sage.combinat
True

```python
>>> from sage.all import *
```

```
>>> W in ComplexReflectionGroups()  # needs sage.combinat
True
```

### Finite

alias of `FiniteComplexReflectionGroups`

#### class ParentMethods

**Bases:** object

**rank()**

Return the rank of `self`.

The rank of `self` is the dimension of the smallest faithfull reflection representation of `self`.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: W.rank()
3
```

### additional_structure()

Return `None`.

Indeed, all the structure complex reflection groups have in addition to groups (simple reflections, ...) is already defined in the super category.

**See also:**

`Category.additional_structure()`

**EXAMPLES:**

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().additional_structure()
```

### example()

Return an example of a complex reflection group.

**EXAMPLES:**
4.28 Common category for Generalized Coxeter Groups or Complex Reflection Groups

The category of complex reflection groups or generalized Coxeter groups.

Finite Coxeter groups can be defined equivalently as groups generated by reflections, or by presentations. Over the last decades, the theory has been generalized in both directions, leading to the study of (finite) complex reflection groups on the one hand, and (finite) generalized Coxeter groups on the other hand. Many of the features remain similar, yet, in the current state of the art, there is no general theory covering both directions.

This is reflected by the name of this category which is about factoring out the common code, tests, and declarations.

A group in this category has:

- A distinguished finite set of generators \((s_i)_I\), called simple reflections. The set \(I\) is called the index set. The name “reflection” is somewhat of an abuse as they can have higher order; still, they are all of finite order: \(s_i^k = 1\) for some \(k\).
- A collection of distinguished reflections which are the conjugates of the simple reflections. For complex reflection groups, they are in one-to-one correspondence with the reflection hyperplanes and share the same index set.
- A collection of reflections which are the conjugates of all the non trivial powers of the simple reflections.
The usual notions of reduced words, length, irreducibility, etc. can be canonically defined from the above.

The following methods must be implemented:

- `ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods.index_set()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods.simple_reflection()`

Optionally one can define analog methods for distinguished reflections and reflections (see below).

At least one of the following methods must be implemented:

- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection_left()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection_right()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods._mul_()`

It's recommended to implement either `_mul_` or both `apply_simple_reflection_left` and `apply_simple_reflection_right`.

See also:

- `complex_reflection_groups.ComplexReflectionGroups`
- `generalized_coxeter_groups.GeneralizedCoxeterGroups`

EXAMPLES:

```python
sage: from sage.categories.complex_reflection_or_generalized_coxeter_groups import ComplexReflectionOrGeneralizedCoxeterGroups
sage: C = ComplexReflectionOrGeneralizedCoxeterGroups(); C
Category of complex reflection or generalized Coxeter groups
sage: C.super_categories()
[Category of finitely generated enumerated groups]
```

(continues on next page)
class ElementMethods
   Bases: object

   apply_conjugation_by_simple_reflection(i)
       Conjugate self by the i-th simple reflection.

       EXAMPLES:

       sage: W = WeylGroup(['A', 3])
       # needs sage.rings.number_field
       sage: w = W.from_reduced_word([3, 1, 2, 1])
       # needs sage.rings.number_field
       sage: w.apply_conjugation_by_simple_reflection(1).reduced_word()  # needs sage.rings.number_field
       [3, 2]

apply_reflections(word, side='right', word_type='all')
      Return the result of the (left/right) multiplication of self by word.

      INPUT:
      • word — a sequence of indices of reflections
      • side — (default: 'right') indicates multiplying from left or right
      • word_type — (default: 'all'): either 'simple', 'distinguished', or 'all'

      EXAMPLES:

      sage: W = ReflectionGroup((1,1,3))
      sage: W.one().apply_reflections([1])
      (1,4)(2,3)(5,6)
      sage: W.one().apply_reflections([2])
      (1,3)(2,5)(4,6)
      sage: W.one().apply_reflections([2,1])
      (1,2,6)(3,4,5)

      sage: W = CoxeterGroups().example()
      sage: w = W.an_element(); w
      (1, 2, 3, 0)
      sage: w.apply_reflections([0,1], word_type='simple')
      (2, 3, 1, 0)
      sage: w
      (1, 2, 3, 0)
sage: w.apply_reflections([0,1], side='left', word_type='simple')
(0, 1, 3, 2)

sage: # needs sage.rings.number_field
sage: W = WeylGroup("A3", prefix='s')

sage: w = W.an_element(); w
s1*s2*s3

sage: AS = W.domain()

sage: r1 = AS.roots()[4]; r1
(0, 1, 0, -1)

sage: r2 = AS.roots()[5]; r2
(0, 0, 1, -1)

sage: w.apply_reflections([r1, r2], word_type=all)
s1

sage: # optional - gap3
sage: W = ReflectionGroup((1,1,3), hyperplane_index_set=['A', 'B', 'C']); W
Irreducible real reflection group of rank 2 and type A2

sage: W.one().apply_reflections([A], word_type=distinguished)
(1,4)(2,3)(5,6)

>>> from sage.all import *

>>> W = ReflectionGroup((1,1,3), hyperplane_index_set=[1,1,3])

>>> W.one().apply_reflections([1])
(1,4)(2,3)(5,6)

>>> W.one().apply_reflections([2])
(1,3)(2,5)(4,6)

>>> W.one().apply_reflections([3])
(1,5)(2,4)(3,6)

>>> W.one().apply_reflections([2,1])
(1,2,6)(3,4,5)

>>> W = CoxeterGroups().example()

>>> w = W.an_element(); w
(1, 2, 3, 0)

>>> w.apply_reflections([Integer(0),Integer(1)], word_type='simple')
(2, 3, 1, 0)

>>> w
(1, 2, 3, 0)

>>> w.apply_reflections([Integer(0),Integer(1)], side='left', word_type='simple')
(0, 1, 3, 2)

>>> # needs sage.rings.number_field
>>> W = WeylGroup("A3", prefix='s')
>>> w = W.an_element(); w
s1*s2*s3
>>> AS = W.domain()
>>> r1 = AS.roots()[Integer(4)]; r1
(0, 1, 0, -1)
>>> r2 = AS.roots()[Integer(5)]; r2
(0, 0, 1, -1)
>>> w.apply_reflections([r1, r2], word_type='all')
s1

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

Return self multiplied by the simple reflection $s[i]$.

INPUT:
- $i$ – an element of the index set
- $side$ – (default: "right") "left" or "right"

This default implementation simply calls apply_simple_reflection_left() or apply_simple_reflection_right().

EXAMPLES:

sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection(0, side="left")
(0, 2, 3, 1)
sage: w.apply_simple_reflection(1, side="left")
(2, 1, 3, 0)
sage: w.apply_simple_reflection(2, side="left")
(1, 3, 2, 0)
sage: w.apply_simple_reflection(0, side="right")
(2, 1, 3, 0)
sage: w.apply_simple_reflection(1, side="right")
(1, 3, 2, 0)
sage: w.apply_simple_reflection(2, side="right")
(1, 2, 0, 3)
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> w = W.an_element(); w
(1, 2, 3, 0)
>>> w.apply_simple_reflection(Integer(0), side="left")
(0, 2, 3, 1)
>>> w.apply_simple_reflection(Integer(1), side="left")
(2, 1, 3, 0)
>>> w.apply_simple_reflection(Integer(2), side="left")
(1, 3, 2, 0)
>>> w.apply_simple_reflection(Integer(0), side="right")
(2, 1, 3, 0)
>>> w.apply_simple_reflection(Integer(1), side="right")
(1, 3, 2, 0)
>>> w.apply_simple_reflection(Integer(2), side="right")
(1, 2, 0, 3)

By default, side is "right":

sage: w.apply_simple_reflection(0)
(2, 1, 3, 0)

Some tests with a complex reflection group:

sage: # needs sage.combinat
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: W = ComplexReflectionGroups().example(); W
5-colored permutations of size 3
sage: w = W.an_element(); w
[[1, 0, 0], [3, 1, 2]]

sage: w.apply_simple_reflection(1, side="left")
[[0, 1, 0], [1, 3, 2]]

sage: w.apply_simple_reflection(2, side="left")
[[1, 0, 0], [3, 2, 1]]

sage: w.apply_simple_reflection(3, side="left")
[[1, 0, 1], [3, 1, 2]]

sage: w.apply_simple_reflection(1, side="right")
[[1, 0, 0], [3, 2, 1]]

sage: w.apply_simple_reflection(2, side="right")
[[1, 0, 0], [2, 1, 3]]

sage: w.apply_simple_reflection(3, side="right")
[[2, 0, 0], [3, 1, 2]]

(continues on next page)
apply_simple_reflection_left \( (i) \)

Return \( \text{self} \) multiplied by the simple reflection \( s[i] \) on the left.

This low level method is used intensively. Coxeter groups are encouraged to override this straightforward implementation whenever a faster approach exists.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection_left(0)
(0, 2, 3, 1)
sage: w.apply_simple_reflection_left(1)
(2, 1, 3, 0)
sage: w.apply_simple_reflection_left(2)
(1, 3, 2, 0)
```

```python
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> w = W.an_element(); w
(1, 2, 3, 0)
>>> w.apply_simple_reflection_left(Integer(0))
(0, 2, 3, 1)
>>> w.apply_simple_reflection_left(Integer(1))
(2, 1, 3, 0)
>>> w.apply_simple_reflection_left(Integer(2))
(1, 3, 2, 0)
```

**EXAMPLES:**

```python
sage: # needs sage.combinat
sage: from sage.categories.complex_reflection_groups import...
    ComplexReflectionGroups
sage: W = ComplexReflectionGroups().example()
sage: w = W.an_element(); w
[[1, 0, 0], [3, 1, 2]]
sage: w.apply_simple_reflection_left(1)
[[0, 1, 0], [1, 3, 2]]
sage: w.apply_simple_reflection_left(2)
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection_left(3)
[[1, 0, 1], [3, 1, 2]]
```
apply_simple_reflection_right\( (i) \)
Return \( self \) multiplied by the simple reflection \( s[i] \) on the right.

This low level method is used intensively. Coxeter groups are encouraged to override this straightforward implementation whenever a faster approach exists.

EXAMPLES:

```
sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection_right(0)
(2, 1, 3, 0)
sage: w.apply_simple_reflection_right(1)
(1, 3, 2, 0)
sage: w.apply_simple_reflection_right(2)
(1, 2, 0, 3)
```

(continues on next page)
apply_simple_reflections (word, side='right', type='simple')

Return the result of the (left/right) multiplication of self by word.

INPUT:
• word – a sequence of indices of simple reflections
• side – (default: 'right') indicates multiplying from left or right

This is a specialized implementation of apply_reflections() for the simple reflections. The rationale for its existence are:
• It can take advantage of apply_simple_reflection, which often is less expensive than computing a product.
• It reduced burden on implementations that would want to provide an optimized version of this method.

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1])
(2, 3, 1, 0)
sage: w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1], side='left')
(0, 1, 3, 2)
```

is_reflection()

Return whether self is a reflection.

EXAMPLES:

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1,1,4))
sage: [t.is_reflection() for t in W.reflections()]
[False, True, False]
```
reflection_length()

Return the reflection length of \texttt{self}.

This is the minimal length of a factorization of \texttt{self} into reflections.

EXAMPLES:

>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup((1,1,2))
>>> sorted([t.reflection_length() for t in W])
[0, 1]

>>> W = ReflectionGroup((2,1,2))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 2, 2, 2]

>>> W = ReflectionGroup((3,1,2))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

>>> W = ReflectionGroup((2,2,2))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 2]

>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 2, 2, 2]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(2)))
>>> sorted([t.reflection_length() for t in W])
[0, 1, 1, 1, 2]
class Irreducible

    Bases: CategoryWithAxiom

class ParentMethods

    Bases: object

    irreducible_components()

        Return a list containing all irreducible components of self as finite reflection groups.

        EXAMPLES:

        sage: W = ColoredPermutations(4, 3)
        sage: W.irreducible_components()  # does not really work
        ([4-colored permutations of size 3])

    distinguished_reflection(i)

        Return the i-th distinguished reflection of self.

        INPUT:
        • i -- an element of the index set of the distinguished reflections.

        See also:
        • distinguished_reflections()
        • hyperplane_index_set()

        EXAMPLES:

        sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=('a','b','c','d','e','f'))
        sage: for i in W.hyperplane_index_set():
        ...     print('%s %s'(i, W.distinguished_reflection(i)))
        a (1,7)(2,4)(5,6)(8,10)(11,12)
        b (1,4)(2,8)(3,5)(7,10)(9,11)
        c (2,5)(3,9)(4,6)(8,11)(10,12)
        d (1,8)(2,7)(3,6)(4,10)(9,12)
        e (1,6)(2,9)(3,8)(5,11)(7,12)
        f (1,11)(3,10)(4,9)(5,7)(6,12)
distinguished_reflections()

Return a finite family containing the distinguished reflections of self, indexed by hyperplane_index_set().

A distinguished reflection is a conjugate of a simple reflection. For a Coxeter group, reflections and distinguished reflections coincide. For a Complex reflection groups this is a reflection acting on the complement of the fixed hyperplane \( H \) as \( \exp(2\pi i/n) \), where \( n \) is the order of the reflection subgroup fixing \( H \).

See also:

- distinguished_reflection()
- hyperplane_index_set()

EXAMPLES:

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1,1,3))
sage: distinguished_reflections = W.distinguished_reflections()
sage: for index in sorted(distinguished_reflections.keys()):
    print('%s %s' % (index, distinguished_reflections[index]))
  1 (1,4)(2,3)(5,6)
  2 (1,3)(2,5)(4,6)
  3 (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((1,1,3), hyperplane_index_set=['a','b','c'])
sage: distinguished_reflections = W.distinguished_reflections()
sage: for index in sorted(distinguished_reflections.keys()):
    print('%s %s' % (index, distinguished_reflections[index]))
  a (1,4)(2,3)(5,6)
  b (1,3)(2,5)(4,6)
  c (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((3,1,1))
  1 (1,2,3)

sage: W = ReflectionGroup((1,1,3), (3,1,2))
  1 (1,6)(2,5)(7,8)
  2 (1,5)(2,7)(6,8)
  3 (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30)
  5 (1,7)(2,6)(5,8)
```
from sage.all import *

# optional - gap3

W = ReflectionGroup((Integer(1), Integer(1), Integer(3)))
distinguished_reflections = W.distinguished_reflections()

for index in sorted(distinguished_reflections.keys):
    print('%s %s
%(index, distinguished_reflections[index])
1 (1,4) (2,3) (5,6)
2 (1,3) (2,5) (4,6)
3 (1,5) (2,4) (3,6)

W = ReflectionGroup((Integer(1), Integer(1), Integer(3)), hyperplane_"index_set=['a', 'b', 'c']")
distinguished_reflections = W.distinguished_reflections()

for index in sorted(distinguished_reflections.keys):
    print('%s %s
%(index, distinguished_reflections[index])
a (1,4) (2,3) (5,6)
b (1,3) (2,5) (4,6)
c (1,5) (2,4) (3,6)

W = ReflectionGroup((Integer(3), Integer(1), Integer(1)))
distinguished_reflections = W.distinguished_reflections()

for index in sorted(distinguished_reflections.keys):
    print('%s %s
%(index, distinguished_reflections[index])
1 (1,6) (2,5) (7,8)
2 (1,5) (2,7) (6,8)
3 (3,9,15) (4,10,16) (12,17,23) (14,18,24) (20,25,29) (21,22,26) (27,28,30)
4 (3,11) (4,12) (9,13) (10,14) (15,19) (16,20) (17,21) (18,22) (23,27) (24,28) (25,26) (29,30)
5 (1,7) (2,6) (5,8)
7 (4,21,27) (10,22,28) (11,13,19) (12,14,20) (16,26,30) (17,18,25) (23,24,29)
8 (3,13) (4,24) (9,19) (10,29) (11,15) (12,26) (14,21) (16,23) (17,30) (18,27) (20,22) (25,28)

from_reduced_word (word, word_type='simple')

Return an element of self from its (reduced) word.

INPUT:
- word -- a list (or iterable) of elements of the index set of self (resp. of the distinguished or of all reflections)
- word_type (default: 'simple'): either 'simple', 'distinguished', or 'all'

If word is [i1, i2, ..., ik], then this returns the corresponding product of simple reflections s_{i_1}s_{i_2}...s_{i_k}.

If word_type is 'distinguished' (resp. 'all'), then the product of the distinguished reflections (resp. all reflections) is returned.

Note: The main use case is for constructing elements from reduced words, hence the name of this method. However, the input word need not be reduced.

See also:
- index_set()
• reflection_index_set()
• hyperplane_index_set()
• apply_simple_reflections()
• reduced_word()
• _test_reduced_word()

EXAMPLES:

```
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: W.from_reduced_word([0,2,0,1])
(0, 3, 1, 2)
sage: W.from_reduced_word((0,2,0,1))
(0, 3, 1, 2)
sage: s[0]*s[2]*s[0]*s[1]
(0, 3, 1, 2)
```

We now experiment with the different values for word_type for the colored symmetric group:

```
sage: # needs sage.combinat
sage: W = ColoredPermutations(1,4)
sage: W.from_reduced_word([1,2,1,2,1,2])
[[0, 0, 0, 0], [1, 2, 3, 4]]
sage: W.from_reduced_word([1, 2, 3]).reduced_word()
[1, 2, 3]
sage: # needs sage.rings.number_field
sage: W = WeylGroup("A3", prefix='s')
sage: AS = W.domain()
sage: r1 = AS.roots()[4]; r1
(0, 1, 0, -1)
sage: r2 = AS.roots()[5]; r2
(0, 0, 1, -1)
sage: W.from_reduced_word([r1, r2], word_type='all')
s3*s2
```

(continues on next page)
group_generators()

Return the simple reflections of self, as distinguished group generators.

See also:

- simple_reflections()
- Groups.ParentMethods.group_generators()
- Semigroups.ParentMethods.semigroup_generators()

EXAMPLES:
The simple reflections are also semigroup generators, even for an infinite group:

```
sage: W = WeylGroup("A",2,1)  # needs sage.rings.number_field

sage: W.semigroup_generators()  # needs sage.rings.number_field
Finite family {0: [-1  1  1]
   [ 0  1  0]
   [ 0  0  1],
1: [ 1  0  0]
   [ 1 -1  1]
   [ 0  0  1],
2: [ 1  0  0]
   [ 0  1  0]
   [ 1  1 -1]}
```
hyperplane_index_set()

Return the index set of the distinguished reflections of self. This is also the index set of the reflection hyperplanes of self, hence the name. This name is slightly abusive since the concept of reflection hyperplanes is not defined for all generalized Coxeter groups. However for all practical purposes this is only used for complex reflection groups, and there this is the desirable name.

See also:

- distinguished_reflection()
- distinguished_reflections()

EXAMPLES:

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1,1,4))
sage: W.hyperplane_index_set()
(1, 2, 3, 4, 5, 6)
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=[1,3,asd',7,9,-11])
sage: W.hyperplane_index_set()
(1, 3, asd', 7, 9, 11)
sage: W = ReflectionGroup((1,1,4),
    ...
    hyperplane_index_set=('a','b','c','d','e','f'))
sage: W.hyperplane_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```

index_set()

Return the index set of (the simple reflections of) self, as a list (or iterable).

See also:

- simple_reflection()
- simple_reflections()

EXAMPLES:
sage: W = CoxeterGroups().Finite().example(); W
The 5-th dihedral group of order 10
sage: W.index_set()
(1, 2)

sage: W = ColoredPermutations(1, 4)  # needs sage.combinat
sage: W.index_set()                  # needs sage.combinat
(1, 2, 3)

sage: W = ReflectionGroup((1,1,4), index_set=(a,b,c))

sage: W = ReflectionGroup((1,1,4), index_set=[1,3,'asdf'])

sage: W = ReflectionGroup((1,1,4), index_set=('a','b','c'))

>>> from sage.all import *
>>> W = CoxeterGroups().Finite().example(); W
The 5-th dihedral group of order 10
>>> W.index_set()
(1, 2)

>>> W = ColoredPermutations(Integer(1), Integer(4))  # needs sage.combinat
>>> W.index_set()                                      # needs sage.combinat
(1, 2, 3)

>>> # optional - gap3
>>> W = ReflectionGroup([1,1,3], [3,1,3], 4); W        # optional - gap3
Reducible complex reflection group of rank 7 and type A2 x G(3,1,3) x ST4
>>> sorted(W.irreducible_component_index_sets())       # optional - gap3
[[1, 2], [3, 4, 5], [6, 7]]

irreducible_component_index_sets()

Return a list containing the index sets of the irreducible components of self as finite reflection groups.

EXAMPLES:

sage: # needs sage.graphs
sage: W = ReflectionGroup([1,1,3], [3,1,3], 4); W        # optional - gap3
Reducible complex reflection group of rank 7 and type A2 x G(3,1,3) x ST4
sage: sorted(W.irreducible_component_index_sets())       # optional - gap3
[[1, 2], [3, 4, 5], [6, 7]]
sorted(W.irreducible_component_index_sets()) # optional - gap3
[[1, 2], [3, 4, 5], [6, 7]]

ALGORITHM:

Take the connected components of the graph on the index set with edges \((i,j)\), where \(s[i]\) and \(s[j]\) do not commute.

**irreducible_components()**

Return the irreducible components of \(self\) as finite reflection groups.

**EXAMPLES:**

```python
sage: # needs sage.graphs
sage: W = ReflectionGroup([1,1,3], [3,1,3], 4)  # optional - gap3
sage: W.irreducible_components()  # optional - gap3
[Irreducible real reflection group of rank 2 and type A2,
 Irreducible complex reflection group of rank 3 and type G(3,1,3),
 Irreducible complex reflection group of rank 2 and type ST4]
```

```python
from sage.all import *
```

```python
sage: # needs sage.graphs
sage: W = ReflectionGroup((1,1,3),(2,1,3)); W  # optional - gap3
Reducible real reflection group of rank 5 and type A2 x B3
sage: W.is_irreducible()  # optional - gap3
False
```

**is_irreducible()**

Return True if \(self\) is irreducible.

**EXAMPLES:**

```python
sage: # needs sage.graphs
sage: W = ColoredPermutations(1,3); W  # needs sage.combinat
1-colored permutations of size 3
sage: W.is_irreducible()  # needs sage.combinat
True
sage: W = ColoredPermutations(Integer(1),Integer(3)); W  # needs sage.combinat
1-colored permutations of size 3
sage: W.is_irreducible()  # needs sage.combinat
True
```

```python
from sage.all import *
```

```python
sage: # needs sage.graphs
sage: W = ColoredPermutations((1,1,3),(2,1,3)); W  # optional - gap3
Reducible real reflection group of rank 5 and type A2 x B3
sage: W.is_irreducible()  # optional - gap3
False
```

(continues on next page)
Reducible real reflection group of rank 5 and type A2 x B3

```python
>>> W.is_irreducible()
False  # optional - gap3
```

\textbf{is\_reducible()}

Return \texttt{True} if \texttt{self} is not irreducible.

\textbf{EXAMPLES:}

```python
sage: # needs sage.graphs
sage: W = ColoredPermutations(1,3); W
      # needs sage.combinat
1-colored permutations of size 3
sage: W.is_reducible()
      # needs sage.combinat
False
sage: W = ReflectionGroup((1,1,3), (2,1,3)); W
      # optional - gap3
Reducible real reflection group of rank 5 and type A2 x B3
sage: W.is_reducible()
      # optional - gap3
True
```

```python
>>> from sage.all import *
```

```python
>>> W = ColoredPermutations(Integer(1),Integer(3)); W
      # needs sage.combinat
1-colored permutations of size 3
```

```python
>>> W.is_reducible()
      # needs sage.combinat
False
```

```python
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)), (Integer(2),
      # optional - gap3
Integer(1),Integer(3))); W
Reducible real reflection group of rank 5 and type A2 x B3
```

```python
>>> W.is_reducible()
      # optional - gap3
True
```

\textbf{number\_of\_irreducible\_components()}

Return the number of irreducible components of \texttt{self}.

\textbf{EXAMPLES:}

```python
sage: # needs sage.graphs
sage: SymmetricGroup(3).number_of_irreducible_components()
1
sage: ColoredPermutations(1,3).number_of_irreducible_components()
      # needs sage.combinat
1
sage: ReflectionGroup((1,1,3),(2,1,3)).number_of_irreducible_components()
      # optional - gap3
2
```

```python
>>> from sage.all import *
```

```python
>>> SymmetricGroup(Integer(3)).number_of_irreducible_components()
1
```

```python
>>> ColoredPermutations(Integer(1),Integer(3)).number_of_irreducible_components()
      # needs sage.combinat
```
number_of_simple_reflections()

Return the number of simple reflections of self.

EXAMPLES:

```python
sage: # needs sage.combinat
sage: W = ColoredPermutations(1,3)
sage: W.number_of_simple_reflections()
2
sage: W = ColoredPermutations(2,3)
sage: W.number_of_simple_reflections()
3
sage: W = ColoredPermutations(4,3)
sage: W.number_of_simple_reflections()
3
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.number_of_simple_reflections()  # optional - gap3
4
```

reflection(i)

Return the i-th reflection of self.

For i in 1, ..., N, this gives the i-th reflection of self.

See also:

- `reflections_index_set()`
- `reflections()`

EXAMPLES:
reflection_index_set()

Return the index set of the reflections of self.

See also:

- reflection()
- reflections()

EXAMPLES:

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1,1,4))
(1, 2, 3, 4, 5, 6)
```

```python
>>> from sage.all import *
>>>
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
>>> W.reflection_index_set()
(1, 2, 3, 4, 5, 6)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), reflection_index_set=[Integer(1),Integer(3),'asdf',Integer(7),Integer(9),Integer(11)])
>>> W.reflection_index_set()
(1, 3, 'asdf', 7, 9, 11)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), reflection_index_set=('a','b','c','d','e','f'))
>>> W.reflection_index_set()
('a', 'b', 'c', 'd', 'e', 'f')
```
reflections()

Return a finite family containing the reflections of self, indexed by reflection_index_set().

See also:

• reflection()
• reflection_index_set()

EXAMPLES:

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1,1,3))
sage: reflections = W.reflections()
sage: for index in sorted(reflections.keys()):
    ....:     print('%s %s
    1 (1,4)(2,3)(5,6)
    2 (1,3)(2,5)(4,6)
    3 (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((1,1,3),
    ....:     reflection_index_set=['a','b','c'])
sage: reflections = W.reflections()
sage: for index in sorted(reflections.keys()):
    ....:     print('%s %s
    a (1,4)(2,3)(5,6)
    b (1,3)(2,5)(4,6)
    c (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((3,1,1))
sage: reflections = W.reflections()
sage: for index in sorted(reflections.keys()):
    ....:     print('%s %s
    1 (1,2,3)
    2 (1,3,2)

sage: W = ReflectionGroup((1,1,3), (3,1,2))
sage: reflections = W.reflections()
sage: for index in sorted(reflections.keys()):
    ....:     print('%s %s
    1 (1,6)(2,5)(7,8)
    2 (1,5)(2,7)(6,8)
    3 (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30)
        26)(29,30)
    5 (1,7)(2,6)(5,8)
        23)(24,26)
        22)(25,28)
```
```
semigroup_generators()

Return the simple reflections of self, as distinguished group generators.

See also:

• simple_reflections()
• Groups.ParentMethods.group_generators()
• Semigroups.ParentMethods.semigroup_generators()

EXAMPLES:

```python
sage: D10 = FiniteCoxeterGroups().example(10)
sage: D10.group_generators()
Finite family {1: (1,), 2: (2,)}
sage: SymmetricGroup(5).group_generators()
Finite family {1: (1,2), 2: (2,3), 3: (3,4), 4: (4,5)}
sage: W = ColoredPermutations(3,2)
# needs sage.combinat
```
The simple reflections are also semigroup generators, even for an infinite group:

\[
\text{sage: } W = \text{WeylGroup}(["A",2,1])
\]

\[
\text{sage: } W.\text{semigroup\_generators}()
\]

\[
\begin{align*}
0: & \begin{bmatrix} -1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
1: & \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \\
2: & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & -1 \end{bmatrix}
\end{align*}
\]

\[
\text{simple\_reflection}(i)
\]

Return the \(i\)-th simple reflection \(s_i\) of \(self\).
INPUT:
• i — an element from the index set

See also:
• index_set()
• simple_reflections()

EXAMPLES:

```
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: W.simple_reflection(1)
(0, 2, 1, 3)
sage: s = W.simple_reflections()
sage: s[1]
(0, 2, 1, 3)
sage: W = ReflectionGroup((1,1,4), index_set=[1,3,asdf])  # optional - gap3
>>> for i in W.index_set():
    # optional - gap3
    ....: print('%s %s'%(i, W.simple_reflection(i)))
1 (1,7)(2,4)(5,6)(8,10)(11,12)
3 (1,4)(2,8)(3,5)(7,10)(9,11)
asdf (2,5)(3,9)(4,6)(8,11)(10,12)
```

```
sage: from sage.all import *
>>> W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
>>> W.simple_reflection(Integer(1))
(0, 2, 1, 3)
>>> s = W.simple_reflections()
>>> s[Integer(1)]
(0, 2, 1, 3)
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)), index_set=[Integer(1),Integer(3),asdf])  # optional - gap3
>>> for i in W.index_set():
    # optional - gap3
    ....: print('%s %s'%(i, W.simple_reflection(i)))
1 (1,7)(2,4)(5,6)(8,10)(11,12)
3 (1,4)(2,8)(3,5)(7,10)(9,11)
asdf (2,5)(3,9)(4,6)(8,11)(10,12)
```

simple_reflection_orders()

Return the orders of the simple reflections.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: W = WeylGroup(['B',3])
sage: W.simple_reflection_orders()
[2, 2, 2]
sage: W = CoxeterGroup(['C',4])
sage: W.simple_reflection_orders()
[2, 2, 2]
sage: SymmetricGroup(5).simple_reflection_orders()
(continues on next page)```
```
[2, 2, 2, 2]  
sage: C = ColoredPermutations(4, 3)  # needs sage.combinat
sage: C.simple_reflection_orders()  # needs sage.combinat
[2, 2, 4]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> W = WeylGroup(['B', Integer(3)])
>>> W.simple_reflection_orders()
[2, 2, 2]
>>> W = CoxeterGroup(['C', Integer(4)])
>>> W.simple_reflection_orders()
[2, 2, 2, 2]
>>> SymmetricGroup(Integer(5)).simple_reflection_orders()
[2, 2, 2, 2]
>>> C = ColoredPermutations(Integer(4), Integer(3))  # needs sage.combinat
>>> C.simple_reflection_orders()  # needs sage.combinat
[2, 2, 4]
```

```
simple_reflections()
Return the simple reflections \((s_i)_{i \in I}\) of self as a family indexed by index_set().

See also:
- simple_reflection()
- index_set()

EXAMPLES:
For the symmetric group, we recognize the simple transpositions:
```
sage: W = SymmetricGroup(4); W
Symmetric group of order 4! as a permutation group
sage: s = W.simple_reflections(); s
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}
sage: s[1]
(1,2)
sage: s[2]
(2,3)
sage: s[3]
(3,4)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> W = SymmetricGroup(Integer(4)); W
Symmetric group of order 4! as a permutation group
>>> s = W.simple_reflections(); s
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}
>>> s[Integer(1)]
(1,2)
>>> s[Integer(2)]
```
Here are the simple reflections for a colored symmetric group and a reflection group:

```python
sage: W = ColoredPermutations(1,3)  # needs sage.combinat
sage: W.simple_reflections()  # needs sage.combinat
Finite family {1: [[0, 0, 0], [2, 1, 3]], 2: [[0, 0, 0], [1, 3, 2]]}

sage: W = ReflectionGroup((1,1,3), index_set=['a','b'])  # optional - gap3
sage: W.simple_reflections()  # optional - gap3
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6)}
```

This default implementation uses `index_set()` and `simple_reflection()`.

Implement `Sets.ParentMethods.some_elements()` by returning some typical elements of `self`.

The result is currently composed of the simple reflections together with the unit and the result of `an_element()`.

**EXAMPLES:**

```python
>>> from sage.all import *
>>> W = ColoredPermutations(Integer(1),Integer(3))  # needs sage.combinat
>>> W.simple_reflections()  # needs sage.combinat
Finite family {1: [[0, 0, 0], [2, 1, 3]], 2: [[0, 0, 0], [1, 3, 2]]}

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)), index_set=['a','b'])  # optional - gap3
>>> W.simple_reflections()  # optional - gap3
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6)}
```

(continues on next page)
\[
\begin{bmatrix}
[0, 0, 0, 0], [2, 1, 3, 4]),
[0, 0, 0, 0], [1, 3, 2, 4]),
[0, 0, 0, 0], [1, 2, 4, 3]),
[0, 0, 0, 0], [1, 2, 3, 4]),
[0, 0, 0, 0], [4, 1, 2, 3])
\end{bmatrix}
\]

```python
>>> from sage.all import *

>>> W = WeylGroup(['A',Integer(3)])

# needs sage.rings.number_field

>>> W.some_elements()

# needs sage.rings.number_field

[[[0, 0, 0, 0], [2, 1, 3, 4]),
[0, 0, 0, 0], [1, 3, 2, 4]),
[0, 0, 0, 0], [1, 2, 4, 3]),
[0, 0, 0, 0], [1, 2, 3, 4]),
[0, 0, 0, 0], [4, 1, 2, 3])
```

```python
class SubcategoryMethods
Bases: object

Irreducible()

Return the full subcategory of irreducible objects of self.

A complex reflection group, or generalized Coxeter group is reducible if its simple reflections can be split in two sets \(X\) and \(Y\) such that the elements of \(X\) commute with that of \(Y\). In particular, the group is then direct product of \(\langle X\rangle\) and \(\langle Y\rangle\). It's irreducible otherwise.

EXAMPLES:

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups

sage: ComplexReflectionGroups().Irreducible()
Category of irreducible complex reflection groups

sage: CoxeterGroups().Irreducible()
Category of irreducible Coxeter groups
```
```
>>> from sage.all import *

>>> from sage.categories.complex_reflection_groups import ComplexReflectionGroups

>>> ComplexReflectionGroups().Irreducible()
Category of irreducible complex reflection groups

>>> CoxeterGroups().Irreducible()
Category of irreducible Coxeter groups
```

super_categories()

Return the super categories of self.

4.28. Common category for Generalized Coxeter Groups or Complex Reflection Groups
EXAMPLES:

```python
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized Coxeter groups]
```

```python
CoxeterGroupAlgebras
```

4.29 Coxeter Group Algebras

```python
class sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras(category, *args)
    Bases: AlgebrasCategory
class ParentMethods
    Bases: object
demazure_lusztig_eigenvectors(q1, q2)
    Return the family of eigenvectors for the Cherednik operators.
    INPUT:
    • self – a finite Coxeter group \( W \)
    • \( q_1, q_2 \) – two elements of the ground ring \( K \)
    The affine Hecke algebra \( H_{q_1, q_2}(\tilde{W}) \) acts on the group algebra of \( W \) through the Demazure-Lusztig operators \( T_i \). Its Cherednik operators \( Y^\lambda \) can be simultaneously diagonalized as long as \( q_1/q_2 \) is not a small root of unity [HST2008].
    This method returns the family of joint eigenvectors, indexed by \( W \).
    See also:
    • demazure_lusztig_operators()
    • sage.combinat.root_system.hecke_algebra_representation.CherednikOperatorsEigenvectors
```

EXAMPLES:

```python
W = WeylGroup(['B', 2])
W.element_class._repr_ = lambda x: ''.join(str(i) for i in x.reduced_word())
K = QQ['q1,q2'].fraction_field()
q1, q2 = K.gens()
KW = W.algebra(K)
E = KW.demazure_lusztig_eigenvectors(q1,q2)
E.keys() # needs sage.rings.number_field
E[w] # needs sage.rings.number_field
```
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(2)])
>>> W.element_class._repr_ = lambda x: ":join(str(i) for i in x.reduced_˓→\rightarrow\text{word}())
>>> K = QQ['q1,q2'].fraction_field()
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> E = KW.demazure_lusztig_eigenvectors(q1,q2)
>>> E.keys()  # needs sage.rings.number_field
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient˓→\rightarrow\text{space})
>>> w = W.an_element()
>>> E[w]  # needs sage.rings.number_field
\begin{pmatrix} 1231 & 123 & 212 & 12 \end{pmatrix}
\begin{pmatrix} 2121 & (\frac{-q2}{-q1+q2}) & 121 & -212 \end{pmatrix}
demazure_lusztig_operator_on_basis(w, i, q1, q2, side='right')
Return the result of applying the $i$-th Demazure Lusztig operator on $w$.

INPUT:
• $w$ – an element of the Coxeter group
• $i$ – an element of the index set
• $q_1, q_2$ – two elements of the ground ring
• $\text{bar}$ – a boolean (default False)

See \texttt{demazure_lusztig_operators()} for details.

EXAMPLES:

\begin{verbatim}
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_ = lambda x: ":join(str(i) for i in x.reduced_˓→\rightarrow\text{word}())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: w = W.an_element()
sage: KW.demazure_lusztig_operator_on_basis(w, 0, q1, q2)
\begin{pmatrix} 323123 & 123 & 212 & 12 \end{pmatrix}
\begin{pmatrix} 1231 & \frac{-q2}{-q1+q2} \cdot 2121 & (\frac{-q2}{-q1+q2}) \cdot 121 & -212 \end{pmatrix}
\begin{pmatrix} 2121 & (\frac{-q2}{-q1+q2}) & 121 & -212 \end{pmatrix}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> W.element_class._repr_ = lambda x: ":join(str(i) for i in x.reduced_˓→\rightarrow\text{word}())
>>> K = QQ['q1,q2']
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> w = W.an_element()
>>> KW.demazure_lusztig_operator_on_basis(w, Integer(0), q1, q2)
\begin{pmatrix} 323123 & 123 & 212 & 12 \end{pmatrix}
\begin{pmatrix} 1231 & \frac{-q2}{-q1+q2} \cdot 2121 & (\frac{-q2}{-q1+q2}) \cdot 121 & -212 \end{pmatrix}
\begin{pmatrix} 2121 & (\frac{-q2}{-q1+q2}) & 121 & -212 \end{pmatrix}
\end{verbatim}
At $q_1 = 1$ and $q_2 = 0$ we recover the action of the isobaric divided differences $\pi_i$:

```python
sage: KW.demazure_lusztig_operator_on_basis(w, 0, 1, 0)
123
sage: KW.demazure_lusztig_operator_on_basis(w, 1, 1, 0)
1231
sage: KW.demazure_lusztig_operator_on_basis(w, 2, 1, 0)
1232
sage: KW.demazure_lusztig_operator_on_basis(w, 3, 1, 0)
123
```

At $q_1 = 1$ and $q_2 = -1$ we recover the action of the simple reflection $s_i$:

```python
sage: KW.demazure_lusztig_operator_on_basis(w, 0, 1, -1)
323123
sage: KW.demazure_lusztig_operator_on_basis(w, 1, 1, -1)
1231
sage: KW.demazure_lusztig_operator_on_basis(w, 2, 1, -1)
1232
sage: KW.demazure_lusztig_operator_on_basis(w, 3, 1, -1)
12
```

demazure_lusztig_operators ($q_1, q_2$, side='right', affine=True)
Return the Demazure Lusztig operators acting on \texttt{self}.

\textbf{INPUT:}

- \(q_1, q_2\) – two elements of the ground ring \(K\)
- \texttt{side} = "left" or "right" (default: "right"); which side to act upon
- \texttt{affine} – a boolean (default: True)

The Demazure-Lusztig operator \(T_i\) is the linear map \(R \rightarrow R\) obtained by interpolating between the simple projection \(\pi_i\) (see \texttt{CoxeterGroups.ElementMethods.simple_projection()}) and the simple reflection \(s_i\) so that \(T_i\) has eigenvalues \(q_1\) and \(q_2\):

\[(q_1 + q_2)\pi_i - q_2 s_i.

The Demazure-Lusztig operators give the usual representation of the operators \(T_i\) of the \(q_1, q_2\) Hecke algebra associated to the Coxeter group.

For a finite Coxeter group, and if \texttt{affine=True}, the Demazure-Lusztig operators \(T_1, \ldots, T_n\) are completed by \(T_0\) to implement the level 0 action of the affine Hecke algebra.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_=\lambda x: ".join(str(i) for i in x.reduced_˓→word())
sage: K = QQ[\"q1,q2\"]
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: x = KW.monomial(W.an_element()); x
123
sage: T[0](x)
(-q2)*323123 + (q1+q2)*123
sage: T[1](x)
q1*1231
sage: T[2](x)
q1*1232
sage: T[3](x)
(q1+q2)*123 + (-q2)*12

sage: T._test_relations()
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> W = WeylGroup(["B",Integer(3)])
>>> W.element_class._repr_=\lambda x: ".join(str(i) for i in x.reduced_˓→word())
>>> K = QQ[\"q1,q2\"]
>>> q1, q2 = K.gens()
>>> KW = W.algebra(K)
>>> T = KW.demazure_lusztig_operators(q1, q2, affine=True)
>>> x = KW.monomial(W.an_element()); x
123
>>> T[Integer(0)](x)
(-q2)*323123 + (q1+q2)*123
>>> T[Integer(1)](x)
q1*1231
>>> T[Integer(2)](x)
q1*1232
>>> T[Integer(3)](x)
(q1+q2)*123 + (-q2)*12
\end{verbatim}

(continues on next page)
\[ \\text{Note: For a finite Weyl group } W, \text{ the level 0 action of the affine Weyl group } \tilde{\mathcal{W}} \text{ only depends on the Coxeter diagram of the affinization, not its Dynkin diagram. Hence it is possible to explore all cases using only untwisted affinizations.} \]

### 4.30 Coxeter Groups

**class** `sage.categories.coxeter_groups.CoxeterGroups`

Bases: `Category_singleton`

The category of Coxeter groups.

A **Coxeter group** is a group \( W \) with a distinguished (finite) family of involutions \( (s_i)_{i \in I} \), called the *simple reflections*, subject to relations of the form \( (s_is_j)^{m_{i,j}} = 1 \).

\( I \) is the *index set* of \( W \) and \( |I| \) is the *rank* of \( W \).

See Wikipedia article Coxeter group for details.

**EXAMPLES:**

```python
sage: C = CoxeterGroups(); C
Category of Coxeter groups
sage: C.super_categories()
[Category of generalized Coxeter groups]

sage: W = C.example(); W
The symmetric group on {0, ..., 3}
sage: W.simple_reflections()
Finite family {0: (1, 0, 2, 3), 1: (0, 2, 1, 3), 2: (0, 1, 3, 2)}
```
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)

```python
sage: S4 = SymmetricGroup(4); S4
# needs sage.groups
Symmetric group of order 4! as a permutation group
sage: S4 in CoxeterGroups().Finite()
# needs sage.groups
True
```

Those will eventually be also in this category:

```python
>>> from sage.all import *
>>> FiniteCoxeterGroups().example()
The 5-th dihedral group of order 10
>>> FiniteWeylGroups().example()
The symmetric group on {0, ..., 3}
>>> WeylGroup(["B", Integer(3)])
   # needs sage.combinat sage.groups
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)

>>> S4 = SymmetricGroup(Integer(4)); S4
   # needs sage.groups
Symmetric group of order 4! as a permutation group
>>> S4 in CoxeterGroups().Finite()
   # needs sage.groups
True
```

Todo: add a demo of usual computations on Coxeter groups.

See also:

- `sage.combinat.root_system`
- `WeylGroups`
- `GeneralizedCoxeterGroups`

**Warning:** It is assumed that morphisms in this category preserve the distinguished choice of simple reflections. In particular, subobjects in this category are parabolic subgroups. In this sense, this category might be better named **Coxeter Systems**. In the long run we might want to have two distinct categories, one for Coxeter groups (with morphisms being just group morphisms) and one for Coxeter systems:

```python
sage: CoxeterGroups().is_full_subcategory(Groups())
False
```
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: CoxeterGroups().is_full_subcategory(GeneralizedCoxeterGroups())
True

>>> from sage.all import *
>>> CoxeterGroups().is_full_subcategory(Groups())
False

>>> from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
>>> CoxeterGroups().is_full_subcategory(GeneralizedCoxeterGroups())
True

Algebras

alias of CoxeterGroupAlgebras

class ElementMethods

Bases: object

absolute_chain()

Return a (saturated) chain in absolute order from 1 to self.

See also:

absolute_chain_reflections()

EXAMPLES:

sage: W = CoxeterGroup(['A', 2, 1])
sage: (r, s, t) = W.simple_reflections()
sage: (r * s * r * t).absolute_chain()
[[1 0 0] [ 0 -1 2] [ 2 1 -2]
 [0 1 0] [-1 0 2] [ 1 2 -2]
 [0 0 1], [ 0 0 1], [ 1 1 -1]]

>>> from sage.all import *
>>> W = CoxeterGroup(['A', Integer(2), Integer(1)])
>>> (r, s, t) = W.simple_reflections()
>>> (r * s * r * t).absolute_chain()
[[1 0 0] [ 0 -1 2] [ 2 1 -2]
 [0 1 0] [-1 0 2] [ 1 2 -2]
 [0 0 1], [ 0 0 1], [ 1 1 -1]]

absolute_chain_reflections()

Return a list of reflections which, when (left) multiplied in order, give self.

This method is based on Theorem 1.1 in [Dy2001], combined with the strong exchange condition. As an example, if $W$ is a type $A_2$ Coxeter group with simple reflections $a, b$, then the absolute chain reflections for the element $w = ab$ is the list $[a, aba]$ as $w = (aba)a = ab$.

See also:

absolute_length(), absolute_chain()

EXAMPLES:
```python
sage: W = CoxeterGroup(
    \("A", 2, 1\)\)
sage: W.one().absolute_chain_reflections()
[]
sage: (r, s, t) = W.simple_reflections()
sage: r.absolute_chain_reflections()
[
    [-1 1 1]
    [ 0 1 0]
    [ 0 0 1]
]
sage: (r * s).absolute_chain_reflections()
[
    [-1 1 1]  [ 0 -1 2]
    [ 0 1 0]  [-1 0 2]
    [ 0 0 1]  [ 0 0 1]
]
sage: (r * s * r * t).absolute_chain_reflections()
[
    [ 0 -1 2]  [-1 -2 4]
    [-1 0 2]  [-2 -1 4]
    [ 0 0 1]  [-1 -1 3]
]
sage: W = CoxeterGroup([\('A', 3, 1\)\])
sage: (r, s, t, u) = W.simple_reflections()
sage: (r * s * t * u).absolute_chain_reflections()
[
    [-1 1 0 1]  [ 0 -1 1 1]  [ 0 0 -1 2]  [-3 2 0 2]
    [ 0 1 0 0]  [-1 0 1 1]  [-1 1 -1 2]  [-2 2 0 1]
    [ 0 0 1 0]  [ 0 0 1 0]  [-1 0 0 2]  [-2 1 1 1]
    [ 0 0 0 1], [ 0 0 0 1], [-1 0 0 2]  [-2 1 0 2]
]
sage: (r * s * t * u * s).absolute_chain_reflections()
[
    [-1 1 0 1]  [ 0 0 -1 2]  [-3 2 0 2]
    [ 0 1 0 0]  [-1 1 -1 2]  [-2 2 0 1]
    [ 0 0 1 0]  [-1 0 0 2]  [-2 1 1 1]
    [ 0 0 0 1], [ 0 0 0 1], [-2 1 0 2]
]
```

```python
>>> from sage.all import *

>>> W = CoxeterGroup([\("A", 3, 1\)\])

>>> W.one().absolute_chain_reflections()
[]

>>> (r, s, t) = W.simple_reflections()

>>> r.absolute_chain_reflections()
[
    [-1 1 1]
    [ 0 1 0]
    [ 0 0 1]
]

>>> (r * s).absolute_chain_reflections()
[
    [-1 1 1]  [ 0 -1 2]
    [ 0 1 0]  [-1 0 2]
    [ 0 0 1]  [ 0 0 1]
]

>>> (r * s * r * t).absolute_chain_reflections()

(continues on next page)```
absolute_covers()  
Return the list of covers of self in absolute order.

See also:

absolute_length()

EXAMPLES:
**absolute_le**(other)

Return whether *self* is smaller than *other* in the absolute order.

A general reflection is an element of the form $w s_i w^{-1}$, where $s_i$ is a simple reflection. The absolute order is defined analogously to the weak order but using general reflections rather than just simple reflections. This partial order can be used to define noncrossing partitions associated with this Coxeter group.

See also:

**absolute_length()**

EXAMPLES:

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3])
sage: s = W.simple_reflections()
sage: w0 = s[1]
sage: w1 = s[1]*s[2]*s[3]
sage: w0.absolute_le(w1)
True
sage: w1.absolute_le(w0)
False
sage: w1.absolute_le(w1)
True
```

>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(3)])
>>> s = W.simple_reflections()
>>> w0 = s[Integer(1)]
>>> w1 = s[Integer(1)]*s[Integer(2)]*s[Integer(3)]
>>> w0.absolute_le(w1)
True
>>> w1.absolute_le(w0)
False
>>> w1.absolute_le(w1)
True

**absolute_length()**

Return the absolute length of *self*.

The absolute length is the length of the shortest expression of the element as a product of reflections. In general, we use Theorem 1.1 in [Dy2001].

See also:

**absolute_le(), absolute_chain()**

EXAMPLES:

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3])
sage: (s[1]*s[2]*s[3]).absolute_length()
3
sage: W = SymmetricGroup(4)
```

(continues on next page)
\begin{verbatim}
needs sage.groups
sage: s = W.simple_reflections()  # needs sage.groups
needs sage.combinat sage.groups
sage: (s[3]*s[2]*s[1]).absolute_length()  # needs sage.combinat sage.groups
3

sage: W = CoxeterGroup(["A",2,1])
sage: (r, s, t) = W.simple_reflections()
sage: (r * s * r * t).absolute_length()
2
sage: W.one().absolute_length()
0
sage: r.absolute_length()
1
sage: (r * s).absolute_length()
2
sage: (r * s * r).absolute_length()
1

sage: W = CoxeterGroup(["A", 3, 1])
sage: (r, s, t, u) = W.simple_reflections()
sage: (r * s * t * u).absolute_length()
4
sage: (r * s * t * u * s).absolute_length()
3

>>> from sage.all import *

>>> W = WeylGroup(["A", Integer(3)])  # needs sage.combinat sage.groups

>>> s = W.simple_reflections()  # needs sage.combinat sage.groups

>>> (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).absolute_length()  # needs sage.combinat sage.groups
3

>>> W = SymmetricGroup(Integer(4))  # needs sage.groups

>>> s = W.simple_reflections()  # needs sage.groups

>>> (s[Integer(2)]*s[Integer(2)]*s[Integer(1)]).absolute_length()  # needs sage.combinat sage.groups
3

>>> W = CoxeterGroup(["A", Integer(2), Integer(1)])

>>> (r, s, t) = W.simple_reflections()

>>> (r * s * r * t).absolute_length()
2

>>> W.one().absolute_length()
0

>>> r.absolute_length()
1

>>> (r * s).absolute_length()
2

>>> (r * s * r).absolute_length()
1

>>> W = CoxeterGroup(["A", Integer(3), Integer(1)])

(continues on next page)
\end{verbatim}
apply_demazure_product \( (element, \text{side}='right', \text{length}_{\text{increasing}}=\text{True}) \)

Return the Demazure or 0-Hecke product of \( \text{self} \) with another Coxeter group element.

See \texttt{CoxeterGroups.ParenMethods.simple_projections()}. 

INPUT:

- **element** – either an element of the same Coxeter group as \( \text{self} \) or a tuple or a list (such as a reduced word) of elements from the index set of the Coxeter group.
- **side** – ‘left’ or ‘right’ (default: ‘right’); the side of \( \text{self} \) on which the element should be applied. If \( \text{side} \) is ‘left’ then the operation is applied on the left.
- **length_{increasing}** – a boolean (default: True) whether to act length increasingly or decreasingly

EXAMPLES:

```python
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['C', 4], prefix="s")
```

```python
sage: v = W.from_reduced_word([\[1,2,3,4,3,1\]])
```

```python
sage: v.apply_demazure_product([1,3,4,3,3])
```

```python
s4*s1*s2*s3*s4*s3*s1
```

```python
sage: v.apply_demazure_product([1,3,4,3,3], side='left')
```

```python
s3*s4*s1*s2*s3*s4*s2*s3*s1
```

```python
apply_simple_projection \( (i, \text{side}='right', \text{length}_{\text{increasing}}=\text{True}) \)

Return the result of the application of the simple projection \( \pi_{i} \) (resp. \( \bar{\pi}_{i} \)) on \( \text{self} \).

INPUT:

- **i** – an element of the index set of the Coxeter group
• side – ‘left’ or ‘right’ (default: ‘right’)
• length_increasing – a boolean (default: True) specifying the direction of the projection

See \texttt{CoxeterGroups.ParentMethods.simple_projections()} for the definition of the simple projections.

\textbf{EXAMPLES:}

```
sage: W = CoxeterGroups().example()
sage: w = W.an_element()
sage: w
(1, 2, 3, 0)
sage: w.apply_simple_projection(2)
(1, 2, 3, 0)
sage: w.apply_simple_projection(2, length_increasing=False)
(1, 2, 0, 3)
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['C', 4], prefix="s")
sage: v = W.from_reduced_word([1,2,3,4,3,1])
sage: v
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(2)
s1*s2*s3*s4*s3*s1*s2
sage: v.apply_simple_projection(2, side='left')
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(1, length_increasing=False)
s1*s2*s3*s4*s3
```

```
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> w = W.an_element()
>>> w
(1, 2, 3, 0)
>>> w.apply_simple_projection(Integer(2))
(1, 2, 3, 0)
>>> w.apply_simple_projection(Integer(2), length_increasing=False)
(1, 2, 0, 3)

>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['C', Integer(4)], prefix="s")
>>> v = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(4),
Integer(3),Integer(1)])
>>> v
s1*s2*s3*s4*s3*s1
>>> v.apply_simple_projection(Integer(2))
s1*s2*s3*s4*s3*s1*s2
>>> v.apply_simple_projection(Integer(2), side='left')
s1*s2*s3*s4*s3*s1
>>> v.apply_simple_projection(Integer(1), length_increasing=False)
s1*s2*s3*s4*s3
```

\textbf{binary_factorizations} \texttt{(predicate=The constant function (...) -> True)}

Return the set of all the factorizations \texttt{self = uv} such that \texttt{l(self) = l(u) + l(v)}.

Iterating through this set is Constant Amortized Time (counting arithmetic operations in the Coxeter group as constant time) complexity, and memory linear in the length of \texttt{self}.

One can pass as optional argument a predicate \texttt{p} such that \texttt{p(u)} implies \texttt{p(u')} for any \texttt{u} left factor of \texttt{self} and \texttt{u'} left factor of \texttt{u}. Then this returns only the factorizations \texttt{self = uv} such \texttt{p(u)} holds.
EXAMPLES:

We construct the set of all factorizations of the maximal element of the group:

```sage
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3])
sage: s = W.simple_reflections()
sage: w0 = W.from_reduced_word([1,2,3,1,2,1])
```

```sage
24
```

```python
>>> from sage.all import *
```

```python
>>> W = WeylGroup(['A', Integer(3)])
```

```python
>>> s = W.simple_reflections()
```

```python
>>> w0 = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(1),
Integer(2),Integer(1)])
```

```python
>>> w0.binary_factorizations().cardinality()  # needs sage.combinat sage.groups
```

```python
24
```

The same number of factorizations, by bounded length:

```python
sage: [w0.binary_factorizations(
....: lambda u: u.length() <= 1
....: ).cardinality() for l in [-1,0,1,2,3,4,5,6]]
[0, 1, 4, 9, 15, 20, 23, 24]
```

```python
>>> from sage.all import *
```

```python
>>> [w0.binary_factorizations(
... lambda u: u.length() <= 1
... ).cardinality() for l in [-Integer(1),Integer(0),Integer(1),Integer(2),Integer(3),
Integer(4),Integer(5),Integer(6)]]
[0, 1, 4, 9, 15, 20, 23, 24]
```

The number of factorizations of the elements just below the maximal element:

```python
sage: [(s[i]*w0).binary_factorizations().cardinality() for i in [1,2,3]]
[12, 12, 12]
```

```python
sage: w0.binary_factorizations(lambda u: False).cardinality()  # needs sage.combinat sage.groups
```

```python
0
```

```python
>>> from sage.all import *
```

```python
>>> [(s[i]*w0).binary_factorizations().cardinality() for i in [Integer(1),Integer(2),Integer(3)]]
[12, 12, 12]
```

```python
>>> w0.binary_factorizations(lambda u: False).cardinality()  # needs sage.combinat sage.groups
```

```python
0
```
\texttt{bruhat\_le}(\texttt{other})

Return whether \texttt{self} \(\leq\) \texttt{other} in the Bruhat order.

INPUT:
\begin{itemize}
  \item \texttt{other} – an element of the same Coxeter group
\end{itemize}

OUTPUT: a boolean

EXAMPLES:

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', 3])
>>> u = W.from_reduced_word([1,2,1])
>>> v = W.from_reduced_word([1,2,3,2,1])
>>> u.bruhat_le(u)  # True
>>> u.bruhat_le(v)  # True
>>> v.bruhat_le(u)  # False
>>> v.bruhat_le(v)  # True
>>> s = W.simple_reflections()
>>> s[1].bruhat_le(W.one())  # False
\end{verbatim}

The implementation uses the equivalent condition that any reduced word for \texttt{other} contains a reduced word for \texttt{self} as subword. See Stembridge, A short derivation of the Möbius function for the Bruhat order. J. Algebraic Combin. 25 (2007), no. 2, 141–148, Proposition 1.1.

Complexity: \(O(l \ast c)\), where \(l\) is the minimum of the lengths of \texttt{u} and \texttt{v}, and \(c\) is the cost of the low level methods \texttt{first\_descent()}, \texttt{has\_descent()}, \texttt{apply\_simple\_reflection()}, etc. Those are typically \(O(n)\), where \(n\) is the rank of the Coxeter group.

\texttt{bruhat\_lower\_covers()}  

Return all elements that \texttt{self} covers in (strong) Bruhat order.

If \(w = \texttt{self}\) has a descent at \(i\), then the elements that \(w\) covers are exactly \(\{ws_i, u_1s_i, u_2s_i, \ldots, u_js_i\}\), where the \(u_k\) are elements that \(ws_i\) covers that also do not have a descent at \(i\).

EXAMPLES:
sage: W = WeylGroup(["A", 3])  # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3,2,3])  # needs sage.combinat sage.groups
sage: print([v.reduced_word() for v in w.bruhat_lower_covers()])  # needs sage.combinat sage.groups
[[3, 2], [2, 3]]

W now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

We now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

sage: covers = tuple([u, v] for u in S for v in W.bruhat_lower_covers(u))

We now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

We now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

We now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

We now show how to construct the Bruhat poset:

sage: W = WeylGroup(["A", 3])

We now show how to construct the Bruhat poset:
for v in W for u in v.bruhat_lower_covers()}{sage: P = Poset((W, covers), cover_relations=True)}  # needs sage.graphs
{sage: P.show()}  # needs sage.graphs sage.plot

Alternatively, one can just use:

{sage: P = W.bruhat_poset()}  # needs sage.combinat sage.graphs sage.groups

The algorithm is taken from Stembridge’s ‘coxeter/weyl’ package for Maple.

**bruhat_lower_covers_reflections()**

Return all 2-tuples of lower_covers and reflections \((v, r)\) where \(v\) is covered by \(self\) and \(r\) is the reflection such that \(self = vr\).

**ALGORITHM:**

See **bruhat_lower_covers()**

**EXAMPLES:**

{sage: W = WeylGroup(['A', 3], prefix="s")}  # needs sage.combinat sage.groups
{sage: w = W.from_reduced_word([3,1,2,1])}  # needs sage.combinat sage.groups
{sage: w.bruhat_lower_covers_reflections()}  # needs sage.combinat sage.groups
[(s1*s2*s1, s1*s2*s3*s2*s1), (s3*s2*s1, s2), (s3*s1*s2, s1)]

**bruhat_upper_covers()**

Return all elements that cover \(self\) in (strong) Bruhat order.
The algorithm works recursively, using the ‘inverse’ of the method described for lower covers \texttt{bruhat_lower_covers()}. Namely, it runs through all \(i\) in the index set. Let \(w\) equal \texttt{self}. If \(w\) has no right descent \(i\), then \(ws_i\) is a cover; if \(w\) has a descent at \(i\), then \(u_js_i\) is a cover of \(w\) where \(u_j\) is a cover of \(ws_i\).

**EXAMPLES:**

```python
sage: W = WeylGroup(['A', 3, 1], prefix="s")  # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([1,2,1])  # needs sage.combinat sage.groups
sage: w.bruhat_upper_covers()  # needs sage.combinat sage.groups
[ s1*s2*s1*s0, s1*s2*s0*s1, s0*s1*s2*s1, s3*s1*s2*s1, s2*s3*s1*s2,
  s1*s2*s3*s1 ]

sage: W = WeylGroup(['A', 3])  # needs sage.combinat sage.groups
sage: w = W.long_element()  # needs sage.combinat sage.groups
sage: w.bruhat_upper_covers()  # needs sage.combinat sage.groups
[]

sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3])

sage: w = W.from_reduced_word([1,2,1])

sage: S = [v for v in W if w in v.bruhat_lower_covers()]
sage: C = w.bruhat_upper_covers()
sage: set(S) == set(C)
True
```

```python
>>> from sage.all import *

>>> W = WeylGroup(['A', Integer(3), Integer(1)], prefix="s")  # needs sage.combinat sage.groups

>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])  # needs sage.combinat sage.groups

>>> w.bruhat_upper_covers()  # needs sage.combinat sage.groups
[ s1*s2*s1*s0, s1*s2*s0*s1, s0*s1*s2*s1, s3*s1*s2*s1, s2*s3*s1*s2,
  s1*s2*s3*s1 ]

>>> W = WeylGroup(['A', Integer(3)])  # needs sage.combinat sage.groups

>>> w = W.long_element()  # needs sage.combinat sage.groups

>>> w.bruhat_upper_covers()  # needs sage.combinat sage.groups
[]

>>> # needs sage.combinat sage.groups

>>> W = WeylGroup(['A', Integer(3)])

>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])

>>> S = [v for v in W if w in v.bruhat_lower_covers()]

>>> C = w.bruhat_upper_covers()

>>> set(S) == set(C)
True
```

\texttt{bruhat_upper_covers_reflections()}

4.30. Coxeter Groups
Return all 2-tuples of covers and reflections $(v, r)$ where $v$ covers self and $r$ is the reflection such that $\text{self} = vr$.

ALGORITHM:

See `bruhat_upper_covers()`

EXAMPLES:

```python
sage: W = WeylGroup(['A', 4], prefix="s")
# needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3,1,2,1])
# needs sage.combinat sage.groups
sage: w.bruhat_upper_covers_reflections()
# needs sage.combinat sage.groups
[(s1*s2*s3*s2*s1, s3), (s2*s3*s1*s2*s1, s2*s3*s2),
 (s3*s4*s1*s2*s1, s4), (s4*s3*s1*s2*s1, s1*s2*s3*s4*s3*s2*s1)]
```

```python
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(4)], prefix="s")
# needs sage.combinat sage.groups
>>> w = W.from_reduced_word([Integer(3),Integer(1),Integer(2),
Integer(1)])
# needs sage.combinat sage.groups
>>> w.bruhat_upper_covers_reflections()
# needs sage.combinat sage.groups
[(s1*s2*s3*s2*s1, s3), (s2*s3*s1*s2*s1, s2*s3*s2),
 (s3*s4*s1*s2*s1, s4), (s4*s3*s1*s2*s1, s1*s2*s3*s4*s3*s2*s1)]
```

canonical_matrix()

Return the matrix of self in the canonical faithful representation.

This is an $n$-dimension real faithful essential representation, where $n$ is the number of generators of the Coxeter group. Note that this is not always the most natural matrix representation, for instance in type $A_n$.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3])
# needs sage.combinat sage.groups
sage: s = W.simple_reflections()
# needs sage.combinat sage.groups
sage: (s[1]*s[2]*s[3]).canonical_matrix()
# needs sage.combinat sage.groups
[ 0 0 -1]
[ 1 0 -1]
[ 0 1 -1]
```

```python
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(3)])
# needs sage.combinat sage.groups
>>> s = W.simple_reflections()
# needs sage.combinat sage.groups
>>> (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).canonical_matrix()
# needs sage.combinat sage.groups
[ 0 0 -1]
[ 1 0 -1]
[ 0 1 -1]
```
coset_representative (index_set, side='right')

Return the unique shortest element of the Coxeter group $W$ which is in the same left (resp. right) coset as self, with respect to the parabolic subgroup $W_I$.

INPUT:

- index_set – a subset (or iterable) of the nodes of the Dynkin diagram
- side – ‘left’ or ‘right’

EXAMPLES:

```python
sage: W = CoxeterGroups().example(5)
sage: s = W.simple_reflections()
sage: w = s[2]*s[1]*s[3]
sage: w.coset_representative([]).reduced_word()
[2, 3, 1]
sage: w.coset_representative([1]).reduced_word()
[2, 3]
sage: w.coset_representative([1,2]).reduced_word()
[2, 3]
sage: w.coset_representative([1,3]).reduced_word()
[2]
sage: w.coset_representative([2,3]).reduced_word()
[2, 1]
sage: w.coset_representative([1,2,3]).reduced_word()
[]
sage: w.coset_representative([], side='left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([1], side='left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([1,2], side='left').reduced_word()
[3]
sage: w.coset_representative([1,3], side='left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([2,3], side='left').reduced_word()
[1]
sage: w.coset_representative([1,2,3], side='left').reduced_word()
[]
```

```python
>>> from sage.all import *
>>> W = CoxeterGroups().example(Integer(5))
>>> s = W.simple_reflections()
>>> w = s[Integer(2)]*s[Integer(1)]*s[Integer(3)]
>>> w.coset_representative([]).reduced_word()
[2, 3, 1]
>>> w.coset_representative([Integer(1)]).reduced_word()
[2, 3]
>>> w.coset_representative([Integer(1),Integer(2)]).reduced_word()
[2, 3]
>>> w.coset_representative([Integer(1),Integer(3)]).reduced_word()
[2]
>>> w.coset_representative([Integer(2),Integer(3)]).reduced_word()
[2, 1]
>>> w.coset_representative([Integer(1),Integer(2),Integer(3)]).reduced_word()
[]
>>> w.coset_representative([], side='left').reduced_word()
(continues on next page)
```
\[
\begin{align*}
&[2, 3, 1] \\
&\text{w.coset_representative([Integer(1)], side='left').reduced_word()} \\
&[2, 3, 1] \\
&\text{w.coset_representative([Integer(1),Integer(2)], side='left')} \\
&\quad \text{.reduced_word()}} \\
&[3] \\
&\text{w.coset_representative([Integer(1),Integer(3)], side='left')} \\
&\quad \text{.reduced_word()}} \\
&[2, 3, 1] \\
&\text{w.coset_representative([Integer(2),Integer(3)], side='left')} \\
&\quad \text{.reduced_word()} \\
&[1] \\
&\text{w.coset_representative([Integer(1),Integer(2),Integer(3)], side='left')} \\
&\quad \text{.reduced_word()}} \\
&[]
\end{align*}
\]

**cover_reflections** *(side='right')*

Return the set of reflections \( t \) such that \( \text{self} t \) covers \( \text{self} \).

If \( \text{side} \) is 'left', \( \text{self} \) covers \( \text{self} \).

**EXAMPLES:**

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 4], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.cover_reflections()
[s3, s2*s3*s2, s4, s1*s2*s3*s4*s3*s2*s1]
sage: w.cover_reflections(side='left')
[s4, s2, s1*s2*s1, s3*s4*s3]
```

```
from sage.all import *

>>> from sage.all import *

>>> W = WeylGroup(['A', Integer(4)], prefix="s")

>>> w = W.from_reduced_word([Integer(3),Integer(1),Integer(2),Integer(1)])

>>> w.cover_reflections()
[s3, s2*s3*s2, s4, s1*s2*s3*s4*s3*s2*s1]

>>> w.cover_reflections(side='left')
[s4, s2, s1*s2*s1, s3*s4*s3]
```

**coxeter_sorting_word** *(c)*

Return the \( c \)-sorting word of \( \text{self} \).

For a Coxeter element \( c \) and an element \( w \), the \( c \)-sorting word of \( w \) is the lexicographic minimal reduced expression of \( w \) in the infinite word \( c^\infty \).

**INPUT:**

- \( c \) - a Coxeter element.

**OUTPUT:**

the \( c \)-sorting word of \( \text{self} \) as a list of integers.

**EXAMPLES:**

```
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
```

(continues on next page)
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]

>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> c = W.from_reduced_word([Integer(0),Integer(2),Integer(1)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1),Integer(0),
                         Integer(1)])
>>> w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]

defodhar_factor_element(w, index_set)
Return Deodhar's Bruhat order factoring element.

INPUT:
• w is an element of the same Coxeter group W as self
• index_set is a subset of Dynkin nodes defining a parabolic subgroup W' of W
It is assumed that v = self and w are minimum length coset representatives for W/W' such that v ≤ w in Bruhat order.

OUTPUT:
Deodhar's element f(v, w) is the unique element of W such that, for all v' and w' in W', vv' ≤ ww' in W if and only if v' ≤ f(v, w) * w' in W' where * is the Demazure product.

EXAMPLES:

sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 5], prefix="s")
sage: v = W.from_reduced_word([5])
sage: w = W.from_reduced_word([4,5,2,3,1,2])
sage: v.deodhar_factor_element(w, [1,3,4])
s3*s1
sage: W = WeylGroup(['C', 2])
sage: w = W.from_reduced_word([2,1])
sage: w.deodhar_factor_element(W.from_reduced_word([2]),[1])
Traceback (most recent call last):
... ValueError: [2, 1] is not of minimum length in its coset for the parabolic subgroup with index set [1]
REFERENCES:
• [Deo1987a]

deodhar_lift_down \( (w, \text{index}_\text{set}) \)
Letting \( v = \text{self} \), given a Bruhat relation \( v W' \geq w W' \) among cosets with respect to the subgroup \( W' \) given by the Dynkin node subset \( \text{index}_\text{set} \), returns the Bruhat-maximum lift \( x \) of \( wW' \) such that \( v \geq x \).

INPUT:
• \( w \) is an element of the same Coxeter group \( W \) as \( \text{self} \).
• \( \text{index}_\text{set} \) is a subset of Dynkin nodes defining a parabolic subgroup \( W' \).

OUTPUT:
The unique Bruhat-maximum element \( x \) in \( W \) such that \( x W' = w W' \) and \( v \geq x \).

See also:
sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_down()

EXAMPLES:

```python
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3], prefix="s")
sage: v = W.from_reduced_word([1,2,3,2])
sage: w = W.from_reduced_word([3,2])
sage: v.deodhar_lift_down(w, [3])
s2*s3*s2

>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(3)], prefix="s")
>>> v = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(2)])
>>> w = W.from_reduced_word([Integer(3),Integer(2)])
>>> v.deodhar_lift_down(w, [Integer(3)])
s2*s3*s2
```

deodhar_lift_up \( (w, \text{index}_\text{set}) \)
Letting \( v = \text{self} \), given a Bruhat relation \( v W' \leq w W' \) among cosets with respect to the subgroup \( W' \) given by the Dynkin node subset \( \text{index}_\text{set} \), returns the Bruhat-minimum lift \( x \) of \( wW' \) such that \( v \leq x \).

INPUT:
• \( w \) is an element of the same Coxeter group \( W \) as \( \text{self} \).
• \( \text{index}_\text{set} \) is a subset of Dynkin nodes defining a parabolic subgroup \( W' \).

OUTPUT:
The unique Bruhat-minimum element \( x \) in \( W \) such that \( x W' = w W' \) and \( v \leq x \).

See also:
sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_down()

EXAMPLES:

```python
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3], prefix="s")
sage: v = W.from_reduced_word([1,2,3])
sage: w = W.from_reduced_word([1,3,2])
```

(continues on next page)
sage: v.deodhar_lift_up(w, [3])
s1*s2*s3*s2

>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(3)], prefix="s")
>>> v = W.from_reduced_word([Integer(1),Integer(2),Integer(3)])
>>> w = W.from_reduced_word([Integer(1),Integer(3),Integer(2)])
>>> v.deodhar_lift_up(w, [Integer(3)])
s1*s2*s3*s2

descents (side='right', index_set=None, positive=False)
Return the descents of self, as a list of elements of the index_set.

INPUT:
• index_set – a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
• side – ‘left’ or ‘right’ (default: ‘right’)
• positive – a boolean (default: False)
The index_set option can be used to restrict to the parabolic subgroup indexed by index_set.

If positive is True, then returns the non-descents instead

Todo: find a better name for positive: complement? non_descent?

Caveat: the return type may change to some other iterable (tuple, …) in the future. Please use keyword arguments also, as the order of the arguments may change as well.

EXAMPLES:

sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]
sage: w.descents()
[1]
sage: w = s[0]*s[2]
sage: w.descents()
[0, 2]

>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)]*s[Integer(1)]
>>> w.descents()
[1]
>>> w = s[Integer(0)]*s[Integer(2)]
>>> w.descents()
[0, 2]

Todo: side, index_set, positive

first_descent (side='right', index_set=None, positive=False)
Return the first left (resp. right) descent of self, as an element of index_set, or None if there is none.
See \texttt{descents()} for a description of the options.

\textbf{EXAMPLES:}

```
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[2]*s[0]
sage: w.first_descent()
0
sage: w = s[0]*s[2]
sage: w.first_descent()
0
sage: w = s[0]*s[1]
sage: w.first_descent()
1
```

```
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(2)]*s[Integer(0)]
>>> w.first_descent()
0
>>> w = s[Integer(0)]*s[Integer(2)]
>>> w.first_descent()
0
>>> w = s[Integer(0)]*s[Integer(1)]
>>> w.first_descent()
1
```

\texttt{has\_descent}(i, side='right', positive=False)

Return whether \(i\) is a (left/right) descent of \(self\).

See \texttt{descents()} for a description of the options.

\textbf{EXAMPLES:}

```
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0] * s[1] * s[2]
sage: w.has_descent(2)
True
```

```
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)] * s[Integer(1)] * s[Integer(2)]
>>> w.has_descent(Integer(2))
True
>>> [ w.has_descent(i) for i in [0,1,2] ]
[False, False, True]
```

```
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)] * s[Integer(1)] * s[Integer(2)]
>>> w.has_descent(Integer(2))
True
>>> [ w.has_descent(i, side='left') for i in [0,1,2] ]
[True, False, False]
```

(continues on next page)
This default implementation delegates the work to \texttt{has_left_descent()} and \texttt{has_right_descent()}.

\subsection*{has\textunderscore full\_support()}

Return whether \texttt{self} has full support.

An element is said to have full support if its support contains all simple reflections.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = CoxeterGroups().example()
sage: w = W.from_reduced_word([1,2,1])
sage: w.has_full_support() False
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.has_full_support() True
\end{verbatim}

\subsection*{has\textunderscore left\_descent\__(i)\n
Return whether \(i\) is a left descent of \texttt{self}.

This default implementation uses that a left descent of \(w\) is a right descent of \(w^{-1}\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = CoxeterGroups().example(); W
The symmetric group on \{0, ..., 3\}
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.has_left_descent(0) True
sage: w.has_left_descent(1) False
sage: w.has_left_descent(2) False
\end{verbatim}

(continues on next page)
has_right_descent\( (i) \)

Return whether \( i \) is a right descent of self.

EXAMPLES:

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on \{0, \ldots, 3\}
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.has_right_descent(0)
False
sage: w.has_right_descent(1)
False
sage: w.has_right_descent(2)
True
```

inversions_as_reflections()

Return the set of reflections \( r \) such that self \( r \) < self.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3], prefix="s") # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3,1,2,1]) # needs sage.combinat sage.groups
sage: w.inversions_as_reflections() # needs sage.combinat sage.groups
[s1, s1*s2*s1, s2, s1*s2*s3*s2*s1]
```

```python
>>> from sage.all import *  
>>> W = CoxeterGroups().example(); W
The symmetric group on \{0, \ldots, 3\}
>>> w = W.an_element(); w
(1, 2, 3, 0)
>>> w.has_right_descent(Integer(0))
False
>>> w.has_right_descent(Integer(1))
False
>>> w.has_right_descent(Integer(2))
True
```

```python
>>> from sage.all import *  
>>> W = WeylGroup(['A', Integer(3)], prefix="s") # needs sage.combinat sage.groups
>>> w = W.from_reduced_word([Integer(3),Integer(1),Integer(2),Integer(1)]) # needs sage.combinat sage.groups
>>> w.inversions_as_reflections() # needs sage.combinat sage.groups
```
\textbf{is\_coxeter\_sortable}(c, \text{sorting\_word}=\text{None})

Return whether self is c-sortable.

Given a Coxeter element \(c\), an element \(w\) is \(c\)-sortable if its \(c\)-sorting word decomposes into a sequence of weakly decreasing subwords of \(c\).

\textbf{INPUT}:

\begin{itemize}
  \item \(c\) – a Coxeter element.
  \item \text{sorting\_word} – sorting word (default: None) used to not recompute the \(c\)-sorting word if already computed.
\end{itemize}

\textbf{OUTPUT}:

\text{is self \(c\)-sortable}

\textbf{EXAMPLES}:

\begin{verbatim}
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
sage: w.is_coxeter_sortable(c)
False
sage: w = W.from_reduced_word([0,2,1,0,2])
sage: w.coxeter_sorting_word(c)
[2, 0, 1, 2, 0]
sage: w.is_coxeter_sortable(c)
True
sage: W = CoxeterGroup(['A', 3])
sage: c = W.from_reduced_word([1,2,3])
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> c = W.from_reduced_word([Integer(0),Integer(2),Integer(1)])
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1),Integer(0), Integer(1)])
>>> w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
>>> w.is_coxeter_sortable(c)
False
>>> w = W.from_reduced_word([Integer(0),Integer(2),Integer(1),Integer(0), Integer(2)])
>>> w.coxeter_sorting_word(c)
[2, 0, 1, 2, 0]
>>> w.is_coxeter_sortable(c)
True
>>> W = CoxeterGroup(['A', Integer(3)])
    \needs sage.combinat sage.groups
    >>> c = W.from_reduced_word([Integer(1),Integer(2),Integer(3)])
    \needs sage.combinat sage.groups
\end{verbatim}

4.30. Coxeter Groups
Number of $c$-sortable elements in $A_3$ (Catalan number):

```python
sage: len([w for w in W if w.is_coxeter_sortable(c)])  # needs sage.rings.number_field
14
```

```python
>>> from sage.all import *

>>> len([w for w in W if w.is_coxeter_sortable(c)])  # needs sage.rings.number_field
14
```

**is_fully_commutative()**

Check if `self` is a fully-commutative element.

We use the characterization that an element $w$ in a Coxeter system $(W, S)$ is fully-commutative if and only if for every pair of generators $s, t \in S$ for which $m(s, t) > 2$, no reduced word of $w$ contains the ‘braid’ word $sts...$ of length $m(s, t)$ as a contiguous subword. See [Ste1996].

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups
sage: W = CoxeterGroup(['A', 3])

sage: len([1 for w in W if w.is_fully_commutative()])
14

sage: W = CoxeterGroup(['B', 3])

sage: len([1 for w in W if w.is_fully_commutative()])
24
```

**is_grassmannian**(side='right')

Return whether `self` is Grassmannian.

**INPUT:**

- `side` – “left” or “right” (default: “right”)

An element is Grassmannian if it has at most one descent on the right (resp. on the left).

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}

sage: s = W.simple_reflections()

sage: W.one().is_grassmannian()
True

sage: (s[1]*s[2]).is_grassmannian()
True

sage: (s[0]*s[1]).is_grassmannian()
True

sage: (s[1]*s[2]*s[1]).is_grassmannian()  # continues on next page
```
False

sage: (s[0]*s[2]*s[1]).is_grassmannian(side="left")
False
sage: (s[0]*s[2]*s[1]).is_grassmannian(side="right")
True
sage: (s[0]*s[2]*s[1]).is_grassmannian()
True

>>> from sage.all import *
>>> W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
>>> s = W.simple_reflections()
>>> W.one().is_grassmannian()
True
>>> s[Integer(1)].is_grassmannian()
True
>>> (s[Integer(1)]*s[Integer(2)]).is_grassmannian()
True
>>> (s[Integer(0)]*s[Integer(1)]).is_grassmannian()
True
>>> (s[Integer(1)]*s[Integer(2)]*s[Integer(1)]).is_grassmannian()
False
>>> (s[Integer(0)]*s[Integer(2)]*s[Integer(1)]).is_grassmannian(side="left →")
False
>>> (s[Integer(0)]*s[Integer(2)]*s[Integer(1)]).is_grassmannian(side="right")
True
>>> (s[Integer(0)]*s[Integer(2)]*s[Integer(1)]).is_grassmannian()
True

kazhdan_lusztig_cell(side='left')

Compute the left, right, or two-sided Kazhdan-Lusztig cell containing the element self depending on the specified side.

Let $C'$ denote the Kazhdan-Lusztig $C'$-basis of the Iwahori-Hecke algebra $H$ of a Coxeter system $(W, S)$. Two elements $x, y$ of the Coxeter group $W$ are said to lie in the same left Kazhdan-Lusztig cell if there exist sequences $x = w_1, w_2, \ldots, w_k = y$ and $y = u_1, u_2, \ldots, u_l = x$ such that for all $1 \leq i < k$ and all $1 \leq j < l$, there exist some Coxeter generators $s, t \in S$ for which $C'_{w_{i+1}}$ appears in $C'_{w_i}$ and $C'_{u_{j+1}}$ appears in $C'_{sC'_{u_j}}$ in $H$. Right and two-sided Kazhdan-Lusztig cells of $W$ are defined similarly; see [Lus2013].

In this function, we compute products in the $C'$ basis by using IwahoriHeckeAlgebra.Cp. As mentioned in that class, installing the optional package coxeter3 is recommended (though not required) before using this function because the package speeds up product computations that are sometimes computationally infeasible without it.

INPUT:
- $w$ – an element of self
- side – (default: 'left') the kind of cell to compute; must be either 'left', 'right', or 'two-sided'

EXAMPLES:

We compute the left cell of the generator $s_1$ in type $A_3$ in three different implementations of the Coxeter group. Note that the choice of implementation affects the representation of elements in the output cell.
but not the method used for the cell computation:

```
sage: W = CoxeterGroup('A3', implementation='permutation')  # ...

sage: s1, s2, s3 = W.simple_reflections()  # ...

sage: s1.kazhdan_lusztig_cell()  # ...
{(1,2,3,12)(4,5,10,11)(6,7,8,9),
 (1,2,10)(3,6,5)(4,7,8)(9,12,11),
 (1,7)(2,4)(5,6)(8,10)(11,12)}
```

The cell computation uses the optional package `coxeter3` in the background if available to speed up the computation, even in the different implementations:

```
sage: # optional - coxeter3, needs sage.combinat sage.groups sage.modules
sage: W = WeylGroup('A3', prefix='s')

sage: s1,s2,s3 = W.simple_reflections()

sage: s1.kazhdan_lusztig_cell()
{(s3*s2*s1, s2*s1, s1)}

sage: W = CoxeterGroup('A3', implementation='coxeter3')

sage: s1,s2,s3 = W.simple_reflections()

sage: s1.kazhdan_lusztig_cell()
{[1], [2, 1], [3, 2, 1]}
```

Next, we compute a right cell and a two-sided cell in $A_3$:

```
sage: # optional - coxeter3, needs sage.combinat sage.groups sage.modules
sage: W = CoxeterGroup('A3', implementation='coxeter3')

sage: s1,s2,s3 = W.simple_reflections()

sage: w = s1 * s3

sage: w.kazhdan_lusztig_cell(side='right')
{[1, 3], [1, 1, 3, 2]}

sage: w.kazhdan_lusztig_cell(side='two-sided')
{[1, 3], [1, 3, 2], [2, 1, 3], [2, 1, 3, 2]}
```
Some slightly longer computations in $B_4$:

```python
sage: W = CoxeterGroup('B4', implementation='coxeter3')
sage: s1, s2, s3, s4 = W.simple_reflections()
sage: s1.kazhdan_lusztig_cell(side='right')  # long time (4 seconds)
{[1],
 [1, 2],
 [1, 2, 3],
 [1, 2, 3, 4],
 [1, 2, 3, 4, 3],
 [1, 2, 3, 4, 3, 2],
 [1, 2, 3, 4, 3, 2, 1],
 ...}
```

```python
sage: (s4*s2*s3*s4).kazhdan_lusztig_cell(side='two-sided')  # long time (8 seconds)
{[2, 3, 1],
 [2, 3, 1, 2],
 [2, 3, 4, 1],
 [2, 3, 4, 1, 2],
 [2, 3, 4, 1, 2, 3],
 [2, 3, 4, 3, 1],
 ...}
```

(continues on next page)
left_inversions_as_reflections()

Return the set of reflections r such that r < self.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3], prefix="s")  # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3, 1, 2])  # needs sage.combinat sage.groups
sage: w.left_inversions_as_reflections()  # needs sage.combinat sage.groups
[s1, s3, s1*s2*s3*s2*s1, s2*s3*s2]
```

length()

Return the length of self.

This is the minimal length of a product of simple reflections giving self.

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: s1 = W.simple_reflection(1)
sage: s2 = W.simple_reflection(2)
sage: s1.length() 1
sage: (s1*s2).length() 2
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[0]
sage: w.length() 3
```

```python
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s1 = W.simple_reflection(Integer(1))
```

(continues on next page)
See also:

reduced_word()

Todo:  Should use reduced_word_iterator (or reverse_iterator)

lower_cover_reflections (side='right')

Return the reflections \( t \) such that \( \text{self} \) covers \( \text{self} t \).

If \( \text{side} \) is 'left', \( \text{self} \) covers \( t \) \( \text{self} \).

EXAMPLES:

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 3],prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.lower_cover_reflections()
[\text{s1} \* \text{s2} \* \text{s3} \* \text{s2} \* \text{s1}, \text{s2}, \text{s1}]
sage: w.lower_cover_reflections(side='left')
[\text{s2} \* \text{s3} \* \text{s2}, \text{s3}, \text{s1}]
```

lower_covers (side='right', index_set=None)

Return all elements that \( \text{self} \) covers in weak order.

INPUT:

- side = 'left' or 'right' (default: 'right')
- index_set = a list of indices or None

OUTPUT: a list

EXAMPLES:


```python
sage: W = WeylGroup(['A', 3])  
     # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3,2,1])  
     # needs sage.combinat sage.groups
sage: [x.reduced_word() for x in w.lower_covers()]  
     # needs sage.combinat sage.groups
[[3, 2]]
```

To obtain covers for left weak order, set the option side to ‘left’:

```python
sage: [x.reduced_word() for x in w.lower_covers(side='left')]  
     # needs sage.combinat sage.groups
[[2, 1]]
sage: w = W.from_reduced_word([3,2,3,1])  
     # needs sage.combinat sage.groups
sage: [x.reduced_word() for x in w.lower_covers()]  
     # needs sage.combinat sage.groups
[[2, 3, 2], [3, 2, 1]]
```

Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:

```python
sage: [x.reduced_word() for x in w.lower_covers(index_set=[1,2])]  
     # needs sage.combinat sage.groups
[[2, 3, 2]]
sage: [x.reduced_word() for x in w.lower_covers(side='left')]  
     # needs sage.combinat sage.groups
[[3, 2, 1], [2, 3, 1]]
```

```python
>>> from sage.all import *  
>>> W = WeylGroup(['A', Integer(3)])  
     # needs sage.combinat sage.groups
>>> w = W.from_reduced_word([Integer(3),Integer(2),Integer(1)])  
     # needs sage.combinat sage.groups
>>> [x.reduced_word() for x in w.lower_covers()]  
     # needs sage.combinat sage.groups
[[3, 2]]
```

Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:

```python
>>> [x.reduced_word() for x in w.lower_covers(index_set=[Integer(1),  
     Integer(2)])]  
     # needs sage.combinat sage.groups
[[2, 3, 2]]
>>> [x.reduced_word() for x in w.lower_covers(side='left')]  
     # needs sage.combinat sage.groups
[[3, 2, 1], [2, 3, 1]]
```

```
min_demazure_product_greater(element)`
Find the unique Bruhat-minimum element \( u \) such that \( v \leq w^* u \) where \( v \) is self, \( w \) is element and * is the Demazure product.

INPUT:
- \( element \) is either an element of the same Coxeter group as self or a list (such as a reduced word) of elements from the index set of the Coxeter group.

EXAMPLES:

```sage
# needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 4], prefix="s")
sage: v = W.from_reduced_word([2, 3, 4, 1, 2])
sage: u = W.from_reduced_word([2, 3, 2, 1])
sage: v.min_demazure_product_greater(u)
s4*s2
sage: v.min_demazure_product_greater([2, 3, 2, 1])
s4*s2
sage: v.min_demazure_product_greater((2, 3, 2, 1))
s4*s2
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(4)], prefix="s")
>>> v = W.from_reduced_word([Integer(2), Integer(3), Integer(4), Integer(1), Integer(2)])
>>> u = W.from_reduced_word([Integer(2), Integer(3), Integer(2), Integer(1)])
>>> v.min_demazure_product_greater(u)
s4*s2
>>> v.min_demazure_product_greater([Integer(2), Integer(3), Integer(2), Integer(1)])
s4*s2
>>> v.min_demazure_product_greater((Integer(2), Integer(3), Integer(2), Integer(1)))
s4*s2
```

\texttt{reduced\_word()}

Return a reduced word for self.

This is a word \([i_1, i_2, \ldots, i_k]\) of minimal length such that \(s_{i_1}s_{i_2}\cdots s_{i_k} = \text{self}\), where the \(s_i\) are the simple reflections.

EXAMPLES:

```sage
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[2]
sage: w.reduced_word()
[0, 1, 2]
sage: w = s[0]*s[2]
sage: w.reduced_word()
[2, 0]
```

```
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)]*s[Integer(1)]*s[Integer(2)]
>>> w.reduced_word()
```

(continues on next page)
**reduced_word_graph()**

Return the reduced word graph of self.

The reduced word graph of an element \( w \) in a Coxeter group is the graph whose vertices are the reduced words for \( w \) (see reduced_word() for a definition of this term), and which has an \( m \)-colored edge between two reduced words \( x \) and \( y \) whenever \( x \) and \( y \) differ by exactly one length-\( m \) braid move (with \( m \geq 2 \)).

This graph is always connected (a theorem due to Tits) and has no multiple edges.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.graphs sage.groups
sage: W = WeylGroup(['A', 3], prefix='s')

sage: w0 = W.long_element()

sage: G = w0.reduced_word_graph()

sage: G.num_verts()
16

sage: len(w0.reduced_words())
16

sage: G.num_edges()
18

sage: len([e for e in G.edges(sort=False) if e[2] == 2])
10

sage: len([e for e in G.edges(sort=False) if e[2] == 3])
8
```

See also:

- reduced_words(), reduced_word_reverse_iterator(),
- length(), reduced_word_graph()
reduced_word_reverse_iterator()  
Return a reverse iterator on a reduced word for self.

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: sigma = s[0]*s[1]*s[2]
sage: rI=sigma.reduced_word_reverse_iterator()
sage: [i for i in rI]
[2, 1, 0]
sage: s[0]*s[1]*s[2]==sigma
True
sage: sigma.length()
3

>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> sigma = s[Integer(0)]*s[Integer(1)]*s[Integer(2)]
>>> rI=sigma.reduced_word_reverse_iterator()
>>> [i for i in rI]
[2, 1, 0]
>>> s[Integer(0)]*s[Integer(1)]*s[Integer(2)]==sigma
True
>>> sigma.length()
3
```

See also:

reduced_word()  
Default implementation: recursively remove the first right descent until the identity is reached (see first_descent() and apply_simple_reflection()).

reduced_words()  
Return all reduced words for self.

See reduced_word() for the definition of a reduced word.

The algorithm uses the Matsumoto property that any two reduced expressions are related by braid relations, see Theorem 3.3.1(ii) in [BB2005].

See also:

braid_orbit()  

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0] * s[2]
sage: sorted(w.reduced_words()) # needs sage.graphs sage.modules
[[0, 2], [2, 0]]
sage: W = WeylGroup(['E', 6]) # needs sage.combinat sage.groups
sage: w = W.from_reduced_word([2,3,4,2]) # needs sage.combinat sage.groups
```
```python
sage: sorted(w.reduced_words())  # needs sage.combinat sage.groups
[[2, 3, 4, 2], [3, 2, 4, 2], [3, 4, 2, 4]]

sage: # optional - gap3, needs sage.combinat sage.groups
sage: W = ReflectionGroup(['A', 3],
    ....:    index_set=['AA', 'BB', '5'])
sage: w = W.long_element()
sage: w.reduced_words()
[['BB', '5', 'AA', 'BB', '5', 'AA'],
 ['5', 'BB', '5', 'AA', 'BB', '5'],
 ['BB', 'AA', 'BB', '5', 'BB', 'AA'],
 ['AA', '5', 'BB', 'AA', '5', 'BB'],
 ['AA', 'BB', '5', 'AA', 'BB', 'AA'],
 ['AA', 'BB', 'AA', '5', 'BB', 'AA'],
 ['AA', 'BB', 'AA', 'BB', 'AA', '5'],
 ['AA', 'BB', 'BB', 'AA', 'BB', 'AA'],
 ['AA', 'BB', 'AA', 'BB', 'AA', 'BB']]

# optional - gap3, needs sage.combinat sage.groups
W = ReflectionGroup(['A', 3],
    ....:    index_set=['AA', 'BB', '5'])
```

```bash
>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)] * s[Integer(2)]
>>> sorted(w.reduced_words())  # needs sage.graphs sage.modules
[[0, 2], [2, 0]]

>>> W = WeylGroup(['E', Integer(6)])  # needs sage.combinat sage.groups
>>> w = W.from_reduced_word([Integer(2),Integer(3),Integer(4), Integer(2)])
# needs sage.combinat sage.groups
>>> sorted(w.reduced_words())  # needs sage.combinat sage.groups
[[2, 3, 4, 2], [3, 2, 4, 2], [3, 4, 2, 4]]
```

```python
# optional - gap3, needs sage.combinat sage.groups
W = ReflectionGroup(['A', Integer(3)],
    ....:    index_set=['AA', 'BB', '5'])
```

(continues on next page)
Todo: The result should be full featured finite enumerated set (e.g., counting can be done much faster than iterating).

See also:

reduced_word(), reduced_word_reverse_iterator(), length(), reduced_word_graph()

reduced_words_iter()
Iterate over all reduced words for self.

See reduced_word() for the definition of a reduced word.

The algorithm uses the Matsumoto property that any two reduced expressions are related by braid relations, see Theorem 3.3.1(ii) in [BB2005].

See also:

braid_orbit_iter()

EXAMPLES:

sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0] * s[2]
sage: sorted(w.reduced_words_iter())
# ...
˓→ needs sage.combinat sage.graphs
[[0, 2], [2, 0]]

>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> s = W.simple_reflections()
>>> w = s[Integer(0)] * s[Integer(2)]
>>> sorted(w.reduced_words_iter())
˓→ needs sage.combinat sage.graphs
[[0, 2], [2, 0]]

reflection_length()
Return the reflection length of self.

The reflection length is the length of the shortest expression of the element as a product of reflections.

See also:

absolute_length()

EXAMPLES:
Category Framework, Release 10.4

sage: W = WeylGroup(['A', Integer(3)])  # needs sage.combinat sage.groups
sage: s = W.simple_reflections()  # needs sage.combinat sage.groups
sage: (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).reflection_length()  # needs sage.combinat sage.groups
3

sage: W = SymmetricGroup(4)  # needs sage.groups
sage: s = W.simple_reflections()  # needs sage.groups
sage: (s[Integer(3)]*s[Integer(2)]*s[Integer(3)]).reflection_length()  # needs sage.combinat sage.groups
1

>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(3)])  # needs sage.combinat sage.groups
>>> s = W.simple_reflections()  # needs sage.combinat sage.groups
>>> (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).reflection_length()  # needs sage.combinat sage.groups
3

>>> W = SymmetricGroup(Integer(4))  # needs sage.groups
>>> s = W.simple_reflections()  # needs sage.combinat sage.groups
>>> (s[Integer(3)]*s[Integer(2)]*s[Integer(3)]).reflection_length()  # needs sage.combinat sage.groups
1

sage: W = CoxeterGroups().example()
sage: w = W.from_reduced_word([1,2,1])
sage: w.support()
{1, 2}

>>> from sage.all import *
>>> W = CoxeterGroups().example()
>>> w = W.from_reduced_word([Integer(1),Integer(2),Integer(1)])
>>> w.support()
{1, 2}

support()

Return the support of self, that is the simple reflections that appear in the reduced expressions of self.

OUTPUT:
The support of self as a set of integers

EXAMPLES:

upper_covers(side='right', index_set=None)

Return all elements that cover self in weak order.

INPUT:
weak_covers(side='right', index_set=None, positive=False)

Return all elements that self covers in weak order.

INPUT:

• side – 'left' or 'right' (default: 'right')
• positive – a boolean (default: False)
• index_set – a list of indices or None

OUTPUT: a list

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3])
# needs sage.combinat sage.groups
sage: w = W.from_reduced_word([3, 2, 1])
# needs sage.combinat sage.groups
sage: [x.reduced_word() for x in w.weak_covers()]
# needs sage.combinat sage.groups
[[3, 2]]
```

To obtain instead elements that cover self, set positive=True:

```python
sage: [x.reduced_word() for x in w.weak_covers(positive=True)]
# needs sage.combinat sage.groups
[[3, 1, 2, 1], [2, 3, 2, 1]]
```

To obtain covers for left weak order, set the option side to ‘left’:

```python
sage: [x.reduced_word() for x in w.weak_covers(side='left')]
[[2, 1]]
sage: w = W.from_reduced_word([3, 2, 3, 1])
sage: [x.reduced_word() for x in w.weak_covers()]
[[2, 3, 2], [3, 2, 1]]
sage: [x.reduced_word() for x in w.weak_covers(side='left')]
[[3, 2, 1], [2, 3, 1]]
```

Covers w.r.t. a parabolic subgroup are obtained with the option index_set:
weak_le (other, side='right')

Comparison in weak order.

INPUT:

• other – an element of the same Coxeter group
• side – ‘left’ or ‘right’ (default: ‘right’)

OUTPUT: a boolean

This returns whether self <= other in left (resp. right) weak order, that is if ‘v’ can be obtained from ‘v’ by length increasing multiplication by simple reflections on the left (resp. right).

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{ # needs sage.combinat sage.groups} \\
\text{sage: } & W = \text{WeylGroup(["A", 3])} \\
\text{sage: } & u = W.from_reduced_word([1,2]) \\
\text{sage: } & v = W.from_reduced_word([1,2,3,2]) \\
\text{sage: } & u.weak_le(u) \\
& \quad \text{True} \\
\text{sage: } & u.weak_le(v) \\
& \quad \text{True} \\
\text{sage: } & v.weak_le(u) \\
& \quad \text{False} \\
\text{sage: } & v.weak_le(v) \\
& \quad \text{True}
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} & \\
\text{>>> # needs sage.combinat sage.groups} \\
\text{>>> W = WeylGroup(["A", Integer(3)])} \\
\text{>>> u = W.from_reduced_word([Integer(1),Integer(2)])} \\
\text{>>> v = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(2)])} \\
\text{>>> u.weak_le(u)} \\
& \quad \text{True} \\
\text{>>> u.weak_le(v)} \\
& \quad \text{True} \\
\text{>>> v.weak_le(u)} \\
& \quad \text{False} \\
\text{>>> v.weak_le(v)} \\
& \quad \text{True}
\end{align*}
\]

Comparison for left weak order is achieved with the option side:

\[
\begin{align*}
\text{sage: } & \text{ u.weak_le(v, side='left') } \\
& \text{ # needs sage.combinat sage.groups} \\
& \quad \text{False}
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} & \\
\text{>>> u.weak_le(v, side='left') } \\
& \text{ # needs sage.combinat sage.groups} \\
& \text{(continues on next page)}
\end{align*}
\]
The implementation uses the equivalent condition that any reduced word for \( u \) is a right (resp. left) prefix of some reduced word for \( v \).

Complexity: \( O(l + c) \), where \( l \) is the minimum of the lengths of \( u \) and of \( v \), and \( c \) is the cost of the low level methods `first_descent()`, `has_descent()`, `apply_simple_reflection()`), etc. Those are typically \( O(n) \), where \( n \) is the rank of the Coxeter group.

We now run consistency tests with permutations:

```python
sage: W = WeylGroup(['A', 3])
# needs sage.combinat sage.groups
sage: P4 = Permutations(4)
sage: def P4toW(w):
    return W.from_reduced_word(w.reduced_word())
sage: for u in P4:
    for v in P4:
        assert u.permutohedron_lequal(v) == P4toW(u).weak_le(P4toW(v))
        assert u.permutohedron_lequal(v, side='left') == P4toW(u).weak_le(P4toW(v), side='left')
```

### Finite

alias of `FiniteCoxeterGroups`

class `ParentMethods`

Bases: `object`

`braid_group_as_finitely_presented_group()`

Return the associated braid group.

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A', 2])
# needs sage.combinat sage.groups
sage: W.braid_group_as_finitely_presented_group()
# needs sage.combinat sage.groups
Finitely presented group < S1, S2 | S1*S2*S1*S2^-1*S1^-1*S2^-1 >
```

```python
sage: W = WeylGroup(['B', 2])
# needs sage.combinat sage.groups
sage: W.braid_group_as_finitely_presented_group()
# needs sage.combinat sage.groups
```
Finitely presented group < S1, S2 | (S1*S2)^2*(S1^-1*S2^-1)^2 >

```
sage: W = ReflectionGroup(['B',3], index_set=["AA","BB","5"])
    # optional - gap3
    gap3
sage: W.braid_group_as_finitely_presented_group()
    # optional - gap3
```

Finitely presented group < SAA, SBB, S5 |
(SAA*SBB)^2*(SAA^-1*SBB^-1)^2, SAA*S5*SAA^-1*S5^-1,
SBB*S5*SBB*S5^-1*SBB^-1*S5^-1 >

```
>>> from sage.all import *
    # needs sage.combinat sage.groups
    >>> W = CoxeterGroup(['A', Integer(2)])
    # needs sage.combinat sage.groups
>>> W.braid_group_as_finitely_presented_group()
    # needs sage.combinat sage.groups
```

Finitely presented group < S1, S2 | S1*S2*S1*S2^-1*S1^-1*S2^-1 >

```
>>> W = WeylGroup(['B', Integer(2)])
    # needs sage.combinat sage.groups
>>> W.braid_group_as_finitely_presented_group()
    # needs sage.combinat sage.groups
```

Finitely presented group < S1, S2 | (S1*S2)^2*(S1^-1*S2^-1)^2 >

```
>>> W = ReflectionGroup(['B',Integer(3)], index_set=["AA","BB","5"])
    # optional - gap3
>>> W.braid_group_as_finitely_presented_group()
    # optional - gap3
```

braid_orbit(word)

Return the braid orbit of a word `word` of indices.

The input word does not need to be a reduced expression of an element.

INPUT:

- `word`: a list (or iterable) of indices in `self.index_set()`

OUTPUT:

a list of all lists that can be obtained from `word` by replacements of braid relations

See `braid_relations()` for the definition of braid relations.

EXAMPLES:

```
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: word = w.reduced_word(); word
[0, 1, 2, 1]

sage: sorted(W.braid_orbit(word))
    # needs sage.combinat sage.groups
[[0, 1, 2, 1], [0, 2, 1, 2], [2, 0, 1, 2]]

sage: sorted(W.braid_orbit([2,1,1,2,1]))
    # needs sage.combinat sage.groups
```

(continues on next page)
```python
W = ReflectionGroup(['A',3], index_set=['AA','BB','5'])
w = W.long_element()
W.braid_orbit(w.reduced_word())
```

```python
W = CoxeterGroups().example()
s = W.simple_reflections()
w = s[Integer(0)] * s[Integer(1)] * s[Integer(2)] * s[Integer(1)]
word = w.reduced_word(); word
```

```python
W = ReflectionGroup(['A',Integer(3)], index_set=['AA','BB','5'])
w = W.long_element()
W.braid_orbit(w.reduced_word())
```

```python
from sage.all import *
W = CoxeterGroups().example()
s = W.simple_reflections()
w = s[Integer(0)] * s[Integer(1)] * s[Integer(2)] * s[Integer(1)]
word = w.reduced_word(); word
```

```python
W = ReflectionGroup(['A',Integer(3)], index_set=['AA','BB','5'])
w = W.long_element()
W.braid_orbit(w.reduced_word())
```
Todo: The result should be full featured finite enumerated set (e.g., counting can be done much faster than iterating).

See also:

reduced_words()

braid_orbit_iter(word)
Iterate over the braid orbit of a word word of indices.
The input word does not need to be a reduced expression of an element.

INPUT:
• word - a list (or iterable) of indices in self.index_set()

OUTPUT:
all lists that can be obtained from word by replacements of braid relations

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: sorted(W.braid_orbit_iter([0, 1, 2, 1]))
[[0, 1, 2, 1], [0, 2, 1, 2], [2, 0, 1, 2]]
```

braid_relations()
Return the braid relations of self as a list of reduced words of the braid relations.

EXAMPLES:

```python
sage: W = WeylGroup("A", 2)
sage: W.braid_relations()
[[[1, 2, 1], [2, 1, 2]]]
```

`from sage.all import *`
>>> W.braid_relations()  # needs sage.combinat sage.groups
[[[1, 2, 1], [2, 1, 2]]]

>>> W = WeylGroup("B", Integer(3))  # needs sage.combinat sage.groups
>>> W.braid_relations()  # needs sage.combinat sage.groups
[[[1, 2, 1], [2, 1, 2]], [[1, 3], [3, 1]], [[2, 3, 2, 3], [3, 2, 3, 2]]]

**bruhat_graph** *(x=None, y=None, edge_labels=False)*

Return the Bruhat graph as a directed graph, with an edge \( u \to v \) if and only if \( u < v \) in the Bruhat order, and \( u = r \cdot v \).

The Bruhat graph \( \Gamma(x, y) \), defined if \( x \leq y \) in the Bruhat order, has as its vertices the Bruhat interval \{\( t \mid x \leq t \leq y \}\}, and as its edges are the pairs \( (u, v) \) such that \( u = r \cdot v \) where \( r \) is a reflection, that is, a conjugate of a simple reflection.

**REFERENCES:**


**EXAMPLES:**

```python
sage: W = CoxeterGroup("H", 3)  # needs sage.combinat sage.groups
sage: G = W.bruhat_graph(); G
Digraph on 120 vertices

sage: s1, s2, s3 = W.simple_reflections()
sage: W.bruhat_graph(s1, s1*s3*s2*s3)
Digraph on 6 vertices

sage: W = WeylGroup("A3", prefix="s")  # needs sage.combinat sage.graphs sage.groups
sage: s1, s2, s3 = W.simple_reflections()
sage: G = W.bruhat_graph(s1*s3, s1*s2*s3*s2*s1); G
Digraph on 10 vertices
```

```python
from sage.all import *
```

```python
>>> W = CoxeterGroup("H", Integer(3))  # needs sage.combinat sage.groups
>>> G = W.bruhat_graph(); G
Digraph on 120 vertices

>>> # needs sage.combinat sage.graphs sage.groups
>>> W = CoxeterGroup("A", Integer(2), Integer(1))
```
s1, s2, s3 = W.simple_reflections()

W.bruhat_graph(s1, s1*s3*s2*s3)
Digraph on 6 vertices

W.bruhat_graph(s1, s3*s2*s3)
Digraph on 0 vertices

W = WeylGroup("A3", prefix="s")

s1, s2, s3 = W.simple_reflections()

G = W.bruhat_graph(s1*s3, s1*s2*s3*s2*s1); G
Digraph on 10 vertices

Check that the graph has the correct number of edges (see Issue #17744):

```
sage: len(G.edges(sort=False))
16
```

```
from sage.all import *

sage: len(G.edges(sort=False))
16
```

\textbf{bruhat\_interval}(x, y)

Return the list of \( t \) such that \( x \leq t \leq y \).

\textbf{EXAMPLES:}

```
sage: W = WeylGroup("A3", prefix="s")

sage: s1, s2, s3 = W.simple_reflections()

sage: W.bruhat_interval(s2, s1*s3*s2*s1*s3)
[\text{s1*s2*s3*s2*s1, s2*s3*s2*s1, s3*s1*s2*s1, s1*s2*s3*s1,}
 \text{s1*s2*s3*s2, s3*s2*s1, s2*s3*s1, s2*s3*s2, s1*s2*s1,}
 \text{s3*s1*s2, s1*s2*s3, s2*s1, s3*s2, s2*s3, s1*s2, s2}]
```

```
sage: W = WeylGroup(['A', 2, 1], prefix="s")

sage: s0, s1, s2 = W.simple_reflections()

sage: W.bruhat_interval(1, s0*s1*s2)
[\text{s0*s1*s2, s1*s2, s0*s2, s0*s1, s2, s1, s0, 1}]
```

```
from sage.all import *

sage: W = WeylGroup("A3", prefix="s")

sage: s1, s2, s3 = W.simple_reflections()

sage: W.bruhat_interval(s2, s1*s3*s2*s1*s3)
[\text{s1*s2*s3*s2*s1, s2*s3*s2*s1, s3*s1*s2*s1, s1*s2*s3*s1,}
 \text{s1*s2*s3*s2, s3*s2*s1, s2*s3*s1, s2*s3*s2, s1*s2*s1,}
 \text{s3*s1*s2, s1*s2*s3, s2*s1, s3*s2, s2*s3, s1*s2, s2}]
```

(continues on next page)
\[s_1*s_2*s_3*s_2, s_3*s_2*s_1, s_2*s_3*s_2, s_1*s_2*s_1, s_3*s_1*s_2, s_1*s_2*s_3, s_2*s_1, s_3*s_2, s_2*s_3, s_1*s_2, s_2\]

\[\text{bruhat_interval_poset}(x, y, \text{facade=False})\]

Return the poset of the Bruhat interval between \(x\) and \(y\) in Bruhat order.

EXAMPLES:

\[
\begin{align*}
\text{sage: } & W = \text{WeylGroup}(\text{"A3"}, \text{prefix="s"}) \quad \text{# needs sage.combinat sage.groups} \\
& s_1, s_2, s_3 = W.\text{simple_reflections()} \quad \text{# needs sage.combinat sage.groups} \\
\text{sage: } & W.\text{bruhat_interval_poset}(s_2, s_1*s_3*s_2*s_1*s_3) \quad \text{# needs sage.combinat sage.groups} \\
& \text{Finite poset containing 16 elements} \\
\text{sage: } & W = \text{WeylGroup}(\text{["A", 2, 1]}, \text{prefix="s"}) \quad \text{# needs sage.combinat sage.groups} \\
\text{sage: } & s_0, s_1, s_2 = W.\text{simple_reflections()} \quad \text{# needs sage.combinat sage.groups} \\
& W.\text{bruhat_interval_poset}(1, s_0*s_1*s_2) \quad \text{# needs sage.combinat sage.groups} \\
& \text{Finite poset containing 8 elements}
\end{align*}
\]

\[\text{canonical_representation}()\]

Return the canonical faithful representation of \texttt{self}.

See also:

To obtain the underlying module with the action, use \texttt{reflection_representation()}.

EXAMPLES:
```python
sage: W = WeylGroup("A3")  # needs sage.combinat sage.groups
sage: W.canonical_representation()  # needs sage.combinat sage.groups
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

```python
>>> from sage.all import *
>>> W = WeylGroup("A3")  # needs sage.combinat sage.groups
>>> W.canonical_representation()  # needs sage.combinat sage.groups
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

**coxeter_diagram()**

Return the Coxeter diagram of self.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.graphs sage.groups
sage: W = CoxeterGroup(['H', 3], implementation="reflection")
sage: G = W.coxeter_diagram(); G
Graph on 3 vertices
[(1, 2, 3), (2, 3, 5)]
sage: CoxeterGroup(G) is W
True
sage: G = Graph([(0, 1, 3), (1, 2, oo)])
sage: W = CoxeterGroup(G)
sage: W.coxeter_diagram() == G
True
sage: CoxeterGroup(W.coxeter_diagram()) is W
True
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs sage.groups
>>> W = CoxeterGroup(['H', Integer(3)], implementation="reflection")
>>> G = W.coxeter_diagram(); G
Graph on 3 vertices
[(1, 2, 3), (2, 3, 5)]
>>> CoxeterGroup(G) is W
True
>>> G = Graph([(Integer(0), Integer(1), Integer(3)), (Integer(1),
Integer(2), oo)])
>>> W = CoxeterGroup(G)
>>> W.coxeter_diagram() == G
True
>>> CoxeterGroup(W.coxeter_diagram()) is W
True
```

**coxeter_element()**
Return a Coxeter element.

The result is the product of the simple reflections, in some order.

**Note:** This implementation is shared with well generated complex reflection groups. It would be nicer to put it in some joint super category; however, in the current state of the art, there is none where it is clear that this is the right construction for obtaining a Coxeter element.

In this context, this is an element having a regular eigenvector (a vector not contained in any reflection hyperplane of \texttt{self}).

**EXAMPLES:**

```sage
sage: # needs sage.combinat sage.groups
sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()
[1, 2, 4, 3]
sage: CoxeterGroup(['F', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['E', 8]).coxeter_element().reduced_word()
[1, 3, 2, 4, 5, 6, 7, 8]
sage: CoxeterGroup(['H', 3]).coxeter_element().reduced_word()
[1, 2, 3]
```

This method is also used for well generated finite complex reflection groups:

```sage
W = ReflectionGroup((1,1,4)) # optional - gap3
W.coxeter_element().reduced_word() # optional - gap3
W = ReflectionGroup((2,1,4)) # optional - gap3
W.coxeter_element().reduced_word() # optional - gap3
W = ReflectionGroup((4,1,4)) # optional - gap3
W.coxeter_element().reduced_word() # optional - gap3
```

(continues on next page)
sage: W = ReflectionGroup((4,4,4))
# optional - gap3
sage: W.coxeter_element().reduced_word()
# optional - gap3
[1, 2, 3, 4]

>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))
# optional - gap3
>>> W.coxeter_element().reduced_word()
# optional - gap3
[1, 2, 3]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(4)))
# optional - gap3
>>> W.coxeter_element().reduced_word()
# optional - gap3
[1, 2, 3, 4]

>>> W = ReflectionGroup((Integer(4),Integer(1),Integer(4)))
# optional - gap3
>>> W.coxeter_element().reduced_word()
# optional - gap3
[1, 2, 3, 4]

>>> W = ReflectionGroup((Integer(4),Integer(4),Integer(4)))
# optional - gap3
>>> W.coxeter_element().reduced_word()
# optional - gap3
[1, 2, 3, 4]

\textbf{coxeter\_matrix()} \hfill
equation* \textup{Return the Coxeter matrix associated to self.}

\textbf{EXAMPLES:}

sage: G = WeylGroup(['A', 3])
# needs sage.combinat sage.groups
sage: G.coxeter_matrix()
# needs sage.combinat sage.groups

\begin{verbatim}
[1 3 2]
[3 1 3]
[2 3 1]
\end{verbatim}

\begin{verbatim}
[1 3 2]
[3 1 3]
[2 3 1]
\end{verbatim}

\textbf{coxeter\_type()}\hfill 
equation* \textup{Return the Coxeter type of self.}

\textbf{EXAMPLES:}

sage: W = CoxeterGroup(['H', 3])
# needs sage.combinat sage.groups
sage: W.coxeter_type()
# needs sage.combinat sage.groups

(continues on next page)
Coxeter type of ['H', 3]

```python
grossdn:
>>> from sage.all import *
>>> W = CoxeterGroup(['H', Integer(3)])
>>> W.coxeter_type()
```

---

```python
demazure_product (Q)

Return the Demazure product of the list Q in self.

INPUT:

- Q is a list of elements from the index set of self.

This returns the Coxeter group element that represents the composition of 0-Hecke or Demazure operators.


EXAMPLES:

```python
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 2])
sage: w = W.demazure_product([2, 2, 1])
sage: w.reduced_word()
[2, 1]
sage: w = W.demazure_product([2, 1, 2, 1, 2])
sage: w.reduced_word()
[1, 2, 1]
sage: W = WeylGroup(['B', 2])
```

---

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(2)])
>>> w = W.demazure_product([Integer(2), Integer(2), Integer(1)])
>>> w.reduced_word()
[2, 1]
```

---

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['B', Integer(2)])
>>> w = W.demazure_product([Integer(2), Integer(1), Integer(2), Integer(1), Integer(2)])
>>> w.reduced_word()
[1, 2, 1]
```
elements_of_length\( (n) \)

Return all elements of length \( n \).

**EXAMPLES:**

```
sage: A = AffinePermutationGroup(['A', 2, 1])
# needs sage.combinat
sage: [len(list(A.elements_of_length(i))) for i in [0..5]]
# needs sage.combinat
[1, 3, 6, 9, 12, 15]
sage: W = CoxeterGroup(['H', 3])
# needs sage.combinat
sage: [len(list(W.elements_of_length(i))) for i in range(4)]
# needs sage.combinat
[1, 3, 5, 7]
sage: W = CoxeterGroup(['A', 2])
# needs sage.combinat
sage: [len(list(W.elements_of_length(i))) for i in range(6)]
# needs sage.combinat
[1, 2, 2, 1, 0, 0]
```

```
>>> from sage.all import *
>>> A = AffinePermutationGroup(['A', Integer(2), Integer(1)])
# needs sage.combinat
... [len(list(A.elements_of_length(i))) for i in (ellipsis_range(Integer(0), Ellipsis, Integer(5))))]
# needs sage.combinat
[1, 3, 6, 9, 12, 15]
```

```
>>> W = CoxeterGroup(['H', Integer(3)])
# needs sage.combinat
... [len(list(W.elements_of_length(i))) for i in range(Integer(4))]
# needs sage.combinat
[1, 3, 5, 7]
```

```
>>> W = CoxeterGroup(['A', Integer(2)])
# needs sage.combinat
... [len(list(W.elements_of_length(i))) for i in range(Integer(6))]
# needs sage.combinat
[1, 2, 2, 1, 0, 0]
```

fully_commutative_elements()

Return the set of fully commutative elements in this Coxeter group.

**See also:**

FullyCommutativeElements

**EXAMPLES:**

```
sage: CoxeterGroup(['A', 3]).fully_commutative_elements()
# needs sage.combinat
```

(continues on next page)
Fully commutative elements of
Finite Coxeter group over Integer Ring with Coxeter matrix:

\[
\begin{bmatrix}
1 & 3 & 2 \\
3 & 1 & 3 \\
2 & 3 & 1
\end{bmatrix}
\]

```python
>>> from sage.all import *

>>> CoxeterGroup(['A', Integer(3)]).fully_commutative_elements() # needs sage.combinat sage.groups
```

```
Fully commutative elements of
Finite Coxeter group over Integer Ring with Coxeter matrix:

\[
\begin{bmatrix}
1 & 3 & 2 \\
3 & 1 & 3 \\
2 & 3 & 1
\end{bmatrix}
\]
```

**grassmannian_elements** *(side='right')*

Return the left or right Grassmannian elements of self as an enumerated set.

**INPUT:**

- `side` – (default: "right") "left" or "right"

**EXAMPLES:**

```python
sage: S = CoxeterGroups().example()
sage: G = S.grassmannian_elements()
sage: G.cardinality()
12
sage: G.list()
[(0, 1, 2, 3), (1, 0, 2, 3), (0, 2, 1, 3), (0, 1, 3, 2),
 (2, 0, 1, 3), (1, 2, 0, 3), (0, 3, 1, 2), (0, 2, 3, 1),
 (3, 0, 1, 2), (1, 3, 2, 0), (2, 3, 0, 1)]
sage: sorted(tuple(w.descents()) for w in G)
[(0, 0), (0, 0), (0, 1), (1, 1), (1, 0), (1, 1), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]
```

```python
>> from sage.all import *

>>> S = CoxeterGroups().example()

>>> G = S.grassmannian_elements(side = "left")

>>> G.cardinality()
12
```

```python
>>> sorted(tuple(w.descents(side = "left")) for w in G)
[(0, 0), (0, 0), (0, 1), (1, 1), (1, 0), (1, 1), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]

```

```python
>> from sage.all import *

>> G = S.grassmannian_elements(side = "left")

>> G.cardinality()
12

>> sorted(tuple(w.descents(side = "left")) for w in G)
[(0, 0), (0, 0), (0, 1), (1, 1), (1, 0), (1, 1), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]
```

**index_set** ()
Return the index set of \texttt{self}.

**EXAMPLES:**

```
sage: # needs sage.combinat sage.groups
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.index_set()
(1, 2)
sage: W = CoxeterGroup([[1,3],[3,1]], index_set=['x', 'y'])
sage: W.index_set()
('x', 'y')
sage: W = CoxeterGroup(['H', 3])
sage: W.index_set()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = CoxeterGroup([[Integer(1),Integer(3)],[Integer(3),Integer(1)]]
>>> W.index_set()
(1, 2)
>>> W = CoxeterGroup([[Integer(1),Integer(3)],[Integer(3),Integer(1)]],
... index_set=['x', 'y'])
>>> W.index_set()
('x', 'y')
>>> W = CoxeterGroup(['H', Integer(3)])
>>> W.index_set()
(1, 2, 3)
```

**kazhdan_lusztig_cells** *(side='left')*

Compute the left, right, or two-sided Kazhdan-Lusztig cells of \texttt{self} if \texttt{self} is finite.

The cells are computed by using \texttt{kazhdan_lusztig_cell()}.

As detailed there, installation of the optional package \texttt{coxeter3} is recommended (though not required) before using this function as it speeds up the computation.

**INPUT:**

• \texttt{side} – (default: \texttt{'left'}) either \texttt{'left'}, \texttt{'right'}, or \texttt{'two-sided'}

**EXAMPLES:**

We compute the right cells in the Coxeter group of type \texttt{A}_2 below. Note that each Coxeter group may be created with multiple implementations, namely, 'reflection' (default), 'permutation', 'matrix', or 'coxeter3'. The choice of implementation affects the representation of elements in the output cells but not the method used for the cell computation:

```
sage: # needs sage.combinat sage.groups
sage: W = CoxeterGroup('A2')
sage: KL_cells = W.kazhdan_lusztig_cells(side='right')
sage: set((tuple(sorted(C, key=lambda w: w.reduced_word()))
...      for C in KL_cells))
{([-1 1] [ 0 -1]
 [ 0 1], [ 1 -1])
 ([-1 0],
 [ 0 -1])
 (}
```

(continues on next page)
We compute the left cells in the Coxeter group of type $A_3$ below. If the optional package `coxeter3` is installed, it runs in the background even if the group is not created with the `coxeter3` implementation:

```python
sage: # optional - coxeter3, needs sage.combinat sage.groups sage.libs.
gap sage.modules sage.rings.number_field
sage: W = CoxeterGroup('A3', implementation='coxeter3')
sage: KL_cells = W.kazhdan_lusztig_cells()
sage: set([tuple(sorted(C)) for C in KL_cells])
{(-1 1), (0 1), (1 0),}
>>> len(KL_cells)
4
```
{([]),
 ([1], [2, 1], [3, 2, 1]),
 ([1, 2], [2], [3, 2]),
 ([1, 2, 1], [1, 3, 2, 1], [2, 1, 3, 2, 1]),
 ([1, 2, 1, 3], [1, 2, 3, 2, 1], [2, 3, 2, 1]),
 ([1, 2, 1, 3, 2], [1, 2, 3, 2], [2, 3, 2]),
 ([1, 2, 1, 3, 2, 1],),
 ([1, 3], [2, 1, 3]),
 ([1, 3, 2], [2, 1, 3, 2])}

sage: len(KL_cells)
10

sage: W = CoxeterGroup('A3', implementation='permutation')

sage: len(W.kazhdan_lusztig_cells())
10

Computing the two sided cells in $B_3$:

```
sage: # optional - coxeter3, needs sage.combinat sage.groups sage.libs.
        gap sage.modules sage.rings.number_field
sage: W = CoxeterGroup('B3', implementation='coxeter3')
sage: b3_cells = W.kazhdan_lusztig_cells('two-sided')
sage: len(b3_cells)
6
sage: set([(tuple(sorted(C)) for C in W.kazhdan_lusztig_cells())])
{([]),
 ([1], [1, 2, 3, 2, 1], [2, 1, 2, 3, 2, 1, 2]),
 ([1, 2], [1, 2, 3, 2], [2], [2, 3, 2], [3, 2]),
 ([1, 2, 3], [2, 3], [3], [3, 2, 3]),
 ([2, 1, 2, 3], [2, 3, 2, 1, 2, 3], [3, 2, 1, 2, 3]),
 ([2, 1, 2, 3, 2], [2, 3, 2, 1, 2, 3, 2], [3, 2, 1, 2, 3, 2]),
 ([2, 1, 2, 3, 2, 1],),
 ([3, 2], [3, 2, 1]),
 ([3, 2, 3], [3, 2, 3, 2], [3, 2, 3, 2]),
 ([3, 2, 3, 2, 1],)}
```
>>> from sage.all import *
>>> # optional - coxeter3, needs sage.combinat sage.groups sage.libs.gap...
    sage.modules sage.rings.number_field
>>> W = CoxeterGroup('B3', implementation='coxeter3')
>>> b3_cells = W.kazhdan_lusztig_cells('two-sided')
>>> len(b3_cells)
6
>>> set([tuple(sorted(C))
...     for C in W.kazhdan_lusztig_cells()])
{(1,),
 (1, 2), (1, 2, 3), (1, 2, 3, 2, 1),
 (1, 2, 3, 2, 1, 2), (1, 2, 3, 2, 1, 2, 3),
 (1, 2, 3, 2, 1, 2, 3, 2, 1), (1, 2, 3, 2, 1, 2, 3, 2, 1, 2)}

random_element_of_length(n)

Return a random element of length n in self.

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

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups
sage: A = AffinePermutationGroup(['A', 7, 1])
sage: p = A.random_element_of_length(10)
sage: p in A
True
sage: p.length() == 10
True

sage: # needs sage.combinat sage.groups
sage: W = CoxeterGroup(['A', 4])
sage: p = W.random_element_of_length(5)
sage: p in W
True
sage: p.length() == 5
True

>>> from sage.all import *

>>> # needs sage.combinat sage.groups
>>> A = AffinePermutationGroup(['A', Integer(7), Integer(1)])
>>> p = A.random_element_of_length(Integer(10))
>>> p in A
True
>>> p.length() == Integer(10)
True

>>> # needs sage.combinat sage.groups
>>> W = CoxeterGroup(['A', Integer(4)])
>>> p = W.random_element_of_length(Integer(5))
>>> p in W
True
>>> p.length() == Integer(5)
True
```

**reflection_representation**(base_ring=None, side='left')

Return the reflection representation of self.

This is also the canonical faithful representation of a Coxeter group.

**INPUT:**

- base_ring – (optional) the base ring; the default is the base ring of `canonical_representation`
- side – ignored

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['D', 4])
sage: W.reflection_representation()
Reflection representation of Finite Coxeter group over Integer Ring with Coxeter matrix:

[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]
```

(continues on next page)
sage: W = CoxeterGroup(['I', 13])
sage: W.reflection_representation()
Reflection representation of Finite Coxeter group over
Universal Cyclotomic Field with Coxeter matrix:
[ 1 13]
[13 1]

sage: W = WeylGroup(['B', 3, 1])
sage: W.reflection_representation(QQ)
Reflection representation of Weyl Group of type ['B', 3, 1]
(as a matrix group acting on the root space)

>>> from sage.all import *

>>> W = CoxeterGroup(['D', Integer(4)])
>>> W.reflection_representation()
Reflection representation of Finite Coxeter group over
Integer Ring with Coxeter matrix:
[ 1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]

>>> W = CoxeterGroup(['I', Integer(13)])
>>> W.reflection_representation()
Reflection representation of Finite Coxeter group over
Universal Cyclotomic Field with Coxeter matrix:
[ 1 13]
[13 1]

>>> W = WeylGroup(['B', Integer(3), Integer(1)])
>>> W.reflection_representation(QQ)
Reflection representation of Weyl Group of type ['B', 3, 1]
(as a matrix group acting on the root space)

sign_representation(base_ring=None)

Return the sign representation of self over base_ring.

INPUT:
• base_ring – (optional) the base ring; the default is \mathbb{Z}

EXAMPLES:

sage: W = WeylGroup(['D', 4])

# needs sage.combinat sage.groups
sage: W.sign_representation(QQ)  # needs sage.combinat sage.groups
Sign representation of
Weyl Group of type ['D', 4] (as a matrix group acting on the ambient...
over Rational Field

sage: # optional - gap3
sage: W = CoxeterGroup(['B', 3], implementation="coxeter3")
sage: W.sign_representation()
Sign representation of Coxeter group of type ['B', 3]
implemented by Coxeter3 over Integer Ring
>>> from sage.all import *
>>> W = WeylGroup(['D', Integer(4)])
˓→ # needs sage.combinat sage.groups
>>> W.sign_representation(QQ)
˓→ needs sage.combinat sage.groups
Sign representation of 
Weyl Group of type ['D', 4] (as a matrix group acting on the ambient␣
˓→ space) over Rational Field

>>> # optional - gap3
>>> W = CoxeterGroup(['B',Integer(3)], implementation="coxeter3")
>>> W.sign_representation()
Sign representation of Coxeter group of type ['B', 3] 
implemented by Coxeter3 over Integer Ring

simple_projection(\(i, \text{side='right', length\_increasing=True}\))
Return the simple projection \(\pi_i\) (or \(\overline{\pi}_i\) if \(\text{length\_increasing}\) is False).

INPUT:
• \(i\) – an element of the index set of \texttt{self}
See \texttt{simple\_projections()} for the options and for the definition of the simple projections.

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: W
The symmetric group on \{0, \ldots, 3\}
sage: s = W.simple_reflections()
sage: sigma = W.an_element()
sage: sigma
(1, 2, 3, 0)
sage: u0 = W.simple_projection(0)
sage: d0 = W.simple_projection(0, length\_increasing=False)
sage: sigma.length()
3
sage: pi=sigma*s[0]
sage: pi.length()
4
sage: u0(sigma)
(2, 1, 3, 0)
sage: pi
(2, 1, 3, 0)
sage: u0(pi)
(2, 1, 3, 0)
sage: d0(sigma)
(1, 2, 3, 0)
sage: d0(pi)
(1, 2, 3, 0)
```
simple_projections \( \text{side='right', length_increasing=True} \)

Return the family of simple projections, also known as 0-Hecke or Demazure operators.

INPUT:
- self — a Coxeter group \( W \)
- side — ‘left’ or ‘right’ (default: ‘right’)
- length_increasing — a boolean (default: True) specifying whether the operator increases or decreases length

This returns the simple projections of \( W \), as a family.

To each simple reflection \( s_i \) of \( W \), corresponds a simple projection \( \pi_i \) from \( W \) to \( W \) defined by:
\[
\pi_i(w) = ws_i \text{ if } i \text{ is not a descent of } w \quad \pi_i(w) = w \text{ otherwise.}
\]

The simple projections \( (\pi_i)_{i \in I} \) move elements down the right permutohedron, toward the maximal element. They satisfy the same braid relations as the simple reflections, but are idempotents \( \pi_i^2 = \pi_i \) not involutions \( s_i^2 = 1 \). As such, the simple projections generate the 0-Hecke monoid.

By symmetry, one can also define the projections \( (\pi_i)_{i \in I} \) (when the option length_increasing is False):
\[
\pi_i(w) = ws_i \text{ if } i \text{ is a descent of } w \quad \pi_i(w) = w \text{ otherwise.}
\]
as well as the analogues acting on the left (when the option side is ‘left’).

EXAMPLES:

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on \{0, ..., 3\}
sage: s = W.simple_reflections()
sage: sigma = W.an_element(); sigma
(1, 2, 3, 0)
sage: pi = W.simple_projections(); pi
Finite family {0: <function ...<lambda> at ...>, 1: <function ...<lambda> at ...>, 2: <function ...<lambda> ...>}
sage: pi[1](sigma)
(1, 3, 2, 0)
sage: W.simple_projection(1)(sigma)
(1, 3, 2, 0)
```
```python
>>> from sage.all import *

>>> W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}

>>> s = W.simple_reflections()
>>> sigma = W.an_element(); sigma
(1, 2, 3, 0)

>>> pi = W.simple_projections(); pi
Finite family {0: <function ...<lambda> at ...>,
    1: <function ...<lambda> at ...>,
    2: <function ...<lambda> ...>}

>>> pi[Integer(1)](sigma)
(1, 3, 2, 0)

>>> W.simple_projection(Integer(1))(sigma)
(1, 3, 2, 0)
```

### standard_coxeter_elements()

Return all standard Coxeter elements in `self`.

This is the set of all elements in `self` obtained from any product of the simple reflections in `self`.

**Note:**
- `self` is assumed to be well-generated.
- This works even beyond real reflection groups, but the conjugacy class is not unique and we only obtain one such class.

**EXAMPLES:**

```python
sage: W = ReflectionGroup(4)  # optional - gap3
sage: sorted(W.standard_coxeter_elements())  # optional - gap3
[(1,7,6,12,23,20) (2,8,17,24,9,5) (3,16,10,19,15,21) (4,14,11,22,18,13),
 (1,10,4,12,21,22) (2,11,19,24,13,3) (5,15,7,17,16,23) (6,18,8,20,14,9)]
```

```python
>>> from sage.all import *

>>> W = ReflectionGroup(Integer(4))  # optional - gap3

>>> sorted(W.standard_coxeter_elements())  # optional - gap3
[(1,7,6,12,23,20) (2,8,17,24,9,5) (3,16,10,19,15,21) (4,14,11,22,18,13),
 (1,10,4,12,21,22) (2,11,19,24,13,3) (5,15,7,17,16,23) (6,18,8,20,14,9)]
```

### weak_order_ideal (predicate, side='right', category=None)

Return a weak order ideal defined by a predicate.

**INPUT:**
- `predicate`: a predicate on the elements of `self` defining an weak order ideal in `self`
- `side`: “left” or “right” (default: “right”)

**OUTPUT:** an enumerated set

**EXAMPLES:**

```python
sage: D6 = FiniteCoxeterGroups().example(5)
sage: I = D6.weak_order_ideal(predicate=lambda w: w.length() <= 3)
sage: I.cardinality()
7
sage: list(I)
[(], [1], [2], [1, 2], [2, 1], [1, 2, 1], [2, 1, 2])
```

4.30. Coxeter Groups 453
We now consider an infinite Coxeter group:

```python
sage: W = WeylGroup(['A',1,1])  # needs sage.groups sage.rings.number_field
sage: I = W.weak_order_ideal(predicate=lambda w: w.length() <= 2)  # needs sage.groups sage.rings.number_field
sage: list(iter(I))  # needs sage.groups sage.rings.number_field
[[1 0], [-1 2], [1 0], [3 -2], [-1 2],
[0 1], [0 1], [2 -1], [2 -1], [-2 3]]
```

Even when the result is finite, some features of `FiniteEnumeratedSets` are not available:

```python
sage: I.cardinality()  # todo: not implemented
5
sage: list(I)  # todo: not implemented
```

unless this finiteness is explicitly specified:

```python
sage: I = W.weak_order_ideal(predicate=lambda w: w.length() <= 2,  # needs sage.groups sage.rings.number_field
....: category=FiniteEnumeratedSets())
sage: I.cardinality()  # needs sage.groups sage.rings.number_field
5
sage: list(I)  # needs sage.groups sage.rings.number_field
[[1 0], [-1 2], [1 0], [3 -2], [-1 2],
[0 1], [0 1], [2 -1], [2 -1], [-2 3]]
```

(continues on next page)
Background

The weak order is returned as a RecursivelyEnumeratedSet_forest. This is achieved by assigning to each element \( u_1 \) of the ideal a single ancestor \( u = u_1 s_i \), where \( i \) is the smallest descent of \( u \).

This allows for iterating through the elements in roughly Constant Amortized Time and constant memory (taking the operations and size of the generated objects as constants).

additional_structure()

Return None.

Indeed, all the structure Coxeter groups have in addition to groups (simple reflections, ...) is already defined in the super category.

See also:

Category.additional_structure()

EXAMPLES:

```python
from sage.all import *
CoxeterGroups().additional_structure()
```

super_categories()

EXAMPLES:

```python
from sage.all import *
CoxeterGroups().super_categories()
```

4.30. Coxeter Groups
4.31 Crystals

class sage.categories.crystals.CrystalHomset(X, Y, category=None)

Bases: Homset

The set of crystal morphisms from one crystal to another.

An $U_q$ $I$-crystal morphism $\Psi : B \rightarrow C$ is a map $\Psi : B \cup \{0\} \rightarrow C \cup \{0\}$ such that:

- $\Psi(0) = 0$.
- If $b \in B$ and $\Psi(b) \in C$, then $wt(\Psi(b)) = wt(b)$, $\varepsilon_i(\Psi(b)) = \varepsilon_i(b)$, and $\varphi_i(\Psi(b)) = \varphi_i(b)$ for all $i \in I$.
- If $b, b' \in B$, $\Psi(b), \Psi(b') \in C$ and $f_i b = b'$, then $f_i \Psi(b) = \Psi(b')$ and $\Psi(b) = e_i \Psi(b')$ for all $i \in I$.

If the Cartan type is unambiguous, it is suppressed from the notation.

We can also generalize the definition of a crystal morphism by considering a map of $\sigma$ of the (now possibly different) Dynkin diagrams corresponding to $B$ and $C$ along with scaling factors $\gamma_i \in \mathbb{Z}$ for $i \in I$. Let $\sigma_i$ denote the orbit of $i$ under $\sigma$. We write objects for $B$ as $X$ with corresponding objects of $C$ as $\hat{X}$. Then a virtual crystal morphism $\Psi$ is a map such that the following holds:

- $\Psi(0) = 0$.
- If $b \in B$ and $\Psi(b) \in C$, then for all $j \in \sigma_i$:
  $$\varepsilon_i(b) = \frac{1}{\gamma_j} \varepsilon_j(\Psi(b)), \quad \varphi_i(b) = \frac{1}{\gamma_j} \varphi_j(\Psi(b)), \quad wt(\Psi(b)) = \sum_i c_i \sum_{j \in \sigma_i} \gamma_j \hat{A}_j,$$
  where $wt(b) = \sum_i c_i A_i$.
- If $b, b' \in B$, $\Psi(b), \Psi(b') \in C$ and $f_i b = b'$, then independent of the ordering of $\sigma_i$, we have:
  $$\Psi(b') = e_i \Psi(b) = \prod_{j \in \sigma_i} \varepsilon_j \Psi(b), \quad \Psi(b') = f_i \Psi(b) = \prod_{j \in \sigma_i} \varphi_j \Psi(b).$$

If $\gamma_i = 1$ for all $i \in I$ and the Dynkin diagrams are the same, then we call $\Psi$ a twisted crystal morphism.

INPUT:

- $X$ – the domain
- $Y$ – the codomain
- category – (optional) the category of the crystal morphisms

See also:

For the construction of an element of the homset, see CrystalMorphismByGenerators and crystal_morphism().

EXAMPLES:

We begin with the natural embedding of $B(2\Lambda_1)$ into $B(\Lambda_1) \otimes B(\Lambda_1)$ in type $A_1$:

```
sage: B = crystals.Tableaux(['A', 1], shape=[2])
sage: F = crystals.Tableaux(['A', 1], shape=[1])
sage: T = crystals.TensorProduct(F, F)
sage: v = T.highest_weight_vectors()[0]; v
[[[1]], [[1]]]
sage: H = Hom(B, T)
sage: psi = H([v])
sage: b = B.highest_weight_vector(); b
```
We now look at the decomposition of $B(\Lambda_1) \otimes B(\Lambda_1)$ into $B(2\Lambda_1) \oplus B(0)$:

```python
sage: B0 = crystals.Tableaux(['A', Integer(1)], shape=[])
sage: D = crystals.DirectSum([B, B0])
sage: H = Hom(T, D)
sage: psi = H(D.module_generators)
sage: psi
['A', 1] Crystal morphism:
  From: Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 1] and shape(s) [[1]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]]]
  To:  Direct sum of the crystals
    (The crystal of tableaux of type ['A', 1] and shape(s) [[2]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]])
  Defn: [[1]], [[1]] |--> [[1, 1]]
    [[2]], [[1]] |--> []
sage: psi.is_isomorphism()
True
```

```python
>>> from sage.all import *
>>> B = crystals.Tableaux(['A', Integer(1)], shape=[Integer(2)])
>>> F = crystals.Tableaux(['A', Integer(1)], shape=[Integer(1)])
>>> T = crystals.TensorProduct(F, F)
>>> v = T.highest_weight_vectors()[Integer(0)]; v
[[[1]], [[1]]]
>>> H = Hom(B, T)
>>> psi = H([v])
>>> psi
['A', 1] Crystal morphism:
  From: Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 1] and shape(s) [[1]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]]]
  To:  Direct sum of the crystals
    (The crystal of tableaux of type ['A', 1] and shape(s) [[2]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]])
  Defn: [[1]], [[1]] |--> [[1, 1]]
    [[2]], [[1]] |--> []
```

```
>>> from sage.all import *
>>> B = crystals.Tableaux(['A', Integer(1)], shape=[Integer(2)])
>>> F = crystals.Tableaux(['A', Integer(1)], shape=[Integer(1)])
>>> T = crystals.TensorProduct(F, F)
>>> v = T.highest_weight_vectors()[Integer(0)]; v
[[[1]], [[1]]]
>>> H = Hom(B, T)
>>> psi = H([v])
>>> psi
['A', 1] Crystal morphism:
  From: Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 1] and shape(s) [[1]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]]]
  To:  Direct sum of the crystals
    (The crystal of tableaux of type ['A', 1] and shape(s) [[2]],
     The crystal of tableaux of type ['A', 1] and shape(s) [[1]])
  Defn: [[1]], [[1]] |--> [[1, 1]]
    [[2]], [[1]] |--> []
sage: psi.is_isomorphism()
True
```

(continues on next page)
The crystal of tableaux of type ['A', 1] and shape(s) [[[]]])
Defn: [[[1]], [[1]]] --> [[1, 1]]
[[[2]], [[1]]] --> [[]]
>>> psi.is_isomorphism()
True

We can always construct the trivial morphism which sends everything to 0:

```python
sage: Binf = crystals.infinity.Tableaux(['B', 2])
sage: B = crystals.Tableaux(['B', 2], shape=[1])
sage: H = Hom(Binf, B)
sage: psi = H(lambda x: None)
sage: psi(Binf.highest_weight_vector())
```

For Kirillov-Reshetikhin crystals, we consider the map to the corresponding classical crystal:

```python
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: B = K.classical_decomposition()
sage: H = Hom(K, B)
sage: psi = H(lambda x: x.lift(), cartan_type=['D',4])
sage: L = [psi(mg) for mg in K.module_generators]; L
[[], [[1], [2]]]
sage: all(x.parent() == B for x in L)
True
```

Next we consider a type $D_4$ crystal morphism where we twist by $3 \leftrightarrow 4$:

```python
sage: B = crystals.Tableaux(['D',4], shape=[1])
sage: H = Hom(B, B)
sage: d = {1:1, 2:2, 3:4, 4:3}
sage: psi = H(B.module_generators, automorphism=d)
sage: b = B.highest_weight_vector()
sage: b.f_string([1,2,3])
[[4]]
sage: b.f_string([1,2,4])
[[-4]]
sage: psi(b.f_string([1,2,3]))
[[-4]]
```
We construct the natural virtual embedding of a type $B_3$ into a type $D_4$ crystal:

```python
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: H = Hom(B, C)
sage: psi = H(C.module_generators)
sage: psi
['B', 3] -> ['D', 4] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['B', 3] and shape(s) [[1]]
  To:  The crystal of tableaux of type ['D', 4] and shape(s) [[2]]
  Defn: [[1]] |--> [[1, 1]]

sage: for b in B: print("{}/|-->{}/").format(b, psi(b))
[[1]] |--> [[1, 1]]
[[2]] |--> [[2, 2]]
[[3]] |--> [[3, 3]]
[[0]] |--> [[3, -3]]
[[1]] |--> [[1, -1]]
```

```python
>>> from sage.all import *
>>> B = crystals.Tableaux(['B','Integer(3)'], shape=[Integer(1)])
>>> C = crystals.Tableaux(['D','Integer(4)'], shape=[Integer(2)])
>>> H = Hom(B, C)
>>> psi = H(C.module_generators)
>>> psi
['B', 3] -> ['D', 4] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['B', 3] and shape(s) [[1]]
  To:  The crystal of tableaux of type ['D', 4] and shape(s) [[2]]
  Defn: [[1]] |--> [[1, 1]]

>>> for b in B: print("{}/|-->{}/").format(b, psi(b))
[[1]] |--> [[1, 1]]
[[2]] |--> [[2, 2]]
[[3]] |--> [[3, 3]]
[[0]] |--> [[3, -3]]
[[1]] |--> [[1, -1]]
```

(continues on next page)
Element

alias of `CrystalMorphismByGenerators`

```python
class sage.categories.crystals.CrystalMorphism(
    parent, cartan_type=None,
    virtualization=None, scaling_factors=None)
```

**Bases**: `Morphism`

A crystal morphism.

**INPUT**:  
- `parent` – a homset  
- `cartan_type` – (optional) a Cartan type; the default is the Cartan type of the domain  
- `virtualization` – (optional) a dictionary whose keys are in the index set of the domain and whose values are lists of entries in the index set of the codomain  
- `scaling_factors` – (optional) a dictionary whose keys are in the index set of the domain and whose values are scaling factors for the weight, $\varepsilon$ and $\varphi$

**`cartan_type()`**

Return the Cartan type of `self`.

**EXAMPLES**:  
```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: psi = Hom(B, B).an_element()
sage: psi.cartan_type()
['A', 2]
```

**`is_injective()`**

Return if `self` is an injective crystal morphism.

**EXAMPLES**:  
```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: psi = Hom(B, B).an_element()
sage: psi.is_injective()
False
```

**`is_surjective()`**

Check if `self` is a surjective crystal morphism.

**EXAMPLES**:  
```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: psi = Hom(B, B).an_element()
sage: psi.is_surjective()
False
```
sage: B = crystals.Tableaux(['C',2], shape=[1,1])

sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)

sage: psi.is_surjective()
False

sage: im_gens = [None, B.module_generators[0]]

sage: psi = C.crystal_morphism(im_gens, codomain=B)

sage: psi.is_surjective()
True

sage: C = crystals.Tableaux(['A',2], shape=[2,1])

sage: W = crystals.elementary.T(['A',2], La[1]+La[2])

sage: T = W.tensor(B)

sage: mg = T(W.module_generators[0], B.module_generators[0])

sage: psi = Hom(C,T)([mg])

sage: psi.is_surjective()
False

scaling_factors()

Return the scaling factors \( \gamma_i \).

EXAMPLES:

sage: B = crystals.Tableaux(['B',3], shape=[1])

sage: C = crystals.Tableaux(['D',4], shape=[2])

sage: psi = B.crystal_morphism(C.module_generators)

sage: psi.scaling_factors()
Finite family {1: 2, 2: 2, 3: 1}

>>> from sage.all import *

>>> B = crystals.Tableaux(['B',Integer(3)], shape=[Integer(1)])

>>> C = crystals.Tableaux(['D',Integer(4)], shape=[Integer(2)])

>>> psi = B.crystal_morphism(C.module_generators)

(continues on next page)
virtualization()

Return the virtualization sets $\sigma_i$.

EXAMPLES:

```python
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: psi = B.crystal_morphism(C.module_generators)
sage: psi.virtualization()
Finite family {1: (1,), 2: (2,), 3: (3, 4)}
```

```python
>>> from sage.all import *
```

```python
>>> B = crystals.Tableaux(['B',Integer(3)], shape=[Integer(1)])
>>> C = crystals.Tableaux(['D',Integer(4)], shape=[Integer(2)])
>>> psi = B.crystal_morphism(C.module_generators)
>>> psi.virtualization()
Finite family {1: (1,), 2: (2,), 3: (3, 4)}
```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: F = crystals.Tableaux(['A',2], shape=[1])
sage: T = crystals.TensorProduct(F, F, F)
sage: H = Hom(T, B)
sage: b = B.highest_weight_vector()
sage: psi = H((None, b, b, None), generators=T.highest_weight_vectors())
sage: psi.im_gens()  
(None, [[1, 1], [2]], [[1, 1], [2]], None)

>>> from sage.all import *
>>> B = crystals.Tableaux(['A',Integer(2)], shape=Integer(2),Integer(1))
>>> F = crystals.Tableaux(['A',Integer(2)], shape=Integer(1))
>>> T = crystals.TensorProduct(F, F, F)
>>> H = Hom(T, B)
>>> b = B.highest_weight_vector()
>>> psi = H((None, b, b, None), generators=T.highest_weight_vectors())
>>> psi.im_gens()  
(None, [[1, 1], [2]], [[1, 1], [2]], None)

image()  
Return the image of self in the codomain as a Subcrystal.

Warning: This assumes that self is a strict crystal morphism.

EXAMPLES:

sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: H = Hom(B, C)
sage: psi = H(C.module_generators)
sage: psi.image()  
Virtual crystal of The crystal of tableaux of type ['D', 4] and shape(s)→[[2]] of type ['B', 3]

>>> from sage.all import *
>>> B = crystals.Tableaux(['B',Integer(3)], shape=Integer(1))
>>> C = crystals.Tableaux(['D',Integer(4)], shape=Integer(2))
>>> H = Hom(B, C)
>>> psi = H(C.module_generators)
>>> psi.image()  
Virtual crystal of The crystal of tableaux of type ['D', 4] and shape(s)→[[2]] of type ['B', 3]

to_module_generator(x)
Return a generator mg and a path of e_i and f_i operations to mg.

OUTPUT:
A tuple consisting of:

• a module generator,

• a list of 'e' and 'f' to denote which operation, and

• a list of matching indices.

EXAMPLES:
```python
sage: B = crystals.elementary.Elementary(['A',2], 2)
sage: psi = B.crystal_morphism(B.module_generators)
sage: psi.to_module_generator(B(4))
(0, ['f', 'f', 'f', 'f'], [2, 2, 2, 2])
sage: psi.to_module_generator(B(-2))
(0, ['e', 'e'], [2, 2])
```

```python
>>> from sage.all import *
>>> B = crystals.elementary.Elementary(['A',Integer(2)], Integer(2))
>>> psi = B.crystal_morphism(B.module_generators)
>>> psi.to_module_generator(B(Integer(4)))
(0, ['f', 'f', 'f', 'f'], [2, 2, 2, 2])
>>> psi.to_module_generator(B(-Integer(2)))
(0, ['e', 'e'], [2, 2])
```

```python
class sage.categories.crystals.Crystals

Bases: Category_singleton

The category of crystals.

See sage.combinat.crystals.crystals for an introduction to crystals.

EXAMPLES:

```python
sage: C = Crystals()
sage: C
Category of crystals
sage: C.super_categories()
[Category of... enumerated sets]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

```python
>>> from sage.all import *
>>> C = Crystals()
>>> C
Category of crystals
>>> C.super_categories()
[Category of... enumerated sets]
>>> C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

Parents in this category should implement the following methods:

- either an attribute _cartan_type or a method cartan_type
- module_generators: a list (or container) of distinct elements which generate the crystal using $f_i$

Furthermore, their elements $x$ should implement the following methods:

- $x.e(i)$ (returning $e_i(x)$)
- $x.f(i)$ (returning $f_i(x)$)
- $x.epsilon(i)$ (returning $\epsilon_i(x)$)
- $x.phi(i)$ (returning $\varphi_i(x)$)

EXAMPLES:
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(Crystals().element_class)
{'optional': [], 'required': ['e', 'epsilon', 'f', 'phi', 'weight']}

>>> from sage.all import *
>>> from sage.misc.abstract_method import abstract_methods_of_class
>>> abstract_methods_of_class(Crystals().element_class)
{'optional': [], 'required': ['e', 'epsilon', 'f', 'phi', 'weight']}

class ElementMethods
    Bases: object

Epsilon()

EXAMPLES:

sage: C = crystals.Letters(['A',5])
sage: C(0).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(1).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(2).Epsilon()
(1, 0, 0, 0, 0, 0)

Phi()

EXAMPLES:

sage: C = crystals.Letters(['A',5])
sage: C(0).Phi()
(0, 0, 0, 0, 0, 0)
sage: C(1).Phi()
(1, 0, 0, 0, 0, 0)
sage: C(2).Phi()
(1, 1, 0, 0, 0, 0)

all_paths_to_highest_weight (index_set=None)
    Iterate over all paths to the highest weight from self with respect to index_set.

INPUT:

4.31. Crystals
• index_set – (optional) a subset of the index set of self

EXAMPLES:

```python
crystals.infinity.Tableaux("A2")
```

```python
b0 = B.highest_weight_vector()
b = b0.f_string([1, 2, 1, 2])
L = b.all_paths_to_highest_weight()
list(L)
```

```python
[2, 1, 2, 1], [2, 2, 1, 1]
```

```python
Y = crystals.infinity.GeneralizedYoungWalls(3)
y0 = Y.highest_weight_vector()
y = y0.f_string([0, 1, 2, 3, 2, 1, 0])
list(y.all_paths_to_highest_weight())
```

```python
[0, 1, 2, 3, 2, 1, 0],
[0, 1, 3, 2, 1, 0],
[0, 3, 1, 2, 1, 0],
[0, 3, 3, 2, 1, 0, 2],
[0, 3, 2, 1, 1, 2, 0]
```

```python
B = crystals.infinity.Tableaux("A3", shape=[4,2,1])
b0 = B.highest_weight_vector()
b = b0.f_string([1, 1, 2, 3])
list(b.all_paths_to_highest_weight())
```

```python
[1, 3, 2, 1], [3, 1, 2, 1], [3, 2, 1, 1]
```

```python
from sage.all import *
```

```python
B = crystals.infinity.Tableaux("A2")
b0 = B.highest_weight_vector()
b = b0.f_string([1, 2, 1, 2])
L = b.all_paths_to_highest_weight()
list(L)
```

```python
[2, 1, 2, 1], [2, 2, 1, 1]
```

```python
Y = crystals.infinity.GeneralizedYoungWalls(3)
y0 = Y.highest_weight_vector()
y = y0.f_string([0, 1, 2, 3, 2, 1, 0])
list(y.all_paths_to_highest_weight())
```

```python
[0, 1, 2, 3, 2, 1, 0],
[0, 1, 3, 2, 1, 0],
[0, 3, 1, 2, 1, 0],
[0, 3, 2, 1, 1, 0, 2],
[0, 3, 3, 2, 1, 0]
```

```python
B = crystals.infinity.Tableaux("A3", shape=[4,2,1])
b0 = B.highest_weight_vector()
b = b0.f_string([1, 1, 2, 3])
list(b.all_paths_to_highest_weight())
```

```python
[1, 3, 2, 1], [3, 1, 2, 1], [3, 2, 1, 1]
```

cartan_type()

Returns the Cartan type associated to self

EXAMPLES:

```python
C = crystals.Letters(['A', 5])
C(1).cartan_type()
```

(continues on next page)


```python
from sage.all import *

C = crystals.Letters(["A", Integer(5)])
C(Integer(1)).cartan_type()
['A', 5]
```

\[ e(i) \]

Return \( e_i \) of \( \text{self} \) if it exists or \( \text{None} \) otherwise.

This method should be implemented by the element class of the crystal.

**EXAMPLES:**

```python
sage: C = Crystals().example(5)
sage: x = C[2]; x
3
sage: x.e(1), x.e(2), x.e(3)
(None, 2, None)

sage: C = Crystals().example(Integer(5))
sage: x = C[Integer(2)]; x
3
sage: x.e(Integer(1)), x.e(Integer(2)), x.e(Integer(3))
(None, 2, None)
```

\[ e_{\text{string}}(\text{list}) \]

Applies \( e_{i_r} \cdots e_{i_1} \) to \( \text{self} \) for \( \text{list} \) as \([i_1, \ldots, i_r]\)

**EXAMPLES:**

```python
sage: C = crystals.Letters(["A",3])
sage: b = C(3)
sage: b.e_string([2,1])
1
sage: b.e_string([1,2])
```

```python
sage: C = crystals.Letters(["A",Integer(3)])
sage: b = C(Integer(3))
```

```python
sage: b.e_string([Integer(2),Integer(1)])
1
sage: b.e_string([Integer(1),Integer(2)])
```

\[ \epsilon(i) \]

**EXAMPLES:**

```python
sage: C = crystals.Letters(["A",5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```

4.31. Crystals
\[ \textbf{f}(i) \]

Return \( f_i \) of \self if it exists or \None otherwise.

This method should be implemented by the element class of the crystal.

\textbf{EXAMPLES:}

```
sage: C = Crystals().example(5)
sage: x = C[1]; x
2
sage: x.f(1), x.f(2), x.f(3)
(\text{None}, 3, \text{None})
```
is_highest_weight (index_set=None)

Return True if self is a highest weight.

Specifying the option index_set to be a subset $I$ of the index set of the underlying crystal, finds all highest weight vectors for arrows in $I$.

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).is_highest_weight()
True
c sage: C(2).is_highest_weight()
False
c sage: C(2).is_highest_weight(index_set = [2,3,4,5])
True
```

is_lowest_weight (index_set=None)

Returns True if self is a lowest weight. Specifying the option index_set to be a subset $I$ of the index set of the underlying crystal, finds all lowest weight vectors for arrows in $I$.

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).is_lowest_weight()
False
c sage: C(6).is_lowest_weight()
True
c sage: C(4).is_lowest_weight(index_set = [1,3])
True
```

phi (i)

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).phi(1)
1
c sage: C(2).phi(1)
0
```
```python
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C(Integer(1)).phi(Integer(1))
1
>>> C(Integer(2)).phi(Integer(1))
0
```

**phi_minus_epsilon (i)**

Return \( \phi_i - \varepsilon_i \) of self.

There are sometimes better implementations using the weight for this. It is used for reflections along a string.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).phi_minus_epsilon(1)
1
```

```python
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C(Integer(1)).phi_minus_epsilon(Integer(1))
1
```

**s (i)**

Return the reflection of self along its \( i \)-string.

**EXAMPLES:**

```python
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: b = C(rows=[[1,1],[3]])
sage: b.s(1)
[[2, 2], [3]]
sage: b.s(2)
[[1, 2], [3]]
sage: T = crystals.Tableaux(['A',2],shape=[4])
sage: t = T(rows=[[1,2,2,2]])
sage: t.s(1)
[[1, 1, 1, 2]]
```

```python
>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> b = C(rows=[[Integer(1),Integer(1)],[Integer(3)]]))
>>> b.s(Integer(1))
[[2, 2], [3]]
>>> b.s(Integer(2))
[[1, 2], [3]]
>>> T = crystals.Tableaux(['A',Integer(2)],shape=[Integer(4)])
>>> t = T(rows=[[Integer(1),Integer(2),Integer(2),Integer(2)]])
>>> t.s(Integer(1))
[[1, 1, 1, 2]]
```

**subcrystal (index_set=None, max_depth=inf, direction='both', contained=None, cartan_type=None, category=None)**

Construct the subcrystal generated by self using \( e_i \) and/or \( f_i \) for all \( i \) in index_set.
INPUT:
- `index_set` – (default: `None`) the index set; if `None` then use the index set of the crystal
- `max_depth` – (default: `infinity`) the maximum depth to build
- `direction` – (default: `'both'`) the direction to build the subcrystal; it can be one of the following:
  - `'both'` – using both $e_i$ and $f_i$
  - `'upper'` – using $e_i$
  - `'lower'` – using $f_i$
- `contained` – (optional) a set (or function) defining the containment in the subcrystal
- `cartan_type` – (optional) specify the Cartan type of the subcrystal
- `category` – (optional) specify the category of the subcrystal

See also:
- `Crystals.ParentMethods.subcrystal()`

EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin([A,3,1], 1, 2)
sage: elt = C(1,4)
sage: list(elt.subcrystal(index_set=[1,3]))
[[[1, 4]], [[1, 3]], [[2, 4]], [[2, 3]]]
sage: list(elt.subcrystal(index_set=[1,3], max_depth=1))
[[[1, 4]], [[2, 4]], [[1, 3]]]
sage: list(elt.subcrystal(index_set=[1,3], direction='upper'))
[[[1, 4]], [[1, 3]]]
sage: list(elt.subcrystal(index_set=[1,3], direction='lower'))
[[[1, 4]], [[2, 4]]]
```

```
>>> from sage.all import *
>>> C = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)],...
     Integer(1), Integer(2))
>>> elt = C(Integer(1),Integer(4))
>>> list(elt.subcrystal(index_set=[Integer(1),Integer(3)]))
[[[1, 4]], [[1, 3]], [[2, 4]], [[2, 3]]]
```

```
tensor (*elts)

Return the tensor product of `self` with the crystal elements `elts`.

EXAMPLES:

```
sage: C = crystals.Letters(['A', 3])
sage: B = crystals.infinity.Tableaux(['A', 3])
sage: c = C[0]
sage: b = B.highest_weight_vector()
sage: t = c.tensor(c, b)
sage: ascii_art(t)

1 1 1
1 1 2 2
```

(continues on next page)
3
sage: tensor([c, c, b]) == t
True
sage: ascii_art(tensor([b, b, c]))
1 1 1 1 1 1
2 2 # 2 2 # 1
3 3

>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(3)])
>>> B = crystals.infinity.Tableaux(['A', Integer(3)])
>>> c = C[Integer(0)]
>>> b = B.highest_weight_vector()
>>> t = c.tensor(c, b)
>>> ascii_art(t)
1 1 1 1 # 2 2
3
>>> tensor([c, c, b]) == t
True
>>> ascii_art(tensor([b, b, c]))
1 1 1 1 1 1
2 2 # 2 2 # 1
3

\textbf{to\_highest\_weight}(\textit{index\_set=None})

Return the highest weight element \( u \) and a list \([i_1, \ldots, i_k]\) such that \( \text{self} = f_{i_1} \cdots f_{i_k} u \), where \( i_1, \ldots, i_k \) are elements in \textit{index\_set}.

By default the \textit{index\_set} is assumed to be the full index set of \textit{self}.

\textbf{EXAMPLES:}

sage: T = crystals.Tableaux(['A',3], shape = [1])
sage: t = T(rows = [[3]])
sage: t.to_highest_weight()
[[[1]], [2, 1]]

sage: T = crystals.Tableaux(['A',3], shape = [2,1])
sage: t = T(rows = [[1,2],[4]])
sage: t.to_highest_weight()
[[[1, 1], [2]], [1, 3, 2]]

sage: t.to_highest_weight(index_set = [3])
[[[1, 2], [3]], [3]]

sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: t = K(rows=[[2],[3]]); t.to_highest_weight(index_set=[1])
[[[1], [3]], [1]]

sage: t.to_highest_weight()
Traceback (most recent call last):
...
ValueError: this is not a highest weight crystal

>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape = [Integer(1)])
>>> t = T(rows = [[Integer(3)]])
>>> t.to_highest_weight()
[[[1]], [2, 1]]
to_lowest_weight \( \text{(index_set=\text{None})} \)

Return the lowest weight element \( u \) and a list \( [i_1, \ldots, i_k] \) such that \( \text{self} = e_{i_1} \cdots e_{i_k} u \), where \( i_1, \ldots, i_k \) are elements in \( \text{index\_set} \).

By default the \text{index\_set} is assumed to be the full index set of \( \text{self} \).

EXAMPLES:

```python
sage: T = crystals.Tableaux(["A", 3], shape = [1])
sage: t = T(rows = [[3]])
sage: t.to_lowest_weight()
[[[4]], [3]]
sage: T = crystals.Tableaux(["A", 3], shape = [2, 1])
sage: t = T(rows = [[1, 2], [4]])
sage: t.to_lowest_weight()
[[[3, 4], [4]], [1, 2, 2, 3]]
sage: t.to_lowest_weight(index_set = [3])
[[[1, 2], [4]], [1]]
sage: K = crystals.KirillovReshetikhin("A", 3, 1)
sage: t = K.module_generator(); t
[[[1], [2]]]
sage: t.to_lowest_weight(index_set=[1, 2, 3])
[[[3], [4]], [2, 1, 3, 2]]
sage: t.to_lowest_weight()
Traceback (most recent call last):
... ValueError: this is not a highest weight crystal
```
```python
>>> K = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)],
   -Integer(2), Integer(1))
>>> t = K.module_generator(); t
[[1], [2]]
>>> t.to_lowest_weight(index_set=[Integer(1), Integer(2), Integer(3)])
[[[3], [4]], [2, 1, 3, 2]]
>>> t.to_lowest_weight()
Traceback (most recent call last):
  ... ValueError: this is not a highest weight crystal
```

### weight()

Return the weight of this crystal element.

This method should be implemented by the element class of the crystal.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: C(1).weight()
(1, 0, 0, 0, 0, 0)
```

```python
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(5)])
>>> C(Integer(1)).weight()
(1, 0, 0, 0, 0, 0)
```

### Finite

- alias of `FiniteCrystals`

#### class MorphismMethods

**Bases:** object

**is_embedding()**

Check if `self` is an injective crystal morphism.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['C', 2], shape=[1,1])
sage: C = crystals.Tableaux(['C', 2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_embedding()
True
```

```python
sage: C = crystals.Tableaux(['A', 2], shape=[2,1])
sage: B = crystals.infinity.Tableaux(['A', 2])
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: W = crystals.elementary.T(['A',2], La[1]+La[2])
sage: T = W.tensor(B)
sage: mg = T(W.module_generators[0], B.module_generators[0])
sage: psi = Hom(C,T)([mg])
sage: psi.is_embedding()
True
```

```python
>>> from sage.all import *

>>> B = crystals.Tableaux(['C', Integer(2)], shape=[Integer(1), Integer(1)])
```

is_isomorphism()
Check if self is a crystal isomorphism.

EXAMPLES:

```
sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_isomorphism()
False
```

is_strict()
Check if self is a strict crystal morphism.

EXAMPLES:

```
sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_strict()
True
```

```
class ParentMethods
    Bases: object

    Lambda()

    Returns the fundamental weights in the weight lattice realization for the root system associated with the crystal

    EXAMPLES:

    sage: C = crystals.Letters(['A', 5])
sage: C.Lambda()
    Finite family {1: (1, 0, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0, 0), 3: (1, 1, 1, 0, 0, 0), 4: (1, 1, 1, 1, 0, 0), 5: (1, 1, 1, 1, 1, 0)}

    sage: from sage.all import *
    >>> C = crystals.Letters(['A', Integer(5)])
    >>> C.Lambda()
    Finite family {1: (1, 0, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0, 0), 3: (1, 1, 1, 0, 0, 0), 4: (1, 1, 1, 1, 0, 0), 5: (1, 1, 1, 1, 1, 0)}

    an_element()
    Returns an element of self
    sage: C = crystals.Letters(['A', 5])
sage: C.an_element()
    sage: C.an_element()

    cartan_type()
    Returns the Cartan type of the crystal

    EXAMPLES:

    sage: C = crystals.Letters(['A', 2])
sage: C.cartan_type()
    ['A', 2]

    sage: from sage.all import *
    >>> C = crystals.Letters(['A', Integer(2)])
    >>> C.cartan_type()
    ['A', 2]

    connected_components()
    Return the connected components of self as subcrystals.

    EXAMPLES:

    sage: B = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: C = crystals.Letters(['A', 2])
sage: T = crystals.TensorProduct(B,C)
sage: T.connected_components()
    [Subcrystal of Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
    The crystal of letters for type ['A', 2]],
    Subcrystal of Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
    The crystal of letters for type ['A', 2]],
    Subcrystal of Full tensor product of the crystals
    [The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
    The crystal of letters for type ['A', 2]]]
from sage.all import *

B = crystals.Tableaux(['A', Integer(2)], shape=[Integer(2), Integer(1)])
C = crystals.Letters(['A', Integer(2)])
T = crystals.TensorProduct(B, C)
T.connected_components()
See also:

For more examples, see \texttt{sage.categories.crystals.CrystalHomset}.

\textbf{EXAMPLES:}

We construct the natural embedding of a crystal using tableaux into the tensor product of single boxes via the reading word:

\begin{verbatim}
>>> from sage.all import *
>>> B = crystals.Tableaux(['A',2], shape=[2,1])
>>> F = crystals.Tableaux(['A',2], shape=[1])
>>> T = crystals.TensorProduct(F, F, F)
>>> mg = T.highest_weight_vectors()[2]; mg
[[[1]], [[2]], [[1]]]
>>> psi = B.crystal_morphism([mg], codomain=T); psi
['A', 2] Crystal morphism:
  From: The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]
  To:   Full tensor product of the crystals
         [The crystal of tableaux of type ['A', 2] and shape(s) [[1]],
          The crystal of tableaux of type ['A', 2] and shape(s) [[1]],
          The crystal of tableaux of type ['A', 2] and shape(s) [[1]]]
  Defn: [[1, 1], [2]] |--> [[[1]], [[2]], [[1]]]

>>> b = B.module_generators[0]
>>> b.pp()
 1 1
  2

>>> psi(b)
[[[1]], [[2]], [[1]]]

>>> psi(b.f(2))
[[[1]], [[3]], [[1]]]

>>> psi(b.f_string([2,1,1]))
[[[2]], [[3]], [[2]]]

>>> lw = b.to_lowest_weight()[0]
>>> lw.pp()
 2 3
  3

>>> psi(lw)
[[[3]], [[3]], [[2]]]

>>> psi(lw) == mg.to_lowest_weight()[0]
True

>>> from sage.all import *
>>> B = crystals.Tableaux(['A',Integer(2)], shape=[2,1])
>>> F = crystals.Tableaux(['A',Integer(2)], shape=[1])
>>> T = crystals.TensorProduct(F, F, F)
>>> mg = T.highest_weight_vectors()[2]; mg
[[[1]], [[2]], [[1]]]
>>> psi = B.crystal_morphism([mg], codomain=T); psi
['A', 2] Crystal morphism:
  From: The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]
  To:   Full tensor product of the crystals
         [The crystal of tableaux of type ['A', 2] and shape(s) [[1]],
          The crystal of tableaux of type ['A', 2] and shape(s) [[1]],
          The crystal of tableaux of type ['A', 2] and shape(s) [[1]]]
  Defn: [[1, 1], [2]] |--> [[[1]], [[2]], [[1]]]

>>> b = B.module_generators[0]
>>> b.pp()
 1 1
  2

(continues on next page)
\end{verbatim}
We now take the other isomorphic highest weight component in the tensor product:

```
sage: mg = T.highest_weight_vectors()[1]; mg
type: Tableaux D(4)
[2]
[1]
[1]
[1]
sage: psi = B.crystal_morphism([mg], codomain=T)
sage: psi(lw)
[[1], [2], [1], [1], [1]]
```

We construct a crystal morphism of classical crystals using a Kirillov-Reshetikhin crystal:

```
sage: B = crystals.Tableaux(['D', 4], shape=[1,1])
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: v = K.module_generators[1]
sage: psi = B.crystal_morphism([v], codomain=K, category=FiniteCrystals())
sage: psi
['D', 4] -> ['D', 4, 1] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['D', 4] and shape(s) [[1, 1]]
  To: Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(2,2)
  Defn: [[1], [2]] |--> [[1], [2]]
sage: b = B.module_generators[0]
sage: psi(b)
[[1], [2]]
sage: psi(b.to_lowest_weight()[0])
[[-2], [-1]]
```

(continues on next page)
We can define crystal morphisms using a different set of generators. For example, we construct an example using the lowest weight vector:

```
sage: B = crystals.Tableaux(['A',2], shape=[1])
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: T = crystals.elementary.T(['A',2], La[2])
sage: Bp = T.tensor(B)
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: x = C.module_generators[0].f_string([1,2])
sage: psi = Bp.crystal_morphism([x], generators=Bp.lowest_weight_vectors())
sage: psi(Bp.highest_weight_vector())
[[1, 1], [2]]
```

We can also use a dictionary to specify the generators and their images:

```
sage: psi = Bp.crystal_morphism({Bp.lowest_weight_vectors()[0]: x})
sage: psi(Bp.highest_weight_vector())
[[1, 1], [2]]
```

We construct a twisted crystal morphism induced from the diagram automorphism of type $A_3^{(1)}$:

```
sage: La = RootSystem(['A',3,1]).weight_lattice().fundamental_weights()
sage: B0 = crystals.GeneralizedYoungWalls(3, La[0])
sage: B1 = crystals.GeneralizedYoungWalls(3, La[1])
```
We construct a virtual crystal morphism from type $G_2$ into type $D_4$:

```python
sage: D = crystals.Tableaux(['D',4], shape=[1,1])
sage: G = crystals.Tableaux(['G',2], shape=[1])
sage: psi = G.crystal_morphism(D.module_generators,
....:     virtualization={1:[2], 2:[1,3,4]},
....:     scaling_factors={1:1, 2:1})
sage: for x in G:
....:     ascii_art(x, psi(x), sep=' |--> ')
....:     print(""")
    1 |-->  2
    1
    2 |-->  3
    2
    3 |--> -3
    3
    0 |--> -3
```

(continues on next page)
```python
>>> from sage.all import *
>>> D = crystals.Tableaux(['D', Integer(4)], shape=[Integer(1), Integer(1)])
>>> G = crystals.Tableaux(['G', Integer(2)], shape=[Integer(1)])
>>> psi = G.crystal_morphism(D.module_generators,
... virtualization={Integer(1):[Integer(2)],
... Integer(2):[Integer(1), Integer(3), Integer(4)]},
... scaling_factors={Integer(1):Integer(1),
... Integer(2):Integer(1)})
>>> for x in G:
...     ascii_art(x, psi(x), sep=' |--> ')
...     print('')
1 |--> 2
<BLANKLINE>
2 |--> 3
<BLANKLINE>
3 |--> -3
<BLANKLINE>
0 |--> -3
<BLANKLINE>
3 |--> -2
<BLANKLINE>
-3 |--> -2
<BLANKLINE>
-2 |--> -1
<BLANKLINE>
-1 |--> -1
```

digraph (subset=None, index_set=None)

Return the DiGraph associated to self.

INPUT:

- `subset` – (optional) a subset of vertices for which the digraph should be constructed
- `index_set` – (optional) the index set to draw arrows

EXAMPLES:

```
sage: C = Crystals().example(5)
sage: C.digraph()
Digraph on 6 vertices
```

```python
>>> from sage.all import *
>>> C = Crystals().example(Integer(5))
```
C.digraph()

Digraph on 6 vertices

The edges of the crystal graph are by default colored using blue for edge 1, red for edge 2, and green for
edge 3:

```python
sage: G = C.digraph()
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external...
  window)
```

One may also overwrite the colors:

```python
sage: G.set_latex_options(color_by_label = {1:"red", 2:"purple", 3:"blue"})
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external...
  window)
```

Or one may add colors to yet unspecified edges:

```python
sage: C.cartan_type()._index_set_coloring[4]="purple"
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external...
  window)
```

Here is an example of how to take the top part up to a given depth of an infinite dimensional crystal:

```python
sage: C = CartanType(['C',2,1])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[0])
sage: S = T.subcrystal(max_depth=3)
```
Here is a way to construct a picture of a Demazure crystal using the \texttt{subset} option:

\begin{verbatim}
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = B.highest_weight_vector()
sage: D = B.demazure_subcrystal(t, [2,1])
sage: list(D)
[[[1, 1], [2]], [[1, 2], [2]], [[1, 1], [3]], [[1, 3], [2]], [[1, 3], [3]]]
sage: view(D)  # optional - dot2tex graphviz, not tested (opens external...window)
\end{verbatim}

We can also choose to display particular arrows using the \texttt{index_set} option:

\begin{verbatim}
>>> from sage.all import *
>>> C = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: G = C.digraph(index_set=[1,3])
sage: len(G.edges(sort=False))
20
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external...window)
\end{verbatim}
Todo: Add more tests.

direct_sum($X$)
Return the direct sum of self with $X$.

EXAMPLES:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: B.direct_sum(C)
Direct sum of the crystals Family
(The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2])
```

As a shorthand, we can use +:

```
sage: B + C
Direct sum of the crystals Family
(The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2])
```

dot_tex()
Return a dot_tex string representation of self.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: C.dot_tex()
digraph G { \n node [ shape=plaintext ]; \n N_0 [ label = " ", texlbl = "$1$" ]; \n N_1 [ label = " ", texlbl = "$2$" ]; \n N_2 [ label = " ", texlbl = "$3$" ]; \n N_0 -> N_1 [ label = " ", texlbl = "1" ]; \n N_1 -> N_2 [ label = " ", texlbl = "2" ]; \n}
```
>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(2)])
>>> C.dot_tex()
'digraph G { \n node [ shape=plaintext ];\n N_0 [ label = " ", texlbl = "$1$" ];\n N_1 [ label = " ", texlbl = "$2$" ];\n N_2 [ label = " ", texlbl = "$3$" ];\n N_0 -> N_1 [ label = " ", texlbl = "1" ];\n N_1 -> \n N_2 [ label = " ", texlbl = "2" ];\n}''

index_set()

Returns the index set of the Dynkin diagram underlying the crystal

EXAMPLES:

sage: C = crystals.Letters(['A', 5])
sage: C.index_set()
(1, 2, 3, 4, 5)

is_connected()

Return True if self is a connected crystal.

EXAMPLES:

sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: B.is_connected()
True
sage: T.is_connected()
False

latex(**options)

Returns the crystal graph as a latex string. This can be exported to a file with self.
latex_file('filename').

EXAMPLES:

sage: T = crystals.Tableaux(['A',2],shape=[1])
sage: T._latex_()
'...tikzpicture...

...view(T) # optional - dot2tex graphviz, not tested (opens external...
One can for example also color the edges using the following options:

```python
sage: T = crystals.Tableaux(['A',2],shape=[1])
sage: T._latex_(color_by_label={0:"black", 1:"red", 2:"blue"})
'...tikzpicture...'
```

**latex_file** (*filename*)

Export a file, suitable for pdflatex, to *filename*.

This requires a proper installation of dot2tex in sage-python. For more information see the documentation for self.latex().

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: fn = tmp_filename(ext='.tex')
sage: C.latex_file(fn)
```

**metapost** (*filename*, **thicklines=False**, **labels=**True, **scaling_factor=1.0**, **tallness=1.0**)

Export a file, suitable for MetaPost, to *filename*.

Root operators $e(1)$ or $f(1)$ move along red lines, $e(2)$ or $f(2)$ along green. The highest weight is in the lower left. Vertices with the same weight are kept close together. The concise labels on the nodes are strings introduced by Berenstein and Zelevinsky and Littelmann; see Littelmann’s paper Cones, Crystals, Patterns, sections 5 and 6.

For Cartan types B2 or C2, the pattern has the form

$a_3a_4a_1$

where $c * a_2 = a_3 = 2 * a_4 = 0$ and $a_1 = 0$, with $c = 2$ for B2, $c = 1$ for C2. Applying $e(2)$ $a_1$ times, $e(1)$ $a_2$ times, $e(2)$ $a_3$ times, $e(1)$ $a_4$ times returns to the highest weight. (Observe that Littelmann writes the roots in opposite of the usual order, so our $e(1)$ is his $e(2)$ for these Cartan types.) For type A2, the pattern has the form

$a_2a_3a_1$

where applying $e(1)$ $a_3$ times, $e(2)$ $a_2$ times then $e(1)$ $a_1$ times returns to the highest weight. These data determine the vertex and may be translated into a Gelfand-Tsetlin pattern or tableau.

**INPUT:**

```
```
• filename – name of the output file, e.g., 'filename.mp'  
• thicklines – (default: True) for thicker edges  
• labels – (default: False) to suppress labeling of the vertices  
• scaling_factor – (default: 1.0) Increasing or decreasing the scaling factor changes the size of the image  
• tallness – (default: 1.0) Increasing makes the image taller without increasing the width

EXAMPLES:

```
sage: C = crystals.Letters(['A', 2])
sage: C.metapost(tmp_filename())
```

```
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(2)])
>>> C.metapost(tmp_filename())

sage: C = crystals.Letters(['A', 5])
sage: C.metapost(tmp_filename())
Traceback (most recent call last):
  ...   
NotImplementedError
```

```
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(5)])
>>> C.metapost(tmp_filename())
Traceback (most recent call last):
  ...   
NotImplementedError
```

`number_of_connected_components()`  
Return the number of connected components of `self`.  

EXAMPLES:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: T.number_of_connected_components()
3
```

```
>>> from sage.all import *

>>> B = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(B,C)
>>> T.number_of_connected_components()
3
```

`plot(**options)`  
Return the plot of `self` as a directed graph.  

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: print(C.plot())
``` Graphics object consisting of 17 graphics primitives
```python
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(5)])

>>> print(C.plot())
Graphics object consisting of 17 graphics primitives

plot3d(**options)

Return the 3-dimensional plot of self as a directed graph.

EXAMPLES:

sage: C = crystals.KirillovReshetikhin(['A',3,1], 2, 1)
sage: print(C.plot3d())
Graphics3d Object

sage: C = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)], ~Integer(2),Integer(1))

>>> print(C.plot3d())
Graphics3d Object

subcrystal(index_set=None, generators=None, max_depth=inf, direction='both', contained=None, virtualization=None, scaling_factors=None, cartan_type=None, category=None)

Construct the subcrystal from generators using $e_i$ and/or $f_i$ for all $i$ in index_set.

INPUT:

- `index_set` -- (default: None) the index set; if None then use the index set of the crystal
- `generators` -- (default: None) the list of generators; if None then use the module generators of the crystal
- `max_depth` -- (default: infinity) the maximum depth to build
- `direction` -- (default: both) the direction to build the subcrystal; it can be one of the following:
  - 'both' -- using both $e_i$ and $f_i$
  - 'upper' -- using $e_i$
  - 'lower' -- using $f_i$
- `contained` -- (optional) a set or function defining the containment in the subcrystal
- `virtualization`, `scaling_factors` -- (optional) dictionaries whose key $i$ corresponds to the sets $\sigma_i$ and $\gamma_i$ respectively used to define virtual crystals; see VirtualCrystal
- `cartan_type` -- (optional) specify the Cartan type of the subcrystal
- `category` -- (optional) specify the category of the subcrystal

EXAMPLES:

sage: C = crystals.KirillovReshetikhin(['A',3,1], 1, 2)
sage: S = list(C.subcrystal(index_set=[1,2])); S
[[[1, 1]], [[1, 2]], [[2, 2]], [[1, 3]], [[2, 3]], [[3, 3]]]
sage: C.cardinality()
10

sage: len(S)
6

sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)]))
[[[1, 4]], [[2, 4]], [[1, 3]], [[2, 3]]]

sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], max_depth=1))
[[[1, 4]], [[2, 4]], [[1, 3]]]

sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], direction='upper'))
```

(continues on next page)
We construct the subcrystal which contains the necessary data to construct the corresponding dual equivalence graph:

```python
sage: C = crystals.Tableaux(['A', 5], shape=[3, 3])
sage: is_wt0 = lambda x: all(x.epsilon(i) == x.phi(i) for i in x.parent().index_set())
sage: def check(x):
    if is_wt0(x):
        return True
    for i in x.parent().index_set()[:-1]:
        L = [x.e(i), x.e_string([i,i+1]), x.f(i), x.f_string([i, i+1])]
        if any(y is not None and is_wt0(y) for y in L):
            return True
    return False
sage: wt0 = [x for x in C if is_wt0(x)]
sage: S = C.subcrystal(contained=check, generators=wt0)
sage: S.module_generators[0]  # (continued on next page)
```
An example of a type $B_2$ virtual crystal inside of a type $A_3$ ambient crystal:

```
sage: A = crystals.Tableaux(['A',3], shape=[2,1,1])
sage: S = A.subcrystal(virtualization={1:[1,3], 2:[2]},
   ....: scaling_factors={1:1,2:1}, cartan_type=['B',2])
sage: B = crystals.Tableaux(['B',2], shape=[1])
sage: S.digraph().is_isomorphic(B.digraph(), edge_labels=True)
True
```

```
sage: C = crystals.Letters(['A', 3])
sage: B = crystals.infinity.Tableaux(['A', 3])
sage: T = C.tensor(C, B); T
Full tensor product of the crystals
[The crystal of letters for type ['A', 3],
 The crystal of letters for type ['A', 3],]
```

4.31. Crystals
The infinity crystal of tableaux of type ['A', 3]

```python
sage: tensor([C, C, B]) is T
True
```

```python
sage: C = crystals.Letters(['A', 2])
sage: T = C.tensor(C, C, generators=[[C(2),C(1),C(1)], [C(1),C(2),C(1)]]); T
```

The tensor product of the crystals
- The crystal of letters for type ['A', 2],
- The crystal of letters for type ['A', 2],
- The crystal of letters for type ['A', 2]

```python
sage: T.module_generators
([2, 1, 1], [1, 2, 1])
```

```python
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(3)])

>>> B = crystals.infinity.Tableaux(['A', Integer(3)])

>>> T = C.tensor(C, B); T
```

Full tensor product of the crystals
- The crystal of letters for type ['A', 3],
- The crystal of letters for type ['A', 3],
- The infinity crystal of tableaux of type ['A', 3]

```python
>>> tensor([C, C, B]) is T
True
```

```python
>>> C = crystals.Letters(['A', Integer(2)])

>>> T = C.tensor(C, C, generators=[[C(Integer(2)), C(Integer(1)), C(Integer(1))], [C(Integer(1)), C(Integer(2)), C(Integer(1))]]); T
```

The tensor product of the crystals
- The crystal of letters for type ['A', 2],
- The crystal of letters for type ['A', 2],
- The crystal of letters for type ['A', 2]

```python
>>> T.module_generators
([2, 1, 1], [1, 2, 1])
```

`weight_lattice_realization()`

Return the weight lattice realization used to express weights in `self`.

This default implementation uses the ambient space of the root system for (non relabelled) finite types and the weight lattice otherwise. This is a legacy from when ambient spaces were partially implemented, and may be changed in the future.

For affine types, this returns the extended weight lattice by default.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
sage: K = crystals.KirillovReshetikhin(['A', 2, 1], 1, 1)
sage: K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

```python
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(5)])

>>> C.weight_lattice_realization()
```

(continues on next page)
class SubcategoryMethods
Bases: object

Methods for all subcategories.

TensorProducts()

Return the full subcategory of objects of self constructed as tensor products.

See also:

• tensor.TensorProductsCategory
• RegressiveCovariantFunctorialConstruction.

EXAMPLES:

```sage
sage: HighestWeightCrystals().TensorProducts()
Category of tensor products of highest weight crystals
```

```sage
>>> from sage.all import *

HighestWeightCrystals().TensorProducts()
Category of tensor products of highest weight crystals
```

class TensorProducts(category, *args)
Bases: TensorProductsCategory

The category of crystals constructed by tensor product of crystals.

extra_super_categories()

EXAMPLES:

```sage
sage: Crystals().TensorProducts().extra_super_categories()
[Category of crystals]
```

```sage
>>> from sage.all import *

[Category of crystals]
```

def example(choice='highwt', **kwds)

Returns an example of a crystal, as per `Category.example()`.

INPUT:

• choice – str [default: ‘highwt’]. Can be either ‘highwt’ for the highest weight crystal of type A, or ‘naive’ for an example of a broken crystal.  
• **kwds – keyword arguments passed onto the constructor for the chosen crystal.

EXAMPLES:

```sage
sage: Crystals().example(choice='highwt', n=5)
Highest weight crystal of type A_5 of highest weight omega_1
sage: Crystals().example(choice='naive')
A broken crystal, defined by digraph, of dimension five.
```
class sage.categories.cw_complexes.CWComplexes
Bases: Category_singleton

The category of CW complexes.
A CW complex is a Closure-finite cell complex in the Weak topology.

REFERENCES:
• Wikipedia article CW_complex

Note: The notion of “finite” is that the number of cells is finite.

EXAMPLES:

sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes(); C
Category of CW complexes

Compact_extra_super_categories()
Return extraneous super categories for CWComplexes().Compact().
A compact CW complex is finite, see Proposition A.1 in [Hat2002].

Todo: Fix the name of finite CW complexes.

EXAMPLES:
```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().Compact()  # indirect doctest
Category of finite finite dimensional CW complexes
sage: CWComplexes().Compact() is CWComplexes().Finite()
True

sage: from sage.all import *
sage: from sage.categories.cw_complexes import CWComplexes

>> from sage.categories.cw_complexes import CWComplexes
>>> CWComplexes().Compact()  # indirect doctest
Category of finite finite dimensional CW complexes
>>> CWComplexes().Compact() is CWComplexes().Finite()
True
```

class Connected(base_category)

Bases: CategoryWithAxiom

The category of connected CW complexes.

class ElementMethods

Bases: object
dimension()

Return the dimension of self.

EXAMPLES:

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.an_element().dimension()
2

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes

>> from sage.categories.cw_complexes import CWComplexes
>>> X = CWComplexes().example()
>>> X.an_element().dimension()
2
```

class Finite(base_category)

Bases: CategoryWithAxiom

Category of finite CW complexes.

A finite CW complex is a CW complex with a finite number of cells.

class ParentMethods

Bases: object
dimension()

Return the dimension of self.

EXAMPLES:

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.dimension()
2
```

4.32. CW Complexes 495
```python
from sage.all import *
from sage.categories.cw_complexes import CWComplexes
X = CWComplexes().example()
X.dimension()
```

**extra_super_categories()**

Return the extra super categories of self.

A finite CW complex is a compact finite-dimensional CW complex.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes().Finite()
sage: C.extra_super_categories()
[Category of finite dimensional CW complexes,
 Category of compact topological spaces]
```

```python
from sage.all import *
from sage.categories.cw_complexes import CWComplexes

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes
>>> X = CWComplexes().example()
>>> X.cells()
```

```python
cells()

Return the cells of self.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: C = X.cells()
sage: sorted((d, C[d]) for d in C.keys())
[(0, (0-cell v,)),
 (1, (0-cell e1, 0-cell e2)),
 (2, (2-cell f,))]
```

**class FiniteDimensional(base_category)**

Bases: `CategoryWithAxiom`

Category of finite dimensional CW complexes.

**class ParentMethods**

Bases: object

cells()

Return the cells of self.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: C = X.cells()
sage: sorted((d, C[d]) for d in C.keys())
[(0, (0-cell v,)),
 (1, (0-cell e1, 0-cell e2)),
 (2, (2-cell f,))]
```
**dimension()**

Return the dimension of self.

EXAMPLES:

```python
def example(self):
    return self.dimension()
```

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.dimension()
2

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes
>>> X = CWComplexes().example()
>>> X.dimension()
2
```

**class SubcategoryMethods**

Bases: object

**Connected()**

Return the full subcategory of the connected objects of self.

EXAMPLES:

```python
def Connected(self):
    return self.category().Connected()
```

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().Connected()
Category of connected CW complexes

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes
>>> CWComplexes().Connected()
Category of connected CW complexes
```

**FiniteDimensional()**

Return the full subcategory of the finite dimensional objects of self.

EXAMPLES:

```python
def FiniteDimensional(self):
    return self.category().FiniteDimensional()
```

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes().FiniteDimensional(); C
Category of finite dimensional CW complexes

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes
>>> C = CWComplexes().FiniteDimensional(); C
Category of finite dimensional CW complexes
```

**super_categories()**

EXAMPLES:

```python
def super_categories(self):
    return [self.category().super_categories()]
```

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().super_categories()
[Category of topological spaces]

>>> from sage.all import *
>>> from sage.categories.cw_complexes import CWComplexes
>>> CWComplexes().super_categories()
[Category of topological spaces]
```

(continues on next page)
4.33 Discrete Valuation Rings (DVR) and Fields (DVF)

```python
>>> CWComplexes().super_categories()
[Category of topological spaces]
```

**class** `sage.categories.discretevaluation.DiscreteValuationFields`

**Bases:** `Category_singleton`

The category of discrete valuation fields

**EXAMPLES:**

```python
sage: Qp(7) in DiscreteValuationFields()  # needs sage.rings.padics
True
sage: TestSuite(DiscreteValuationFields()).run()
```

```python
>>> from sage.all import *
```

```python
>>> Qp(Integer(7)) in DiscreteValuationFields()  # needs sage.rings.padics
True
>>> TestSuite(DiscreteValuationFields()).run()
```

**class** `ElementMethods`

**Bases:** `object`

```python
valuation()
```

Return the valuation of this element.

**EXAMPLES:**

```python
sage: # needs sage.rings.padics
sage: x = Qp(5)(50)
sage: x.valuation()
2
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.rings.padics
>>> x = Qp(Integer(5))(Integer(50))
>>> x.valuation()
2
```

**class** `ParentMethods`

**Bases:** `object`

```python
residue_field()
```

Return the residue field of the ring of integers of this discrete valuation field.

**EXAMPLES:**

```python
sage: Qp(5).residue_field()  # needs sage.rings.padics
Finite Field of size 5
```

(continues on next page)
sage: K.<u> = LaurentSeriesRing(QQ)
sage: K.residue_field()
Rational Field

```python
>>> from sage.all import *

>>> Qp(Integer(5)).residue_field()
˓→ # needs sage.rings.padics
Finite Field of size 5

K = LaurentSeriesRing(QQ, names=('u',)); (u,) = K._first_ngens(1)
>>> K.residue_field()
Rational Field
```

**uniformizer()**

Return a uniformizer of this ring.

**EXAMPLES:**

```python
sage: Qp(5).uniformizer()
˓→ # needs sage.rings.padics
5 + O(5^21)
```

```python
>>> from sage.all import *

>>> Qp(Integer(5)).uniformizer()
˓→ # needs sage.rings.padics
5 + O(5^21)
```

**super_categories()**

**EXAMPLES:**

```python
sage: DiscreteValuationFields().super_categories()
[Category of fields]
```

```python
>>> from sage.all import *

>>> DiscreteValuationFields().super_categories()
[Category of fields]
```

**class** `sage.categories.discrete_valuation.DiscreteValuationRings`

**Bases:** `Category_singleton`

The category of discrete valuation rings

**EXAMPLES:**

```python
sage: GF(7)['x'] in DiscreteValuationRings()
True

sage: TestSuite(DiscreteValuationRings()).run()
```

```python
>>> from sage.all import *

>>> GF(Integer(7))['x'] in DiscreteValuationRings()
True

>>> TestSuite(DiscreteValuationRings()).run()
```

**class** `ElementMethods`

**Bases:** `object`

4.33. Discrete Valuation Rings (DVR) and Fields (DVF) 499
euclidean_degree()

Return the Euclidean degree of this element.

gcd(other)

Return the greatest common divisor of self and other, normalized so that it is a power of the distinguished uniformizer.

is_unit()

Return True if self is invertible.

EXAMPLES:

```python
sage: # needs sage.rings.padics
sage: x = Zp(5)(50)
sage: x.is_unit()
False

sage: # needs sage.rings.padics
sage: x = Zp(7)(50)
sage: x.is_unit()
True
```

```python
>>> from sage.all import *

>>> # needs sage.rings.padics

>>> x = Zp(Integer(5))(Integer(50))

>>> x.is_unit()
False

>>> # needs sage.rings.padics

>>> x = Zp(Integer(7))(Integer(50))

>>> x.is_unit()
True
```

lcm(other)

Return the least common multiple of self and other, normalized so that it is a power of the distinguished uniformizer.

quo_rem(other)

Return the quotient and remainder for Euclidean division of self by other.

EXAMPLES:

```python
sage: R.<q> = GF(5)[[]]

sage: (q^2 + q).quo_rem(q)
(1 + q, 0)

sage: (q + 1).quo_rem(q^2)
(0, 1 + q)
```

```python
>>> from sage.all import *


>>> R = GF(Integer(5))[[q]]; (q,) = R._first_ngens(1)

>>> (q^Integer(2) + q).quo_rem(q)
(1 + q, 0)

>>> (q + Integer(1)).quo_rem(q^Integer(2))
(0, 1 + q)
```

valuation()

Return the valuation of this element.
EXAMPLES:

```
sage: # needs sage.rings.padics
sage: x = Zp(5)(50)
sage: x.valuation()
2
```

```python
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> x = Zp(Integer(5))(Integer(50))
>>> x.valuation()
2
```

class ParentMethods

Bases: object

`residue_field()`

Return the residue field of this ring.

EXAMPLES:

```
sage: Zp(5).residue_field()  # needs sage.rings.padics
Finite Field of size 5

sage: K.<u> = QQ[['u']]
sage: K.residue_field()
Rational Field
```

```python
>>> from sage.all import *
>>> Zp(Integer(5)).residue_field()  # needs sage.rings.padics
Finite Field of size 5

>>> K = QQ[['u']]; (u,) = K._first_ngens(1)
>>> K.residue_field()
Rational Field
```

`uniformizer()`

Return a uniformizer of this ring.

EXAMPLES:

```
sage: Zp(5).uniformizer()  # needs sage.rings.padics
5 + O(5^21)

sage: K.<u> = QQ[['u']]
sage: K.uniformizer()
u
```

```python
>>> from sage.all import *
>>> Zp(Integer(5)).uniformizer()  # needs sage.rings.padics
5 + O(5^21)

>>> K = QQ[['u']]; (u,) = K._first_ngens(1)
```
(continues on next page)
super_categories()

EXAMPLES:

```python
sage: DiscreteValuationRings().super_categories()
[Category of euclidean domains]
```

```python
from sage.all import *

DiscreteValuationRings().super_categories()
[Category of euclidean domains]
```

### 4.34 Distributive Magmas and Additive Magmas

```python
class sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas:
    pass
```

Bases: `CategoryWithAxiom_singleton`

The category of sets \((S, +, \ast)\) with \(\ast\) distributing on 

\(+\).

This is similar to a ring, but \(+\) and \(\ast\) are only required to be (additive) magmas.

EXAMPLES:

```
sage: from sage.categories.distributive_magmas_and_additive_magmas import *
    DistributiveMagmasAndAdditiveMagmas
sage: C = DistributiveMagmasAndAdditiveMagmas(); C
Category of distributive magmas and additive magmas
sage: C.super_categories()
[Category of magmas and additive magmas]
```

```
from sage.all import *

from sage.categories.distributive_magmas_and_additive_magmas import *

C = DistributiveMagmasAndAdditiveMagmas(); C
Category of distributive magmas and additive magmas

C.super_categories()
[Category of magmas and additive magmas]
```

```python
class AdditiveAssociative(base_category):
    pass
```

Bases: `CategoryWithAxiom_singleton`

```python
class AdditiveCommutative(base_category):
    pass
```

Bases: `CategoryWithAxiom_singleton`

```python
class AdditiveUnital(base_category):
    pass
```

Bases: `CategoryWithAxiom_singleton`

```python
class Associative(base_category):
    pass
```

Bases: `CategoryWithAxiom_singleton`
AdditiveInverse
alias of $Rngs$

Unital
alias of $Semirings$

class CartesianProducts (category, *args)
   Bases: CartesianProductsCategory

   extra_super_categories()
   Implement the fact that a Cartesian product of magmas distributing over additive magmas is a magma distributing over an additive magma.

   EXAMPLES:

   sage: C = (Magmas() & AdditiveMagmas()).Distributive().CartesianProducts()
   sage: C.extra_super_categories()
   [Category of distributive magmas and additive magmas]
   sage: C.axioms()
   frozenset({'Distributive'})

   >>> from sage.all import *
   >>> C = (Magmas() & AdditiveMagmas()).Distributive().CartesianProducts()
   >>> C.extra_super_categories()
   [Category of distributive magmas and additive magmas]
   >>> C.axioms()
   frozenset({'Distributive'})


class ParentMethods
   Bases: object

4.35 Division rings

class sage.categories.division_rings.DivisionRings (base_category)
   Bases: CategoryWithAxiom_singleton

   The category of division rings

   A division ring (or skew field) is a not necessarily commutative ring where all non-zero elements have multiplicative inverses

   EXAMPLES:

   sage: DivisionRings()
   Category of division rings
   sage: DivisionRings().super_categories()
   [Category of domains]

   >>> from sage.all import *
   >>> DivisionRings()
   Category of division rings
   >>> DivisionRings().super_categories()
   [Category of domains]

4.35. Division rings 503
Commutative
  alias of Fields

class ElementMethods
  Bases: object

Finite_extra_super_categories()
  Return extraneous super categories for DivisionRings().Finite().

EXAMPLES:
  Any field is a division ring:

  sage: Fields().is_subcategory(DivisionRings())
  True

  >>> from sage.all import *
  >>> Fields().is_subcategory(DivisionRings())
  True

  This method specifies that, by Weddeburn theorem, the reciprocal holds in the finite case: a finite division ring is commutative and thus a field:

  sage: DivisionRings().Finite_extra_super_categories()
  (Category of commutative magmas,)
  sage: DivisionRings().Finite()
  Category of finite enumerated fields

  >>> from sage.all import *
  >>> DivisionRings().Finite_extra_super_categories()
  (Category of commutative magmas,)
  >>> DivisionRings().Finite()
  Category of finite enumerated fields

  Warning: This is not implemented in DivisionRings.Finite.extra_super_categories because the categories of finite division rings and of finite fields coincide. See the section Deduction rules in the documentation of axioms.

class ParentMethods
  Bases: object

extra_super_categories()
  Return the Domains category.

  This method specifies that a division ring has no zero divisors, i.e. is a domain.

  See also:
  The Deduction rules section in the documentation of axioms

EXAMPLES:

  sage: DivisionRings().extra_super_categories()
  (Category of domains,)
  sage: "NoZeroDivisors" in DivisionRings().axioms()
  True
>>> from sage.all import *
>>> DivisionRings().extra_super_categories()
(Category of domains,)
>>> "NoZeroDivisors" in DivisionRings().axioms()
True

4.36 Domains

class sage.categories.domains.Domains(base_category)
   Bases: CategoryWithAxiom_singleton

The category of domains

A domain (or non-commutative integral domain), is a ring, not necessarily commutative, with no nonzero zero divisors.

EXAMPLES:

sage: C = Domains(); C
Category of domains
sage: C.super_categories()
[Category of rings]

Commutative
   alias of IntegralDomains

class ElementMethods
   Bases: object

class ParentMethods
   Bases: object

super_categories()
   EXAMPLES:

sage: Domains().super_categories()
[Category of rings]

>>> from sage.all import *
>>> Domains().super_categories()
[Category of rings]
4.37 Enumerated sets

class sage.categories.enumerated_sets.EnumeratedSets(base_category)
    Bases: CategoryWithAxiom_singleton

The category of enumerated sets

An enumerated set is a finite or countable set or multiset $S$ together with a canonical enumeration of its elements; conceptually, this is very similar to an immutable list. The main difference lies in the names and the return type of the methods, and of course the fact that the list of elements is not supposed to be expanded in memory. Whenever possible one should use one of the two sub-categories FiniteEnumeratedSets or InfiniteEnumeratedSets.

The purpose of this category is threefold:

- to fix a common interface for all these sets;
- to provide a bunch of default implementations;
- to provide consistency tests.

The standard methods for an enumerated set $S$ are:

- $S$.cardinality(): the number of elements of the set. This is the equivalent for $\text{len}$ on a list except that the return value is specified to be a Sage Integer or infinity, instead of a Python int.
- $\text{iter}(S)$: an iterator for the elements of the set;
- $S$.list(): a fresh list of the elements of the set, when possible; raises a $\text{NotImplementedError}$ if the list is predictably too large to be expanded in memory.
- $S$.tuple(): a tuple of the elements of the set, when possible; raises a $\text{NotImplementedError}$ if the tuple is predictably too large to be expanded in memory.
- $S$.unrank($n$): the $n$-th element of the set when $n$ is a Sage Integer. This is the equivalent for $l[n]$ on a list.
- $S$.rank($e$): the position of the element $e$ in the set; This is equivalent to $l$.index($e$) for a list except that the return value is specified to be a Sage Integer, instead of a Python int.
- $S$.first(): the first object of the set; it is equivalent to $S$.unrank(0).
- $S$.next($e$): the object of the set which follows $e$; it is equivalent to $S$.unrank($S$.rank($e$) + 1).
- $S$.random_element(): a random generator for an element of the set. Unless otherwise stated, and for finite enumerated sets, the probability is uniform.

For examples and tests see:

- $\text{FiniteEnumeratedSets().example()}$
- $\text{InfiniteEnumeratedSets().example()}$

EXAMPLES:

```
sage: EnumeratedSets()
Category of enumerated sets
sage: EnumeratedSets().super_categories()
[Category of sets]
sage: EnumeratedSets().all_super_categories()
[Category of enumerated sets, Category of sets, Category of sets with partial maps, Category of objects]
```
class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

    class ParentMethods
        Bases: object
        first()
            Return the first element.
            EXAMPLES:

            sage: cartesian_product([ZZ]*10).first()
            (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

            sage: cartesian_product([ZZ]*Integer(10)).first()
            (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

    class ElementMethods
        Bases: object
        rank()
            Return the rank of self in its parent.
            See also EnumeratedSets.ElementMethods.rank()
            EXAMPLES:

            sage: F = FiniteSemigroups().example(('a','b','c'))
            sage: L = list(F)
            sage: L[7].rank()
            7
            sage: all(x.rank() == i for i,x in enumerate(L))
            True

            >>> from sage.all import *
            >>> F = FiniteSemigroups().example(('a','b','c'))
            >>> L = list(F)
            >>> L[Integer(7)].rank()
            7
            >>> all(x.rank() == i for i,x in enumerate(L))
            True

Finite
    alias of FiniteEnumeratedSets

Infinite
    alias of InfiniteEnumeratedSets

4.37. Enumerated sets
class ParentMethods
    Bases: object

    first()
    The "first" element of self.

    self.first() returns the first element of the set self. This is a generic implementation from the
category EnumeratedSets() which can be used when the method __iter__ is provided.

    EXAMPLES:

    sage: C = FiniteEnumeratedSets().example()
sage: C.first() # indirect doctest
    1

    is_empty()
    Return whether this set is empty.

    EXAMPLES:

    sage: F = FiniteEnumeratedSet([1,2,3])
sage: F.is_empty()
    False
    sage: F = FiniteEnumeratedSet([])
sage: F.is_empty()
    True

    >>> from sage.all import *
    >>> F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])
    >>> F.is_empty()
    False
    >>> F = FiniteEnumeratedSet([])
    >>> F.is_empty()
    True

    iterator_range(start=None, stop=None, step=None)
    Iterate over the range of elements of self starting at start, ending at stop, and stepping by step.

    See also:
    unrank(), unrank_range()

    EXAMPLES:

    sage: # needs sage.combinat
    sage: P = Partitions()
sage: list(P.iterator_range(stop=5))
    [[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(0, 5))
    [[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(3, 5))
    [[1, 1], [3]]
sage: list(P.iterator_range(3, 10))

    (continues on next page)
list(P.iterator_range(3, 10, 2))
[[1, 1], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
sage: it = P.iterator_range(3)
sage: [next(it) for x in range(10)]
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1], [5]]
sage: it = P.iterator_range(3, step=2)
sage: [next(it) for x in range(5)]
[[1, 1], [2, 1], [4], [2, 2], [1, 1, 1, 1]]
sage: next(P.iterator_range(stop=-3))
Traceback (most recent call last):
... Not ImplementedError: cannot list an infinite set
sage: next(P.iterator_range(start=-3))
Traceback (most recent call last):
... Not ImplementedError: cannot list an infinite set

>>> from sage.all import *
>>> # needs sage.combinat
>>> P = Partitions()

list(P.iterator_range(stop=Integer(5)))
[[], [1], [2], [1, 1], [3]]
list(P.iterator_range(Integer(0), Integer(5)))
[[], [1], [2], [1, 1], [3]]
list(P.iterator_range(Integer(3), Integer(5)))
[[1, 1], [3]]
list(P.iterator_range(Integer(3), Integer(10)))
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
list(P.iterator_range(Integer(3), Integer(10), Integer(2)))
[[1, 1], [2, 1], [4], [2, 2]]

>>> it = P.iterator_range(Integer(3))

list(P.iterator_range(start=Integer(5)))
[[1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1], [5]]
list(P.iterator_range(start=Integer(3), step=Integer(2)))
[[1, 1], [2, 1], [4], [2, 2], [1, 1, 1, 1]]

list(P.iterator_range(start=-Integer(3)))
Traceback (most recent call last):
... Not ImplementedError: cannot list an infinite set
list(P.iterator_range(start=-Integer(3)))
Traceback (most recent call last):
... Not ImplementedError: cannot list an infinite set

list()
Return a list of the elements of self.

The elements of set \( x \) are created and cached on the first call of \( x\.list() \). Then each call of \( x\.list() \) returns a new list from the cached result. Thus in looping, it may be better to do for e in \( x \); not for e in \( x\.list() \):

If \( x \) is not known to be finite, then an exception is raised.

EXAMPLES:

```
sage: (GF(3)^2).list() # needs sage.modules
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
sage: R = Integers(11)
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: 1 = R.list(); 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: 1.remove(0); 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: C = FiniteEnumeratedSets().example()
sage: C.list()
[1, 2, 3]
```

```
>>> from sage.all import *
>>> (GF(Integer(3))**Integer(2)).list() # needs sage.modules
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
>>> R = Integers(Integer(11))
>>> R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> 1 = R.list(); 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> 1.remove(Integer(0)); 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> C = FiniteEnumeratedSets().example()
>>> C.list()
[1, 2, 3]
```

map \((f, name, is\textunderscore injective=\text{None})\)

Return the image \( \{ f(x) | x \in \text{self} \} \) of this enumerated set by \( f \), as an enumerated set.

INPUT:

• is\_injective – boolean (default: True) whether to assume that \( f \) is injective.

EXAMPLES:

```
sage: R = Compositions(4).map(attrcall('partial_sums')); R
Image of Compositions of 4 by The map *\.partial_sums()
from Compositions of 4
sage: R.cardinality()
8
sage: R.list()
```

(continues on next page)
next (obj)

The “next” element after obj in self.

self.next(e) returns the element of the set self which follows e. This is a generic implementation from the category EnumeratedSets() which can be used when the method __iter__ is provided.

Remark: this is the default (brute force) implementation of the category EnumeratedSets(). Its
complexity is \( O(r) \), where \( r \) is the rank of \( \text{obj} \).

EXAMPLES:

```python
sage: C = InfiniteEnumeratedSets().example()
sage: C._next_from_iterator(10) # indirect doctest
11
```

```python
>>> from sage.all import *
>>> C = InfiniteEnumeratedSets().example()
>>> C._next_from_iterator(Integer(10)) # indirect doctest
11
```

TODO: specify the behavior when \( \text{obj} \) is not in \( \text{self} \).

**random_element()**

Return a random element in \( \text{self} \).

Unless otherwise stated, and for finite enumerated sets, the probability is uniform.

This is a generic implementation from the category \( \text{EnumeratedSets()} \). It raises a \( \text{NotImplementedError} \) since one does not know whether the set is finite.

EXAMPLES:

```python
sage: class broken(UniqueRepresentation, Parent):
    ....:     def __init__(self):
    ....:         Parent.__init__(self, category = EnumeratedSets())
sage: broken().random_element()
Traceback (most recent call last):
... NotImplementedError: unknown cardinality
```

```python
>>> from sage.all import *
>>> class broken(UniqueRepresentation, Parent):
...     def __init__(self):
...         Parent.__init__(self, category = EnumeratedSets())
>>> broken().random_element()
Traceback (most recent call last):
... NotImplementedError: unknown cardinality
```

**rank(\(x\))**

The rank of an element of \( \text{self} \)

\( \text{self.rank}(x) \) returns the rank of \( x \), that is its position in the enumeration of \( \text{self} \). This is an integer between 0 and \( n-1 \) where \( n \) is the cardinality of \( \text{self} \), or None if \( x \) is not in \( \text{self} \).

This is the default (brute force) implementation from the category \( \text{EnumeratedSets()} \) which can be used when the method __iter__ is provided. Its complexity is \( O(r) \), where \( r \) is the rank of \( \text{obj} \). For infinite enumerated sets, this won’t terminate when \( x \) is not in \( \text{self} \).

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: list(C)
[1, 2, 3]
sage: C.rank(3) # indirect doctest
2
sage: C.rank(5) # indirect doctest
```

```python
sage: C = FiniteEnumeratedSets().example()
sage: list(C)
[1, 2, 3]
sage: C.rank(3) # indirect doctest
2
sage: C.rank(5) # indirect doctest
```
some_elements()

Return some elements in self.

See TestSuite for a typical use case.

This is a generic implementation from the category EnumeratedSets() which can be used when the method __iter__ is provided. It returns an iterator for up to the first 100 elements of self.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: list(C.some_elements()) # indirect doctest
[1, 2, 3]
```

tuple()

Return a tuple of the elements of self.

The tuple of elements of x is created and cached on the first call of x.tuple(). Each following call of x.tuple() returns the same tuple.

For looping, it may be better to do for e in x:, not for e in x.tuple():.

If x is not known to be finite, then an exception is raised.

EXAMPLES:

```python
sage: (GF(3)^2).tuple()  # needs sage.modules
((0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2))
sage: R = Integers(11)
sage: l = R.tuple(); l
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sage: l is R.tuple()  
True
```

```python
>>> from sage.all import *

>>> (GF=Integer(3))^Integer(2).tuple()  # needs sage.modules
((0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2))

>>> R = Integers(Integer(11))

>>> l = R.tuple(); l
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

>>> l is R.tuple() 
True
```
\textbf{unrank} \((r)\)

The \(r\)-th element of \texttt{self}

\texttt{self.unrank(r)} returns the \(r\)-th element of \texttt{self}, where \(r\) is an integer between 0 and \(n-1\) where \(n\) is the cardinality of \texttt{self}.

This is the default (brute force) implementation from the category \texttt{EnumeratedSets()} which can be used when the method \texttt{__iter__} is provided. Its complexity is \(O(r)\), where \(r\) is the rank of \texttt{obj}.

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: C = FiniteEnumeratedSets().example()
  sage: C.unrank(2) # indirect doctest
  3
  sage: C._unrank_from_iterator(5)
  Traceback (most recent call last):
    ... ValueError: the rank must be in the range from 0 to 2
  sage: ZZ._unrank_from_iterator(-1)
  Traceback (most recent call last):
    ... ValueError: the rank must be greater than or equal to 0

  >>> from sage.all import *
  >>> C = FiniteEnumeratedSets().example()
  >>> C.unrank(Integer(2)) # indirect doctest
  3
  >>> C._unrank_from_iterator(Integer(5))
  Traceback (most recent call last):
    ... ValueError: the rank must be in the range from 0 to 2
  >>> ZZ._unrank_from_iterator(-Integer(1))
  Traceback (most recent call last):
    ... ValueError: the rank must be greater than or equal to 0
\end{verbatim}

\textbf{unrank_range} \((start=None, stop=None, step=None)\)

Return the range of elements of \texttt{self} starting at \texttt{start}, ending at \texttt{stop}, and stepping by \texttt{step}.

\textbf{See also:}

\texttt{unrank()}, \texttt{iterator_range()}

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: # needs sage.combinat
  sage: P = Partitions()
  sage: P.unrank_range(stop=5)
  [[], [1], [2], [1, 1], [3]]
  sage: P.unrank_range(0, 5)
  [[], [1], [2], [1, 1], [3]]
  sage: P.unrank_range(3, 5)
  [[1, 1], [3]]
  sage: P.unrank_range(3, 10)
  [[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
  sage: P.unrank_range(3, 10, 2)
  [[1, 1], [2, 1], [4], [2, 2]]
  sage: P.unrank_range(3)
  Traceback (most recent call last):
  ...

(continues on next page)
... Not ImplementedError: cannot list an infinite set

sage: P.unrank_range(stop=-3)
Traceback (most recent call last):
...
Not ImplementedError: cannot list an infinite set

sage: P.unrank_range(start=-3)
Traceback (most recent call last):
...
Not ImplementedError: cannot list an infinite set

>>> from sage.all import *
>>> # needs sage.combinat

>>> P = Partitions()

>>> P.unrank_range(stop=Integer(5))
[[], [1], [2], [1, 1], [3]]

>>> P.unrank_range(Integer(0), Integer(5))
[[], [1], [2], [1, 1], [3]]

>>> P.unrank_range(Integer(3), Integer(5))
[[1, 1], [3]]

>>> P.unrank_range(Integer(3), Integer(10))
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]

>>> P.unrank_range(Integer(3), Integer(10), Integer(2))
[[1, 1], [2, 1], [4], [2, 2]]

>>> P.unrank_range(Integer(3))
Traceback (most recent call last):
...
Not ImplementedError: cannot list an infinite set

>>> P.unrank_range(stop=-Integer(3))
Traceback (most recent call last):
...
Not ImplementedError: cannot list an infinite set

>>> P.unrank_range(start=-Integer(3))
Traceback (most recent call last):
...
Not ImplementedError: cannot list an infinite set

Additional structure

Return None.

Indeed, morphisms of enumerated sets are not required to preserve the enumeration.

See also:

Category.additional_structure()

Examples:

sage: EnumeratedSets().additional_structure()

super_categories()

Examples:

4.37. Enumerated sets 515
**4.38 Euclidean domains**

AUTHORS:

- Teresa Gomez-Diaz (2008): initial version
- Julian Rueth (2013-09-13): added euclidean degree, quotient remainder, and their tests

```python
class sage.categories.euclidean_domains.EuclideanDomains
    Bases: Category_singleton

The category of constructive euclidean domains, i.e., one can divide producing a quotient and a remainder where the remainder is either zero or its `ElementMethods.euclidean_degree()` is smaller than the divisor.

EXAMPLES:
```
```
```python
>>> from sage.all import *
```
```
```
```python
class ElementMethods
    Bases: object

    `euclidean_degree()`

    Return the degree of this element as an element of an Euclidean domain, i.e., for elements $a$, $b$ the euclidean degree $f$ satisfies the usual properties:
    1. if $b$ is not zero, then there are elements $q$ and $r$ such that $a = bq + r$ with $r = 0$ or $f(r) < f(b)$
    2. if $a$, $b$ are not zero, then $f(a) \leq f(ab)$

Note: The name `euclidean_degree` was chosen because the euclidean function has different names in different contexts, e.g., absolute value for integers, degree for polynomials.

OUTPUT:

For non-zero elements, a natural number. For the zero element, this might raise an exception or produce some other output, depending on the implementation.

EXAMPLES:
```python
sage: R.<x> = QQ[]
sage: x.euclidean_degree()
1
sage: ZZ.one().euclidean_degree()
1
```

```python
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> x.euclidean_degree()
1
>>> ZZ.one().euclidean_degree()
1
```

### `gcd` *(other)*

Return the greatest common divisor of this element and `other`.

**INPUT:**

- `other` – an element in the same ring as `self`

**ALGORITHM:**

Algorithm 3.2.1 in [Coh1993].

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ, sparse=True)
sage: EuclideanDomains().element_class.gcd(x, x+1)
-1
```

```python
>>> from sage.all import *
>>> R = PolynomialRing(QQ, sparse=True, names=('x',)); (x,) = R._first_ngens(1)
>>> EuclideanDomains().element_class.gcd(x, x+Integer(1))
-1
```

### `quo_rem` *(other)*

Return the quotient and remainder of the division of this element by the non-zero element `other`.

**INPUT:**

- `other` – an element in the same euclidean domain

**OUTPUT:**

a pair of elements

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: x.quo_rem(x)
(1, 0)
```

```python
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> x.quo_rem(x)
(1, 0)
```

### class `ParentMethods`

**Bases:** `object`

---

4.38. Euclidean domains
gcd_free_basis (elts)
Compute a set of coprime elements that can be used to express the elements of elts.

INPUT:
• elts – A sequence of elements of self.

OUTPUT:
A GCD-free basis (also called a coprime base) of elts; that is, a set of pairwise relatively prime elements of self such that any element of elts can be written as a product of elements of the set.

ALGORITHM:
Naive implementation of the algorithm described in Section 4.8 of Bach & Shallit [BS1996].

EXAMPLES:

```
sage: ZZ.gcd_free_basis([1])
[]
sage: ZZ.gcd_free_basis([4, 30, 14, 49])
[2, 15, 7]
sage: Pol.<x> = QQ[]
sage: sorted(Pol.gcd_free_basis([
    (x+1)^3*(x+2)^3*(x+3),
    (x+1)*(x+2)*(x+3),
    (x+1)*(x+2)*(x+4)]))
[x + 3, x + 4, x^2 + 3*x + 2]
```

is_euclidean_domain()
Return True, since this in an object of the category of Euclidean domains.

EXAMPLES:

```
sage: Parent(QQ,category=EuclideanDomains()).is_euclidean_domain()
True
```

super_categories()
EXAMPLES:

```
sage: EuclideanDomains().super_categories()
[Category of principal ideal domains]
```
>>> from sage.all import *
>>> EuclideanDomains().super_categories()
[Category of principal ideal domains]

4.39 Fields

class sage.categories.fields.Fields(base_category)

Bases: CategoryWithAxiom_singleton

The category of (commutative) fields, i.e. commutative rings where all non-zero elements have multiplicative inverses

EXAMPLES:

```
sage: K = Fields()
sage: K
Category of fields
sage: Fields().super_categories()
[Category of euclidean domains,
 Category of division rings,
 Category of noetherian rings]
sage: K(IntegerRing())
Rational Field
sage: K(PolynomialRing(GF(3), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Finite Field of size 3
sage: K(RealField())
# needs sage.rings.real_mpfr
Real Field with 53 bits of precision
```

class ElementMethods

Bases: object

euclidean_degree()

Return the degree of this element as an element of an Euclidean domain.
In a field, this returns 0 for all but the zero element (for which it is undefined).

**EXAMPLES:**

```python
sage: QQ.one().euclidean_degree()
0
```

```python
>>> from sage.all import *

>>> QQ.one().euclidean_degree()
0
```

**factor()**

Return a factorization of `self`.

Since `self` is either a unit or zero, this function is trivial.

**EXAMPLES:**

```python
sage: x = GF(7)(5)
sage: x.factor()
5
```

```python
sage: RR(0).factor()  # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
ArithmeticError: factorization of 0.000000000000000 is not defined
```

```python
>>> from sage.all import *

>>> x = GF(Integer(7))(Integer(5))

>>> x.factor()
5
```

```python
>>> RR(Integer(0)).factor()  # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
ArithmeticError: factorization of 0.000000000000000 is not defined
```

**gcd(other)**

Greatest common divisor.

**Note:** Since we are in a field and the greatest common divisor is only determined up to a unit, it is correct to either return zero or one. Note that fraction fields of unique factorization domains provide a more sophisticated `gcd`.

**EXAMPLES:**

```python
sage: K = GF(5)
sage: K(2).gcd(K(1))
1
```

```python
sage: K(0).gcd(K(0))
0
```

```python
sage: all(x.gcd(y) == (0 if x == 0 and y == 0 else 1)
.....  for x in K for y in K)
True
```
For field of characteristic zero, the gcd of integers is considered as if they were elements of the integer ring:

```
sage: gcd(15.0, 12.0)  # needs sage.rings.real_mpfr
3.00000000000000
```

But for other floating point numbers, the gcd is just 0.0 or 1.0:

```
sage: gcd(3.2, 2.18)  # needs sage.rings.real_mpfr
1.00000000000000

sage: gcd(0.0, 0.0)  # needs sage.rings.real_mpfr
0.000000000000000
```

AUTHOR:
- Simon King (2011-02) – Issue #10771
- Vincent Delecroix (2015) – Issue #17671

`inverse_of_unit()`
Return the inverse of this element.

`EXAMPLES:`

```
sage: x = polygen(ZZ, 'x')
sage: NumberField(x^7 + 2, 'a')(2).inverse_of_unit()  # needs sage.rings.number_field
1/2
```
Trying to invert the zero element typically raises a `ZeroDivisionError`:

```python
>>> from sage.all import *
>>> QQ(0).inverse_of_unit()
Traceback (most recent call last):
  ... ZeroDivisionError: rational division by zero
```

To catch that exception in a way that also works for non-units in more general rings, use something like:

```python
sage: try:
    QQ(0).inverse_of_unit()
except ArithmeticError:
    pass
```

Also note that some “fields” allow one to invert the zero element:

```python
sage: RR(0).inverse_of_unit()
+inf
```

```
>>> from sage.all import *
>>> RR(Integer(0)).inverse_of_unit()
+infinity
```

`is_unit()`

Returns True if self has a multiplicative inverse.

**EXAMPLES:**

```python
sage: QQ(2).is_unit()
True
sage: QQ(0).is_unit()
False
```

```python
>>> from sage.all import *
>>> QQ(Integer(2)).is_unit()
True
>>> QQ(Integer(0)).is_unit()
False
```
**lcm** *(other)*

Least common multiple.

**Note:** Since we are in a field and the least common multiple is only determined up to a unit, it is correct to either return zero or one. Note that fraction fields of unique factorization domains provide a more sophisticated lcm.

**EXAMPLES:**

```python
sage: GF(2)(1).lcm(GF(2)(0))
0
sage: GF(2)(1).lcm(GF(2)(1))
1

>>> from sage.all import *

>>> GF(Integer(2))(Integer(1)).lcm(GF(Integer(2))(Integer(0)))
0
>>> GF(Integer(2))(Integer(1)).lcm(GF(Integer(2))(Integer(1)))
1
```

For field of characteristic zero, the lcm of integers is considered as if they were elements of the integer ring:

```python
sage: lcm(15.0, 12.0)  # needs sage.rings.real_mpfr
60.0

>>> from sage.all import *

>>> lcm(RealNumber(15.0), RealNumber('12.0'))  # needs sage.rings.real_mpfr
60.0
```

But for others floating point numbers, it is just 0.0 or 1.0:

```python
sage: lcm(3.2, 2.18)  # needs sage.rings.real_mpfr
1.0
sage: lcm(0.0, 0.0)  # needs sage.rings.real_mpfr
0.0

>>> from sage.all import *

>>> lcm(RealNumber('3.2'), RealNumber('2.18'))  # needs sage.rings.real_mpfr
1.0
>>> lcm(RealNumber('0.0'), RealNumber('0.0'))  # needs sage.rings.real_mpfr
0.0
```

**AUTHOR:**
- Simon King (2011-02) – Issue #10771
- Vincent Delecroix (2015) – Issue #17671
\texttt{quo\_rem}(\texttt{other})

Return the quotient with remainder of the division of this element by \texttt{other}.

\textbf{INPUT:}
- \texttt{other} – an element of the field

\textbf{EXAMPLES:}

\begin{verbatim}
sage: f, g = QQ(1), QQ(2)
sage: f.quo_rem(g)
(1/2, 0)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> f, g = QQ(Integer(1)), QQ(Integer(2))
>>> f.quo_rem(g)
(1/2, 0)
\end{verbatim}

\texttt{xgcd}(\texttt{other})

Compute the extended \texttt{gcd} of \texttt{self} and \texttt{other}.

\textbf{INPUT:}
- \texttt{other} – an element with the same parent as \texttt{self}

\textbf{OUTPUT:}
A tuple \((r, s, t)\) of elements in the parent of \texttt{self} such that \(r = s \cdot \texttt{self} + t \cdot \texttt{other}\). Since the computations are done over a field, \(r\) is zero if \texttt{self} and \texttt{other} are zero, and one otherwise.

\textbf{AUTHORS:}
- Julian Rueth (2012-10-19): moved here from \texttt{sage.structure.element.FieldElement}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: K = GF(5)
sage: K(2).xgcd(K(1))
(1, 3, 0)
sage: K(0).xgcd(K(4))
(1, 0, 4)
sage: K(1).xgcd(K(1))
(1, 1, 0)
sage: GF(5)(0).xgcd(GF(5)(0))
(0, 0, 0)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> K = GF(Integer(5))
>>> K(Integer(2)).xgcd(K(Integer(1)))
(1, 3, 0)
>>> K(Integer(0)).xgcd(K(Integer(4)))
(1, 0, 4)
>>> K(Integer(1)).xgcd(K(Integer(1)))
(1, 1, 0)
>>> GF(Integer(5))(Integer(0)).xgcd(GF(Integer(5))(Integer(0)))
(0, 0, 0)
\end{verbatim}

The \texttt{xgcd} of non-zero floating point numbers will be a triple of floating points. But if the input are two integral floating points the result is a floating point version of the standard \texttt{gcd} on \(\mathbb{Z}\):

\begin{verbatim}
sage: xgcd(12.0, 8.0) #˓→ needs sage.rings.real_mpfr
(4.00000000000000, 1.00000000000000, -1.00000000000000)
\end{verbatim}

(continues on next page)
sage: xgcd(3.1, 2.98714)  # needs sage.rings.real_mpfr
(1.00000000000000, 0.322580645161290, 0.000000000000000)
sage: xgcd(0.0, 1.1)  # needs sage.rings.real_mpfr
(1.00000000000000, 0.000000000000000, 0.909090909090909)

>>> from sage.all import *
>>> xgcd(RealNumber('12.0'), RealNumber('8.0'))  # needs sage.rings.real_mpfr
(4.00000000000000, 1.00000000000000, -1.00000000000000)

>>> from sage.all import *
>>> xgcd(RealNumber('3.1'), RealNumber('2.98714'))  # needs sage.rings.real_mpfr
(1.00000000000000, 0.322580645161290, 0.000000000000000)

>>> from sage.all import *
>>> xgcd(RealNumber('0.0'), RealNumber('1.1'))  # needs sage.rings.real_mpfr
(1.00000000000000, 0.000000000000000, 0.909090909090909)

Finite
alias of FiniteFields
class ParentMethods
    Bases: object

    fraction_field()
    Returns the fraction field of self, which is self.

    EXAMPLES:

    sage: QQ.fraction_field() is QQ
    True

    >>> from sage.all import *
    >>> QQ.fraction_field() is QQ
    True

    is_field(proof=True)
    Returns True as self is a field.

    EXAMPLES:

    sage: QQ.is_field()
    True
    sage: Parent(QQ, category=Fields()).is_field()
    True

    >>> from sage.all import *
    >>> QQ.is_field()
    True
    >>> Parent(QQ, category=Fields()).is_field()
    True
is_integrally_closed()

Return True, as per IntegralDomain.is_integrally_closed(): for every field $F$, $F$ is its own field of fractions, hence every element of $F$ is integral over $F$.

EXAMPLES:

```python
sage: QQ.is_integrally_closed()
True
sage: QQbar.is_integrally_closed() # needs sage.rings.number_field
True
sage: Z5 = GF(5); Z5
Finite Field of size 5
sage: Z5.is_integrally_closed()
True
```

is_perfect()

Return whether this field is perfect, i.e., its characteristic is $p = 0$ or every element has a $p$-th root.

EXAMPLES:

```python
sage: QQ.is_perfect()
True
sage: GF(2).is_perfect()
True
sage: FunctionField(GF(2), 'x').is_perfect()
False
```

vector_space(*args, **kwds)

Gives an isomorphism of this field with a vector space over a subfield.

This method is an alias for free_module, which may have more documentation.

INPUT:

- base – a subfield or morphism into this field (defaults to the base field)
- basis – a basis of the field as a vector space over the subfield; if not given, one is chosen automatically
- map – whether to return maps from and to the vector space

OUTPUT:
• $V$ – a vector space over base
• from $V$ – an isomorphism from $V$ to this field
• to $V$ – the inverse isomorphism from this field to $V$

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: K.<a> = Qq(125)
sage: V, fr, to = K.vector_space()
sage: v = V([1, 2, 3])
sage: fr(v, 7)
(3*a^2 + 2*a + 1) + O(5^7)
```

```python
>>> from sage.all import *
>>> K = Qq(Integer(125), names=('a',)); (a,) = K._first_ngens(1)
>>> V, fr, to = K.vector_space()
>>> v = V([Integer(1), Integer(2), Integer(3)])
>>> fr(v, Integer(7))
(3*a^2 + 2*a + 1) + O(5^7)
```

extra_super_categories()

EXAMPLES:

```
sage: Fields().extra_super_categories()
[Category of euclidean domains, Category of noetherian rings]
```

```python
>>> from sage.all import *
>>> Fields().extra_super_categories()
[Category of euclidean domains, Category of noetherian rings]
```

### 4.40 Filtered Algebras

**class** `sage.categories.filtered_algebras.FilteredAlgebras(base_category)`

**Bases:** `FilteredModulesCategory`

The category of filtered algebras.

An algebra $A$ over a commutative ring $R$ is **filtered** if $A$ is endowed with a structure of a filtered $R$-module (whose underlying $R$-module structure is identical with that of the $R$-algebra $A$) such that the indexing set $I$ (typically $I = \mathbb{N}$) is also an additive abelian monoid, the unity 1 of $A$ belongs to $F_0$, and we have $F_i \cdot F_j \subseteq F_{i+j}$ for all $i, j \in I$.

**EXAMPLES:**

```
sage: Algebras(ZZ).Filtered()
Category of filtered algebras over Integer Ring
sage: Algebras(ZZ).Filtered().super_categories()
[Category of algebras over Integer Ring,
 Category of filtered modules over Integer Ring]
```

```python
>>> from sage.all import *
>>> Algebras(ZZ).Filtered()
Category of filtered algebras over Integer Ring
>>> Algebras(ZZ).Filtered().super_categories()
```

(continues on next page)
REFERENCES:

- Wikipedia article Filtered_algebra

```python
class ParentMethods
    Bases: object

    graded_algebra()

    Return the associated graded algebra to self.

Todo: Implement a version of the associated graded algebra which does not require self to have a distinguished basis.
```

EXAMPLES:

```python
c sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
c sage: A.graded_algebra()
Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring
```

4.41 Filtered Algebras With Basis

A filtered algebra with basis over a commutative ring \( R \) is a filtered algebra over \( R \) endowed with the structure of a filtered module with basis (with the same underlying filtered-module structure). See FilteredAlgebras and FilteredModulesWithBasis for these two notions.

```python
class sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis(base_category)
```

Bases: FilteredModulesCategory

The category of filtered algebras with a distinguished homogeneous basis.

A filtered algebra with basis over a commutative ring \( R \) is a filtered algebra over \( R \) endowed with the structure of a filtered module with basis (with the same underlying filtered-module structure). See FilteredAlgebras and FilteredModulesWithBasis for these two notions.

EXAMPLES:

```python
c sage: C = AlgebrasWithBasis(ZZ).Filtered(); C
Category of filtered algebras with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
```
>>> from sage.all import *

Category of algebras with basis over Integer Ring,
Category of filtered algebras over Integer Ring,
Category of filtered modules with basis over Integer Ring

C = AlgebrasWithBasis(ZZ).Filtered(); C
Category of filtered algebras with basis over Integer Ring

sorted(C.super_categories(), key=str)

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

    from_graded_conversion()
    Return the inverse of the canonical $R$-module isomorphism $A \to \text{gr } A$ induced by the basis of $A$ (where $A = \text{``self''}$). This inverse is an isomorphism $\text{gr } A \to A$.

    This is an isomorphism of $R$-modules, not of algebras. See the class documentation AssociatedGradedAlgebra.

    See also:
    to_graded_conversion()

    EXAMPLES:

    sage: A = Algebras(QQ).WithBasis().Filtered().example()
    sage: p = A.an_element() + A.algebra_generators()['x'] + 2; p
    U['x']^2*U['y']^2*U['z']^3 + 3*U['x'] + 3*U['y'] + 3
    sage: q = A.to_graded_conversion()(p)
    sage: A.from_graded_conversion()(q) == p
    True
    sage: q.parent() is A.graded_algebra()
    True

graded_algebra()
    Return the associated graded algebra to self.

    See AssociatedGradedAlgebra for the definition and the properties of this.

    If the filtered algebra self with basis is called $A$, then this method returns $\text{gr } A$. The method to_graded_conversion() returns the canonical $R$-module isomorphism $A \to \text{gr } A$ induced by
the basis of $A$, and the method `from_graded_conversion()` returns the inverse of this isomorphism. The method `projection()` projects elements of $A$ onto $\text{gr}A$ according to their place in the filtration on $A$.

**Warning:** When not overridden, this method returns the default implementation of an associated graded algebra – namely, `AssociatedGradedAlgebra(self)`, where `AssociatedGradedAlgebra` is `AssociatedGradedAlgebra`. But many instances of `FilteredAlgebrasWithBasis` override this method, as the associated graded algebra often is (isomorphic) to a simpler object (for instance, the associated graded algebra of a graded algebra can be identified with the graded algebra itself). Generic code that uses associated graded algebras (such as the code of the `induced_graded_map()` method below) should make sure to only communicate with them via the `to_graded_conversion()`, `from_graded_conversion()`, and `projection()` methods (in particular, do not expect there to be a conversion from `self` to `self.graded_algebra()`; this currently does not work for Clifford algebras). Similarly, when overriding `graded_algebra()`, make sure to accordingly redefine these three methods, unless their definitions below still apply to your case (this will happen whenever the basis of your `graded_algebra()` has the same indexing set as `self`, and the partition of this indexing set according to degree is the same as for `self`).

**Todo:** Maybe the thing about the conversion from `self` to `self.graded_algebra()` on the Clifford at least could be made to work? (I would still warn the user against ASSUMING that it must work – as there is probably no way to guarantee it in all cases, and we shouldn’t require users to mess with element constructors.)

**EXAMPLES:**

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.graded_algebra()
Graded Algebra of An example of a filtered algebra with basis:
the universal enveloping algebra of
Lie algebra of $\mathbb{R}^3$ with cross product over Integer Ring
```

```python
>>> from sage.all import *
>>> A = AlgebrasWithBasis(ZZ).Filtered().example()
>>> A.graded_algebra()
Graded Algebra of An example of a filtered algebra with basis:
the universal enveloping algebra of
Lie algebra of $\mathbb{R}^3$ with cross product over Integer Ring
```

`induced_graded_map(other, f)`

Return the graded linear map between the associated graded algebras of `self` and `other` canonically induced by the filtration-preserving map $f : \text{self} \to \text{other}$. Let $A$ and $B$ be two filtered algebras with basis, and let $(F_i)_{i \in I}$ and $(G_i)_{i \in I}$ be their filtrations. Let $f : A \to B$ be a linear map which preserves the filtration (i.e., satisfies $f(F_i) \subseteq G_i$ for all $i \in I$). Then, there is a canonically defined graded linear map $\text{gr}f : \text{gr}A \to \text{gr}B$ which satisfies

$$(\text{gr}f)(p_i(a)) = p_i(f(a)) \quad \text{for all } i \in I \text{ and } a \in F_i,$$

where the $p_i$ on the left hand side is the canonical projection from $F_i$ onto the $i$-th graded component of $\text{gr}A$, while the $p_i$ on the right hand side is the canonical projection from $G_i$ onto the $i$-th graded component of $\text{gr}B$. 

**INPUT:**
• other – a filtered algebra with basis
• f – a filtration-preserving linear map from self to other (can be given as a morphism or as a function)
OUTPUT:
The graded linear map $gr f$.
EXAMPLES:
Example 1.
We start with the universal enveloping algebra of the Lie algebra $\mathbb{R}^3$ (with the cross product serving as Lie bracket):

```python
sage: A = AlgebrasWithBasis(QQ).Filtered().example(); A
An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Rational Field
sage: M = A.indices(); M
Free abelian monoid indexed by {'x', 'y', 'z'}
sage: x, y, z = [A.basis()[M.gens()[i]] for i in "xyz"
```

Let us define a stupid filtered map from $A$ to itself:

```python
sage: def map_on_basis(m):
....:     d = m.dict()
....:     i = d.get('x', 0); j = d.get('y', 0); k = d.get('z', 0)
....:     g = (y ** (i+j)) * (z ** k)
....:     if i > 0:
....:         g += i * (x ** (i-1)) * (y ** j) * (z ** k)
....:     return g
sage: f = A.module_morphism(on_basis=map_on_basis, codomain=A)
```

```python
sage: f(x)
U[y] + 1
sage: f(x*y*z)
U[y]^2*U[z] + U[y]*U[z]
```

```python
>>> from sage.all import *
```
g = (y ** (i+j)) * (z ** k)
if i > Integer(0):
    g += i * (x ** (i-Integer(1))) * (y ** j) * (z ** k)
return g

```python
f = A.module_morphism(on_basis=map_on_basis,
codomain=A)

f(x)
U[y] + 1
f(x*y*z)
U[y]^2*U[z] + U[y]*U[z]
f(x*x*y*z)
U[y]^3*U[z] + 2*U[x]*U[y]*U[z]
f(A.one())
1
f(y*z)
U[y]*U[z]
```

(There is nothing here that is peculiar to this universal enveloping algebra; we are only using its module structure, and we could just as well be using a polynomial algebra in its stead.)

We now compute \( gr f \)

```python
sage: grA = A.graded_algebra(); grA
Graded Algebra of An example of a filtered algebra with
basis: the universal enveloping algebra of Lie algebra
of \( \mathbb{R}^3 \) with cross product over Rational Field
sage: xx, yy, zz = [A.to_graded_conversion()(i) for i in [x, y, z]]
sage: xx+yy*zz
bar(U[y]*U[z]) + bar(U[x])
sage: grf = A.induced_graded_map(A, f); grf
Generic endomorphism of Graded Algebra of An example
of a filtered algebra with basis: the universal
enveloping algebra of Lie algebra of \( \mathbb{R}^3 \) with cross
product over Rational Field
sage: grf(xx)
bar(U[y])
sage: grf(xx*yy*zz)
bar(U[y]^2*U[z])
sage: grf(xx*xx*yy*zz)
bar(U[y]^3*U[z])
sage: grf(grA.one())
1
sage: grf(yy*zz)
bar(U[y]*U[z])
sage: grf(yy*zz-2*yy)
bar(U[y]*U[z]) - 2*bar(U[y])
```
of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of $\mathbb{R}^3$ with cross product over Rational Field

```python
>>>
```

Example 2.

We shall now construct $g$ for a different map $f$ out of the same $A$; the new map $f$ will lead into a graded algebra already, namely into the algebra of symmetric functions:

```python
sage: # needs sage.combinat sage.modules
sage: h = SymmetricFunctions(QQ).h()

sage: def map_on_basis(m):
    # redefining map_on_basis
    d = m.dict()
    i = d.get('x', 0); j = d.get('y', 0); k = d.get('z', 0)
    g += i * (h[1] ** (i+j+k))
    return g

sage: f = A.module_morphism(on_basis=map_on_basis,
    ...:     codomain=h)  # redefining f

sage: f(x)
2*h[1]
sage: f(y)
h[]
sage: f(z)
h[]
sage: f(y**2)
h[2]
sage: f(x**2)
3*h[1, 1]
sage: f(x*y*z)
h[1] + h[1, 1, 1]
sage: f(x*x*y*y*z)
2*h[1, 1, 1, 1, 1] + h[2, 1, 1]
sage: f(A.one())
h[]
```

```python
>>>
```

(continues on next page)
The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:

```python
sage: # needs sage.combinat sage.modules
sage: grh = h.graded_algebra(); grh is h
True
sage: grf = A.induced_graded_map(h, f); grf
Generic morphism:
  From: Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of $\mathbb{R}^3$ with cross product over Rational Field
  To: Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(xx)
2*h[1]
sage: grf(yy)
0
sage: grf(zz)
0
sage: grf(yy**2)
h[2]
sage: grf(xx**2)
3*h[1, 1]
sage: grf(xx*yy*zz)
h[1, 1, 1]
sage: grf(xx*xx*yy*yy*zz)
2*h[1, 1, 1, 1, 1]
sage: grf(grA.one())
h[]
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> grh = h.graded_algebra(); grh is h
True
(continues on next page)
```
Example 3.

After having had a graded algebra as the codomain, let us try to have one as the domain instead. Our new $f$ will go from $h$ to $A$:

```python
sage: # needs sage.combinat sage.modules
sage: def map_on_basis(lam):
    # redefining map_on_basis
    ....:     return x ** (sum(lam)) + y ** (len(lam))
sage: f = h.module_morphism(on_basis=map_on_basis,
    ....:     codomain=A)  # redefining f
sage: f(h[1])
U['x'] + U['y']
sage: f(h[2])
U['x']^2 + U['y']
sage: f(h[1, 1])
U['x']^2 + U['y']^2
sage: f(h[2, 2])
U['x']^4 + U['y']^2
sage: f(h[3, 2, 1])
U['x']^6 + U['y']^3
sage: f(h.one())
2
sage: grf = h.induced_graded_map(A, f); grf
Generic morphism:
  From: Symmetric Functions over Rational Field in the homogeneous basis
  To:   Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Rational Field
sage: grf(h[1])
bar(U['x']) + bar(U['y'])
```

(continues on next page)
Example 4.

The construct $\text{grf}$ also makes sense when $f$ is a filtration-preserving map between graded algebras.

```
sage: # needs sage.combinat sage.modules
sage: def map_on_basis(lam):  # redefining map_on_basis
... return h[lam] + h[len(lam)]
```
Example 5.
For another example, let us compute $\text{gr} \, f$ for a map $f$ between two Clifford algebras:

```python
sage: # needs sage.modules
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: B = CliffordAlgebra(Q, names=['u','v']); B
The Clifford algebra of the Quadratic form in 2 variables over Integer Ring with coefficients:
[ 1 2 ]
[ * 3 ]
sage: m = Matrix(ZZ, [[1, 2], [1, -1]])
sage: f = B.lift_module_morphism(m, names=['x','y'])
sage: A = f.domain(); A
The Clifford algebra of the Quadratic form in 2 variables over Integer Ring with coefficients:
[ 6 0 ]
[ * 3 ]
sage: x, y = A.gens()
sage: f(x)
u + v
sage: f(y)
2*u - v
sage: f(x**2)
6
sage: f(x*y)
-3*u*v + 3
sage: grA = A.graded_algebra(); grA
The exterior algebra of rank 2 over Integer Ring
sage: A.to_graded_conversion()(x)
x
sage: A.to_graded_conversion()(y)
y
sage: A.to_graded_conversion()(x*y)
x*y
sage: u = A.to_graded_conversion()(x*y+1); u
x*y + 1
sage: A.from_graded_conversion()(u)
x*y + 1
sage: A.projection(2)(x**y+1)
x*y
sage: A.projection(1)(x+2*y-2)
x + 2*y
sage: grf = A.induced_graded_map(B, f); grf
Generic morphism:
    From: The exterior algebra of rank 2 over Integer Ring
    To:  The exterior algebra of rank 2 over Integer Ring
sage: grf(A.to_graded_conversion()(x))
u + v
sage: grf(A.to_graded_conversion()(y))
2*u - v
sage: grf(A.to_graded_conversion()(x**2))
6
sage: grf(A.to_graded_conversion()(x*y))
-3*u*v
sage: grf(grA.one())
1
```

>>> from sage.all import *
>>> # needs sage.modules
(continues on next page)
Q = QuadraticForm(ZZ, Integer(2), [Integer(1), Integer(2), Integer(3)])
B = CliffordAlgebra(Q, names=[u', v'])

The Clifford algebra of the Quadratic form in 2
variables over Integer Ring with coefficients:
[ 1 2 ]
[ * 3 ]
m = Matrix(ZZ, [[Integer(1), Integer(2)], [Integer(1), -Integer(1)]]))
f = B.lift_module_morphism(m, names=['x', 'y'])
A = f.domain(); A

The Clifford algebra of the Quadratic form in 2
variables over Integer Ring with coefficients:
[ 6 0 ]
[ * 3 ]
x, y = A.gens()
f(x)
u + v
f(y)
2*u - v
f(x**Integer(2))
6
f(x*y)
-3*u*v + 3
grA = A.graded_algebra(); grA
The exterior algebra of rank 2 over Integer Ring
A.to_graded_conversion()(x)
x
A.to_graded_conversion()(y)
y
A.to_graded_conversion()(x*y)
x*y
u = A.to_graded_conversion()(x*y+Integer(1)); u
x*y + 1
A.from_graded_conversion()(u)
x*y + 1
A.projection(Integer(2))(x*y+Integer(1))
x*y
A.projection(Integer(1))(x+Integer(2)*y-Integer(2))
x + 2*y
grf = A.induced_graded_map(B, f); grf

Generic morphism:
From: The exterior algebra of rank 2 over Integer Ring
To: The exterior algebra of rank 2 over Integer Ring
grf(A.to_graded_conversion()(x))
u + v
grf(A.to_graded_conversion()(y))
2*u - v
grf(A.to_graded_conversion()(x**Integer(2)))
6
grf(A.to_graded_conversion()(x*y))
-3*u*v
grf(grA.one())

projection(i)
Return the i-th projection p_i : F_i \to G_i (in the notations of the class documentation Associated-GradedAlgebra, where A = ""self")
This method actually does not return the map \( p_i \) itself, but an extension of \( p_i \) to the whole \( R \)-module \( A \). This extension is the composition of the \( R \)-module isomorphism \( A \to \text{gr}A \) with the canonical projection of the graded \( R \)-module \( \text{gr}A \) onto its \( i \)-th graded component \( G_i \). The codomain of this map is \( \text{gr}A \), although its actual image is \( G_i \). The map \( p_i \) is obtained from this map by restricting its domain to \( F_i \) and its image to \( G_i \).

**EXAMPLES:**

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: p = A.an_element() + A.algebra_generators()[x] + 2; p
U[x]^2*U[y]^2*U[z]^3 + 3*U[x] + 3*U[y] + 3
sage: q = A.projection(7)(p); q
bar(U[x]^2*U[y]^2*U[z]^3)
sage: q.parent() is A.graded_algebra()
True
sage: A.projection(8)(p)
0
```

**to_graded_conversion()**

Return the canonical \( R \)-module isomorphism \( A \to \text{gr}A \) induced by the basis of \( A \) (where \( A = \text{``self''} \)).

This is an isomorphism of \( R \)-modules, not of algebras. See the class documentation `AssociatedGradedAlgebra`.

See also:

`from_graded_conversion()`

**EXAMPLES:**

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: p = A.an_element() + A.algebra_generators()[x] + Integer(2); p
U[x]^2*U[y]^2*U[z]^3 + 3*U[x] + 3*U[y] + 3
sage: q = A.to_graded_conversion()(p); q
bar(U[x]^2*U[y]^2*U[z]^3) + 3*bar(U[x]) + 3*bar(U[y]) + 3*bar(1)
sage: q.parent() is A.graded_algebra()
True
```

```python
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> p = A.an_element() + A.algebra_generators()[x] + Integer(2); p
U[x]^2*U[y]^2*U[z]^3 + 3*U[x] + 3*U[y] + 3
>>> q = A.to_graded_conversion()(p); q
bar(U[x]^2*U[y]^2*U[z]^3) + 3*bar(U[x]) + 3*bar(U[y]) + 3*bar(1)
>>> q.parent() is A.graded_algebra()
True
```
4.42 Filtered Modules

A filtered module over a ring $R$ with a totally ordered indexing set $I$ (typically $I = \mathbb{N}$) is an $R$-module $M$ equipped with a family $(F_i)_{i \in I}$ of $R$-submodules satisfying $F_i \subseteq F_j$ for all $i, j \in I$ having $i \leq j$, and $M = \bigcup_{i \in I} F_i$. This family is called a filtration of the given module $M$.

**Todo:** Implement a notion for decreasing filtrations: where $F_j \subseteq F_i$ when $i \leq j$.

**Todo:** Implement filtrations for all concrete categories.

**Todo:** Implement gr as a functor.

```python
class sage.categories.filtered_modules.FilteredModules(base_category):
    Bases: FilteredModulesCategory
    The category of filtered modules over a given ring $R$.

    A filtered module over a ring $R$ with a totally ordered indexing set $I$ (typically $I = \mathbb{N}$) is an $R$-module $M$ equipped with a family $(F_i)_{i \in I}$ of $R$-submodules satisfying $F_i \subseteq F_j$ for all $i, j \in I$ having $i \leq j$, and $M = \bigcup_{i \in I} F_i$. This family is called a filtration of the given module $M$.

    EXAMPLES:

    sage: Modules(ZZ).Filtered()
    Category of filtered modules over Integer Ring
    sage: Modules(ZZ).Filtered().super_categories()
    [Category of modules over Integer Ring]

    >>> from sage.all import *
    >>> Modules(ZZ).Filtered()
    Category of filtered modules over Integer Ring
    >>> Modules(ZZ).Filtered().super_categories()
    [Category of modules over Integer Ring]
```

REFERENCES:

• Wikipedia article Filtration_(mathematics)

```python
class Connected(base_category):
    Bases: CategoryWithAxiom_over_base_ring
    class SubcategoryMethods
        Bases: object
        Connected()
        Return the full subcategory of the connected objects of self.

        A filtered $R$-module $M$ with filtration $(F_0, F_1, F_2, \ldots)$ (indexed by $\mathbb{N}$) is said to be connected if $F_0$ is isomorphic to $R$.

        EXAMPLES:
```
sage: Modules(ZZ).Filtered().Connected()
Category of filtered connected modules over Integer Ring
sage: Coalgebras(QQ).Filtered().Connected()
Category of filtered connected coalgebras over Rational Field
sage: AlgebrasWithBasis(QQ).Filtered().Connected()
Category of filtered connected algebras with basis over Rational Field

>>> from sage.all import *
>>> Modules(ZZ).Filtered().Connected()
Category of filtered connected modules over Integer Ring
>>> Coalgebras(QQ).Filtered().Connected()
Category of filtered connected coalgebras over Rational Field
>>> AlgebrasWithBasis(QQ).Filtered().Connected()
Category of filtered connected algebras with basis over Rational Field

extra_super_categories()
Add VectorSpaces to the super categories of self if the base ring is a field.

EXAMPLES:

sage: Modules(QQ).Filtered().is_subcategory(VectorSpaces(QQ))
True
sage: Modules(ZZ).Filtered().extra_super_categories()
[]

>>> from sage.all import *
>>> Modules(QQ).Filtered().is_subcategory(VectorSpaces(QQ))
True
>>> Modules(ZZ).Filtered().extra_super_categories()
[]

This makes sure that Modules(QQ).Filtered() returns an instance of FilteredModules and not a join category of an instance of this class and of VectorSpaces(QQ):

sage: type(Modules(QQ).Filtered())
<class 'sage.categories.vector_spaces.VectorSpaces.Filtered_with_category'>

>>> from sage.all import *
>>> type(Modules(QQ).Filtered())
<class 'sage.categories.vector_spaces.VectorSpaces.Filtered_with_category'>

Todo: Get rid of this workaround once there is a more systematic approach for the alias Modules(QQ) -> VectorSpaces(QQ). Probably the latter should be a category with axiom, and covariant constructions should play well with axioms.

class sage.categories.filtered_modules.FilteredModulesCategory(base_category)
    Bases: RegressiveCovariantConstructionCategory, Category_over_base_ring

EXAMPLES:

sage: C = Algebras(QQ).Filtered()
sage: C
Category of filtered algebras over Rational Field
sage: C.base_category()
Category of algebras over Rational Field

```python
sage: sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field,
 Category of filtered vector spaces over Rational Field]
```

```python
sage: AlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field

sage: HopfAlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field
```

```python
>>> from sage.all import *

>>> C = Algebras(QQ).Filtered()

>>> C
Category of filtered algebras over Rational Field

>>> C.base_category()
Category of algebras over Rational Field

>>> sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field,
 Category of filtered vector spaces over Rational Field]

>>> AlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field

>>> HopfAlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field
```

4.43 Filtered Modules With Basis

A filtered module with basis over a ring $R$ means (for the purpose of this code) a filtered $R$-module $M$ with filtration $(F_i)_{i \in I}$ (typically $I = \mathbb{N}$) endowed with a basis $(b_j)_{j \in I}$ of $M$ and a partition $J = \bigcup_{i \in I} J_i$ of the set $J$ (it is allowed that some $J_i$ are empty) such that for every $n \in I$, the subfamily $(b_j)_{j \in U_n}$, where $U_n = \bigcup_{i \leq n} J_i$, is a basis of the $R$-submodule $F_n$.

For every $i \in I$, the $R$-submodule of $M$ spanned by $(b_j)_{j \in J_i}$ is called the $i$-th graded component (aka the $i$-th homogeneous component) of the filtered module with basis $M$; the elements of this submodule are referred to as homogeneous elements of degree $i$.

See the class documentation `FilteredModulesWithBasis` for further details.

```python
class sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis(base_category)

Bases: FilteredModulesCategory
```

The category of filtered modules with a distinguished basis.

A filtered module with basis over a ring $R$ means (for the purpose of this code) a filtered $R$-module $M$ with filtration $(F_i)_{i \in I}$ (typically $I = \mathbb{N}$) endowed with a basis $(b_j)_{j \in I}$ of $M$ and a partition $J = \bigcup_{i \in I} J_i$ of the set $J$ (it is allowed that some $J_i$ are empty) such that for every $n \in I$, the subfamily $(b_j)_{j \in U_n}$, where $U_n = \bigcup_{i \leq n} J_i$, is a basis of the $R$-submodule $F_n$.

For every $i \in I$, the $R$-submodule of $M$ spanned by $(b_j)_{j \in J_i}$ is called the $i$-th graded component (aka the $i$-th homogeneous component) of the filtered module with basis $M$; the elements of this submodule are referred to as homogeneous elements of degree $i$. The $R$-module $M$ is the direct sum of its $i$-th graded components over all $i \in I$, and thus becomes a graded $R$-module with basis. Conversely, any graded $R$-module with basis canonically becomes a filtered $R$-module with basis (by defining $F_n = \bigoplus_{i \leq n} G_i$ where $G_i$ is the $i$-th graded component, and
defining $J_i$ as the indexing set of the basis of the $i$-th graded component). Hence, the notion of a filtered $R$-module with basis is equivalent to the notion of a graded $R$-module with basis.

However, the category of filtered $R$-modules with basis is not the category of graded $R$-modules with basis. Indeed, the morphisms of filtered $R$-modules with basis are defined to be morphisms of $R$-modules which send each $F_n$ of the domain to the corresponding $F_n$ of the target; in contrast, the morphisms of graded $R$-modules with basis must preserve each homogeneous component. Also, the notion of a filtered algebra with basis differs from that of a graded algebra with basis.

**Note:** Currently, to make use of the functionality of this class, an instance of `FilteredModulesWithBasis` should fulfill the contract of a `CombinatorialFreeModule` (most likely by inheriting from it). It should also have the indexing set $J$ encoded as its `_indices` attribute, and `_indices.subset(size=i)` should yield the subset $J_i$ (as an iterable). If the latter conditions are not satisfied, then `basis()` must be overridden.

**Note:** One should implement a `degree_on_basis` method in the parent class in order to fully utilize the methods of this category. This might become a required abstract method in the future.

**EXAMPLES:**

```python
tsage: C = ModulesWithBasis(ZZ).Filtered(); C
Category of filtered modules with basis over Integer Ring
tsage: sorted(C.super_categories(), key=str)
[Category of filtered modules over Integer Ring,
  Category of modules with basis over Integer Ring]
tsage: C is ModulesWithBasis(ZZ).Filtered()
True
```

```python
>>> from sage.all import *

>>> C = ModulesWithBasis(ZZ).Filtered(); C
Category of filtered modules with basis over Integer Ring
>>> sorted(C.super_categories(), key=str)
[Category of filtered modules over Integer Ring,
  Category of modules with basis over Integer Ring]
>>> C is ModulesWithBasis(ZZ).Filtered()
True
```

class ElementMethods

Bases: object

`degree()`

The degree of a nonzero homogeneous element `self` in the filtered module.

**Note:** This raises an error if the element is not homogeneous. To compute the maximum of the degrees of the homogeneous summands of a (not necessarily homogeneous) element, use `maximal_degree()` instead.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,4)))
```

(continues on next page)
An example in a graded algebra:

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis):
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).homogeneous_degree()
2
sage: (y*x).homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
sage: A.one().homogeneous_degree()
0

>>> from sage.all import *
>>> A = AlgebrasWithBasis(QQ).Filtered().example()
>>> x,y,z = A.algebra_generators()
>>> (x*y).homogeneous_degree()
2
>>> (y*x).homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
>>> A.one().homogeneous_degree()
0

degree_on_basis(m)
Return the degree of the basis element indexed by m in self.

EXAMPLES:

sage: A = GradedModulesWithBasis(QQ).example() # needs sage.combinat sage.modules
sage: A.degree_on_basis(Partition((2,1)))
# needs sage.combinat sage.modules
3
sage: A.degree_on_basis(Partition((4,2,1,1,1,1)))
# needs sage.combinat sage.modules
10

>>> from sage.all import *
>>> A = GradedModulesWithBasis(QQ).example() # needs sage.combinat sage.modules
>>> A.degree_on_basis(Partition((Integer(2),Integer(1)))) # needs sage.combinat sage.modules
3
>>> A.degree_on_basis(Partition((Integer(4),Integer(2),Integer(1),
-Integer(1),Integer(1),Integer(1)))) # needs sage.
- combinat sage.modules
10

homogeneous_component(n)
Return the homogeneous component of degree n of the element self.

Let m be an element of a filtered R-module M with basis. Then, m can be uniquely written in the form 
\[ m = \sum_{i \in I} m_i \]
where each \( m_i \) is a homogeneous element of degree i. For \( n \in I \), we define the homogeneous component of degree n of the element m to be \( m_n \).

EXAMPLES:
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A.an_element(); x
sage: x.homogeneous_component(-1)
0
sage: x.homogeneous_component(0)
2*P[]
sage: x.homogeneous_component(1)
2*P[1]
sage: x.homogeneous_component(2)
3*P[2]
sage: x.homogeneous_component(3)
0

sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Graded().example()
sage: x = A.an_element(); x
sage: x.homogeneous_component(-1)
0
sage: x.homogeneous_component(0)
2*P[]
sage: x.homogeneous_component(1)
2*P[1]
sage: x.homogeneous_component(2)
3*P[2]
sage: x.homogeneous_component(3)
0

sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: G = A.algebra_generators()
sage: g = A.an_element() - 2 * G['x'] * G['y']; g
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y']
+ 2*U['x'] + 3*U['y'] + 1
sage: g.homogeneous_component(-1)
0
sage: g.homogeneous_component(0)
1
sage: g.homogeneous_component(2)
-2*U['x']*U['y']

sage: g.homogeneous_component(5)
0
sage: g.homogeneous_component(7)
U['x']^2*U['y']^2*U['z']^3
sage: g.homogeneous_component(8)
0

>>> from sage.all import *

>>> # needs sage.combinat sage.modules

>>> A = ModulesWithBasis(ZZ).Filtered().example()
>>> x = A.an_element(); x

>>> x.homogeneous_component(-Integer(1))
0

>>> x.homogeneous_component(Integer(0))
2*P[]
homogeneous_degree()

The degree of a nonzero homogeneous element self in the filtered module.

Note: This raises an error if the element is not homogeneous. To compute the maximum of the degrees of the homogeneous summands of a (not necessarily homogeneous) element, use maximal_degree() instead.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: G = A.algebra_generators()
sage: g = A.an_element() - Integer(2) * G['x'] * G['y']; g
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y']
   + 2*U['x'] + 3*U['y'] + 1
sage: g.homogeneous_component(-Integer(1))
0
sage: g.homogeneous_component(Integer(0))
1
sage: g.homogeneous_component(Integer(2))
-2*U['x']*U['y']
sage: g.homogeneous_component(Integer(5))
0
sage: g.homogeneous_component(Integer(7))
U['x']^2*U['y']^2*U['z']^3
sage: g.homogeneous_component(Integer(8))
0
```
An example in a graded algebra:

sage: # needs sage.combinat sage.modules
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.homogeneous_degree()
2
sage: (x^3 + 4*y^2).homogeneous_degree()
6
sage: ((1 + x)^3).homogeneous_degree()
... 
ValueError: element is not homogeneous

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis):

sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).homogeneous_degree()
is_homogeneous()  
Return whether the element self is homogeneous.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3, 2, 1)))
sage: y = A(Partition((4, 4, 1)))
sage: z = A(Partition((2, 2, 2)))
sage: (3*x).is_homogeneous()  
True
sage: (x - y).is_homogeneous()  
False
sage: (x + 2*z).is_homogeneous()  
True
```

Here is an example with a graded algebra:

```python
sage: # needs sage.combinat sage.modules
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: (3*x).is_homogeneous()
```

(continues on next page)
Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis, or at least on a definition of homogeneous components):

```
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).is_homogeneous()
True
sage: (y*x).is_homogeneous()
False
sage: A.one().is_homogeneous()
True
sage: A.zero().is_homogeneous()
True
sage: (A.one()+x).is_homogeneous()
False
```

**maximal_degree()**

The maximum of the degrees of the homogeneous components of `self`. This is also the smallest $i$ such that `self` belongs to $F_i$. Hence, it does not depend on the basis of the parent of `self`.

See also:

- `homogeneous_degree()`

**EXAMPLES:**

```python
>>> from sage.all import *
>>> A = AlgebrasWithBasis(QQ).Filtered().example()
>>> x,y,z = A.algebra_generators()
>>> (x*y).is_homogeneous()
True
>>> (y*x).is_homogeneous()
False
>>> A.one().is_homogeneous()
True
>>> A.zero().is_homogeneous()
True
>>> (A.one()+x).is_homogeneous()
False
```
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: x.maximal_degree()
6
sage: (x + 2*z).maximal_degree()
6
sage: (y - x).maximal_degree()
9
sage: (3*z).maximal_degree()
6

>>> from sage.all import *

sage: # needs sage.combinat sage.modules
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.maximal_degree()
2
sage: (x^3 + 4*y^2).maximal_degree()
6
sage: ((1 + x)^3).maximal_degree()
6

Let us now test a filtered algebra:

sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
The SageMath command `maximal_degree()` returns the maximum degree of a polynomial. Here are some examples:

```
sage: (x*y).maximal_degree()
sage: 2
sage: (y*x).maximal_degree()
sage: 2
sage: A.one().maximal_degree()
sage: 0
sage: A.zero().maximal_degree()
Traceback (most recent call last):
  ... ValueError: the zero element does not have a well-defined degree
sage: (A.one()+x).maximal_degree()
sage: 1
```

The `truncate(n)` method returns the sum of the homogeneous components of degree strictly less than `n` of the object. Here are examples for different bases:

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = AlgebrasWithBasis(QQ).Filtered().example()
>>> x,y,z = A.algebra_generators()
>>> (x*y).maximal_degree()
sage: 2
>>> (y*x).maximal_degree()
sage: 2
>>> A.one().maximal_degree()
sage: 0
>>> A.zero().maximal_degree()
Traceback (most recent call last):
  ... ValueError: the zero element does not have a well-defined degree
>>> (A.one()+x).maximal_degree()
sage: 1
```

```
truncaten(n)

Return the sum of the homogeneous components of degree strictly less than n of self.

See `homogeneous_component()` for the notion of a homogeneous component.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A.an_element(); x
```

```
sage: x.truncate(0)
0
sage: x.truncate(1)
2*P[]
```

```
sage: x.truncate(2)
2*P[] + 2*P[1]
```

```
sage: x.truncate(3)
```

```
sage: # needs sage.combinat sage.modules
sage: A = ModulesWithBasis(ZZ).Graded().example()
```

```
sage: x = A.an_element(); x
```

```
sage: x.truncate(0)
0
sage: x.truncate(1)
0
```

(continues on next page)
2*P[]  
sage: x.truncate(2)  
2*P[] + 2*P[1]  
sage: x.truncate(3)  

sage: A = AlgebrasWithBasis(ZZ).Filtered().example()  
sage: G = A.algebra_generators()  
sage: g = A.an_element() - 2 * G['x'] * G['y']; g  
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y']  
+ 2*U['x'] + 3*U['y'] + 1  
sage: g.truncate(-1)  
0  
sage: g.truncate(0)  
0  
sage: g.truncate(2)  
2*U['x'] + 3*U['y'] + 1  
sage: g.truncate(3)  
-2*U['x']*U['y'] + 2*U['x'] + 3*U['y'] + 1  
sage: g.truncate(5)  
-2*U['x']*U['y'] + 2*U['x'] + 3*U['y'] + 1  
sage: g.truncate(7)  
-2*U['x']*U['y'] + 2*U['x'] + 3*U['y'] + 1  
sage: g.truncate(8)  
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y']  
+ 2*U['x'] + 3*U['y'] + 1

>>> from sage.all import *  
>>> # needs sage.combinat sage.modules  
>>> A = ModulesWithBasis(ZZ).Filtered().example()  
>>> x = A.an_element(); x  
>>> x.truncate(Integer(0))  
0  
>>> x.truncate(Integer(1))  
2*P[]  
>>> x.truncate(Integer(2))  
2*P[] + 2*P[1]  
>>> x.truncate(Integer(3))  

>>> # needs sage.combinat sage.modules  
>>> A = ModulesWithBasis(ZZ).Graded().example()  
>>> x = A.an_element(); x  
>>> x.truncate(Integer(0))  
0  
>>> x.truncate(Integer(1))  
2*P[]  
>>> x.truncate(Integer(2))  
2*P[] + 2*P[1]  
>>> x.truncate(Integer(3))  

>>> A = AlgebrasWithBasis(ZZ).Filtered().example()  
>>> G = A.algebra_generators()
class FiniteDimensional(base_category)

Bases: CategoryWithAxiom_over_base_ring

class ParentMethods

Bases: object

hilbert_series(prec=None)

Return the Hilbert series of self as a polynomial.

Let $R$ be a commutative ring (with unit). Let $M = \bigcup_{n=0}^{\infty} M_n$ be a filtered $R$-module. The Hilbert series of $M$ is the formal power series

$$H(t) = \sum_{n=0}^{\infty} \ell(M_n/M_{n-1}) t^n,$$

where $\ell(N)$ is the length of $N$, which is the longest chain of submodules (over $R$), and by convention $M_{-1} = \{0\}$. By the assumptions of the category, $M_n/M_{n-1}$ is a free $R$-module, and so $\ell(M_n/M_{n-1})$ is equal to the rank of $M_n/M_{n-1}$.

EXAMPLES:

```python
sage: OS = hyperplane_arrangements.braid(3).orlik_solomon_algebra(QQ)
sage: OS.hilbert_series()
2*t^2 + 3*t + 1

sage: OS = matroids.Uniform(5, 3).orlik_solomon_algebra(ZZ)
sage: OS.hilbert_series()
t^3 + 3*t^2 + 3*t + 1

sage: OS = matroids.PG(2, 3).orlik_solomon_algebra(ZZ['x','y'])
sage: OS.hilbert_series()
27*t^3 + 39*t^2 + 13*t + 1
```

```python
>>> from sage.all import *

>>> OS = hyperplane_arrangements.braid(Integer(3)).orlik_solomon_algebra(QQ)
>>> OS.hilbert_series()
```
2*t^2 + 3*t + 1

```python
>>> OS = matroids.Uniform(Integer(5), Integer(3)).orlik_solomon_algebra(ZZ)
>>> OS.hilbert_series()
t^3 + 3*t^2 + 3*t + 1

>>> OS = matroids.PG(Integer(2), Integer(3)).orlik_solomon_algebra(ZZ['x','y'])
>>> OS.hilbert_series()
27*t^3 + 39*t^2 + 13*t + 1
```

```python
class ParentMethods
    Bases: object

    basis(d=None)

    Return the basis for (the d-th homogeneous component of) self.

    INPUT:
    • d – (default: None) nonnegative integer or None

    OUTPUT:

    If d is None, returns the basis of the module. Otherwise, returns the basis of the homogeneous component of degree d (i.e., the subfamily of the basis of the whole module which consists only of the basis vectors lying in \( F_d \setminus \bigcup_{i<d} F_i \)).

    The basis is always returned as a family.

    EXAMPLES:

    ```python
    sage: A = ModulesWithBasis(ZZ).Filtered().example()
    sage: A.basis(4)
    Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions of the integer 4}
    ```

    ```python
    >>> from sage.all import *
    >>> A = ModulesWithBasis(ZZ).Filtered().example()
    >>> A.basis(Integer(4))
    Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions of the integer 4}
    ```

    Without arguments, the full basis is returned:

    ```python
    sage: A.basis()
    Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions}
    ```

    ```python
    sage: A.basis()
    Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions}
    ```

    ```python
    >>> from sage.all import *
    >>> A.basis()
    Lazy family (Term map from Partitions to An example of a
    ```
filtered module with basis: the free module on partitions over Integer Ring(i)_(i in Partitions)

>>> A.basis()
Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i)_(i in Partitions)

Checking this method on a filtered algebra. Note that this will typically raise a `NotImplementedError` when this feature is not implemented.

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Traceback (most recent call last):
  ...  
NotImplementedError: infinite set
```

```python
>>> from sage.all import *

>>> A = AlgebrasWithBasis(ZZ).Filtered().example()
>>> A.basis(Integer(4))
Traceback (most recent call last):
  ...  
NotImplementedError: infinite set
```

Without arguments, the full basis is returned:

```python
sage: A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring(i)_(i in Free abelian monoid indexed by {'x', 'y', 'z'})
```

```python
>>> from sage.all import *

>>> A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring(i)_(i in Free abelian monoid indexed by {'x', 'y', 'z'})
```

An example with a graded algebra:

```python
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.basis()
Lazy family (Term map from Subsets of {0,1} to The exterior algebra of rank 2 over Rational Field(i)_(i in Subsets of {0,1})
```

```python
>>> from sage.all import *

>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_  
˓→ngens(2)
>>> E.basis()
Lazy family (Term map from Subsets of {0,1} to The exterior algebra of rank 2 over Rational Field(i)_(i in Subsets of {0,1})
```

4.43. Filtered Modules With Basis 557
**from_graded_conversion()**

Return the inverse of the canonical $R$-module isomorphism $A \rightarrow \text{gr} A$ induced by the basis of $A$ (where $A = "\text{self}"$). This inverse is an isomorphism $\text{gr} A \rightarrow A$.

This is an isomorphism of $R$-modules. See the class documentation AssociatedGradedAlgebra.

See also:

**to_graded_conversion()**

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: p = -2 * A.an_element(); p
sage: q = A.to_graded_conversion()(p); q
sage: A.from_graded_conversion()(q) == p
True
sage: q.parent() is A.graded_algebra()
True
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = Modules(QQ).WithBasis().Filtered().example()
>>> p = -Integer(2) * A.an_element(); p
>>> q = A.to_graded_conversion()(p); q
>>> A.from_graded_conversion()(q) == p
True
>>> q.parent() is A.graded_algebra()
True
```

**graded_algebra()**

Return the associated graded module to `self`.

See AssociatedGradedAlgebra for the definition and the properties of this.

If the filtered module `self` with basis is called $A$, then this method returns $\text{gr} A$. The method `to_graded_conversion()` returns the canonical $R$-module isomorphism $A \rightarrow \text{gr} A$ induced by the basis of $A$, and the method `from_graded_conversion()` returns the inverse of this isomorphism. The method `projection()` projects elements of $A$ onto $\text{gr} A$ according to their place in the filtration on $A$.

**Warning:** When not overridden, this method returns the default implementation of an associated graded module – namely, AssociatedGradedAlgebra(self), where AssociatedGradedAlgebra is AssociatedGradedAlgebra. But some instances of FilteredModulesWithBasis override this method, as the associated graded module often is (isomorphic) to a simpler object (for instance, the associated graded module of a graded module can be identified with the graded module itself). Generic code that uses associated graded modules (such as the code of the induced_graded_map() method below) should make sure to only communicate with them via the to_graded_conversion(), from_graded_conversion() and projection() methods (in particular, do not expect there to be a conversion from `self` to `self.graded_algebra()`; this currently does not work for Clifford algebras). Similarly, when overriding `graded_algebra()`, make sure to accordingly redefine these three methods,
EXCEPT THEIR DEFINITIONS BELOW STILL APPLY TO YOUR CASE (THIS WILL HAPPEN WHENEVER THE BASIS OF YOUR
\texttt{graded}\textunderscore\texttt{algebra}() HAS THE SAME INDEXING SET AS \texttt{self}, AND THE PARTITION OF THIS INDEXING SET
ACCORDING TO DEGREE IS THE SAME AS FOR \texttt{self}).

\textbf{EXAMPLES}:

\begin{verbatim}
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: A.graded_algebra()
Graded Module of An example of a filtered module with basis:
  the free module on partitions over Integer Ring

from sage.all import *
A = ModulesWithBasis(ZZ).Filtered().example()
A.graded_algebra()
Graded Module of An example of a filtered module with basis:
  the free module on partitions over Integer Ring

hilbert\textunderscore\texttt{series}\ (\texttt{prec}=\texttt{None})

Return the Hilbert series of \texttt{self}.

Let $R$ be a commutative ring (with unit). Let $M = \bigcup_{n=0}^{\infty} M_n$ be a filtered $R$-module. The \textit{Hilbert series}
of $M$ is the formal power series

$$H(t) = \sum_{n=0}^{\infty} \ell(M_n/M_{n-1}) t^n,$$

where $\ell(N)$ is the \textit{length} of $N$, which is the longest chain of submodules (over $R$), and by convention
$M_{-1} = \{0\}$. By the assumptions of the category, $M_n/M_{n-1}$ is a free $R$-module, and so $\ell(M_n/M_{n-1})$
is equal to the rank of $M_n/M_{n-1}$.

INPUT:

\begin{itemize}
  \item \texttt{prec} -- (default: $\infty$) the precision
\end{itemize}

OUTPUT:

If the precision is finite, then this returns an element in the \texttt{PowerSeriesRing} over $\mathbb{Z}$. Otherwise
it returns an element in the \texttt{LazyPowerSeriesRing} over $\mathbb{Z}$.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.hilbert_series()
1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + O(t^7)
sage: A.hilbert_series(Integer(10))
1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + 15*t^7 + 22*t^8 + 30*t^9 + O(t^10)

from sage.all import *
A = GradedModulesWithBasis(ZZ).example()
A.hilbert_series()
1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + 0(t^7)
A.hilbert_series(Integer(10))
1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + 15*t^7 + 22*t^8 + 30*t^9 + O(t^10)
\end{verbatim}

\textbf{homogeneous\textunderscore\texttt{component}} ($d$)

Return the $d$-th homogeneous component of \texttt{self}.

4.43. Filtered Modules With Basis
EXAMPLES:

```python
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.homogeneous_component(4)
Degree 4 homogeneous component of An example of a graded module
with basis: the free module on partitions over Integer Ring
```

```python
>>> from sage.all import *
>>> A = GradedModulesWithBasis(ZZ).example()
>>> A.homogeneous_component(Integer(4))
Degree 4 homogeneous component of An example of a graded module
with basis: the free module on partitions over Integer Ring
```

**homogeneous_component_basis** \((d)\)

Return a basis for the \(d\)-th homogeneous component of \(self\).

EXAMPLES:

```python
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.homogeneous_component_basis(4)
Lazy family (Term map
from Partitions
  to An example of a graded module with basis: the free module
    on partitions over Integer Ring(i))_{i in Partitions of the integer ←4}
```

```python
sage: # needs sage.modules
cat = GradedModulesWithBasis(ZZ)
C = CombinatorialFreeModule(ZZ, ['a', 'b'], category=cat)
C.degree_on_basis = lambda x: 1 if x == 'a' else 2
C.homogeneous_component_basis(1)
Finite family {'a': B['a']}
```

```python
>>> from sage.all import *
>>> A = GradedModulesWithBasis(ZZ).example()
>>> A.homogeneous_component_basis(Integer(4))
Lazy family (Term map
from Partitions
  to An example of a graded module with basis: the free module
    on partitions over Integer Ring(i))_{i in Partitions of the integer ←4}
```

```python
>>> # needs sage.modules
cat = GradedModulesWithBasis(ZZ)
C = CombinatorialFreeModule(ZZ, ['a', 'b'], category=cat)
C.degree_on_basis = lambda x: Integer(1) if x == 'a' else Integer(2)
C.homogeneous_component_basis(Integer(1))
Finite family {'a': B['a']}
```

**induced_graded_map** *(other, f)*

Return the graded linear map between the associated graded modules of \(self\) and \(other\) canonically induced by the filtration-preserving map \(f : self \rightarrow other\).

Let \(A\) and \(B\) be two filtered modules with basis, and let \((F_i)_{i \in I}\) and \((G_i)_{i \in I}\) be their filtrations. Let
\( f : A \to B \) be a linear map which preserves the filtration (i.e., satisfies \( f(F_i) \subseteq G_i \) for all \( i \in I \)). Then, there is a canonically defined graded linear map \( \text{gr} f : \text{gr} A \to \text{gr} B \) which satisfies

\[
(\text{gr} f)(p_i(a)) = p_i(f(a)) \quad \text{for all } i \in I \text{ and } a \in F_i,
\]

where the \( p_i \) on the left hand side is the canonical projection from \( F_i \) onto the \( i \)-th graded component of \( \text{gr} A \), while the \( p_i \) on the right hand side is the canonical projection from \( G_i \) onto the \( i \)-th graded component of \( \text{gr} B \).

**INPUT:**
- \( \text{other} \) - a filtered algebra with basis
- \( f \) - a filtration-preserving linear map from \( \text{self} \) to \( \text{other} \) (can be given as a morphism or as a function)

**OUTPUT:**
The graded linear map \( \text{gr} f \).

**EXAMPLES:**

**Example 1.**

We start with the free \( \mathbb{Q} \)-module with basis the set of all partitions:

```python
sage: A = Modules(QQ).WithBasis().Filtered().example(); A
# needs sage.combinat sage.modules
An example of a filtered module with basis: the free module on partitions over Rational Field

sage: M = A.indices(); M
# needs sage.combinat sage.modules
Partitions

sage: p1, p2, p21, p321 = [A.basis()[Partition(i)]
# needs sage.combinat sage.modules
.....: for i in [[1], [2], [2,1], [3,2,1]]

>>> from sage.all import *

>>> A = Modules(QQ).WithBasis().Filtered().example(); A
# needs sage.combinat sage.modules
An example of a filtered module with basis: the free module on partitions over Rational Field

>>> M = A.indices(); M
# needs sage.combinat sage.modules
Partitions

>>> p1, p2, p21, p321 = [A.basis()[Partition(i)]
# needs sage.combinat sage.modules
....: for i in [[Integer(1)], [Integer(2)],
....: [Integer(2),Integer(1)], [Integer(3),Integer(2),Integer(1)]]

Let us define a map from \( A \) to itself which acts on the basis by sending every partition \( \lambda \) to the sum of the conjugates of all partitions \( \mu \) for which \( \lambda / \mu \) is a horizontal strip:

```python
sage: # needs sage.combinat sage.modules
def map_on_basis(lam):
    ....: def mus(k):
    ....:     return lam.remove_horizontal_border_strip(k)
    ....: return A.sum_of_monomials([Partition(mu).conjugate()
    ....:     for k in range(sum(lam) + 1)
    ....:     for mu in mus(k)])

sage: f = A.module_morphism(on_basis=map_on_basis,
.....:     codomain=A)
```

(continues on next page)
We now compute $\mathfrak{g} f$

```python
sage: # needs sage.combinat.sage.modules
sage: grA = A.graded_algebra(); grA
Graded Module of An example of a filtered module with basis:
the free module on partitions over Rational Field
sage: pp1, pp2, pp21, pp321 = [A.to_graded_conversion()(i) for i in [p1, p2, p21, p321]]
...:   for i in [p1, p2, p21, p321]]
sage: grf = A.induced_graded_map(A, f); grf
Generic endomorphism of Graded Module of
An example of a filtered module with basis:
the free module on partitions over Rational Field
sage: grf(pp1)
Bbar[[1]]
sage: grf(pp2 + 4 * pp21)
Bbar[[1, 1]] + 4*Bbar[[2, 1]]
```
Example 2.

We shall now construct $g f$ for a different map $f$ out of the same $A$; the new map $f$ will lead into a graded algebra already, namely into the algebra of symmetric functions:

```
sage: # needs sage.combinat sage.modules
sage: h = SymmetricFunctions(QQ).h()
sage: def map_on_basis(lam):
.... def mus(k):
.... return lam.remove_horizontal_border_strip(k)
.... return h.sum_of_monomials([Partition(mu).conjugate()
.... for k in range(sum(lam) + 1)
.... for mu in mus(k)])
sage: f = A.module_morphism(on_basis=map_on_basis,
.... codomain=h) # redefining f
sage: f(p1)
h[] + h[1]
sage: f(p2)
h[] + h[1] + h[1, 1]
sage: f(A.zero())
0
sage: f(p2 - 3*p1)
-2*h[] - 2*h[1] + h[1, 1]
```

(continues on next page)
The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:

```python
sage: # needs sage.combinat sage.modules
sage: grh = h.graded_algebra(); grh
  True
sage: grf = A.induced_graded_map(h, f); grf
  Generic morphism:
  From: Graded Module of An example of a filtered module with basis: the free module on partitions over Rational Field
  To:  Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(pp1)
  h[1]
sage: grf(pp2)
  h[1, 1]
sage: grf(pp321)
  h[3, 2, 1]
sage: grf(pp2 - Integer(3)*pp1)
-3*h[1] + h[1, 1]
sage: grf(pp21)
  h[2, 1]
sage: grf(grA.zero())
  0
```

Example 3.

After having had a graded module as the codomain, let us try to have one as the domain instead. Our new $f$ will go from $h$ to $A$:  

$$-2 \cdot h[\cdot] - 2 \cdot h[1] + h[1, 1]$$
sage: # needs sage.combinat sage.modules
sage: def map_on_basis(lam): # redefining map_on_basis
    ....:     def mus(k):
    ....:         return lam.remove_horizontal_border_strip(k)
    ....:     return A.sum_of_monomials([Partition(mu).conjugate()
    ....:                                  for k in range(sum(lam) + 1)
    ....:                                  for mu in mus(k)])
    ....:
    ....:     f = h.module_morphism(on_basis=map_on_basis,
    ....:                            codomain=A) # redefining f

sage: f(h[1])
P[1] + P[1]
sage: f(h[2])
P[1] + P[1] + P[1, 1]
sage: f(h[1, 1])
sage: f(h[2, 2])
P[1, 1] + P[2, 1] + P[2, 2]
sage: f(h[3, 2, 1])

sage: grf = h.induced_graded_map(A, f); grf
Generic morphism:
    From: Symmetric Functions over Rational Field
    in the homogeneous basis
    To:   Graded Module of An example of a filtered
    module with basis: the free module on partitions
    over Rational Field

sage: grf(h[1])
Bbar[[1]]
sage: grf(h[2])
Bbar[[1], [1]]
sage: grf(h[3, 2, 1])
Bbar[[3, 2, 1]]
sage: grf(h.one())
Bbar[[1], [1], [1]]

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> def map_on_basis(lam): # redefining map_on_basis
...     def mus(k):
...         return lam.remove_horizontal_border_strip(k)
...     return A.sum_of_monomials([Partition(mu).conjugate()
...                                  for k in range(sum(lam) + Integer(1))
...                                  for mu in mus(k)])
>>> f = h.module_morphism(on_basis=map_on_basis,
...                        codomain=A) # redefining f

>>> f(h[Integer(1)])
P[1] + P[1]
>>> f(h[Integer(2)])
P[1] + P[1] + P[1, 1]
>>> f(h[Integer(1), Integer(1)])
Example 4.

The construct $\text{gr} f$ also makes sense when $f$ is a filtration-preserving map between graded modules.

```python
sage: # needs sage.combinat sage.modules
sage: def map_on_basis(lam): # redefining map_on_basis
sage: ....: def mus(k):
sage: ....: return lam.remove_horizontal_border_strip(k)
sage: ....: return h.sum_of_monomials([Partition(mu).conjugate() for k in range(sum(lam) + 1) for mu in mus(k)])
.sage: f = h.module_morphism(on_basis=map_on_basis,
sage: ....: codomain=h) # redefining f
.sage: f(h[1])
h[1] + h[1]
sage: f(h[2])
h[1] + h[1] + h[1, 1]
sage: f(h[1, 1])
h[1] + h[2]
sage: f(h[2, 1])
sage: f(h.one())
h[]
sage: grf = h.induced_graded_map(h, f); grf
Generic endomorphism of
Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(h[1])
h[1]
sage: grf(h[2])
```

(continues on next page)
projection \( i \)

Return the \( i \)-th projection \( p_i : F_i \to G_i \) (in the notations of the class documentation Associated-GradedAlgebra, where \( A = \text{"self"} \)).

This method actually does not return the map \( p_i \) itself, but an extension of \( p_i \) to the whole \( R \)-module \( A \). This extension is the composition of the \( R \)-module isomorphism \( A \to \text{gr } A \) with the canonical projection of the graded \( R \)-module \( \text{gr } A \) onto its \( i \)-th graded component \( G_i \). The codomain of this map is \( \text{gr } A \), although its actual image is \( G_i \). The map \( p_i \) is obtained from this map by restricting its domain to \( F_i \) and its image to \( G_i \).

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: A = Modules(ZZ).WithBasis().Filtered().example()
```
```python
sage: p = -2 * A.an_element(); p
sage: q = A.projection(2)(p); q
-6*Bbar[[2]]
sage: q.parent() is A.graded_algebra()
True
sage: A.projection(3)(p)
0

```  

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = Modules(ZZ).WithBasis().Filtered().example()
>>> p = -Integer(2) * A.an_element(); p
>>> q = A.projection(Integer(2))(p); q
-6*Bbar[[2]]
>>> q.parent() is A.graded_algebra()
True
```  

### `to_graded_conversion()`

Return the canonical $R$-module isomorphism $A \to \text{gr} A$ induced by the basis of $A$ (where $A = \text{``self''}$).

This is an isomorphism of $R$-modules. See the class documentation `AssociatedGradedAlgebra`.

See also:

`from_graded_conversion()`

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: p = -2 * A.an_element(); p
```

```python
```  

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = Modules(QQ).WithBasis().Filtered().example()
>>> p = -Integer(2) * A.an_element(); p
```  

### `class Subobjects` *(category, *args)*

Bases: `SubobjectsCategory`

### `class ElementMethods`

Bases: `object`
degree()

Return the degree of self.

EXAMPLES:

```python
sage: # needs sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: S = E.submodule([x + y, x*y - y*z, y])
sage: B = S.basis()
[x, y, x*y - y*z]
sage: B[0].degree()
1
sage: B[1].degree()
1
sage: (B[0] + 3*B[1]).degree()
1
```

The degree of inhomogeneous elements is not defined (following the behavior of the exterior algebra):

```python
sage: (B[0] + B[2]).degree()
# needs sage.modules
Traceback (most recent call last):
... ValueError: element is not homogeneous
```

We can still get the maximal degree:

```python
sage: (B[0] + B[2]).maximal_degree()
# needs sage.modules
2
```

(continues on next page)
maximal_degree()

The maximum of the degrees of the homogeneous components of self.

This is also the smallest $i$ such that self belongs to $F_i$. Hence, it does not depend on the basis of the parent of self.

See also:

homogeneous_degree()

EXAMPLES:

```
sage: # needs sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: F = E.submodule([x + 1, x*y - 1])
sage: B = F.basis()
sage: [B[0].lift(), B[1].lift()]
[-x*y + 1, x*y + x]
sage: B[0].maximal_degree()
2
sage: B[1].maximal_degree()
2
```

class ParentMethods

Bases: object

degree_on_basis($m$)

Return the degree of the basis element indexed by $m$ in self.

EXAMPLES:

```
sage: # needs sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: S = E.submodule([x + y, x*y - y*z, y])
sage: B = S.basis()
[x, y, x*y - y*z]
sage: S.degree_on_basis(0)
1
sage: S.degree_on_basis(1)
1
```
### 4.44 Finite Complex Reflection Groups

*class* `sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups(base_category)`

The category of finite complex reflection groups.

See [ComplexReflectionGroups](#) for the definition of complex reflection group. In the finite case, most of the information about the group can be recovered from its *degrees* and *codegrees*, and to a lesser extent to the explicit realization as subgroup of $GL(V)$. Hence the most important optional methods to implement are:

- `ComplexReflectionGroups.Finite.ParentMethods.degrees()`,
- `ComplexReflectionGroups.Finite.ParentMethods.codegrees()`,

Finite complex reflection groups are completely classified. In particular, if the group is irreducible, then it’s uniquely determined by its degrees and codegrees and whether it’s reflection representation is *primitive* or not (see [LT2009] Chapter 2.1 for the definition of primitive).

**See also:**

Wikipedia article Complex_reflection_groups

**EXAMPLES:**

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite()
Category of finite complex reflection groups
sage: ComplexReflectionGroups().Finite().super_categories()
[Category of complex reflection groups, Category of finite groups, Category of finite finitely generated semigroups]
```
An example of a finite reflection group:

```python
sage: W = ComplexReflectionGroups().Finite().example(); W
Reducible real reflection group of rank 4 and type A2 x B2
sage: W.reflections()
Finite family {1: (1,8)(2,5)(9,12), 2: (1,5)(2,9)(8,12),
3: (3,10)(4,7)(11,14), 4: (3,6)(4,11)(10,13),
5: (1,9)(2,8)(5,12), 6: (4,14)(6,13)(7,11),
7: (3,13)(6,10)(7,14)}
```

W is in the category of complex reflection groups:

```python
sage: W in ComplexReflectionGroups().Finite()  # optional - gap3
True
```

class ElementMethods

Bases: object

character_value()

Return the value at self of the character of the reflection representation given by to_matrix().

EXAMPLES:

```python
sage: W = ColoredPermutations(1,3); W
1-colored permutations of size 3
sage: [t.character_value() for t in W]  # needs sage.combinat sage.groups
[3, 1, 1, 0, 0, 1]
```
Note that this could be a different (faithful) representation than that given by the corresponding root system:

```python
sage: W = ReflectionGroup((1,1,3)); W
Irreducible real reflection group of rank 2 and type A2
sage: [t.character_value() for t in W]  # optional - gap3
[2, 0, 0, -1, -1, 0]

sage: W = ColoredPermutations(2,2); W  # needs sage.combinat
2-colored permutations of size 2
sage: [t.character_value() for t in W]  # needs sage.combinat sage.groups
[2, 0, 0, -2, 0, 0, 0, 0]

sage: W = ColoredPermutations(3,1); W  # needs sage.combinat
3-colored permutations of size 1
sage: [t.character_value() for t in W]  # needs sage.combinat sage.groups
[1, zeta3, -zeta3 - 1]
```

**reflection_length** *(in_unitary_group=False)*

Return the reflection length of self.

This is the minimal numbers of reflections needed to obtain self.

**INPUT:**

- **in_unitary_group** *(default: False)* if True, the reflection length is computed in the unitary group which is the dimension of the move space of self.
EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3)) # optional - gap3
sage: sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 2, 2]

sage: W = ReflectionGroup((2,1,2)) # optional - gap3
sage: sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 1, 2, 2, 2]

sage: W = ReflectionGroup((2,2,2)) # optional - gap3
sage: sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 2]

sage: W = ReflectionGroup((3,1,2)) # optional - gap3
sage: sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3))) → # optional - gap3
>>> sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 2, 2]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(2))) → # optional - gap3
>>> sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 1, 2, 2, 2]

>>> W = ReflectionGroup((Integer(2),Integer(2),Integer(2))) → # optional - gap3
>>> sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 2]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2))) → # optional - gap3
>>> sorted([t.reflection_length() for t in W]) # optional - gap3
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

**to_matrix()**

Return the matrix presentation of self acting on a vector space $V$.

EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3)) # optional - gap3
sage: [t.to_matrix() for t in W] # optional - gap3
[[1 0]
 [0 1]],
[[1 1]
 [0 0]],
[-1 0]
 [0 1]
[-1 -1]
 [0 1]
[ 0 -1]
 [1 0]
[ 0 1]
 [1 1],
[-1 -1]
 [0 1]
[-1 0]
 [1 0]]

sage: W = ColoredPermutations(1,3) # needs sage.combinat
sage: [t.to_matrix() for t in W] # needs sage.combinat sage.groups
[[1 0 0]
 [0 1 0]
 [0 0 1]],
[[1 0 0]
 [0 1 0]
 [0 0 1]],
[[0 0 1]
 [1 0 0]
 [0 0 1]],
[[0 1 0]
 [0 0 1]
 [1 0 0]],
[[0 1 0]
 [0 1 0]
 [0 0 1]]
```

(continues on next page)
A different representation is given by the colored permutations:

```python
sage: W = ColoredPermutations(3, 1)  # needs sage.combinat
sage: [t.to_matrix() for t in W]  # needs sage.combinat sage.groups
[[1], [zeta3], [-zeta3 - 1]]
```

```python
>>> from sage.all import *
>>> W = ColoredPermutations(Integer(3), Integer(1))  # needs sage.combinat
>>> [t.to_matrix() for t in W]  # needs sage.combinat sage.groups
[[1], [zeta3], [-zeta3 - 1]]
```

## 4.44. Finite Complex Reflection Groups

A different representation is given by the colored permutations:

```python
sage: W = ColoredPermutations(3, 1)  # needs sage.combinat
sage: [t.to_matrix() for t in W]  # needs sage.combinat sage.groups
[[1], [zeta3], [-zeta3 - 1]]
```

```python
>>> from sage.all import *
>>> W = ColoredPermutations(Integer(3), Integer(1))  # needs sage.combinat
>>> [t.to_matrix() for t in W]  # needs sage.combinat sage.groups
[[1], [zeta3], [-zeta3 - 1]]
```

### class Irreducible (base_category)

Bases: CategoryWithAxiom

```python
class ParentMethods
Bases: object

absolute_order_ideal (gens=None, in_unitary_group=True, return_lengths=False)

Return all elements in self below given elements in the absolute order of self.

This order is defined by

\[ \omega \leq_R \tau \iff \ell_R(\omega) + \ell_R(\omega^{-1}\tau) = \ell_R(\tau), \]

where \( \ell_R \) denotes the reflection length.

This is, if in_unitary_group is False, then

\[ \ell_R(w) = \min\{\ell : w = r_1 \cdots r_i, r_i \in R\}, \]
and otherwise

$$\ell_R(w) = \dim \ker(w - 1).$$

**Note:** If `gens` are not given, `self` is assumed to be well-generated.

**INPUT:**

- `gens` *(default: None)* if one or more elements are given, the order ideal in the absolute order generated by `gens` is returned. Otherwise, the standard Coxeter element is used as unique maximal element.
- `in_unitary_group` *(default: True)* determines the length function used to compute the order. For real groups, both possible orders coincide, and for complex non-real groups, the order in the unitary group is much faster to compute.
- `return_lengths` *(default: False)* whether or not to also return the lengths of the elements.

**EXAMPLES:**

```python
sage: # optional - gap3
sage: W = ReflectionGroup((1, 1, 3))
sage: sorted(w.reduced_word() for w in W.absolute_order_ideal())
[[], [1], [1, 2], [1, 2, 1], [2]]
sage: sorted(w.reduced_word() for w in W.absolute_order_ideal(W.from_reduced_word([2, -1])))
[[], [1], [1, 2, 1], [2], [2, 1]]
sage: sorted(w.reduced_word() for w in W.absolute_order_ideal(W.from_reduced_word([2])))
[[], [2]]
sage: W = CoxeterGroup(["A", 3])
# needs sage.combinat sage.groups
sage: len(list(W.absolute_order_ideal()))
14
sage: W = CoxeterGroup(["A", 2])
# needs sage.combinat sage.groups
sage: for (w, l) in W.absolute_order_ideal(return_lengths=True):
    print(w.reduced_word(), l)
[1, 2] 2
[1, 2, 1] 1
[2] 1
[1] 1
[] 0

>>> from sage.all import *
>>> # optional - gap3
>>> W = ReflectionGroup((Integer(1), Integer(1), Integer(3)))
>>> sorted(w.reduced_word() for w in W.absolute_order_ideal())
[[], [1], [1, 2], [1, 2, 1], [2]]
>>> sorted(w.reduced_word() for w in W.absolute_order_ideal(W.from_reduced_word([Integer(2), Integer(1)])))
(continues on next page)```
sorted([w.reduced_word() for w in W.absolute_order_ideal(W.from_reduced_word([Integer(2)]))])

W = CoxeterGroup(['A', Integer(3)])
# needs sage.combinat, sage.groups
len(list(W.absolute_order_ideal()))
# needs sage.combinat, sage.groups
14

W = CoxeterGroup(['A', Integer(2)])
# needs sage.combinat, sage.groups
for (w, l) in W.absolute_order_ideal(return_lengths=True):
# needs sage.combinat, sage.groups
... print(w.reduced_word(), l)
[1, 2] 2
[1, 2, 1] 1
[2] 1
[1] 1
[] 0

absolute_poset (in_unitary_group=False)

Return the poset induced by the absolute order of self as a finite lattice.

INPUT:
• in_unitary_group – (default: False) if False, the relation is given by \(\sigma \leq \tau\) if False, the relation is given by \(\sigma \leq \tau\) if \(l_R(\sigma) + l_R(\sigma^{-1}\tau) = l_R(\tau)\) If True, the relation is given by \(\sigma \leq \tau\) if \(\dim(\text{Fix}(\sigma)) + \dim(\text{Fix}(\sigma^{-1}\tau)) = \dim(\text{Fix}(\tau))\)

See also:
noncrossing_partition_lattice()

EXAMPLES:

```
sage: P = ReflectionGroup((1,1,3)).absolute_poset(); P
Finite poset containing 6 elements

sage: sorted([w.reduced_word() for w in P])

sage: W = ReflectionGroup(4); W
Irreducible complex reflection group of rank 2 and type ST4
sage: W.absolute_poset()
Finite poset containing 24 elements
```

```
>>> from sage.all import *

>>> P = ReflectionGroup((Integer(1),Integer(1),Integer(3))).absolute_poset(); P
Finite poset containing 6 elements

>>> sorted([w.reduced_word() for w in P])
(continues on next page)"
coxeter_number()

Return the Coxeter number of an irreducible reflection group.

This is defined as $\frac{N + N^*}{n}$ where $N$ is the number of reflections, $N^*$ is the number of reflection hyperplanes, and $n$ is the rank of self.

EXAMPLES:

```python
sage: W = ReflectionGroup(31) # optional - gap3
sage: W.coxeter_number() # optional - gap3
30
```

generalized_noncrossing_partitions $m, c=\text{None}, \text{positive}=\text{False}$

Return the set of all chains of length $m$ in the noncrossing partition lattice of self, see `noncrossing_partition_lattice()`.

Note: self is assumed to be well-generated.

INPUT:

- $c$ – (default: None) if an element $c$ in self is given, it is used as the maximal element in the interval
- $\text{positive}$ – (default: False) if True, only those generalized noncrossing partitions of full support are returned

EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3)) # optional - gap3
sage: chains = W.generalized_noncrossing_partitions(2) # optional - gap3
sage: sorted([w.reduced_word() for w in chain] for chain in chains) # optional - gap3
[[[], [], [1, 2]]],
[[[], [1], [2]]],
[[[], [1, 2], []]],
[[[], [1, 2, 1], [1]]],
[[[], [2], [1, 2, 1]]],
[[[1], [], [2]]],
```

(continues on next page)
([1], [2], []),
([1], [2], [], [1]),
([1], [2], [1], [1]),
([2], [1], [1], [1]),
([2], [1], [1], [], [])

sage: chains = W.generalized_noncrossing_partitions(2, positive=True)

sage: sorted([w.reduced_word() for w in chain])
[[[], [], [1, 2]],
 [ [], [1], [2]],
 [ [], [1], [], [2]],
 [ [], [1], [2], [1]],
 [ [], [2], [1], [2]],
 [ [1], [], [2]],
 [ [1], [2], []],
 [ [1], [2], [1], []],
 [ [2], [], [1], [2]],
 [ [2], [1], [], [2]],
 [ [2], [1], [2], []]]

>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))

noncrossing_partition_lattice (c=None, L=None, in_unitary_group=True)
Return the interval \([1, c]\) in the absolute order of \(\text{self}\) as a finite lattice.

See also:

\texttt{absolute_order_ideal()}

INPUT:

- \(c\) (default: None) if an element \(c\) in \(\text{self}\) is given, it is used as the maximal element in the interval.

- \(L\) (default: None) if a subset \(L\) (must be hashable!) of \(\text{self}\) is given, it is used as the underlying set (only cover relations are checked).

- \texttt{in\_unitary\_group} (default: False) if False, the relation is given by \(\sigma \leq \tau\) if \(l_R(\sigma) + l_R(\sigma^{-1}\tau) = l_R(\tau)\); if True, the relation is given by \(\sigma \leq \tau\) if \(\dim(\text{Fix}(\sigma)) + \dim(\text{Fix}(\sigma^{-1}\tau)) = \dim(\text{Fix}(\tau))\)

Note: If \(L\) is given, the parameter \(c\) is ignored.

EXAMPLES:

\begin{verbatim}
sage: W = SymmetricGroup(4) \# needs sage.groups
sage: W.noncrossing_partition_lattice() \# needs sage.combinat sage.groups
Finite lattice containing 14 elements

sage: W = WeylGroup(['G', 2]) \# needs sage.combinat sage.groups
sage: W.noncrossing_partition_lattice() \# needs sage.combinat sage.groups
Finite lattice containing 8 elements

sage: # optional - gap3
sage: W = ReflectionGroup((1,1,3))

sage: sorted(w.reduced_word() for w in W.noncrossing_partition_lattice())
[[], [1], [1, 2], [1, 2, 1], [2]]

sage: c21 = W.from_reduced_word([2,1])

sage: sorted(w.reduced_word() for w in W.noncrossing_partition_lattice(c21))
[[], [1, 2, 1], [2], [2, 1]]

sage: c2 = W.from_reduced_word([2])

sage: sorted(w.reduced_word() for w in W.noncrossing_partition_lattice(c2))
[[], [2]]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> W = SymmetricGroup(Integer(4)) \# needs sage.groups

>>> W.noncrossing_partition_lattice() \# needs sage.combinat sage.groups
Finite lattice containing 14 elements

>>> W = WeylGroup(['G', Integer(2)]) \# needs sage.combinat sage.groups

>>> W.noncrossing_partition_lattice() \# needs sage.combinat sage.groups
\end{verbatim}

(continues on next page)
example()

Return an example of an irreducible complex reflection group.

EXAMPLES:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().Irreducible()
sage: C.example()  # optional - gap3
Irreducible complex reflection group of rank 3 and type G(4,2,3)
```

class ParentMethods

Bases: object

```python
base_change_matrix()
```

Return the base change from the standard basis of the vector space of \( \text{self} \) to the basis given by the independent roots of \( \text{self} \).

Todo: For non-well-generated groups there is a conflict with construction of the matrix for an element.

EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[1 0]
[0 1]
```

4.44. Finite Complex Reflection Groups 581
sage: W = ReflectionGroup((3,1,2))  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[1 0]
[1 1]

sage: W = ReflectionGroup((4,2,2))  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[1 0]
[E(4) 1]

>>> from sage.all import *
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))  #...
˓→optional - gap3
>>> W.base_change_matrix()  # optional - gap3
[1 0]
[0 1]

>>> W = ReflectionGroup(Integer(23))  # optional - gap3
>>> W.base_change_matrix()  # optional - gap3
[1 0 0]
[0 1 0]
[0 0 1]

>>> W = ReflectionGroup((Integer(3),Integer(1),Integer(2)))  #...
˓→optional - gap3
>>> W.base_change_matrix()  # optional - gap3
[1 0]
[1 1]

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(2)))  #...
˓→optional - gap3
>>> W.base_change_matrix()  # optional - gap3
[1 0]
[E(4) 1]

\textbf{cardinality}()

Return the cardinality of \texttt{self}.

It is given by the product of the degrees of \texttt{self}.

\textbf{EXAMPLES}:

sage: # needs sage.combinat sage.groups
sage: W = ColoredPermutations(1,3)
sage: W.cardinality()
6
sage: W = ColoredPermutations(2,3)
sage: W.cardinality()
48
sage: W = ColoredPermutations(4,3)
sage: W.cardinality()
384

sage: # optional - gap3, needs sage.combinat sage.groups
sage: W = ReflectionGroup((4,2,3))
```python
sage: W.cardinality()
192

>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = ColoredPermutations(Integer(1),Integer(3))
>>> W.cardinality()
6
>>> W = ColoredPermutations(Integer(2),Integer(3))
>>> W.cardinality()
48
>>> W = ColoredPermutations(Integer(4),Integer(3))
>>> W.cardinality()
384

>>> # optional - gap3, needs sage.combinat sage.groups
>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(3)))
>>> W.cardinality()
192

codegrees()

Return the codegrees of self.

OUTPUT: a tuple of Sage integers

EXAMPLES:

```n
```
```
degrees()

Return the degrees of self.

OUTPUT: a tuple of Sage integers

EXAMPLES:

```sage
sage: W = ColoredPermutations(1, 4)  # needs sage.combinat
sage: W.degrees()  # needs sage.combinat
(2, 3, 4)

sage: W = ColoredPermutations(3, 3)  # needs sage.combinat
sage: W.degrees()  # needs sage.combinat
(3, 6, 9)

sage: W = ReflectionGroup(31)  # optional - gap3
sage: W.degrees()  # optional - gap3
(8, 12, 20, 24)
```

is_real()

Return whether self is real.

A complex reflection group is real if it is isomorphic to a reflection group in \(GL(V)\) over a real vector space \(V\). Equivalently its character table has real entries.

This implementation uses the following statement: an irreducible complex reflection group is real if and only if 2 is a degree of self with multiplicity one. Hence, in general we just need to compare the number of occurrences of 2 as degree of self and the number of irreducible components.

EXAMPLES:

```sage
>>> from sage.all import *

>>> W = ColoredPermutations(Integer(1), Integer(4))  # needs sage.combinat
>>> W.degrees()  # needs sage.combinat
(2, 3, 4)

>>> W = ColoredPermutations(Integer(3), Integer(3))  # needs sage.combinat
>>> W.degrees()  # needs sage.combinat
(3, 6, 9)

>>> W = ReflectionGroup(Integer(31))  # optional - gap3
>>> W.degrees()  # optional - gap3
(8, 12, 20, 24)
```
Todo: Add an example of non-real finite complex reflection group that is generated by order 2 reflections.

is_well_generated()

Return whether self is well-generated.

A finite complex reflection group is well generated if the number of its simple reflections coincides with its rank.

See also:

ComplexReflectionGroups.Finite.WellGenerated()

Note:

• All finite real reflection groups are well generated.
• The complex reflection groups of type $G(r, 1, n)$ and of type $G(r, r, n)$ are well generated.
• The complex reflection groups of type $G(r, p, n)$ with $1 < p < r$ are not well generated.
• The direct product of two well generated finite complex reflection group is still well generated.

EXAMPLES:

sage: W = ColoredPermutations(1,3)  # needs sage.combinat
sage: W.is_well_generated()  # needs sage.combinat
True

sage: W = ColoredPermutations(4,3)  # needs sage.combinat
sage: W.is_well_generated()  # needs sage.combinat
False
The Milnor fiber poset of a finite complex reflection group \( W \) is defined as the poset of (right) standard cosets \( gWJ \), where \( J \) is a subset of the index set \( I \) of \( W \), ordered by reverse inclusion. This is conjecturally a meet semilattice if and only if \( W \) is well-generated.

\begin{verbatim}
>>> from sage.all import *
>>> W = ColoredPermutations(Integer(1),Integer(3))
                      # needs sage.combinat
>>> W.is_well_generated()
                      # needs sage.combinat
True

>>> W = ColoredPermutations(Integer(4),Integer(3))
                      # needs sage.combinat
>>> W.is_well_generated()
                      # needs sage.combinat
True

>>> W = ReflectionGroup(((Integer(4),Integer(2),Integer(3)))
>>> W.is_well_generated()
False

>>> W = ReflectionGroup(((Integer(4),Integer(4),Integer(3)))
>>> W.is_well_generated()
True
\end{verbatim}

\section*{milnor_fiber_poset()} Return the Milnor fiber poset of \texttt{self}.

\begin{verbatim}
sage: # needs sage.combinat
sage: W = ColoredPermutations(3, 2)
sage: P = W.milnor_fiber_poset()
sage: P
Finite meet-semilattice containing 34 elements

sage: R.<x> = ZZ[]
sage: sum(x**P.rank(elt) \texttt{ for elt in P})
18*x^2 + 15*x + 1

sage: # optional - gap3
sage: W = ReflectionGroup(4)
sage: P = W.milnor_fiber_poset(); P
Finite meet-semilattice containing 41 elements

sage: sum(x**P.rank(elt) \texttt{ for elt in P})
24*x^2 + 16*x + 1
\end{verbatim}
sage: # optional - gap3
sage: W = ReflectionGroup([4,2,2])

sage: W.is_well_generated()
False

sage: P = W.milnor_fiber_poset(); P
Finite poset containing 47 elements

sage: sum(x**P.rank(elt) for elt in P)
16*x^3 + 24*x^2 + 6*x + 1

sage: P.is_meet_semilattice()
False

>>> from sage.all import *

>>> # needs sage.combinat

>>> W = ColoredPermutations(Integer(3), Integer(2))

>>> P = W.milnor_fiber_poset()

>>> P
Finite meet-semilattice containing 34 elements

>>> R = ZZ['x']; (x,) = R._first_ngens(1)

>>> sum(x**P.rank(elt) for elt in P)
18*x^2 + 15*x + 1

>>> # optional - gap3

>>> W = ReflectionGroup(Integer(4))

>>> P = W.milnor_fiber_poset(); P
Finite meet-semilattice containing 41 elements

>>> sum(x**P.rank(elt) for elt in P)
24*x^2 + 16*x + 1

>>> # optional - gap3

>>> W = ReflectionGroup([Integer(4),Integer(2),Integer(2)])

>>> W.is_well_generated()
False

>>> P = W.milnor_fiber_poset(); P
Finite poset containing 47 elements

>>> sum(x**P.rank(elt) for elt in P)
16*x^3 + 24*x^2 + 6*x + 1

>>> P.is_meet_semilattice()
False

number_of_reflection_hyperplanes()

Return the number of reflection hyperplanes of self.

This is also the number of distinguished reflections. For real groups, this coincides with the number of reflections.

This implementation uses that it is given by the sum of the codegrees of self plus its rank.

See also:

number_of_reflections()

EXAMPLES:

sage: # needs sage.combinat

sage: W = ColoredPermutations(1,3)

sage: W.number_of_reflection_hyperplanes()
3

sage: W = ColoredPermutations(2,3)
number_of_reflections()

Return the number of reflections of self.

For real groups, this coincides with the number of reflection hyperplanes.

This implementation uses that it is given by the sum of the degrees of self minus its rank.

See also:

number_of_reflection_hyperplanes()

EXAMPLES:

sage: [SymmetricGroup(i).number_of_reflections() for i in range(int(8))]  # needs sage.groups sage.rings.number_field
[0, 0, 1, 3, 6, 10, 15, 21]

sage: W = ColoredPermutations(1,3)
sage: W.number_of_reflections()  # needs sage.combinat sage.groups
3

sage: W = ColoredPermutations(2,3)
sage: W.number_of_reflections()  # needs sage.combinat sage.groups
9

sage: W = ColoredPermutations(4,3)
sage: W.number_of_reflections()  # needs sage.combinat sage.groups
21

sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.number_of_reflections()  # optional - gap3
15

```python
>>> from sage.all import *

>>> [SymmetricGroup(i).number_of_reflections() for i in range(int(Integer(8)))]
[0, 0, 1, 3, 6, 10, 15, 21]
```

```python
>>> # needs sage.combinat sage.groups
>>> W = ColoredPermutations(Integer(1),Integer(3))

>>> W.number_of_reflections()
3

>>> W = ColoredPermutations(Integer(2),Integer(3))

>>> W.number_of_reflections()
9

>>> W = ColoredPermutations(Integer(4),Integer(3))

>>> W.number_of_reflections()
21

>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(3)))  # optional - gap3

>>> W.number_of_reflections()  # optional - gap3
15
```

**rank()**

Return the rank of self.

The rank of self is the dimension of the smallest faithful reflection representation of self.

This default implementation uses that the rank is the number of degrees().

**See also:**

ComplexReflectionGroups.rank()

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups
sage: W = ColoredPermutations(1,3)

sage: W.rank()
2
```

```python
sage: W = ColoredPermutations(2,3)

sage: W.rank()
3
```

```python
sage: W = ColoredPermutations(4,3)

sage: W.rank()
3
```

```python
sage: # optional - gap3, needs sage.combinat sage.groups
sage: W = ReflectionGroup((4,2,3))

sage: W.rank()
3
```

```python
>>> from sage.all import *

>>> # needs sage.combinat sage.groups

>>> W = ColoredPermutations(Integer(1),Integer(3))

>>> W.rank()
```

(continues on next page)
2
>>> W = ColoredPermutations(Integer(2),Integer(3))
>>> W.rank()
3
>>> W = ColoredPermutations(Integer(4),Integer(3))
>>> W.rank()
3
>>> # optional - gap3, needs sage.combinat sage.groups
>>> W = ReflectionGroup((Integer(4),Integer(2),Integer(3)))
>>> W.rank()
3

class SubcategoryMethods
    Bases: object

    WellGenerated()

        Return the full subcategory of well-generated objects of self.

        A finite complex generated group is well generated if it is isomorphic to a subgroup of the general linear group \( GL_n \) generated by \( n \) reflections.

        See also:

        ComplexReflectionGroups.Finite.ParentMethods.is_well_generated()

    EXAMPLES:

    sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
    sage: C = ComplexReflectionGroups().Finite().WellGenerated(); C
    Category of well generated finite complex reflection groups

    >>> from sage.all import *
    >>> from sage.categories.complex_reflection_groups import ComplexReflectionGroups
    >>> C = ComplexReflectionGroups().Finite().WellGenerated(); C
    Category of well generated finite complex reflection groups

    Here is an example of a finite well-generated complex reflection group:

    sage: W = C.example(); W
    # optional - gap3
    Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)

    All finite Coxeter groups are well generated:

    sage: CoxeterGroups().Finite().is_subcategory(C)
    True
    sage: SymmetricGroup(3) in C
    #-- needs sage.groups
    True
>> from sage.all import *
>>> CoxeterGroups().Finite().is_subcategory(C)
True
>>> SymmetricGroup(Integer(3)) in C
... # needs sage.groups
True

Note: The category of well generated finite complex reflection groups is currently implemented as an axiom. See discussion on Issue #11187. This may be a bit of overkill. Still it’s nice to have a full subcategory.

class WellGenerated(base_category)
    Bases: CategoryWithAxiom

class Irreducible(base_category)
    Bases: CategoryWithAxiom

The category of finite irreducible well-generated finite complex reflection groups.

class ParentMethods
    Bases: object

catalan_number(positive=False, polynomial=False)
    Return the Catalan number associated to self.
    It is defined by
    \[ \prod_{i=1}^{n} \frac{d_i + h}{d_i}, \]
    where \( d_1, \ldots, d_n \) are the degrees and where \( h \) is the Coxeter number. See [Ar2006] for further information.

    INPUT:
    • positive – optional boolean (default False) if True, return instead the positive Catalan number
    • polynomial – optional boolean (default False) if True, return instead the \( q \)-analogue as a polynomial in \( q \)

    Note:
    • For the symmetric group \( S_n \), it reduces to the Catalan number \( \frac{1}{n+1} \binom{2n}{n} \).
    • The Catalan numbers for \( G(r, 1, n) \) all coincide for \( r > 1 \).

EXAMPLES:

sage: [ColoredPermutations(1,n).catalan_number() #
        # needs sage.combinat
        ....: for n in [3,4,5]]
[5, 14, 42]

sage: [ColoredPermutations(2,n).catalan_number() #
        # needs sage.combinat
        ....: for n in [3,4,5]]
[20, 70, 252]
sage: [ReflectionGroup((2,2,n)).catalan_number()  # optional - gap3
....: for n in [3,4,5]]
[14, 50, 182]

>>> from sage.all import *
>>> [ColoredPermutations(Integer(1),n).catalan_number()  # needs sage.combinat
... for n in [3,4,5]]
[5, 14, 42]

>>> [ColoredPermutations(Integer(2),n).catalan_number()  # needs sage.combinat
... for n in [3,4,5]]
[20, 70, 252]

>>> [ReflectionGroup((Integer(2),Integer(2),n)).catalan_number()  # optional - gap3
... for n in [3,4,5]]
[14, 50, 182]

coxeter_number()

Return the Coxeter number of a well-generated, irreducible reflection group. This is defined to be the order of a regular element in \texttt{self}, and is equal to the highest degree of \texttt{self}.

See also:
ComplexReflectionGroups.Finite.Irreducible()

Note: This method overwrites the more general method for complex reflection groups since the expression given here is quicker to compute.

EXAMPLES:

sage: W = ColoredPermutations(1,3)  # needs sage.combinat
sage: W.coxeter_number()  # needs sage.combinat
3

sage: W = ColoredPermutations(4,3)  # needs sage.combinat
sage: W.coxeter_number()  # needs sage.combinat
12

sage: W = ReflectionGroup((4,4,3))  # optional - gap3
sage: W.coxeter_number()  # optional - gap3
8

>>> from sage.all import *
>>> W = ColoredPermutations(Integer(1),Integer(3))  # needs sage.combinat
>>> W.coxeter_number()  # needs sage.combinat
>>> W = ColoredPermutations(Integer(4), Integer(3))  # needs sage.combinat
>>> W.coxeter_number()  # needs sage.combinat
12

>>> W = ReflectionGroup((Integer(4), Integer(4), Integer(3)))  # optional - gap3
>>> W.coxeter_number()  # optional - gap3
8

\textbf{fuss\_catalan\_number}(m, positive=False, polynomial=False)

Return the $m$-th Fuss-Catalan number associated to \texttt{self}.

This is defined by
\[
\prod_{i=1}^{n} \frac{d_i + mh}{d_i},
\]
where $d_1, \ldots, d_n$ are the degrees and $h$ is the Coxeter number.

**INPUT:**
- \texttt{positive} – optional boolean (default False) if True, return instead the positive Fuss-Catalan number
- \texttt{polynomial} – optional boolean (default False) if True, return instead the $q$-analogue as a polynomial in $q$

See [Ar2006] for further information.

**Note:**
- For the symmetric group $S_n$, it reduces to the Fuss-Catalan number $\frac{1}{mn+1} \binom{(m+1)n}{n}$.
- The Fuss-Catalan numbers for $G(r, 1, n)$ all coincide for $r > 1$.

**EXAMPLES:**

\begin{verbatim}
sage: W = ColoredPermutations(1,3)  # needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[5, 12, 22]
sage: W = ColoredPermutations(1,4)  # needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[14, 55, 140]
sage: W = ColoredPermutations(1,5)  # needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[42, 273, 969]
sage: W = ColoredPermutations(2,2)  # needs sage.combinat
\end{verbatim}
needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[6, 15, 28]
sage: W = ColoredPermutations(2,3)  # needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[20, 84, 220]
sage: W = ColoredPermutations(2,4)  # needs sage.combinat
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]  # needs sage.combinat
[70, 495, 1820]

from sage.all import *
>>> W = ColoredPermutations(Integer(1),Integer(3))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[5, 12, 22]
>>> W = ColoredPermutations(Integer(1),Integer(4))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[14, 55, 140]
>>> W = ColoredPermutations(Integer(1),Integer(5))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[42, 273, 969]

>>> W = ColoredPermutations(Integer(2),Integer(2))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[6, 15, 28]

>>> W = ColoredPermutations(Integer(2),Integer(3))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[20, 84, 220]

>>> W = ColoredPermutations(Integer(2),Integer(4))  # needs sage.combinat
>>> [W.fuss_catalan_number(i) for i in [Integer(1),Integer(2),
Integer(3)]]  # needs sage.combinat
[70, 495, 1820]

number_of_reflections_of_full_support()

Return the number of reflections with full support.
EXAMPLES:

```python
sage: W = Permutations(4)
sage: W.number_of_reflections_of_full_support()
1

sage: W = ColoredPermutations(1,4)  # needs sage.combinat
sage: W.number_of_reflections_of_full_support()
1

sage: W = CoxeterGroup("B3")  # needs sage.combinat sage.groups
sage: W.number_of_reflections_of_full_support()  # needs sage.combinat sage.groups
3

sage: W = ColoredPermutations(3,3)  # needs sage.combinat
sage: W.number_of_reflections_of_full_support()  # needs sage.combinat
3

>>> from sage.all import *
>>> W = Permutations(Integer(4))
>>> W.number_of_reflections_of_full_support()
1

>>> W = ColoredPermutations(Integer(1),Integer(4))  # needs sage.combinat
>>> W.number_of_reflections_of_full_support()
1

>>> W = CoxeterGroup("B3")  # needs sage.combinat sage.groups
>>> W.number_of_reflections_of_full_support()  # needs sage.combinat sage.groups
3

>>> W = ColoredPermutations(Integer(3),Integer(3))  # needs sage.combinat
>>> W.number_of_reflections_of_full_support()  # needs sage.combinat
3```

`rational_catalan_number(p, polynomial=False)`

Return the $p$-th rational Catalan number associated to `self`.

It is defined by

\[
\prod_{i=1}^{n} \frac{p + (p(d_i - 1)) \mod h}{d_i},
\]

where $d_1, \ldots, d_n$ are the degrees and $h$ is the Coxeter number. See [STW2016] for this formula.

**INPUT:**

- `polynomial` – optional boolean (default `False`) if `True`, return instead the $q$-analogue as a polynomial in $q
**EXAMPLES:**

```
sage: W = ColoredPermutations(1,3)  # needs sage.combinat
sage: [W.rational_catalan_number(p) for p in [5,7,8]]  # needs sage.combinat
[7, 12, 15]
sage: W = ColoredPermutations(2,2)  # needs sage.combinat
sage: [W.rational_catalan_number(p) for p in [7,9,11]]  # needs sage.combinat
[10, 15, 21]
```

```
>>> from sage.all import *
>>> W = ColoredPermutations(Integer(1),Integer(3))  # needs sage.combinat
>>> [W.rational_catalan_number(p) for p in [Integer(5),Integer(7), Integer(8)]]  # needs sage.combinat
[7, 12, 15]
>>> W = ColoredPermutations(Integer(2),Integer(2))  # needs sage.combinat
>>> [W.rational_catalan_number(p) for p in [Integer(7),Integer(9), Integer(11)]]  # needs sage.combinat
[10, 15, 21]
```

### example()

Return an example of an irreducible well-generated complex reflection group.

**EXAMPLES:**

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated().Irreducible()
sage: C.example()  # needs sage.combinat
4-colored permutations of size 3
```

### class ParentMethods

Bases: object

```
coxeter_element()  # needs sage.combinat
```

Return a Coxeter element.

The result is the product of the simple reflections, in some order.

**Note:** This implementation is shared with well generated complex reflection groups. It would be...
nearer to put it in some joint super category; however, in the current state of the art, there is none
where it is clear that this is the right construction for obtaining a Coxeter element.

In this context, this is an element having a regular eigenvector (a vector not contained in any reflection
hyperplane of self).

EXAMPLES:

```sage
# needs sage.combinat sage.groups
sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()
[1, 2, 4, 3]
sage: CoxeterGroup(['F', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['E', 8]).coxeter_element().reduced_word()
[1, 3, 2, 4, 5, 6, 7, 8]
sage: CoxeterGroup(['H', 3]).coxeter_element().reduced_word()
[1, 2, 3]
```

```python
>>> from sage.all import *

>>> CoxeterGroup(['A', Integer(4)]).coxeter_element().reduced_word()
[1, 2, 3, 4]
>>> CoxeterGroup(['B', Integer(4)]).coxeter_element().reduced_word()
[1, 2, 3, 4]
>>> CoxeterGroup(['D', Integer(4)]).coxeter_element().reduced_word()
[1, 2, 4, 3]
>>> CoxeterGroup(['F', Integer(4)]).coxeter_element().reduced_word()
[1, 2, 3, 4]
>>> CoxeterGroup(['E', Integer(8)]).coxeter_element().reduced_word()
[1, 3, 2, 4, 5, 6, 7, 8]
>>> CoxeterGroup(['H', Integer(3)]).coxeter_element().reduced_word()
[1, 2, 3]
```

This method is also used for well generated finite complex reflection groups:

```sage
W = ReflectionGroup((1,1,4))
W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3]
W = ReflectionGroup((2,1,4))
W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3]
W = ReflectionGroup((4,1,4))
W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3]
W = ReflectionGroup((4,4,4))
W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3]
```

```python
>>> from sage.all import *

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))

(continues on next page)
\begin{verbatim}
>>> W.coxeter_element().reduced_word()   # optional - gap3
[1, 2, 3]

>>> W = ReflectionGroup((Integer(2),Integer(1),Integer(4)))  # optional - gap3
>>> W.coxeter_element().reduced_word()   # optional - gap3
[1, 2, 3, 4]

>>> W = ReflectionGroup((Integer(4),Integer(1),Integer(4)))  # optional - gap3
>>> W.coxeter_element().reduced_word()   # optional - gap3
[1, 2, 3, 4]

>>> W = ReflectionGroup((Integer(4),Integer(4),Integer(4)))  # optional - gap3
>>> W.coxeter_element().reduced_word()   # optional - gap3
[1, 2, 3, 4]
\end{verbatim}

**coxeter_elements()**

Return the (unique) conjugacy class in self containing all Coxeter elements.

A Coxeter element is an element that has an eigenvalue $e^{2\pi i/h}$ where $h$ is the Coxeter number.

In case of finite Coxeter groups, these are exactly the elements that are conjugate to one (or, equivalently, all) standard Coxeter element, this is, to an element that is the product of the simple generators in some order.

**See also:**

`standard_coxeter_elements()`

**EXAMPLES:**

```sage
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: sorted(c.reduced_word()         # optional - gap3
....:     for c in W.coxeter_elements())
[[1, 2], [2, 1]]

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: sorted(c.reduced_word()         # optional - gap3
....:     for c in W.coxeter_elements())
[[1, 2, 1, 3, 2], [1, 2, 3], [1, 3, 2],
 [2, 1, 3], [2, 1, 3, 2, 1], [3, 2, 1]]
```

```sage
>>> from sage.all import *    
>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(3)))  # optional - gap3
>>> sorted(c.reduced_word()         # optional - gap3
... for c in W.coxeter_elements())
[[1, 2], [2, 1]]

>>> W = ReflectionGroup((Integer(1),Integer(1),Integer(4)))  # optional - gap3
>>> sorted(c.reduced_word()         # optional - gap3
... for c in W.coxeter_elements())
[[1, 2, 1, 3, 2], [1, 2, 3], [1, 3, 2],
 [2, 1, 3], [2, 1, 3, 2, 1], [3, 2, 1]]
```
is_well_generated()

Return True as self is well-generated.

EXAMPLES:

sage: W = ReflectionGroup((3,1,2)) # optional - gap3
sage: W.is_well_generated() # optional - gap3
True

milnor_fiber_complex()

Return the Milnor fiber complex of self.

The Milnor fiber complex of a finite well-generated complex reflection group \( W \) is the simplicial complex whose face poset is given by \( \text{milnor_fiber_poset}() \). When \( W \) is an irreducible Shephard group, it is also an equivariant strong deformation retract of the Milnor fiber \( f_1^{-1}(1) \), where \( f_1 : V \to \mathbb{C} \) is the polynomial invariant of smallest degree acting on the reflection representation \( V \).

When \( W \) is a Coxeter group, this is isomorphic to the Coxeter complex of \( W \).

EXAMPLES:

sage: W = ColoredPermutations(3, 2) # needs sage.combinat
sage: C = W.milnor_fiber_complex() # needs sage.combinat
sage: C.homology() # needs sage.combinat
{0: 0, 1: \( \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \)}

sage: W = ReflectionGroup(5) # optional - gap3
sage: C = W.milnor_fiber_complex() # optional - gap3
sage: C.homology() # optional - gap3
{0: 0, 1: \( \mathbb{Z}^{25} \)}

standard_coxeter_elements()

Return all standard Coxeter elements in self.

This is the set of all elements in self obtained from any product of the simple reflections in self.

4.4. Finite Complex Reflection Groups 599
Note:
• self is assumed to be well-generated.
• This works even beyond real reflection groups, but the conjugacy class is not unique and we only obtain one such class.

EXAMPLES:

```python
# optional - gap3
W = ReflectionGroup(4)
sage: sorted(W.standard_coxeter_elements())  # optional - gap3
[(1,7,6,12,23,20)(2,8,17,24,9,5)(3,16,10,19,15,21)(4,14,11,22,18,13),
 (1,10,4,12,21,22)(2,11,19,24,13,3)(5,15,7,17,16,23)(6,18,8,20,14,9)]
```

```python
# optional - gap3
W = ReflectionGroup(Integer(4))
sage: sorted(W.standard_coxeter_elements())  # optional - gap3
[(1,7,6,12,23,20)(2,8,17,24,9,5)(3,16,10,19,15,21)(4,14,11,22,18,13),
 (1,10,4,12,21,22)(2,11,19,24,13,3)(5,15,7,17,16,23)(6,18,8,20,14,9)]
```

```python
from sage.all import *
>>> from sage.categories.complex_reflection_groups import ComplexReflectionGroups
>>> C = ComplexReflectionGroups().Finite().WellGenerated()
>>> C.example()  # optional - gap3
Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)
```

```python
# optional - gap3
C = ComplexReflectionGroups().Finite().WellGenerated()
>>> C.example()  # optional - gap3
Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)
```

```python
# optional - gap3
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated()
```

```python
# optional - gap3
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```

```python
# optional - gap3
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```

EXAMPLES:

```python
# optional - gap3
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated()
sage: C.example()  # optional - gap3
Reducible complex reflection group of rank 4 and type A2 x B2
```

```python
# optional - gap3
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```

```python
# optional - gap3
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```

example() method:

Return an example of a well-generated complex reflection group.

EXAMPLES:

```python
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated()
sage: C.example()  # optional - gap3
Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)
```

example() method:

Return an example of a complex reflection group.

EXAMPLES:

```python
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite().example()  # optional - gap3
Reducible real reflection group of rank 4 and type A2 x B2
```

```python
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```

```python
from sage.categories.complex_reflection_groups import ComplexReflectionGroups
```
4.45 Finite Coxeter Groups

class sage.categories.finite_coxeter_groups.FiniteCoxeterGroups(base_category)

Bases: CategoryWithAxiom

The category of finite Coxeter groups.

EXAMPLES:

sage: CoxeterGroups().Finite()
Category of finite Coxeter groups
sage: FiniteCoxeterGroups().super_categories()
[Category of finite generalized Coxeter groups,  
 Category of Coxeter groups]

sage: G = CoxeterGroups().Finite().example()
sage: G.cayley_graph(side = "right").plot()
Graphics object consisting of 40 graphics primitives

Here are some further examples:

sage: WeylGroups().Finite().example()
The symmetric group on {0, ..., 3}
sage: WeylGroup(['B', 3])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)

Those other examples will eventually be also in this category:

sage: SymmetricGroup(4)
Symmetric group of order 4! as a permutation group
sage: DihedralGroup(5)
Dihedral group of order 10 as a permutation group

sage: SymmetricGroup(Integer(4))
Symmetric group of order 4! as a permutation group
sage: DihedralGroup(Integer(5))
Dihedral group of order 10 as a permutation group
class ElementMethods

    Bases: object

    absolute_length()

    Return the absolute length of self.

    The absolute length is the length of the shortest expression of the element as a product of reflections.
    For finite Coxeter groups, the absolute length is the codimension of the 1-eigenspace of the element
    (Lemmas 1-3 in [Car1972a]).

    For permutations in the symmetric groups, the absolute length is the size minus the number of its disjoint
    cycles.

    See also:

    absolute_le()  # See also absolute_le()

    EXAMPLES:

    sage: W = WeylGroup( ['A', 3] )  # needs sage.combinat sage.groups
    sage: s = W.simple_reflections()  # needs sage.combinat sage.groups
    sage: (s[1]*s[2]*s[3]).absolute_length()  # needs sage.combinat sage.groups
    3
    sage: W = SymmetricGroup(4)  # needs sage.groups
    sage: s = W.simple_reflections()  # needs sage.groups
    sage: (s[3]*s[2]*s[1]).absolute_length()  # needs sage.combinat sage.groups
    3

    >>> from sage.all import *
    >>> W = WeylGroup( ['A', Integer(3)] )  # needs sage.combinat sage.groups
    >>> s = W.simple_reflections()  # needs sage.combinat sage.groups
    >>> (s[Integer(1)]*s[Integer(2)]*s[Integer(3)]).absolute_length()  # needs sage.combinat sage.groups
    3
    >>> W = SymmetricGroup(Integer(4))  # needs sage.groups
    >>> s = W.simple_reflections()  # needs sage.groups
    >>> (s[Integer(3)]*s[Integer(2)]*s[Integer(1)]).absolute_length()  # needs sage.combinat sage.groups
    3

    bruhat_upper_covers()

    Returns all the elements that cover self in Bruhat order.

    EXAMPLES:

    sage: w = W.from_reduced_word([3,2])

    (continues on next page)
Recursive algorithm: write $w$ for self. If $i$ is a non-descent of $w$, then the covers of $w$ are exactly \{ $wsi$, $u_1si$, $u_2si$, ..., $u_jsi$ \}, where the $u_k$ are those covers of $wsi$ that have a descent at $i$.

**covered_reflections_subgroup()**

Return the subgroup of $W$ generated by the conjugates by $w$ of the simple reflections indexed by right descents of $w$.

This is used to compute the shard intersection order on $W$.

**EXAMPLES:**
Return the Coxeter-Knuth graph of type $A$.

The Coxeter-Knuth graph of type $A$ is generated by the Coxeter-Knuth relations which are given by $aa + 1a \sim a + 1aa + 1, abc \sim acb$ if $b < a < c$ and $abc \sim bac$ if $a < c < b$.

EXAMPLES:

```python
>>> from sage.all import *
>>> W = CoxeterGroup(["A", Integer(3)], base_ring=ZZ)
>>> len(W.long_element().covered_reflections_subgroup())
24
>>> s = W.simple_reflection(Integer(1))
>>> Gs = s.covered_reflections_subgroup()
>>> len(Gs)
2
>>> s in [u.lift() for u in Gs]
True
>>> len(W.one().covered_reflections_subgroup())
1
```
```python
>>> w = W.from_reduced_word([Integer(1), Integer(2), Integer(1), Integer(3), Integer(2)])
>>> D = w.coxeter_knuth_graph()
>>> D.vertices(sort=True)
[(1, 2, 1, 3, 2),
 (1, 2, 3, 1, 2),
 (2, 1, 2, 3, 2),
 (2, 1, 3, 2, 3),
 (2, 3, 1, 2, 3)]
>>> D.edges(sort=True)
[((1, 2, 1, 3, 2), (1, 2, 3, 1, 2), None),
 ((1, 2, 1, 3, 2), (2, 1, 2, 3, 2), None),
 ((2, 1, 2, 3, 2), (2, 1, 3, 2, 3), None),
 ((2, 1, 3, 2, 3), (2, 3, 1, 2, 3), None)]
```

```python
>>> w = W.from_reduced_word([Integer(1), Integer(3)])
>>> D = w.coxeter_knuth_graph()
>>> D.vertices(sort=True)
[(1, 3), (3, 1)]
>>> D.edges(sort=False)
[]
```

\[\text{coxeter_knuth_neighbor}(w)\]

Return the Coxeter-Knuth (oriented) neighbors of the reduced word \(w\) of \(\text{self}\).

**INPUT:**

- \(w\) – reduced word of \(\text{self}\)

The Coxeter-Knuth relations are given by \(aa + 1a \sim a + 1aa + 1\), \(abc \sim acb\) if \(b < a < c\) and \(abc \sim bac\) if \(a < c < b\). This method returns all neighbors of \(w\) under the Coxeter-Knuth relations oriented from left to right.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',4], prefix='s')
sage: word = [1,2,1,3,2]
sage: w = W.from_reduced_word(word)
sage: w.coxeter_knuth_neighbor(word)
{(1, 2, 3, 1, 2), (2, 1, 2, 3, 2)}
```

```python
sage: word = [1,2,1,3,2,4,3]
sage: w = W.from_reduced_word(word)
sage: w.coxeter_knuth_neighbor(word)
{(1, 2, 1, 3, 4, 2, 3), (1, 2, 3, 1, 2, 4, 3), (2, 1, 2, 3, 2, 4, 3)}
```

```python
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(4)], prefix='s')
>>> word = [Integer(1), Integer(2), Integer(1), Integer(3), Integer(2), Integer(4), Integer(3)]
>>> w = W.from_reduced_word(word)
>>> w.coxeter_knuth_neighbor(word)
{(1, 2, 1, 3, 4, 2, 3), (1, 2, 3, 1, 2, 4, 3), (2, 1, 2, 3, 2, 4, 3)}
```
**is_coxeter_element()**

Return whether this is a Coxeter element.

This is, whether `self` has an eigenvalue \( e^{2\pi i/h} \) where \( h \) is the Coxeter number.

See also:

`coxeter_elements()`

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A',2])
sage: c = prod(W.gens())
sage: c.is_coxeter_element()
True
sage: W.one().is_coxeter_element()
False

sage: W = WeylGroup(['G', 2])
sage: c = prod(W.gens())
sage: c.is_coxeter_element()
True
sage: W.one().is_coxeter_element()
False
```

```python
>>> from sage.all import *
>>> W = CoxeterGroup(['A',Integer(2)])
>>> c = prod(W.gens())
>>> c.is_coxeter_element()
True
>>> W.one().is_coxeter_element()
False

>>> W = WeylGroup(['G', Integer(2)])
>>> c = prod(W.gens())
>>> c.is_coxeter_element()
True
>>> W.one().is_coxeter_element()
False
```

**class ParentMethods**

Bases: object

Ambiguity resolution: the implementation of `some_elements` is preferable to that of `FiniteGroups`. The same holds for `__iter__`, although a breadth first search would be more natural; at least this maintains backward compatibility after Issue #13589.

**bhz_poset()**

Return the Bergeron-Hohlweg-Zabrocki partial order on the Coxeter group.

This is a partial order on the elements of a finite Coxeter group \( W \), which is distinct from the Bruhat order, the weak order and the shard intersection order. It was defined in [BHZ2005].

This partial order is not a lattice, as there is no unique maximal element. It can be succinctly defined as follows.

Let \( u \) and \( v \) be two elements of the Coxeter group \( W \). Let \( S(u) \) be the support of \( u \). Then \( u \preceq v \) if and only if \( v_{S(u)} = u \) (here \( v = v'l'v_l \) denotes the usual parabolic decomposition with respect to the standard parabolic subgroup \( W_I \)).
See also:

\texttt{bruhat\_poset()}, \texttt{shard\_poset()}, \texttt{weak\_poset()}

EXAMPLES:

\begin{verbatim}
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: P = W.bhz_poset(); P
Finite poset containing 24 elements
sage: P.relations_number()
103
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: len(P.maximal_elements())
13
sage: from sage.all import *
sage: W = CoxeterGroup(['A',Integer(3)], base_ring=ZZ)
sage: P = W.bhz_poset(); P
Finite poset containing 24 elements
sage: P.relations_number()
103
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: len(P.maximal_elements())
13
\end{verbatim}

\texttt{bruhat\_poset} (\texttt{facade=False})

Return the Bruhat poset of \texttt{self}.

See also:

\texttt{bhz\_poset()}, \texttt{shard\_poset()}, \texttt{weak\_poset()}

EXAMPLES:

\begin{verbatim}
sage: W = WeylGroup(['A', 2])
sage: P = W.bruhat_poset()
sage: P
Finite poset containing 6 elements
sage: P.show()

>>> from sage.all import *

>>> W = WeylGroup(['A', Integer(2)])

>>> P = W.bruhat_poset()

>>> P
Finite poset containing 6 elements

>>> P.show()

Here are some typical operations on this poset:

\begin{verbatim}
sage: W = WeylGroup(['A', 3])
sage: P = W.bruhat_poset()
sage: u = W.from_reduced_word([3,1])
sage: v = W.from_reduced_word([3,2,1,2,3])
sage: P(u) <= P(v)
True
sage: len(P.interval(P(u), P(v)))
10
\end{verbatim}

(continues on next page)
By default, the elements of $P$ are aware that they belong to $P$:

```sage
sage: P.an_element().parent()
Finite poset containing 24 elements
```

If instead one wants the elements to be plain elements of the Coxeter group, one can use the `facade` option:

```sage
sage: P = W.bruhat_poset(facade=True)
```

See also:

`Poset()` for more on posets and facade posets.

Todo:
- Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
- The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.

```
cambrian_lattice(c, on_roots=False)
```

Return the $c$-Cambrian lattice on delta sequences.


Delta sequences are certain 2-colored minimal factorizations of $c$ into reflections.

INPUT:
• c – a standard Coxeter element in self (as a tuple, or as an element of self)
• on_roots (default: False) – if on_roots is True, the lattice is realized on roots rather than on reflections. In order for this to work, the ElementMethod reflection_to_root must be available.

EXAMPLES:

```sage
sage: CoxeterGroup(['A', 2]).cambrian_lattice((1,2))
Finite lattice containing 5 elements
sage: CoxeterGroup(['B', 2]).cambrian_lattice((1,2))
Finite lattice containing 6 elements
sage: CoxeterGroup(['G', 2]).cambrian_lattice((1,2))
Finite lattice containing 8 elements
```

```sage
>>> from sage.all import *
>>> CoxeterGroup(['A', Integer(2)]).cambrian_lattice((Integer(1), Integer(2)))
Finite lattice containing 5 elements
>>> CoxeterGroup(['B', Integer(2)]).cambrian_lattice((Integer(1), Integer(2)))
Finite lattice containing 6 elements
>>> CoxeterGroup(['G', Integer(2)]).cambrian_lattice((Integer(1), Integer(2)))
Finite lattice containing 8 elements
```

codegrees ()

Return the codegrees of the Coxeter group.
These are just the degrees minus 2.

EXAMPLES:

```sage
sage: CoxeterGroup(['A', 4]).codegrees()
(0, 1, 2, 3)
sage: CoxeterGroup(['B', 4]).codegrees()
(0, 2, 4, 6)
sage: CoxeterGroup(['D', 4]).codegrees()
(0, 2, 2, 4)
sage: CoxeterGroup(['F', 4]).codegrees()
(0, 4, 6, 10)
sage: CoxeterGroup(['E', 8]).codegrees()
(0, 6, 10, 12, 16, 18, 22, 28)
sage: CoxeterGroup(['H', 3]).codegrees()
(0, 4, 8)
sage: WeylGroup(['A',3], ['A',3], ['B',2]).codegrees()
(0, 1, 2, 0, 1, 2, 0, 2)
```

```sage
>>> from sage.all import *
>>> CoxeterGroup(['A', Integer(4)]).codegrees()
(0, 1, 2, 3)
>>> CoxeterGroup(['B', Integer(4)]).codegrees()
(0, 2, 4, 6)
>>> CoxeterGroup(['D', Integer(4)]).codegrees()
```

(continues on next page)
(0, 2, 2, 4)

```python
>>> CoxeterGroup(['F', Integer(4)]).codegrees()
(0, 4, 6, 10)
```

```python
>>> CoxeterGroup(['E', Integer(8)]).codegrees()
(0, 6, 10, 12, 16, 18, 22, 28)
```

```python
>>> CoxeterGroup(['H', Integer(3)]).codegrees()
(0, 4, 8)
```

```python
>>> WeylGroup(["A",Integer(3)], ["A",Integer(3)], ["B",Integer(2)]).codegrees()
(0, 1, 2, 0, 1, 2, 0, 2)
```

**coxeter_complex()**

Return the Coxeter complex of `self`.

Let $W$ be a Coxeter group, and let $X$ be the corresponding Tits cone, which is constructed as the $W$ orbit of the fundamental chamber in the reflection representation. The Coxeter complex of $W$ is the simplicial complex $(X \setminus \{0\})/\mathbb{R}_{>0}$. The face poset of this simplicial complex is given by the `coxeter_poset()`. When $W$ is a finite group, then the Coxeter complex is homeomorphic to an $(n - 1)$-dimensional sphere, where $n$ is the rank of $W$.

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A', 3])
sage: C = W.coxeter_complex()
sage: C
Simplicial complex with 14 vertices and 24 facets
sage: C.homology()
{0: 0, 1: 0, 2: 0}
```

```python
sage: W = WeylGroup(['B', 3])
sage: C = W.coxeter_complex()
sage: C
Simplicial complex with 26 vertices and 48 facets
sage: C.homology()
{0: 0, 1: 0, 2: 0}
```

```python
sage: W = CoxeterGroup(['I', 7])
sage: C = W.coxeter_complex()
sage: C
Simplicial complex with 14 vertices and 14 facets
sage: C.homology()
{0: 0, 1: 0, 2: 1}
```

```python
sage: W = CoxeterGroup(['H', 3])
sage: C = W.coxeter_complex()
sage: C
Simplicial complex with 62 vertices and 120 facets
sage: C.homology()
{0: 0, 1: 0, 2: 0}
```

```python
# optional - gap3
sage: W = CoxeterGroup(['H', 3], implementation="permutation")
sage: C = W.coxeter_complex()
sage: C
Simplicial complex with 62 vertices and 120 facets
sage: C.homology()
```

(continues on next page)
Return the Coxeter poset of self.

Let $W$ be a Coxeter group. The Coxeter poset is defined as the set of (right) standard cosets $gW_J$, where $J$ is a subset of the index set $I$ of $W$, ordered by reverse inclusion.

This is equal to the face poset of the Coxeter complex.

EXAMPLES:

```python
coxeter_poset()
>>> from sage.all import *
>>> W = CoxeterGroup(['A', Integer(3)])
>>> C = W.coxeter_complex()
>>> C
Simplicial complex with 14 vertices and 24 facets
>>> C.homology()
{0: 0, 1: 0, 2: ℤ}

>>> W = WeylGroup(['B', Integer(3)])
>>> C = W.coxeter_complex()
>>> C
Simplicial complex with 26 vertices and 48 facets
>>> C.homology()
{0: 0, 1: 0, 2: ℤ}

>>> W = CoxeterGroup(['I', Integer(7)])
>>> C = W.coxeter_complex()
>>> C
Simplicial complex with 14 vertices and 14 facets
>>> C.homology()
{0: 0, 1: ℤ}

>>> W = CoxeterGroup(['H', Integer(3)])
>>> C = W.coxeter_complex()
>>> C
Simplicial complex with 14 vertices and 24 facets
>>> C.homology()
{0: 0, 1: 0, 2: ℤ}

>>> # optional - gap3
>>> W = CoxeterGroup(['H', Integer(3)], implementation="permutation")
>>> C = W.coxeter_complex()
>>> C
Simplicial complex with 14 vertices and 24 facets
>>> C.homology()
{0: 0, 1: 0, 2: ℤ}
```
sage: W = CoxeterGroup(['I', 7])
sage: P = W.coxeter_poset()
sage: P
Finite meet-semilattice containing 29 elements
sage: P.rank()
2

sage: W = CoxeterGroup(['H', 3])
sage: P = W.coxeter_poset()
sage: P
Finite meet-semilattice containing 363 elements
sage: P.rank()
3

sage: # optional - gap3
sage: W = CoxeterGroup(['H', 3], implementation="permutation")
sage: P = W.coxeter_poset()
sage: P
Finite meet-semilattice containing 363 elements
sage: P.rank()
3

>>> from sage.all import *

>>> W = CoxeterGroup(['A', Integer(3)])
>>> P = W.coxeter_poset()
>>> P
Finite meet-semilattice containing 75 elements
>>> P.rank()
3

>>> W = WeylGroup(['B', Integer(3)])
>>> P = W.coxeter_poset()
>>> P
Finite meet-semilattice containing 147 elements
>>> P.rank()
3

>>> W = CoxeterGroup(['I', Integer(7)])
>>> P = W.coxeter_poset()
>>> P
Finite meet-semilattice containing 29 elements
>>> P.rank()
2

>>> W = CoxeterGroup(['H', Integer(3)])
>>> P = W.coxeter_poset()
>>> P
Finite meet-semilattice containing 363 elements
>>> P.rank()
3
### degrees()

Return the degrees of the Coxeter group.

The output is an increasing list of integers.

**EXAMPLES:**

```python
sage: CoxeterGroup(['A', 4]).degrees()
(2, 3, 4, 5)
sage: CoxeterGroup(['B', 4]).degrees()
(2, 4, 6, 8)
sage: CoxeterGroup(['D', 4]).degrees()
(2, 4, 4, 6)
sage: CoxeterGroup(['F', 4]).degrees()
(2, 6, 8, 12)
sage: CoxeterGroup(['E', 8]).degrees()
(2, 8, 12, 14, 18, 20, 24, 30)
sage: CoxeterGroup(['H', 3]).degrees()
(2, 6, 10)
sage: WeylGroup([['A',3], ['A',3], ['B',2]]).degrees()
(2, 3, 4, 2, 3, 4, 2, 4)
```

### inversion_sequence(word)

Return the inversion sequence corresponding to the word in indices of simple generators of self.

If `word` corresponds to \([w_0, w_1, \ldots, w_k]\), the output is \([w_0, w_0 w_1, \ldots, w_0 w_1 \ldots w_k \ldots w_1 w_0]\).

**INPUT:**

- `word` - a word in the indices of the simple generators of self.

**EXAMPLES:**
```sage
sage: CoxeterGroup(["A", 2]).inversion_sequence([1,2,1])
[[[-1 1] [ 0 -1] [ 1 0]
[ 0 1], [-1 0], [ 1 -1] ]

sage: [t.reduced_word() for t in CoxeterGroup(["A",3]).inversion_sequence([2,1,3,2,1,3])]
[[2], [1, 2, 1], [2, 3, 2], [1, 2, 3, 2, 1], [3], [1]]
```

```python
>>> from sage.all import *

>>> CoxeterGroup(["A", Integer(2)].inversion_sequence([Integer(1), Integer(2), Integer(1)])
[[[-1 1] [ 0 -1] [ 1 0]
[ 0 1], [-1 0], [ 1 -1] ]

>>> [t.reduced_word() for t in CoxeterGroup(["A",Integer(3)].inversion_sequence([Integer(2),Integer(1),Integer(3),Integer(2),Integer(1), Integer(3)])]
[[2], [1, 2, 1], [2, 3, 2], [1, 2, 3, 2, 1], [3], [1]]
```

### is_real()

Return True since self is a real reflection group.

**EXAMPLES:**

```sage
sage: CoxeterGroup(["F",4]).is_real()
True
d
sage: CoxeterGroup(["H",4]).is_real()
True
```

```python
>>> from sage.all import *

>>> CoxeterGroup(["F", Integer(4)].is_real()
True

>>> CoxeterGroup(["H", Integer(4)].is_real()
True
```

### long_element(index_set=None, as_word=False)

Return the longest element of self, or of the parabolic subgroup corresponding to the given index_set.

**INPUT:**

- **index_set** – a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
- **as_word** – boolean (default False). If True, then return instead a reduced decomposition of the longest element.

Should this method be called maximal_element? longest_element?

**EXAMPLES:**

```sage
sage: D10 = FiniteCoxeterGroups().example(10)
sage: D10.long_element()
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
sage: D10.long_element([1])
```

(continues on next page)
One can require instead a reduced word for \( w_0 \):

\[
\text{sage: } A3 = \text{CoxeterGroup([}'A', 3])
\]

\[
\text{sage: } A3.\text{long_element(as_word=}'True'\})
\]

\[
[1, 2, 1, 3, 2, 1]
\]

```python
>>> from sage.all import *

>>> A3 = CoxeterGroup(['A', Integer(3)])

>>> A3.long_element(as_word=True)

[1, 2, 1, 3, 2, 1]
```

\[m\text{-Cambrian lattice}\( (c, m=1, on_roots=False)\]

Return the \( m \)-Cambrian lattice on \( m \)-delta sequences.


The \( m \)-delta sequences are certain \( m \)-colored minimal factorizations of \( c \) into reflections.

INPUT:
- \( c \) – a Coxeter element of \( self \) (as a tuple, or as an element of \( self \))
- \( m \) – a positive integer (default: 1)
- on_roots (default: False) – if on_roots is True, the lattice is realized on roots rather than on reflections. In order for this to work, the ElementMethod reflection_to_root must be available.

EXAMPLES:

\[
\text{sage: } \text{CoxeterGroup([}'A', 2])}\text{.m_cambrian_lattice(}'1,2'\})
\]

Finite lattice containing 5 elements

\[
\text{sage: } \text{CoxeterGroup([}'A', 2])}\text{.m_cambrian_lattice(}'1,2',2)\}
\]

Finite lattice containing 12 elements

4.45. Finite Coxeter Groups
>>> from sage.all import *

>>> CoxeterGroup(['A',Integer(2)]).m_cambrian_lattice((Integer(1),
Integer(2)))
Finite lattice containing 5 elements

>>> CoxeterGroup(['A',Integer(2)]).m_cambrian_lattice((Integer(1),
Integer(2)),Integer(2))
Finite lattice containing 12 elements

permutahedron (point=None, base_ring=None)

Return the permutahedron of self,

This is the convex hull of the point point in the weight basis under the action of self on the underlying vector space $V$.

See also:

permutahedron()

INPUT:

- point – optional, a point given by its coordinates in the weight basis (default is $(1, 1, 1, \ldots)$)
- base_ring – optional, the base ring of the polytope

Note: The result is expressed in the root basis coordinates.

Note: If function is too slow, switching the base ring to RDF will almost certainly speed things up.

EXAMPLES:

sage: W = CoxeterGroup(['H',3], base_ring=RDF)
sage: W.permutahedron()
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 120 vertices

sage: W = CoxeterGroup(['I',7])
sage: W.permutahedron()
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 14 vertices

sage: W.permutahedron(base_ring=RDF)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 14 vertices

sage: W = ReflectionGroup(['A',3]) # optional -- gap3
sage: W.permutahedron() # optional -- gap3
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices

sage: W = ReflectionGroup(['A',3],['B',2]) # optional -- gap3
(continues on next page)
sage: W.permutahedron()  # optional - gap3
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 192 vertices

---

```python
>>> from sage.all import *
>>> W = CoxeterGroup(["H",Integer(3)], base_ring=RDF)
>>> W.permutahedron()
doctest:warning ...
UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 120 vertices

>>> W = CoxeterGroup(["I",Integer(7)])
>>> W.permutahedron()
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 14 vertices

>>> W.permutahedron(base_ring=RDF)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 14 vertices

>>> W = ReflectionGroup(["A",Integer(3)])  # optional - gap3
>>> W.permutahedron()  # optional - gap3
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices

>>> W = ReflectionGroup(["A",Integer(3)],["B",Integer(2)])  # optional - gap3
>>> W.permutahedron()  # optional - gap3
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 192 vertices
```

reflections_from_w0()
Return the reflections of self using the inversion set of \( w_0 \).

**EXAMPLES:**

```python
sage: WeylGroup(['A',2]).reflections_from_w0()

\[
\begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

sage: WeylGroup(['A',3]).reflections_from_w0()

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix}
\]
```

```python
>>> from sage.all import *

>>> WeylGroup(['A',Integer(2)]).reflections_from_w0()

\[
\begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

>>> WeylGroup(['A',Integer(3)]).reflections_from_w0()

\[
\begin{pmatrix}
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{pmatrix}
\]
```

**shard_poset**(side='right')

Return the shard intersection order attached to \( W \).

This is a lattice structure on \( W \), introduced in [Rea2009]. It contains the noncrossing partition lattice, as the induced lattice on the subset of \( c \)-sortable elements.

The partial order is given by simultaneous inclusion of inversion sets and subgroups attached to every element.

The precise description used here can be found in [STW2018].

Another implementation for the symmetric group is available as `shard_poset()`.

See also:

`bhz_poset()`, `bruhat_poset()`, `weak_poset()`

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: SH = W.shard_poset(); SH

Finite lattice containing 24 elements

sage: SH.is_graded()

True

sage: SH.characteristic_polynomial()

(continues on next page)```
w0()  
Return the longest element of self.
This attribute is deprecated, use long_element() instead.

EXAMPLES:

```python
sage: D8 = FiniteCoxeterGroups().example(8)
sage: D8.w0
(1, 2, 1, 2, 1, 2, 1, 2)
sage: D3 = FiniteCoxeterGroups().example(3)
sage: D3.w0
(1, 2, 1)
```

weak_lattice(side='right', facade=False)

INPUT:
- side – “left”, “right”, or “twosided” (default: “right”)
- facade – a boolean (default: False)

Returns the left (resp. right) poset for weak order. In this poset, \( u \) is smaller than \( v \) if some reduced word of \( u \) is a right (resp. left) factor of some reduced word of \( v \).

See also:
bhz_poset(), bruhat_poset(), shard_poset()

EXAMPLES:

```python
>>> from sage.all import *
>>> W = CoxeterGroup(['A', Integer(3)], base_ring=ZZ)
>>> SH = W.shard_poset(); SH
Finite lattice containing 24 elements
>>> SH.is_graded()
True
>>> SH.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
>>> SH.f_polynomial()
34*q^3 + 22*q^2 + q
```

w0()

Return the longest element of self.
This attribute is deprecated, use long_element() instead.

EXAMPLES:

```python
sage: D8 = FiniteCoxeterGroups().example(8)
sage: D8.w0
(1, 2, 1, 2, 1, 2, 1, 2)
sage: D3 = FiniteCoxeterGroups().example(3)
sage: D3.w0
(1, 2, 1)
```
```python
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(2)])
>>> P = W.weak_poset()
>>> P
Finite lattice containing 6 elements
>>> P.show()
This poset is in fact a lattice:
```
```
sage: W = WeylGroup(['B', 3])
sage: P = W.weak_poset(side = "left")
sage: P.is_lattice()
True
```
```
>>> from sage.all import *
>>> W = WeylGroup(['B', Integer(3)])
>>> P = W.weak_poset(side = "left")
>>> P.is_lattice()
True
```
so this method has an alias `weak_lattice()`:
```
sage: W.weak_lattice(side = "left") is W.weak_poset(side = "left")
True
```
```
>>> from sage.all import *
>>> W.weak_lattice(side = "left") is W.weak_poset(side = "left")
True
```
As a bonus feature, one can create the left-right weak poset:
```
sage: W = WeylGroup(['A',2])
sage: P = W.weak_poset(side = "twosided")
sage: P.show()
sage: len(P.hasse_diagram().edges(sort=False))
8
```
```
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(2)])
>>> P = W.weak_poset(side = "twosided")
>>> P.show()
>>> len(P.hasse_diagram().edges(sort=False))
8
```
This is the transitive closure of the union of left and right order. In this poset, \( u \) is smaller than \( v \) if some reduced word of \( u \) is a factor of some reduced word of \( v \). Note that this is not a lattice:
```
sage: P.is_lattice()
False
```
```
>>> from sage.all import *
>>> P.is_lattice()
False
```
By default, the elements of \( P \) are aware of that they belong to \( P \):
```
If instead one wants the elements to be plain elements of the Coxeter group, one can use the \texttt{facade} option:

\begin{verbatim}
from sage.all import *

P = W.weak_poset(facade = True)
P.an_element().parent()
\end{verbatim}

\begin{verbatim}
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient→space)
\end{verbatim}

See also:

\texttt{Poset()} for more on posets and facade posets.

\textbf{Todo}:

- Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
- The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.

\texttt{weak_poset}(side='right', facade=False)

\textbf{INPUT}:

- \texttt{side} – “left”, “right”, or “twosided” (default: “right”)
- \texttt{facade} – a boolean (default: \texttt{False})

Returns the left (resp. right) poset for weak order. In this poset, \(u\) is smaller than \(v\) if some reduced word of \(u\) is a right (resp. left) factor of some reduced word of \(v\).

\textbf{See also}:

\texttt{bhz_poset()}, \texttt{bruhat_poset()}, \texttt{shard_poset()}.
sage: W = WeylGroup(["B", 3])
sage: P = W.weak_poset(side = "left")
sage: P.is_lattice()
True

```
>>> from sage.all import *
>>> W = WeylGroup(["B", Integer(3)])
>>> P = W.weak_poset(side = "left")
>>> P.is_lattice()
True
```

so this method has an alias `weak_lattice()`:

```
sage: W.weak_lattice(side = "left") is W.weak_poset(side = "left")
True
```

As a bonus feature, one can create the left-right weak poset:

```
sage: W = WeylGroup(["A",2])
sage: P = W.weak_poset(side = "twosided")
sage: P.show()
sage: len(P.hasse_diagram().edges(sort=False))
8
```

```
>>> from sage.all import *
>>> W = WeylGroup(["A",Integer(2)])
>>> P = W.weak_poset(side = "twosided")
>>> P.show()
>>> len(P.hasse_diagram().edges(sort=False))
8
```

This is the transitive closure of the union of left and right order. In this poset, \(u\) is smaller than \(v\) if some reduced word of \(u\) is a factor of some reduced word of \(v\). Note that this is not a lattice:

```
sage: P.is_lattice()
False
```

```
>>> from sage.all import *
>>> P.is_lattice()
False
```

By default, the elements of \(P\) are aware of that they belong to \(P\):

```
sage: P.an_element().parent()
Finite poset containing 6 elements
```

```
>>> from sage.all import *
>>> P.an_element().parent()
Finite poset containing 6 elements
```

If instead one wants the elements to be plain elements of the Coxeter group, one can use the `facade` option:
sage: P = W.weak_poset(facade = True)
sage: P.an_element().parent()
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient...space)

See also:
Poset() for more on posets and facade posets.

Todo:
- Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
- The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.

extra_super_categories()

EXAMPLES:
sage: CoxeterGroups().Finite().super_categories()
[Category of finite generalized Coxeter groups,
 Category of Coxeter groups]

>>> from sage.all import *
>>> CoxeterGroups().Finite().super_categories()
[Category of finite generalized Coxeter groups,
 Category of Coxeter groups]

4.46 Finite Crystals

class sage.categories.finite_crystals.FiniteCrystals(base_category)

Bases: CategoryWithAxiom_singleton

The category of finite crystals.

EXAMPLES:
sage: C = FiniteCrystals()
sage: C
Category of finite crystals
sage: C.super_categories()
[Category of crystals, Category of finite enumerated sets]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1

>>> from sage.all import *
>>> C = FiniteCrystals()
>>> C
Category of finite crystals

(continues on next page)
C.super_categories()
[Category of crystals, Category of finite enumerated sets]
C.example()
Highest weight crystal of type A_3 of highest weight omega_1

class TensorProducts(category, *args)
    Bases: TensorProductsCategory
    The category of finite crystals constructed by tensor product of finite crystals.
    extra_super_categories()
    EXAMPLES:
    sage: FiniteCrystals().TensorProducts().extra_super_categories()
    [Category of finite crystals]

>>> from sage.all import *
>>> FiniteCrystals().TensorProducts().extra_super_categories()
[Category of finite crystals]

element (n=3)
    Returns an example of highest weight crystals, as per Category.example().
    EXAMPLES:
    sage: B = FiniteCrystals().example(); B
    Highest weight crystal of type A_3 of highest weight omega_1

>>> from sage.all import *
>>> B = FiniteCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1

extra_super_categories()
    EXAMPLES:
    sage: FiniteCrystals().extra_super_categories()
    [Category of finite enumerated sets]

>>> from sage.all import *
>>> FiniteCrystals().extra_super_categories()
[Category of finite enumerated sets]

4.47 Finite dimensional algebras with basis

Todo: Quotients of polynomial rings.
    Quotients in general.
    Matrix rings.

REFERENCES:
    • [CR1962]
class sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis

Bases: CategoryWithAxiom_over_base_ring

The category of finite dimensional algebras with a distinguished basis.

EXAMPLES:

```python
sage: C = FiniteDimensionalAlgebrasWithBasis(QQ); C
Category of finite dimensional algebras with basis over Rational Field
sage: C.super_categories()
[Category of algebras with basis over Rational Field,  
Category of finite dimensional magmatic algebras with basis over Rational Field]
```

```python
from sage.all import *
C = FiniteDimensionalAlgebrasWithBasis(QQ); C
```

```python
Cellular

cellular.

Let $R$ be a commutative ring. A $R$-algebra $A$ is a cellular algebra if it has a cell datum, which is a tuple $(\Lambda, i, M, C)$, where $\Lambda$ is finite poset with order $\geq$, if $\mu \in \Lambda$ then $T(\mu)$ is a finite set and

$$C: \prod_{\mu \in \Lambda} T(\mu) \times T(\mu) \rightarrow A; (\mu, s, t) \mapsto c^\mu_{st}$$

is an injective map

such that the following holds:

- The set $\{c^\mu_{st} \mid \mu \in \Lambda, s, t \in T(\mu)\}$ is a basis of $A$.
- If $a \in A$ and $\mu \in \Lambda, s, t \in T(\mu)$ then:

$$ac^\mu_{st} = \sum_{u \in T(\mu)} r_a(s, u)c^\mu_{ut} \pmod{A^>\mu},$$

where $A^>\mu$ is spanned by

$$\{c^\nu_{ab} \mid \nu > \mu \text{ and } a, b \in T(\nu)\}.$$ 

Moreover, the scalar $r_a(s, u)$ depends only on $a, s$ and $u$ and, in particular, is independent of $t$.
- The map $\iota: A \rightarrow A; c^\mu_{st} \mapsto c^\mu_{ts}$ is an algebra anti-isomorphism.
A cellular basis for $A$ is any basis of the form $\{e^\mu_{st} \mid \mu \in \Lambda, s, t \in T(\mu)\}$.

Note that in particular, the scalars $r_a(u, s)$ in the second condition do not depend on $t$.

REFERENCES:
- [GrLe1996]
- [KX1998]
- [Mat1999]
- Wikipedia article Cellular_algebra

class ElementMethods
Bases: object

cellular_involution()

Return the cellular involution on self.

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: elt = S([3,1,2,4])
sage: ci = elt.cellular_involution(); ci
7/48*[1, 3, 2, 4] + 49/48*[2, 3, 1, 4]
- 1/48*[3, 1, 2, 4] - 7/48*[3, 2, 1, 4]
sage: ci.cellular_involution()
[3, 1, 2, 4]
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> S = SymmetricGroupAlgebra(QQ, Integer(4))
>>> elt = S([Integer(3),Integer(1),Integer(2),Integer(4)])
>>> ci = elt.cellular_involution(); ci
7/48*[1, 3, 2, 4] + 49/48*[2, 3, 1, 4]
- 1/48*[3, 1, 2, 4] - 7/48*[3, 2, 1, 4]
>>> ci.cellular_involution()
[3, 1, 2, 4]
```

class ParentMethods
Bases: object

cell_module($\mu$, **kwds)

Return the cell module indexed by $\mu$.

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 3) # needs sage.groups.sage.modules
sage: S.cell_module(Partition([2,1])) # needs sage.combinat.sage.groups.sage.modules
Cell module indexed by [2, 1] of Cellular basis of Symmetric group algebra of order 3 over Rational Field
```

```python
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3)) ...
```

(continues on next page)


\[ \text{cell_module_indices}(\mu) \]

Return the indices of the cell module of \text{self} indexed by \mu.

This is the finite set \( M(\lambda) \).

**EXAMPLES:**

```
>>> S = SymmetricGroupAlgebra(QQ, 3)  #...
>>> S.cell_module_indices([2,1])  #...
Standard tableaux of shape [2, 1]
```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))       # needs sage.groups sage.modules
>>> dict(S.cells())                               # needs sage.groups sage.modules
{[1, 1, 1]: Standard tableaux of shape [1, 1, 1],
 [2, 1]: Standard tableaux of shape [2, 1],
 [3]: Standard tableaux of shape [3]}

**cellular_basis()**

Return the cellular basis of `self`.

**EXAMPLES:**

sage: S = SymmetricGroupAlgebra(QQ, 3)            # needs sage.groups sage.modules
sage: S.cellular_basis()                          # needs sage.groups sage.modules
Cellular basis of Symmetric group algebra of order 3 over Rational Field

```python
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))   # needs sage.groups sage.modules
>>> for b in S.basis(): b, S.cellular_involution(b)   # needs sage.groups sage.modules
([1, 2, 3], [1, 2, 3])
([1, 3, 2], 49/48*[1, 3, 2] + 7/48*[2, 3, 1] - 7/48*[3, 1, 2] - 1/48*[3, 2, 1])
([2, 1, 3], [2, 1, 3])
([2, 3, 1], -7/48*[1, 3, 2] - 1/48*[2, 3, 1] + 49/48*[3, 1, 2] + 7/48*[3, 2, 1])
([3, 1, 2], 7/48*[1, 3, 2] + 49/48*[2, 3, 1] - 1/48*[3, 1, 2] - 7/48*[3, 2, 1])
([3, 2, 1], -1/48*[1, 3, 2] - 7/48*[2, 3, 1] + 7/48*[3, 1, 2] + 49/48*[3, 2, 1])
```

(continues on next page)
Return a parameterization of the simple modules of self.

The set of simple modules are parameterized by $\lambda \in \Lambda$ such that the cell module bilinear form $\Phi_{\lambda} \neq 0$.

**EXAMPLES:**

```python
sage: TL = TemperleyLiebAlgebra(5, 30, QQ)  # semisimple
sage: len(TL.radical_basis())
0
sage: TL.simple_module_parameterization()
(1, 3, 5)

sage: TL = TemperleyLiebAlgebra(5, 1, QQ)  # not semisimple
sage: len(TL.radical_basis())
24
sage: TL.simple_module_parameterization()
(1, 3, 5)

sage: TL = TemperleyLiebAlgebra(6, 30, QQ)  # semisimple
sage: all(TL.cell_module(la).dimension() == TL.cell_module(la).simple_module().dimension() for la in TL.simple_module_parameterization())
True
sage: TL.simple_module_parameterization()
(0, 2, 4, 6)

sage: TL = TemperleyLiebAlgebra(6, 0, QQ)  # not semisimple
sage: TL.simple_module_parameterization()
(2, 4, 6)
```

```python
>>> from sage.all import *
>>> TL = TemperleyLiebAlgebra(Integer(5), Integer(30), QQ)  # semisimple
>>> len(TL.radical_basis())
0
>>> TL.simple_module_parameterization()
(1, 3, 5)

>>> TL = TemperleyLiebAlgebra(Integer(5), Integer(1), QQ)  # not semisimple
>>> len(TL.radical_basis())
24
>>> TL.simple_module_parameterization()
(1, 3, 5)

>>> TL = TemperleyLiebAlgebra(Integer(6), Integer(30), QQ)  # semisimple
>>> len(TL.radical_basis())
0
>>> TL.simple_module_parameterization()
(0, 2, 4, 6)
```

```python
>>> TL = TemperleyLiebAlgebra(Integer(5), Integer(30), QQ)  # semisimple
>>> len(TL.radical_basis())
0
>>> TL.simple_module_parameterization()
(1, 3, 5)

>>> TL = TemperleyLiebAlgebra(Integer(5), Integer(1), QQ)  # not semisimple
>>> len(TL.radical_basis())
24
>>> TL.simple_module_parameterization()
(1, 3, 5)

>>> TL = TemperleyLiebAlgebra(Integer(6), Integer(30), QQ)  # semisimple
>>> len(TL.radical_basis())
0
>>> TL.simple_module_parameterization()
(0, 2, 4, 6)
```
```python
>>> all(TL.cell_module(la).dimension() == TL.cell_module(la).simple_module().dimension() for la in TL.simple_module_parameterization())
True
>>> TL.simple_module_parameterization()
(0, 2, 4, 6)

>>> TL = TemperleyLiebAlgebra(Integer(6), Integer(0), QQ)  # not...
˓→semisimple
>>> TL.simple_module_parameterization()
(2, 4, 6)
```

**class TensorProducts** *(category, *args)*

Bases: `TensorProductsCategory`

The category of cellular algebras constructed by tensor product of cellular algebras.

**class ParentMethods**

Bases: `object`

**cell_module_indices** *(mu)*

Return the indices of the cell module of `self` indexed by `mu`.

This is the finite set $M(\lambda)$.

**EXAMPLES:**

```python
sage: # needs sage.groups sage.modules
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: T = S2.tensor(S3)
sage: T.cell_module_indices((Integer(1), Integer(1)), (Integer(2), Integer(1)))
The Cartesian product of (Standard tableaux of shape [1, 1], Standard tableaux of shape [2, 1])
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> S2 = SymmetricGroupAlgebra(QQ, Integer(2))
>>> S3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> T = S2.tensor(S3)
>>> T.cell_module_indices([(Integer(1), Integer(1)), [Integer(2), Integer(1)])
The Cartesian product of (Standard tableaux of shape [1, 1], Standard tableaux of shape [2, 1])
```

**cell_poset()**

Return the cell poset of `self`.

**EXAMPLES:**

```python
sage: # needs sage.groups sage.modules
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: T = S2.tensor(S3)
sage: T.cell_poset()  #...
˓→needs sage.combinat sage.graphs
Finite poset containing 6 elements
```
cellular_involution()

Return the image of the cellular involution of the basis element indexed by \( i \).

EXAMPLES:

```
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> S2 = SymmetricGroupAlgebra(QQ, Integer(2))
>>> S3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> T = S2.tensor(S3)
>>> for b in T.basis(): b, T.cellular_involution(b)
(([1, 2] # [1, 2, 3], [1, 2] # [1, 2, 3])
([1, 2] # [1, 3, 2],
  49/48*[1, 2] # [1, 3, 2] + 7/48*[1, 2] # [2, 3, 1]
  - 7/48*[1, 2] # [3, 1, 2] - 1/48*[1, 2] # [3, 2, 1])
([1, 2] # [2, 1, 3],
  7/48*[1, 2] # [1, 2, 3] + 49/48*[1, 2] # [1, 3, 2]
  - 1/48*[1, 2] # [3, 1, 2] - 7/48*[1, 2] # [3, 2, 1])
([1, 2] # [2, 3, 1],
  -7/48*[1, 2] # [1, 3, 2] - 1/48*[1, 2] # [2, 3, 1]
  + 49/48*[1, 2] # [3, 1, 2] + 7/48*[1, 2] # [3, 2, 1])
((2, 1] # [1, 2, 3], [2, 1] # [1, 2, 3])
((2, 1] # [1, 3, 2],
  49/48*[2, 1] # [1, 3, 2] + 7/48*[2, 1] # [2, 3, 1]
  - 7/48*[2, 1] # [3, 1, 2] - 1/48*[2, 1] # [3, 2, 1])
((2, 1] # [2, 1, 3], [2, 1] # [2, 1, 3])
((2, 1] # [2, 3, 1],
  + 49/48*[2, 1] # [3, 1, 2] + 7/48*[2, 1] # [3, 2, 1])
((2, 1] # [3, 1, 2],
  7/48*[2, 1] # [1, 3, 2] + 49/48*[2, 1] # [2, 3, 1]
  - 1/48*[2, 1] # [3, 1, 2] - 7/48*[2, 1] # [3, 2, 1])
((2, 1] # [3, 2, 1],
  + 7/48*[2, 1] # [3, 1, 2] + 49/48*[2, 1] # [3, 2, 1])
```

(continues on next page)
extra_super_categories()

Tensor products of cellular algebras are cellular.

EXAMPLES:

```sage
sage: cat = Algebras(QQ).FiniteDimensional().WithBasis()
sage: cat.Cellular().TensorProducts().extra_super_categories()
[Category of finite dimensional cellular algebras with basis over Rational Field]
```

```python
>>> from sage.all import *
>>> cat = Algebras(QQ).FiniteDimensional().WithBasis()
>>> cat.Cellular().TensorProducts().extra_super_categories()
[Category of finite dimensional cellular algebras with basis over Rational Field]
```

class ElementMethods

Bases: object

`on_left_matrix(base_ring=None, action=<built-in function mul>, side='left')`

Return the matrix of the action of self on the algebra.

INPUT:

- `base_ring` – the base ring for the matrix to be constructed
- `action` – a bivariate function (default: `operator.mul()`)
- `side` – ‘left’ or ‘right’ (default: ‘left’)

EXAMPLES:

```sage
sage: # needs sage.groups sage.modules
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
```
AUTHORS: Mike Hansen, …

\texttt{to\_matrix}(\text{base\_ring}=\text{None}, \text{action}=<\text{built-in function mul}>), \text{side}=\text{'left'}$

Return the matrix of the action of self on the algebra.

INPUT:

- \texttt{base\_ring} – the base ring for the matrix to be constructed
- \texttt{action} – a bivariate function (default: \texttt{operator.mul()})
• side – ‘left’ or ‘right’ (default: ‘left’)

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3([2,1,3])
sage: a.to_matrix(side='left')
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
sage: a.to_matrix(side='right')
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 0 0 0 1]
[0 0 0 1 0 0]
sage: a.to_matrix(base_ring=RDF, side="left")
[0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0]
[1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0 0.0]
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> a = QS3([[Integer(2),Integer(1),Integer(3)]])
>>> a.to_matrix(side='left')
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
>>> a.to_matrix(side='right')
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 0 0 0 1]
[0 0 0 1 0 0]
>>> a.to_matrix(base_ring=RDF, side="left")
[0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0]
[1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0 0.0]
```

AUTHORS: Mike Hansen, …

class ParentMethods
    Bases: object
cartan_invariants_matrix()

Return the Cartan invariants matrix of the algebra.

OUTPUT: a matrix of non negative integers

Let $A$ be this finite dimensional algebra and $(S_i)_{i \in I}$ be representatives of the right simple modules of $A$. Note that their adjoints $S_i^*$ are representatives of the left simple modules.

Let $(P^L_i)_{i \in I}$ and $(P^R_i)_{i \in I}$ be respectively representatives of the corresponding indecomposable projective left and right modules of $A$. In particular, we assume that the indexing is consistent so that $S_i^* = \text{top } P^L_i$ and $S_i = \text{top } P^R_i$.

The Cartan invariant matrix $(C_{i,j})_{i,j \in I}$ is a matrix of non negative integers that encodes much of the representation theory of $A$; namely:

- $C_{i,j}$ counts how many times $S_i^* \otimes S_j$ appears as composition factor of $A$ seen as a bimodule over itself;
- $C_{i,j} = \dim \text{Hom}_A(P^R_j, P^R_i)$;
- $C_{i,j}$ counts how many times $S_j$ appears as composition factor of $P^R_i$;
- $C_{i,j} = \dim \text{Hom}_A(P^L_i, P^L_j)$;
- $C_{i,j}$ counts how many times $S_i^*$ appears as composition factor of $P^L_j$.

In the commutative case, the Cartan invariant matrix is diagonal. In the context of solving systems of multivariate polynomial equations of dimension zero, $A$ is the quotient of the polynomial ring by the ideal generated by the equations, the simple modules correspond to the roots, and the numbers $C_{i,i}$ give the multiplicities of those roots.

Note: For simplicity, the current implementation assumes that the index set $I$ is of the form $\{0, \ldots, n-1\}$. Better indexations will be possible in the future.

ALGORITHM:
The Cartan invariant matrix of $A$ is computed from the dimension of the summands of its Peirce decomposition.

See also:
- `peirce_decomposition()`
- `isotypic_projective_modules()`

EXAMPLES:

For a semisimple algebra, in particular for group algebras in characteristic zero, the Cartan invariants matrix is the identity:

```python
sage: A3 = SymmetricGroup(3).algebra(QQ)  # needs sage.combinat sage.groups sage.modules
sage: A3.cartan_invariants_matrix()  # needs sage.combinat sage.groups sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```

```python
>>> from sage.all import *
>>> A3 = SymmetricGroup(Integer(3)).algebra(QQ)  # needs sage.combinat sage.groups sage.modules
>>> A3.cartan_invariants_matrix()  # needs sage.combinat sage.groups sage.modules
[1 0 0]
[0 1 0]
[0 0 1]
```
For the path algebra of a quiver, the Cartan invariants matrix counts the number of paths between two vertices:

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()
sage: A.cartan_invariants_matrix()
# needs sage.modules sage.rings.number_field
[1 2]
[0 1]
```

In the commutative case, the Cartan invariant matrix is diagonal:

```python
sage: Z12 = Monoids().Finite().example(); Z12
An example of a finite multiplicative monoid: the integers modulo 12
sage: A = Z12.algebra(QQ)
# needs sage.modules
sage: A.cartan_invariants_matrix()
# needs sage.modules sage.rings.number_field
[1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0]
[0 0 2 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0]
[0 0 0 0 2 0 0 0 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 2 0]
[0 0 0 0 0 0 0 0 1]
```

With the algebra of the 0-Hecke monoid:

```python
>>> from sage.all import *
>>> Z12 = Monoids().Finite().example(); Z12
An example of a finite multiplicative monoid: the integers modulo 12
>>> A = Z12.algebra(QQ)
# needs sage.modules
>>> A.cartan_invariants_matrix()
# needs sage.modules sage.rings.number_field
[1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0]
[0 0 2 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0]
[0 0 0 0 2 0 0 0 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 2 0]
[0 0 0 0 0 0 0 0 1]
```
center()

Return the center of self.

See also:

center_basis()

EXAMPLES:

sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: center = A.center(); center
Center of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: center in Algebras(QQ).WithBasis().FiniteDimensional().Commutative()
True
sage: center.dimension()
1
sage: center.basis()
Finite family {0: B[0]}

sage: [c.lift() for c in center.basis()]
[x + y]
The center of a semisimple algebra is semisimple:

```
sage: A = DihedralGroup(6).algebra(QQ) # needs sage.groups sage.modules
sage: A.center() in Algebras(QQ).Semisimple() # needs sage.groups sage.modules
True
```

```
sage: from sage.all import *
sage: A = DihedralGroup(Integer(6)).algebra(QQ) # needs sage.groups sage.modules
sage: A.center() in Algebras(QQ).Semisimple() # needs sage.groups sage.modules
True
```

Todo:
- Pickling by construction, as A.center()?
- Lazy evaluation of _repr_

```
center_basis()
```

Return a basis of the center of self.

OUTPUT:
- a list of elements of self.

See also:
```
center()
```

EXAMPLES:

```
sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field

sage: A = DihedralGroup(6).algebra(QQ) # needs sage.groups sage.modules
```
(continues on next page)
ideal_submodule (gens, side='left', category=None, *args, **opts)

Return the side ideal of self generated by gens as a submodule.

Todo: This is not generally compatible with the implementation of the ideals. This method should be folded into the ideal method after the corresponding classes are refactored to be compatible.

EXAMPLES:

sage: scoeffs = {('a','e'): {'a':1}, ('b','e'): {'a':1, 'b':1},
            ...
            ('c','d'): {'a':1}, ('c','e'): {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: MS = MatrixSpace(QQ, 5)
sage: I = MS.ideal_submodule([bg.adjoint_matrix() for bg in L.lie_algebra_generators()])
sage: I.dimension() 25

idempotent_lift (x)

Lift an idempotent of the semisimple quotient into an idempotent of self.

Let $A$ be this finite dimensional algebra and $\pi$ be the projection $A \to \overline{A}$ on its semisimple quotient. Let $\overline{\pi}$ be an idempotent of $\overline{A}$, and $x$ any lift thereof in $A$. This returns an idempotent $e$ of $A$ such that $\overline{\pi}(e) = \pi(x)$ and $e$ is a polynomial in $x$.

INPUT:

- $x$ – an element of $A$ that projects on an idempotent $\overline{\pi}$ of the semisimple quotient of $A$. Alternatively one may give as input the idempotent $\overline{\pi}$, in which case some lift thereof will be taken for $x$.

OUTPUT: the idempotent $e$ of self

ALGORITHM:
Iterate the formula $1 - (1 - x^2)^2$ until having an idempotent.

See [CR1962] for correctness and termination proofs.

EXAMPLES:

```python
sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()
sage: S = A.semisimple_quotient()
sage: A.idempotent_lift(S.basis()['x'])
x
sage: A.idempotent_lift(A.basis()['y'])
y
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> A = Algebras(QQ).FiniteDimensional().WithBasis().example()
>>> S = A.semisimple_quotient()
>>> A.idempotent_lift(S.basis()['x'])
x
>>> A.idempotent_lift(A.basis()['y'])
y
Todo: Add some non trivial example

**is_commutative()**

Return whether `self` is a commutative algebra.

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: S4 = SymmetricGroupAlgebra(QQ, 4)
sage: S4.is_commutative()
False
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S2.is_commutative()
True
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> S4 = SymmetricGroupAlgebra(QQ, Integer(4))
>>> S4.is_commutative()
False
>>> S2 = SymmetricGroupAlgebra(QQ, Integer(2))
>>> S2.is_commutative()
True
```

**is_identity_decomposition_into_orthogonal_idempotents()**

Return whether `l` is a decomposition of the identity into orthogonal idempotents.

**INPUT:**

- `l` - a list or iterable of elements of `self`

**EXAMPLES:**

```python
sage: # needs sage.graphs sage.modules
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
```

(continues on next page)
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows $a:x\rightarrow y$ and $b:x\rightarrow y$) over Rational Field

```
sage: x,y,a,b = A.algebra_generators(); x,y,a,b
(x, y, a, b)
sage: A.is_identity_decomposition_into_orthogonal_idempotents([A.one()])
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x, y])
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x + a, y -\rightarrow a])
True
```

```
>>> from sage.all import *

>>> A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows $a:x\rightarrow y$ and $b:x\rightarrow y$) over Rational Field

>>> x,y,a,b = A.algebra_generators(); x,y,a,b
(x, y, a, b)

>>> A.is_identity_decomposition_into_orthogonal_idempotents([A.one()])
True

>>> A.is_identity_decomposition_into_orthogonal_idempotents([x, y])
True

>>> A.is_identity_decomposition_into_orthogonal_idempotents([x + a, y -\rightarrow a])
True
```

Here the idempotents do not sum up to 1:

```
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x])
# needs sage.graphs sage.modules
False
```

```
>>> from sage.all import *

>>> A.is_identity_decomposition_into_orthogonal_idempotents([x])
# needs sage.graphs sage.modules
False
```

Here $1 + x$ and $-x$ are neither idempotent nor orthogonal:

```
sage: A.is_identity_decomposition_into_orthogonal_idempotents([1 + x, -\rightarrow x])
# needs sage.graphs sage.modules
False
```

```
>>> from sage.all import *

>>> A.is_identity_decomposition_into_orthogonal_idempotents([Integer(1) +\rightarrow x, -x])
# needs sage.graphs sage.modules
False
```

With the algebra of the 0-Hecke monoid:

```
sage: # needs sage.combinat sage.groups sage.modules sage.rings.number_\rightarrow field
```

(continues on next page)
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
>>> idempotents = A.orthogonal_idempotents_central_mod_radical()
>>> A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True

Here are some more counterexamples:
1. Some orthogonal elements summing to 1 but not being idempotent:

sage: from sage.all import *

```python
>>> class PQAlgebra(CombinatorialFreeModule):
...     def __init__(self, F, p):
...         # Construct the quotient algebra F[x] / p,
...         # where p is a univariate polynomial.
...         R = parent(p); x = R.gen()
...         I = R.ideal(p)
...         self._xbar = R.quotient(I).gen()
...         basis_keys = [self._xbar**i for i in range(p.degree())]
...         CombinatorialFreeModule.__init__(self, F, basis_keys,
...         category=Algebras(F).FiniteDimensional).
...     def x(self):
...         return self(self._xbar)
...     def one(self):
...         return self.basis()[self.base_ring().one()]
...     def product_on_basis(self, w1, w2):
...         return self.from_vector(vector(w1*w2))

sage: R.<x> = PolynomialRing(QQ)

sage: A = PQAlgebra(QQ, x**3 - x**2 + x + 1); y = A.x()

sage: a, b = y, 1 - y

sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
False
```

(continues on next page)
... def one(self):
... return self.basis()[self.base_ring().one()]

... def product_on_basis(self, w1, w2):
... return self.from_vector(vector(w1*w2))

R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
A = PQAlgebra(QQ, x**Integer(3) - x**Integer(2) + x + Integer(1));
→ y = A.x()
>>> a, b = y, Integer(1) - y
>>> A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
False

For comparison:

sage: # needs sage.libs.pari sage.modules
sage: A = PQAlgebra(QQ, x**2 - x); y = A.x()
sage: a, b = y, 1-y
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, A.
→ zero(), b))
True
sage: A = PQAlgebra(QQ, x**3 - x**2 + x - 1); y = A.x()
sage: a = (y**2 + 1) / Integer(2)
sage: b = Integer(1) - a
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
True

>>> from sage.all import *

2. Some idempotents summing to 1 but not orthogonal:

sage: # needs sage.libs.pari sage.modules
sage: R.<x> = PolynomialRing(GF(2))
sage: A = PQAlgebra(GF(2), x)
sage: a = A.one()
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, a, ⨁
a))
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, a, ⨁
a, a))
False

>>> from sage.all import *

(continues on next page)
# needs sage.libs.pari sage.modules

```python
>>> R = PolynomialRing(GF(Integer(2)), names=('x',)); (x,) = R._first_ 
   ngens(1)
```

```python
>>> A = PQAlgebra(GF(Integer(2)), x)
>>> a = A.one()
>>> A.is_identity_decomposition_into_orthogonal_idempotents((a,))
True
>>> A.is_identity_decomposition_into_orthogonal_idempotents((a, a, a))
False
```

3. Some orthogonal idempotents not summing to the identity:

```python
>>> from sage.all import *
```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules sage.rings.number_field
>>> A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x→y and b:x→y) over Rational Field
>>> Q = A.isotypic_projective_modules(side="left"); Q
[Free module generated by {0} over Rational Field,
 Free module generated by {0, 1, 2} over Rational Field]
>>> [[x.lift() for x in Qi.basis()]
... for Qi in Q]
[[x],
 [y, a, b]]

We check that the sum of the dimensions of the isotypic projective modules is the dimension of self:

```
sage: sum([Qi.dimension() for Qi in Q]) == A.dimension() #... needs sage.graphs sage.modules sage.rings.number_field
True
```

See also:

• orthogonal_idempotents_central_mod_radical()
• peirce_decomposition()

**orthogonal_idempotents_central_mod_radical()**

Return a family of orthogonal idempotents of self that project on the central orthogonal idempotents of the semisimple quotient.

**OUTPUT:**

• a list of orthogonal idempotents obtained by lifting the central orthogonal idempotents of the semisimple quotient.

**ALGORITHM:**

The orthogonal idempotents of $A$ are obtained by lifting the central orthogonal idempotents of the semisimple quotient $\overline{A}$.

Namely, let $(\overline{f_i})$ be the central orthogonal idempotents of the semisimple quotient of $A$. We recursively construct orthogonal idempotents of $A$ by the following procedure: assuming $(f_i)_{i<n}$ is a set of already constructed orthogonal idempotent, we construct $f_k$ by idempotent lifting of $(1-f)g(1-f)$, where $g$ is any lift of $\overline{c_k}$ and $f = \sum_{i<k} f_i$.

See [CR1962] for correctness and termination proofs.

See also:

• idempotent_lift()

**EXAMPLES:**

```
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
(continues on next page)```
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x→y and b:x→y) over Rational Field

```python
sage: A.orthogonal_idempotents_central_mod_radical()
# needs sage.rings.number_field
(x, y)
```

```python
from sage.all import *

# needs sage.graphs sage.modules
A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x→y and b:x→y) over Rational Field

# needs sage.rings.number_field
sage: A.orthogonal_idempotents_central_mod_radical()
(x, y)
```

```python
class Z12(Monoids().Finite().example): Z12
An example of a finite multiplicative monoid: the integers modulo 12

sage: Z12 = Monoids().Finite().example(); Z12
sage: idempotents = A.orthogonal_idempotents_central_mod_radical()
sage: sorted(idempotents, key=str)
  1/2*B[9] - 1/2*B[3],
  B[0])
sage: sum(idempotents) == 1
True
sage: all(e*e == e for e in idempotents)
True
sage: all(e*f == 0 and f*e == 0
    ....:   for e in idempotents
    ....:     for f in idempotents if e != f)
True
```

```python
from sage.all import *

# needs sage.rings.number_field

Z12 = Monoids().Finite().example(); Z12
An example of a finite multiplicative monoid: the integers modulo 12

# needs sage.rings.number_field
```
\[
\frac{1}{4}B[1] - \frac{1}{4}B[5] + \frac{1}{4}B[7] - \frac{1}{4}B[11] - \frac{1}{2}B[4] + \frac{1}{2}B[8],
\]

\[
B[0]
\]

>>> sum(idempotents) == Integer(1)
True
>>> all(e*e == e for e in idempotents)
True
>>> all(e*f == Integer(0) and f*e == Integer(0)
...    for e in idempotents for f in idempotents if e != f)
True

This is best tested with:

```python
sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents) # needs sage.graphs sage.modules sage.rings.number_field
True

>>> from sage.all import *

>>> A.is_identity_decomposition_into_orthogonal_idempotents(idempotents) # needs sage.graphs sage.modules sage.rings.number_field
True
```

We construct orthogonal idempotents for the algebra of the 0-Hecke monoid:

```python
sage: # needs sage.combinat sage.graphs sage.groups sage.modules
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
sage: idempotents = A.orthogonal_idempotents_central_mod_radical()
sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True

>>> from sage.all import *

>>> # needs sage.combinat sage.graphs sage.groups sage.modules
>>> from sage.monoids.hecke_monoid import HeckeMonoid
>>> A = HeckeMonoid(SymmetricGroup(Integer(4))).algebra(QQ)
>>> idempotents = A.orthogonal_idempotents_central_mod_radical()
>>> A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

peirce_decomposition(idempotents=None, check=True)

Return a Peirce decomposition of self.

Let \((e_i)\) be a collection of orthogonal idempotents of \(A\) with sum 1. The Peirce decomposition of \(A\) is the decomposition of \(A\) into the direct sum of the subspaces \(e_i A e_j\).

With the default collection of orthogonal idempotents, one has

\[
\dim e_i A e_j = C_{i,j} \dim S_i \dim S_j
\]

where \((S_i)\) are the simple modules of \(A\) and \((C_{i,j})\) is the Cartan invariants matrix.

INPUT:

- idempotents – a list of orthogonal idempotents \((e_i)\) of the algebra that sum to 1 (default: the idempotents returned by orthogonal_idempotents_central_mod_radical())
- check – (default: True) whether to check that the idempotents are indeed orthogonal and idempotent and sum to 1

4.47. Finite dimensional algebras with basis
OUTPUT:
A list of lists \( l \) such that \( l[i][j] \) is the subspace \( e_i A e_j \).

See also:

- \texttt{orthogonal_idempotents_central_mod_radical()}
- \texttt{cartan_invariants_matrix()}

EXAMPLES:

```
sage: # needs sage.graphs sage.groups sage.modules sage.rings.number_field
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows \(a:x\rightarrow y\) and \(b:x\rightarrow y\)) over Rational Field
sage: A.orthogonal_idempotents_central_mod_radical()
(x, y)
sage: decomposition = A.peirce_decomposition(); decomposition
[[Free module generated by \{0\} over Rational Field,
  Free module generated by \{0, 1\} over Rational Field],
 [Free module generated by \{} over Rational Field,
  Free module generated by \{0\} over Rational Field]]
sage: [[[[x.lift() for x in decomposition[i][j].basis()]
      for i in range(2)]
     for j in range(2)]
[[[x], [a, b]],
 []]
```

We recover that the group algebra of the symmetric group \( S_4 \) is a block matrix algebra:

```
>>> from sage.all import *
>>> # needs sage.groups sage.modules sage.rings.number_field
>>> A = SymmetricGroup(4).algebra(QQ)
>>> decomposition = A.peirce_decomposition()  # long time
sage: decomposition
[[[decomposition[i][j].dimension() # long time (4s)
  for j in range(len(decomposition))]# long time (4s)
   for i in range(len(decomposition))]# long time (4s)
[[9, 0, 0, 0, 0],
 [0, 9, 0, 0, 0],
 [0, 0, 4, 0, 0],
[0, 0, 0, 4, 0],
[0, 0, 0, 0, 4]]
```

(continues on next page)
The dimension of each block is $d^2$, where $d$ is the dimension of the corresponding simple module of $S_i$. The latter are given by:

```
sage: [p.standard_tableaux().cardinality() for p in Partitions(4)]  # needs sage.combinat
[1, 3, 2, 3, 1]
```

```
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.rings.number_field
sage: idemp = A.orthogonal_idempotents_central_mod_radical()  # needs sage.rings.number_field
sage: A.peirce_summand(idemp[0], idemp[1])  # needs sage.rings.number_field
Free module generated by {0, 1} over Rational Field
sage: A.peirce_summand(idemp[1], idemp[0])  # needs sage.rings.number_field
Free module generated by {} over Rational Field
```

peirce_summand($e_i, e_j$)

Return the Peirce decomposition summand $e_i A e_j$.

INPUT:

- **self** – an algebra $A$
- **$e_i, e_j$** – two idempotents of $A$

OUTPUT: $e_i A e_j$, as a subspace of $A$.

See also:

- **peirce_decomposition()**
- **principal_ideal()**

EXAMPLES:

```
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.rings.number_field
sage: idemp = A.orthogonal_idempotents_central_mod_radical()  # needs sage.rings.number_field
sage: A.peirce_summand(idemp[0], idemp[1])  # needs sage.rings.number_field
Free module generated by {0, 1} over Rational Field
sage: A.peirce_summand(idemp[1], idemp[0])  # needs sage.rings.number_field
Free module generated by {} over Rational Field
```

(continues on next page)
We recover the \(2 \times 2\) block of \(\mathbb{Q}[S_4]\) corresponding to the unique simple module of dimension 2 of the symmetric group \(S_4\):

```
\begin{verbatim}
sage: # needs sage.combinat sage.groups sage.rings.number_field
sage: A4 = SymmetricGroup(4).algebra(QQ)
sage: e = A4.central_orthogonal_idempotents()[2]
sage: A4.peirce_summand(e, e)
Free module generated by {0, 1, 2, 3} over Rational Field
\end{verbatim}
```

```
>>> from sage.all import *
```
so the left ideal generated by $x$ is one-dimensional:

```python
sage: Ax = A.principal_ideal(x, side='left'); Ax
Free module generated by (0) over Rational Field
sage: [B.lift() for B in Ax.basis()]
[x]
```

Multiplication on the left by $x$ annihilates only $x$ and fixes the other basis elements:

```python
sage: x*x, x*y, x*a, x*b
(x, 0, a, b)
```

so the right ideal generated by $x$ is 3-dimensional:

```python
sage: xA = A.principal_ideal(x, side='right'); xA
Free module generated by (0, 1, 2) over Rational Field
sage: [B.lift() for B in xA.basis()]
[x, a, b]
```

See also:

- `peirce_summand()`
- `radical()`

Return the Jacobson radical of `self`.

This uses `radical_basis()`, whose default implementation handles algebras over fields of characteristic zero or fields of characteristic $p$ in which we can compute $x^{1/p}$. 4.47. Finite dimensional algebras with basis
See also:

\texttt{radical\_basis()}, \texttt{semisimple\_quotient()}

EXAMPLES:

```python
sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
( containing the arrows a:x->y and b:x->y) over Rational Field
sage: radical = A.radical(); radical
Radical of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
( containing the arrows a:x->y and b:x->y) over Rational Field
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
( containing the arrows a:x->y and b:x->y) over Rational Field
>>> radical = A.radical(); radical
Radical of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
( containing the arrows a:x->y and b:x->y) over Rational Field
```

The radical is an ideal of \(A\), and thus a finite dimensional non unital associative algebra:

```python
sage: # needs sage.graphs sage.modules
sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: radical in AssociativeAlgebras(QQ).WithBasis().FiniteDimensional()
True
sage: radical in Algebras(QQ)
False
```

```python
sage: radical.dimension()
2
sage: radical.basis()
Finite family {0: B[0], 1: B[1]}
sage: radical.ambient() is A
True
sage: [c.lift() for c in radical.basis()]
[a, b]
```

```python
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> from sage.categories.associative_algebras import AssociativeAlgebras
>>> radical in AssociativeAlgebras(QQ).WithBasis().FiniteDimensional()
True
>>> radical in Algebras(QQ)
False

>>> # needs sage.graphs sage.modules
>>> radical.dimension()
2
>>> radical.basis()
```

(continues on next page)
Finite family {0: B[0], 1: B[1]}
>>> radical.ambient() is A
True
>>> [c.lift() for c in radical.basis()]
[a, b]

Todo:
- Tell Sage that the radical is in fact an ideal;
- Pickling by construction, as A.center();
- Lazy evaluation of _repr_.

radical_basis()

Return a basis of the Jacobson radical of this algebra.

Note: This implementation handles algebras over fields of characteristic zero (using Dixon’s lemma) or fields of characteristic \( p \) in which we can compute \( x^{1/p} \) [FR1985], [Eb1989].

OUTPUT:
- a list of elements of self.

See also:
radical(), Algebras.Semisimple

EXAMPLES:

```sage
sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: A.radical_basis()
(a, b)
```

```sage
>>> from sage.all import *
```

```sage
A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
>>> A.radical_basis()
(a, b)
```

We construct the group algebra of the Klein Four-Group over the rationals:

```sage
sage: A = KleinFourGroup().algebra(QQ)  # needs sage.groups sage.modules
```

```sage
>>> from sage.all import *
```

```sage
A = KleinFourGroup().algebra(QQ)  # needs sage.groups sage.modules
```

This algebra belongs to the category of finite dimensional algebras over the rationals:
Since the field has characteristic 0, Maschke’s Theorem tells us that the group algebra is semisimple. So its radical is the zero ideal:

```python
sage: A in Algebras(QQ).Semisimple()  #_
← needs sage.groups sage.modules
True
sage: A.radical_basis()  #_
← needs sage.groups sage.modules
()
```

Let’s work instead over a field of characteristic 2:

```python
sage: A = KleinFourGroup().algebra(GF(2))  #_
← needs sage.groups sage.modules
sage: A in Algebras(GF(2)).Semisimple()  #_
← needs sage.groups sage.modules
False
sage: A.radical_basis()  #_
← needs sage.groups sage.modules
(() + (1,2)(3,4), (3,4) + (1,2)(3,4), (1,2) + (1,2)(3,4))
```

We now implement the algebra \( A = K[x]/(x^p - 1) \), where \( K \) is a finite field of characteristic \( p \), and check its radical; alas, we currently need to wrap \( A \) to make it a proper `ModulesWithBasis`:

```python
sage: # needs sage.modules
sage: class AnAlgebra(CombinatorialFreeModule):
...... def __init__(self, F):
...... R.<x> = PolynomialRing(F)
...... I = R.ideal(x**F.characteristic() - F.one())
```
....: self._xbar = R.quotient(I).gen()
....: basis_keys = [self._xbar**i for i in range(F.
˓→characteristic())]
....: CombinatorialFreeModule.__init__(self, F, basis_keys,
˓→category=Algebras(F).FiniteDimensional().
˓→WithBasis())
....: def one(self):
....: return self.basis()[self.base_ring().one()]
....: def product_on_basis(self, w1, w2):
....: return self.from_vector(vector(w1*w2))

sage: AnAlgebra(GF(3)).radical_basis() # needs sage.libs.pari
(B[1] + 2*B[xbar^2], B[xbar] + 2*B[xbar^2])

sage: AnAlgebra(GF(16,'a')).radical_basis() # needs sage.rings.finite_rings
(B[1] + B[xbar],)

sage: AnAlgebra(GF(49,'a')).radical_basis() # needs sage.rings.finite_rings
˓→6])

We compute the radical basis in a subalgebra using the inherited product:

sage: scoeffs = {('a','e'): {'a':1}, ('b','e'): {'a':1, 'b':1},
˓→('c','d'): {'a':1}, ('c','e'): {'c':1}}

sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)

sage: MS = MatrixSpace(QQ, 5)

sage: A = MS.subalgebra([bg.adjoint_matrix() for bg in L.lie_algebra_
semisimple_quotient()

Return the semisimple quotient of self.

This is the quotient of self by its radical.

See also:

radical()

EXAMPLES:

sage: # needs sage.graphs sage.modules
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: a,b,x,y = sorted(A.basis())
sage: S = A.semisimple_quotient(); S
Semisimple quotient of An example of a finite dimensional algebra with_
--basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: S in Algebras(QQ).Semisimple()
True
sage: S.basis()
Finite family {'x': B['x'], 'y': B['y']}
sage: xs,ys = sorted(S.basis())
sage: (xs + ys) * xs
B['x']
(containing the arrows $a:x\rightarrow y$ and $b:x\rightarrow y$) over Rational Field

```python
>>> S in Algebras(QQ).Semisimple()
True
>>> S.basis()
Finite family {'x': B['x'], 'y': B['y']}
>>> xs,ys = sorted(S.basis())
>>> (xs + ys) * xs
B['x']
```

Sanity check: the semisimple quotient of the $n$-th descent algebra of the symmetric group is of dimension the number of partitions of $n$:

```python
 sage: [ DescentAlgebra(QQ,n).B().semisimple_quotient().dimension()  
     for n in range(6) ]
[1, 1, 2, 3, 5, 7]
 sage: [Partitions(n).cardinality() for n in range(10)]
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

```python
>>> from sage.all import *
>>> [ DescentAlgebra(QQ,n).B().semisimple_quotient().dimension()  
     for n in range(Integer(6)) ]
[1, 1, 2, 3, 5, 7]
>>> [Partitions(n).cardinality() for n in range(Integer(10))]
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

Todo:
- Pickling by construction, as `A.semisimple_quotient()`?
- Lazy evaluation of `_repr_`

**subalgebra** *(gens, category=None, *args, **opts)*

Return the subalgebra of `self` generated by `gens`.

**EXAMPLES:**

```python
 sage: scoeffs = {('a','e'): {'a':1}, ('b','e'): {'a':1, 'b':1},  
              ('c','d'): {'a':1}, ('c','e'): {'c':1}}
 sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
 sage: MS = MatrixSpace(QQ, 5)
 sage: A = MS.subalgebra([bg.adjoint_matrix() for bg in L.lie_algebra_  
                       .generators()])
 sage: A.dimension()
7
 sage: L.<x,y,z> = LieAlgebra(GF(3), {('x','y'): {'x':1, 'y':1}, ('y','z'  
           ->): {'y':1}})
 sage: MS = MatrixSpace(L.base_ring(), L.dimension())
 sage: gens = [b.adjoint_matrix() for b in L.basis()]
 sage: A = MS.subalgebra(gens)
 sage: A.dimension()
5
```
```python
>>> from sage.all import *
>>> scoeffs = {('a','e'): {'a':Integer(1), 'b':Integer(1)},
    ...
    ('c','d'): {'a':Integer(1), 'c':Integer(1)},
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e'));
>>> (a, b, c, d, e) = L._first_ngens(5)
>>> MS = MatrixSpace(QQ, Integer(5))
>>> A = MS.subalgebra([bg.adjoint_matrix() for bg in L.lie_algebra_generators()])
>>> A.dimension() 7
```

```python
>>> L = LieAlgebra(GF(Integer(3)), {('x', 'z'): {'x':Integer(1), 'y':Integer(1)},
    ...
    ('y', 'z'): {'y':Integer(1)}}, names=('x', 'y', 'z'));
>>> (x, y, z) = L._first_ngens(3)
>>> MS = MatrixSpace(L.base_ring(), L.dimension())
>>> gens = [b.adjoint_matrix() for b in L.basis()]
>>> A = MS.subalgebra(gens)
>>> A.dimension() 5
```

### class SubcategoryMethods

**Bases:** object

**Cellular()**

Return the full subcategory of the cellular objects of `self`.

See also:

Wikipedia article Cellular_algebra

**EXAMPLES:**

```
sage: Algebras(QQ).FiniteDimensional().WithBasis().Cellular()
Category of finite dimensional cellular algebras with basis over Rational Field
```

```
>>> from sage.all import *
>>> Algebras(QQ).FiniteDimensional().WithBasis().Cellular()
Category of finite dimensional cellular algebras with basis over Rational Field
```

### 4.48 Finite dimensional bialgebras with basis

The category of finite dimensional bialgebras with a distinguished basis

**EXAMPLES:**

```
sage: C = FiniteDimensionalBialgebrasWithBasis(QQ); C
Category of finite dimensional bialgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
  Category of finite dimensional algebras with basis over Rational Field]
```

(continues on next page)
sage: C.is_Bialgebras(QQ).WithBasis().FiniteDimensional()
True

>> from sage.all import *
>> C = FiniteDimensionalBialgebrasWithBasis(QQ); C
Category of finite dimensional bialgebras with basis over Rational Field
>> sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
 Category of finite dimensional algebras with basis over Rational Field]
>> C.is_Bialgebras(QQ).WithBasis().FiniteDimensional()
True

4.49 Finite dimensional coalgebras with basis

sage.categories.finite_dimensional_coalgebras_with_basis.FiniteDimensionalCoalgebrasWithBasis

The category of finite dimensional coalgebras with a distinguished basis

EXAMPLES:

sage: C = FiniteDimensionalCoalgebrasWithBasis(QQ); C
Category of finite dimensional coalgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of coalgebras with basis over Rational Field,
 Category of finite dimensional vector spaces with basis over Rational Field]
>> C.is_Coalgebras(QQ).WithBasis().FiniteDimensional()
True

>> from sage.all import *
>> C = FiniteDimensionalCoalgebrasWithBasis(QQ); C
Category of finite dimensional coalgebras with basis over Rational Field
>> sorted(C.super_categories(), key=str)
[Category of coalgebras with basis over Rational Field,
 Category of finite dimensional vector spaces with basis over Rational Field]
>> C.is_Coalgebras(QQ).WithBasis().FiniteDimensional()
True

4.50 Finite Dimensional Graded Lie Algebras With Basis

AUTHORS:

- Eero Hakavuori (2018-08-16): initial version

class sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis

Bases: CategoryWithAxiom_over_base_ring

Category of finite dimensional graded Lie algebras with a basis.

A grading of a Lie algebra $\mathfrak{g}$ is a direct sum decomposition $\mathfrak{g} = \bigoplus_i V_i$ such that $[V_i, V_j] \subset V_{i+j}$.

EXAMPLES:
sage: C = LieAlgebras(ZZ).WithBasis().FiniteDimensional().Graded(); C
Category of finite dimensional graded Lie algebras with basis over Integer Ring
sage: C.super_categories()
[Category of graded Lie algebras with basis over Integer Ring,
 Category of finite dimensional filtered modules with basis over Integer Ring,
 Category of finite dimensional Lie algebras with basis over Integer Ring]
sage: C is LieAlgebras(ZZ).WithBasis().FiniteDimensional().Graded()
True

class ParentMethods
    Bases: object

    homogeneous_component_as_submodule(d)

    Return the d-th homogeneous component of self as a submodule.

    EXAMPLES:

sage: C = LieAlgebras(QQ).WithBasis().Graded()
sage: C = C.FiniteDimensional().Stratified().Nilpotent()
sage: L = LieAlgebra(QQ, {(x,y): {z: 1}}, # needs sage.combinat sage.modules
....:   nilpotent=True, category=C)
sage: L.homogeneous_component_as_submodule(2) # needs sage.combinat sage.modules
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 0 1]

>>> from sage.all import *
>>> C = LieAlgebras(QQ).WithBasis().Graded(); C
Category of finite dimensional graded Lie algebras with basis over Integer Ring
>>> C.super_categories()
[Category of graded Lie algebras with basis over Integer Ring,
 Category of finite dimensional filtered modules with basis over Integer Ring,
 Category of finite dimensional Lie algebras with basis over Integer Ring]
>>> C is LieAlgebras(QQ).WithBasis().FiniteDimensional().Graded()
True

class Stratified(base_category)
    Bases: CategoryWithAxiom_over_base_ring

    Category of finite dimensional stratified Lie algebras with a basis.
A stratified Lie algebra is a graded Lie algebra that is generated as a Lie algebra by its homogeneous component of degree 1. That is to say, for a graded Lie algebra $L = \bigoplus_{k=1}^{M} L_k$, we have $L_{k+1} = [L_1, L_k]$.

**EXAMPLES:**

```python
sage: C = LieAlgebras(QQ).WithBasis().Graded().Stratified().FiniteDimensional()
sage: C
Category of finite dimensional stratified Lie algebras with basis over...

>>> from sage.all import *

>>> C = LieAlgebras(QQ).WithBasis().Graded().Stratified().FiniteDimensional()

>>> C
Category of finite dimensional stratified Lie algebras with basis over...
```

A finite-dimensional stratified Lie algebra is nilpotent:

```python
sage: C is C.Nilpotent()
True

>>> from sage.all import *

>>> C is C.Nilpotent()
True
```

```python
class ParentMethods
Bases: object

 degree_on_basis(m) 
Return the degree of the basis element indexed by m.

    If the degrees of the basis elements are not defined, they will be computed. By assumption the stratification $L_1 \oplus \cdots \oplus L_s$ of self is such that each component $L_k$ is spanned by some subset of the basis.

    The degree of a basis element $X$ is therefore the largest index $k$ such that $X \in L_k \oplus \cdots \oplus L_s$. The space $L_k \oplus \cdots \oplus L_s$ is by assumption the $k$-th term of the lower central series.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: C = LieAlgebras(QQ).WithBasis().Graded()

sage: C = C.FiniteDimensional().Stratified().Nilpotent()

sage: sc = {('X','Y'): {('Z'): 1}}

sage: L.<X,Y,Z> = LieAlgebra(QQ, sc, nilpotent=True, category=C)

sage: L.degree_on_basis(X.leading_support())

>>> sage: X.degree()

1

>>> sage: Y.degree()

1

>>> sage: L[X, Y]

Z

>>> sage: Z.degree()

2
```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> C = LieAlgebras(QQ).WithBasis().Graded()
>>> C = C.FiniteDimensional().Stratified().Nilpotent()
>>> sc = {('X', 'Y'): {'Z': Integer(1)}}
>>> L = LieAlgebra(QQ, sc, nilpotent=True, category=C, names=('X', 'Y', 'Z'))

4.51 Finite dimensional Hopf algebras with basis

class sage.categories.finite_dimensional_hopf_algebras_with_basis.FiniteDimensionalHopfAlgebrasWithBasis

    Bases: CategoryWithAxiom_over_base_ring

The category of finite dimensional Hopf algebras with a distinguished basis.

EXAMPLES:

sage: FiniteDimensionalHopfAlgebrasWithBasis(QQ)
Category of finite dimensional Hopf algebras with basis over Rational Field

sage: FiniteDimensionalHopfAlgebrasWithBasis(QQ).super_categories()
[Category of Hopf algebras with basis over Rational Field, 
 Category of finite dimensional algebras with basis over Rational Field]

>>> from sage.all import *
>>> FiniteDimensionalHopfAlgebrasWithBasis(QQ)
Category of finite dimensional Hopf algebras with basis over Rational Field

>>> FiniteDimensionalHopfAlgebrasWithBasis(QQ).super_categories()
[Category of Hopf algebras with basis over Rational Field, 
 Category of finite dimensional algebras with basis over Rational Field]

class ElementMethods

    Bases: object

class ParentMethods

    Bases: object
4.52 Finite Dimensional Lie Algebras With Basis

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

```python
class sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis:

    Bases: CategoryWithAxiom_over_base_ring

    Category of finite dimensional Lie algebras with a basis.

    Todo: Many of these tests should use non-abelian Lie algebras and need to be added after Issue #16820.
```

```python
class ElementMethods:

    Bases: object

    adjoint_matrix(sparse=False)

    Return the matrix of the adjoint action of self.

    EXAMPLES:

    sage: # needs sage.combinat sage.modules
    sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
    sage: L.an_element().adjoint_matrix()
    [0 0 0]
    [0 0 0]
    [0 0 0]
    sage: L.an_element().adjoint_matrix(sparse=True).is_sparse()
    True

    >>> from sage.all import *
    >>> # needs sage.combinat sage.modules
    >>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
    >>> L.an_element().adjoint_matrix()
    [0 0 0]
    [0 0 0]
    [0 0 0]
    >>> L.an_element().adjoint_matrix(sparse=True).is_sparse()
    True

    sage: # needs sage.combinat sage.modules
    sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})
    sage: x.adjoint_matrix()
    [0 1]
    [0 0]
    sage: y.adjoint_matrix()
    [-1 0]
    [ 0 0]

    >>> from sage.all import *
    >>> # needs sage.combinat sage.modules
    >>> L = LieAlgebra(QQ, {('x','y'): {'x':Integer(1)}, names=('x', 'y')))
    >>> (x, y,) = L._first_ngens(2)
```

(continues on next page)
We verify that this forms a representation:

```python
sage: # needs sage.combinat sage.modules
sage: sl3 = lie_algebras.sl(QQ, 3)
sage: e1, e2 = sl3.e(1), sl3.e(2)
sage: e12 = e1.bracket(e2)
sage: E1, E2 = e1.adjoint_matrix(), e2.adjoint_matrix()
sage: E1 * E2 - E2 * E1 == e12.adjoint_matrix()
True
```
Nilpotent

Alias of FiniteDimensionalNilpotentLieAlgebrasWithBasis

class ParentMethods

Bases: object

as_finite_dimensional_algebra()

Return self as a FiniteDimensionalAlgebra.

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.cross_product(QQ)
sage: x, y, z = L.basis()
sage: F = L.as_finite_dimensional_algebra()
sage: X, Y, Z = F.basis()
sage: x.bracket(y)
Z
sage: X * Y
Z

casimir_element(order=2, UEA=None, force_generic=False, basis=False)

Return a Casimir element of order order in the universal enveloping algebra of self.

A Casimir element of order \( k \) is a distinguished basis element for the center of \( U(\mathfrak{g}) \) of homogeneous degree \( k \) (that is, it is an element of \( U_k \setminus U_{k-1} \), where \( \{U_i\}_{i=0}^{\infty} \) is the natural filtration of \( U(\mathfrak{g}) \)). When \( \mathfrak{g} \) is a simple Lie algebra, then this spans \( Z(U(\mathfrak{g}))_k \).

INPUT:
• order – (default: 2) the order of the Casimir element
• UEA – (optional) the universal enveloping algebra implementation to return the result in
• force_generic – (default: False) if True for the quadratic order, then this uses the default algorithm; otherwise this is ignored
• basis – (default: False) if True, this returns a basis of all Casimir elements of order order as a list

ALGORITHM:
For the quadratic order (i.e., order=2), then this uses \( K^U \), the inverse of the Killing form matrix, to compute \( C(2) = \sum_{i,j} K_{ij} X_i \cdots X_j \), where \( \{X_1, \ldots, X_n\} \) is a basis for \( \mathfrak{g} \). Otherwise this solves the
system of equations

\[
\sum_{i_1, \ldots, i_k} \kappa_{i_1 \cdots i_k} X_{i_1} \cdots X_{i_k}.
\]

for the symmetric tensor \( \kappa_{i_1 \cdots i_k} \), where \( k \) is the order. This system comes from \([X_i, C_{(k)}] = 0\) with

\[
C_{(k)} = \sum_{i_1, \ldots, i_k} \kappa_{i_1 \cdots i_k} X_{i_1} \cdots X_{i_k}.
\]

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebra(QQ, cartan_type=['A', 1])
sage: C = L.casimir_element(); C
1/8*b1^2 + 1/2*b0*b2 - 1/4*b1
sage: U = L.universal_enveloping_algebra()
sage: all(g * C == C * g for g in U.gens())
True
sage: U = L.pbw_basis()
sage: C = L.casimir_element(UEA=U); C
1/2*PBW[alpha[1]]*PBW[-alpha[1]] + 1/8*PBW[alphacheck[1]]^2
- 1/4*PBW[alphacheck[1]]
```

(continues on next page)
\[ 2 \cdot \text{PBW}[\alpha[2]] \cdot \text{PBW}[-\alpha[2]] + 1/2 \cdot \text{PBW}[\text{alphacheck}[2]]^2 \rightarrow \text{PBW}[\alpha[2]] , \\
2 \cdot \text{PBW}[\alpha[1]] \cdot \text{PBW}[-\alpha[1]] + 1/2 \cdot \text{PBW}[\text{alphacheck}[1]]^2 \rightarrow \text{PBW}[\alpha[1]] \]

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
L = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
C = L.casimir_element(); C
1/8*b1^2 + 1/2*b0*b2 - 1/4*b1
>>> U = L.universal_enveloping_algebra()
>>> all(g * C == C * g for g in U.gens())
True
>>> U = L.pbw_basis()
>>> C = L.casimir_element(UEA=U); C
1/2*PBW[alpha[1]]*PBW[-alpha[1]] + 1/8*PBW[alphacheck[1]]^2
- 1/4*PBW[alphacheck[1]]
>>> all(g * C == C * g for g in U.algebra_generators())
True
```

```python
>>> # needs sage.combinat sage.modules
L = LieAlgebra(QQ, cartan_type=['B', Integer(2)])
U = L.pbw_basis()
C = L.casimir_element(UEA=U)
>>> all(g * C == C * g for g in U.algebra_generators())
True
>>> # needs sage.combinat sage.modules
L = LieAlgebra(QQ, cartan_type=['C', Integer(3)])
U = L.pbw_basis()
C = L.casimir_element(UEA=U)
>>> all(g * C == C * g for g in U.algebra_generators())
True
>>> # needs sage.combinat sage.modules
L = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
C4 = L.casimir_element(order=Integer(4), UEA=L.pbw_basis()); C4
4*PBW[alpha[1]]^2*PBW[-alpha[1]]^2
+ 2*PBW[alpha[1]]*PBW[alphacheck[1]]^2*PBW[-alpha[1]]
+ 1/4*PBW[alphacheck[1]]^4 - PBW[alphacheck[1]]^3
- 4*PBW[alpha[1]]*PBW[-alpha[1]] + 2*PBW[alphacheck[1]]
>>> all(g * C4 == C4 * g for g in L.pbw_basis().algebra_generators())
True
```

```python
>>> # needs sage.combinat sage.modules
g = LieAlgebra(QQ, cartan_type=['D', Integer(2)])
U = g.pbw_basis()
U.casimir_element(Integer(2), basis=True)
2*PBW[alpha[1]]*PBW[-alpha[1]] + 1/2*PBW[alphacheck[1]]^2 \rightarrow \text{PBW}[\alpha[1]] ]
```

4.52. Finite Dimensional Lie Algebras With Basis
Todo: Use the symmetry of the tensor to reduce the number of equations and/or variables to solve.

center()

Return the center of self.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: Z = L.center(); Z
An example of a finite dimensional Lie algebra with basis: the
3-dimensional abelian Lie algebra over Rational Field
sage: Z.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

centralizer(S)

Return the centralizer of S in self.

INPUT:
• S—a subalgebra of self or a list of elements that represent generators for a subalgebra

See also:
centralizer_basis()

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.centralizer([a + b, 2*a + c]); S
An example of a finite dimensional Lie algebra with basis: the
3-dimensional abelian Lie algebra over Rational Field
sage: S.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

(continues on next page)
the 3-dimensional abelian Lie algebra over Rational Field

```python
S.basis_matrix()
```

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

centralizer_basis(S)

Return a basis of the centralizer of \( S \) in \( self \).

INPUT:

- \( S \) – a subalgebra of \( self \) or a list of elements that represent generators for a subalgebra

See also:

- centralizer()

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.centralizer_basis([a + b, 2*a + c])
```

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

```python
sage: # needs sage.combinat sage.modules
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: H.centralizer_basis(H)
```

```
[z]
```

```python
sage: # needs sage.combinat sage.groups sage.modules
sage: D = DescentAlgebra(QQ, 4).D()
sage: L = LieAlgebra(associative=D)
sage: L.centralizer_basis(L)
```

```
[D{}, D{1} + D{1, 2} + D{2, 3} + D{3}, D{1, 2, 3} + D{1, 3} + D{2}]
```

```python
sage: D.center_basis()
```

```
[D{}, D{1} + D{1, 2} + D{2, 3} + D{3}, D{1, 2, 3} + D{1, 3} + D{2}]
```

```python
sage: scoeffs = {('a','d'): {'a':1}, ('a','e'): {'b':-1}, ...
   ('b','d'): {'b':1}, ('b','e'): {'a':1}, ...
   ('d','e'): {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.centralizer_basis([a, c])
```

```
[a, b, c]
```

```python
sage: L.centralizer_basis([a, e])
```

```
[c]
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
```

```python
>>> L.centralizer_basis([a + b, Integer(2)*a + c])
```

```
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```
chevalley_eilenberg_complex(M=None, dual=False, sparse=True, ncpus=None)

Return the Chevalley-Eilenberg complex of self.

Let \( g \) be a Lie algebra and \( M \) be a right \( g \)-module. The *Chevalley-Eilenberg complex* is the chain complex on

\[
C^*(g, M) = M \otimes \bigwedge^* g,
\]

where the differential is given by

\[
d(m \otimes g_1 \wedge \cdots \wedge g_p) = \sum_{i=1}^{p} (-1)^{i+1} (mg_i) \otimes g_1 \wedge \cdots \wedge \hat{g}_i \wedge \cdots \wedge g_p + \sum_{1 \leq i < j \leq p} (-1)^{i+j} m \otimes [g_i, g_j] \wedge g_1 \wedge \cdots \wedge \hat{g}_i \wedge \cdots \wedge g_p.
\]

**INPUT:**
- \( M \) (default: the trivial 1-dimensional module) one of the following:
  - a module \( M \) with an action of self
  - a dictionary whose keys are basis elements and values are matrices representing a Lie algebra homomorphism defining the representation
- \( \text{dual} \) (default: \( \text{False} \)) if \( \text{True} \), causes the dual of the complex to be computed
- \( \text{sparse} \) (default: \( \text{True} \)) whether to use sparse or dense matrices
- \( \text{ncpus} \) (optional) how many cpus to use

**EXAMPLES:**

sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.sl(ZZ, 2)
sage: C = L.chevalley_eilenberg_complex(); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
[[-2  0  0]  [0]  (continues on next page)
[ 0 1 0 ]  [0]  
[0 0 0]  [ 0 0 -2]  [0]  
0 <-- C_0 <-------- C_1 <---------- C_2 <---- C_3 <-- 0

sage: # needs sage.combinat sage.modules
sage: L = LieAlgebra(QQ, cartan_type=['C',2])

sage: C = L.chevalley_eilenberg_complex()  # long time
sage: [C.free_module_rank(i) for i in range(11)]  # long time
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

sage: # needs sage.combinat sage.modules
sage: g = lie_algebras.sl(QQ, 2)

sage: E, F, H = g.basis()

sage: n = g.subalgebra([F, H])

sage: ascii_art(n.chevalley_eilenberg_complex())

[ 0
[0 0]
[0 0 0 0]
0 <-- C_0 <---------- C_1 <---------- C_2 <-- 0

sage: L.<x,y> = LieAlgebra(QQ, {(x,y): {y:1}})

sage: f = ({x: Matrix([[1,0],[0,0]]), y: Matrix([[0,1],[0,0]])})

sage: C = L.chevalley_eilenberg_complex(f); C
Chain complex with at most 3 nonzero terms over Rational Field

sage: ascii_art(C)

[ 0 -1]
[ 2 0]
[1 0 0 1]  [ 0 0]
[0 0 0 0]  [ 0 1]
0 <-- C_0 <---------- C_1 <---------- C_2 <-- 0

sage: ascii_art(L.chevalley_eilenberg_complex(f, sparse=False))

[ 0 -1]
[ 2 0]
[1 0 0 1]  [ 0 0]
[0 0 0 0]  [ 0 1]
0 <-- C_0 <---------- C_1 <---------- C_2 <-- 0

>>> from sage.all import *

>>> # needs sage.combinat sage.modules

>>> L = lie_algebras.sl(ZZ, Integer(2))

>>> C = L.chevalley_eilenberg_complex(); C
Chain complex with at most 4 nonzero terms over Integer Ring

>>> ascii_art(C)

[-2 0 0]  [0]
[ 0 1 0]  [0]
[0 0 0]  [ 0 0 -2]  [0]
0 <-- C_0 <---------- C_1 <---------- C_2 <---- C_3 <-- 0

>>> # needs sage.combinat sage.modules

>>> L = LieAlgebra(QQ, cartan_type=['C',Integer(2)])

>>> C = L.chevalley_eilenberg_complex()  # long time

>>> [C.free_module_rank(i) for i in range(Integer(11))]  # long time
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

>>> g = lie_algebras.sl(QQ, Integer(2))
>>> E, F, H = g.basis()
>>> n = g.subalgebra([F, H])

```python
>>> ascii_art(n.chevalley_eilenberg_complex())

```

```text
[ 0]
[0 0] 
[ 0] 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
```

```python
>>> L = LieAlgebra(QQ, {('x', 'y'): {'y': Integer(1)}}, names=('x', 'y',));
˓→ (x, y) = L._first_ngens(2)

```python
>>> f = ({x: Matrix([[Integer(1), Integer(0)],[Integer(0), Integer(0)]],...
˓→ y: Matrix([[Integer(0), Integer(1)],[Integer(0), Integer(0)]])})

```python
>>> C = L.chevalley_eilenberg_complex(f); C

```

```
Chain complex with at most 3 nonzero terms over Rational Field
```

```python
>>> ascii_art(C)

```

```text
[ 0 -1]
[ 2 0] 0 <-- C_0 <---------- C_1 <-------- C_2 <-- 0
```

```python
>>> ascii_art(L.chevalley_eilenberg_complex(f, sparse=False))

```

```text
[ 0 -1]
[ 2 0] 0 <-- C_0 <---------- C_1 <-------- C_2 <-- 0
```

REFERENCES:

- Wikipedia article Lie_algebra_cohomology#Chevalley-Eilenberg_complex
- [Wei1994] Chapter 7

doctest: cohomology (deg=None, M=None, sparse=True, ncpus=None)

Return the Lie algebra cohomology of self.

The Lie algebra cohomology is the cohomology of the Chevalley-Eilenberg cochain complex (which is the dual of the Chevalley-Eilenberg chain complex).

Let g be a Lie algebra and M a left g-module. It is known that $H^0(g; M)$ is the subspace of g-invariants of M:

$$H^0(g; M) = M^g = \{ m \in M \mid gm = 0 \text{ for all } g \in g \}.$$  

Additionally, $H^1(g; M)$ is the space of derivations $g \to M$ modulo the space of inner derivations, and $H^2(g; M)$ is the space of equivalence classes of Lie algebra extensions of $g$ by $M$.

EXAMPLES:

```sage
sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.so(QQ, 4)

```sage
sage: L.cohomology()

```sage
{0: Vector space of dimension 1 over Rational Field,
}
1: Vector space of dimension 0 over Rational Field,
2: Vector space of dimension 0 over Rational Field,
3: Vector space of dimension 2 over Rational Field,
4: Vector space of dimension 0 over Rational Field,
5: Vector space of dimension 0 over Rational Field,
6: Vector space of dimension 1 over Rational Field

```
sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.Heisenberg(QQ, 2)
sage: L.cohomology()
{0: Vector space of dimension 1 over Rational Field,
  1: Vector space of dimension 4 over Rational Field,
  2: Vector space of dimension 5 over Rational Field,
  3: Vector space of dimension 5 over Rational Field,
  4: Vector space of dimension 4 over Rational Field,
  5: Vector space of dimension 1 over Rational Field}
```

```
sage: # needs sage.combinat sage.modules
d = {(x, y): {y: 2}}
sage: L.<x,y> = LieAlgebra(ZZ, d)
sage: L.cohomology()
{0: Z, 1: Z, 2: C2}
```

```
>>> from sage.all import *

>>> # needs sage.combinat sage.modules
L = lie_algebras.so(QQ, Integer(4))
>>> L.cohomology()
{0: Vector space of dimension 1 over Rational Field,
  1: Vector space of dimension 0 over Rational Field,
  2: Vector space of dimension 0 over Rational Field,
  3: Vector space of dimension 2 over Rational Field,
  4: Vector space of dimension 0 over Rational Field,
  5: Vector space of dimension 0 over Rational Field,
  6: Vector space of dimension 1 over Rational Field}
```

```
>>> # needs sage.combinat sage.modules
L = lie_algebras.Heisenberg(QQ, Integer(2))
>>> L.cohomology()
{0: Vector space of dimension 1 over Rational Field,
  1: Vector space of dimension 4 over Rational Field,
  2: Vector space of dimension 5 over Rational Field,
  3: Vector space of dimension 5 over Rational Field,
  4: Vector space of dimension 4 over Rational Field,
  5: Vector space of dimension 1 over Rational Field}
```

```
>>> d = {(x', y'): {y': Integer(2)}}
>>> L = LieAlgebra(ZZ, d, names=('x', 'y')); (x, y,) = L._first_ngens(2)
>>> L.cohomology()
{0: Z, 1: Z, 2: C2}
```

See also:

`chevalley_ellenberg_complex()`

REFERENCES:

- Wikipedia article Lie_algebra_cohomology

4.52. Finite Dimensional Lie Algebras With Basis
derivations_basis()  
Return a basis for the Lie algebra of derivations of self as matrices.  
A derivation $D$ of an algebra is an endomorphism of $A$ such that  
$$D([a, b]) = [D(a), b] + [a, D(b)]$$  
for all $a, b \in A$. The set of all derivations form a Lie algebra.  
EXAMPLES:  
We construct the derivations of the Heisenberg Lie algebra:

```
sage: # needs sage.combinat sage.modules  
sage: H = lie_algebras.Heisenberg(QQ, 1)  
sage: H.derivations_basis()  
([0 0 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0]  
[0 0 0] [0 0 0] [1 0 0] [0 1 0] [0 0 0] [0 0 0]  
[0 0 1], [0 0 0], [0 0 0], [0 0 1], [1 0 0], [0 1 0]  
)  
```

```
>>> from sage.all import *  
>>> # needs sage.combinat sage.modules  
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))  
>>> H.derivations_basis()  
([0 0 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0]  
[0 0 0] [0 0 0] [1 0 0] [0 1 0] [0 0 0] [0 0 0]  
[0 0 1], [0 0 0], [0 0 0], [0 0 1], [1 0 0], [0 1 0]  
)  
```

We construct the derivations of $sl_2$:  
```
sage: # needs sage.combinat sage.modules  
sage: sl2 = lie_algebras.sl(QQ, 2)  
sage: sl2.derivations_basis()  
([ 1 0 0] [ 0 1 0] [ 0 0 0]  
[ 0 0 0] [ 0 0 -1/2] [ 1 0 0]  
[ 0 0 -1], [ 0 0 0], [ 0 -2 0]  
)  
```

```
>>> from sage.all import *  
>>> # needs sage.combinat sage.modules  
>>> sl2 = lie_algebras.sl(QQ, Integer(2))  
>>> sl2.derivations_basis()  
([ 1 0 0] [ 0 1 0] [ 0 0 0]  
[ 0 0 0] [ 0 0 -1/2] [ 1 0 0]  
[ 0 0 -1], [ 0 0 0], [ 0 -2 0]  
)  
```

We verify these are derivations:

```
sage: # needs sage.combinat sage.modules  
sage: D = [sl2.module_morphism(matrix=M, codomain=sl2)  
.....:: for M in sl2.derivations_basis()]  
```
REFERENCES:

Wikipedia article Derivation_(differential_algebra)

\texttt{derived\_series()}

Return the derived series \((g^{(i)})\), of \texttt{self} where the rightmost \(g^{(k)} = g^{(k+1)} = \cdots\).

We define the derived series of a Lie algebra \(g\) recursively by
\(g^{(0)} := g\) and
\(g^{(k+1)} = [g^{(k)}, g^{(k)}]\)

and recall that \(g^{(k)} \supseteq g^{(k+1)}\). Alternatively we can express this as

\[ g \supseteq \mathfrak{g} \supseteq \mathfrak{g} \left[ \mathfrak{g}, \mathfrak{g} \right] \supseteq \left[ \left[ \mathfrak{g}, \mathfrak{g} \right], \left[ \mathfrak{g}, \mathfrak{g} \right] \right] \supseteq \cdots.\]

EXAMPLES:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.derived_series()

(An example of a finite dimensional Lie algebra with basis: 
the 3-dimensional abelian Lie algebra over Rational Field, 
An example of a finite dimensional Lie algebra with basis: 
the 0-dimensional abelian Lie algebra over Rational Field 
with basis matrix: [])
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.derived_series()

(An example of a finite dimensional Lie algebra with basis: 
the 3-dimensional abelian Lie algebra over Rational Field, 
An example of a finite dimensional Lie algebra with basis: 
the 0-dimensional abelian Lie algebra over Rational Field 
with basis matrix: [])
\end{verbatim}

\begin{verbatim}
sage: # needs sage.combinat sage.modules
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):<'x':1})
sage: L.derived_series()

(Lie algebra on 2 generators (x, y) over Rational Field, 
Ideal (x) of Lie algebra on 2 generators (x, y) over Rational Field, 
Ideal () of Lie algebra on 2 generators (x, y) over Rational Field)
\end{verbatim}
```
sage: scoeffs = {('a','d'): {'a':1}, ('a','e'): {'b':-1},
            ('b','d'): {'b':1}, ('b','e'): {'a':1},
            ('d','e'): {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.derived_series()
(Lie algebra on 5 generators (a, b, c, d, e) over Rational Field,
  Ideal (a, b, c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field,
  Ideal () of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field)
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebra(QQ, {('x','y'): {'x':Integer(1)}}, names=('x', 'y',));
    (x, y) = L._first_ngens(2)
>>> L.derived_series()
(Lie algebra on 2 generators (x, y) over Rational Field,
  Ideal (x) of Lie algebra on 2 generators (x, y) over Rational Field,
  Ideal () of Lie algebra on 2 generators (x, y) over Rational Field)
```

```
>>> scoeffs = {('a','d'): {'a':Integer(1)}, ('a','e'): {'b':-Integer(1)},
            ('b','d'): {'b':Integer(1)}, ('b','e'): {'a':Integer(1)},
            ('d','e'): {'c':Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e',)); (a, b,
    c, d, e,) = L._first_ngens(5)
>>> L.derived_series()
(Lie algebra on 5 generators (a, b, c, d, e) over Rational Field,
  Ideal (a, b, c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field,
  Ideal () of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field)
```

#### derived_subalgebra()

Return the derived subalgebra of `self`.

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.derived_subalgebra()
An example of a finite dimensional Lie algebra with basis:
  the 0-dimensional abelian Lie algebra over Rational Field with basis matrix:
  []
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.derived_subalgebra()
An example of a finite dimensional Lie algebra with basis:
  the 0-dimensional abelian Lie algebra over Rational Field with basis matrix:
  []
```

If `self` is semisimple, then the derived subalgebra is `self`:
Category Framework, Release 10.4

```python
sage: # needs sage.combinat sage.modules
sage: s13 = LieAlgebra(QQ, cartan_type=['A', 2])
sage: s13.derived_subalgebra()
Lie algebra of ['A', 2] in the Chevalley basis
sage: s13 is s13.derived_subalgebra()
True

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> s13 = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> s13.derived_subalgebra()
Lie algebra of ['A', 2] in the Chevalley basis
>>> s13 is s13.derived_subalgebra()
True

faithful_representation(algorithm=None)

Return a faithful representation of self.

By Ado's and Iwasawa's theorems, every finite dimensional Lie algebra has a faithful finite dimensional representation.

INPUT:

- algorithm – one of the following depending on the classification of the Lie algebra:

  Nilpotent:
  - 'regular' – use the universal enveloping algebra quotient
  - 'minimal' – construct the minimal representation (for precise details, see the documentation of

  Solvable:
  - Not implemented

  General case
  - 'generic' – generic algorithm (only implemented currently for positive characteristic)

Note that the algorithm for any more generic cases can be used in the specialized cases. For instance, using 'generic' for any Lie algebra (e.g., even if nilpotent) will use the generic implementation.

EXAMPLES:

sage: H2 = lie_algebras.Heisenberg(QQ, 2)
sage: H2.is_nilpotent()
True
sage: F = H2.faithful_representation(); F
Faithful 16 dimensional representation of Heisenberg algebra of rank 2 over Rational Field
sage: M = H2.faithful_representation(algorithm="minimal"); M
Minimal faithful representation of Heisenberg algebra of rank 2 over Rational Field
sage: M.dimension()
4
sage: H2.faithful_representation(algorithm="invalid")
Traceback (most recent call last):
...
ValueError: invalid algorithm 'invalid'
```
\[
\text{\texttt{\ldots:\quad\{('b','d'):\{('b':1), ('b','e'): \{('a':1),}
\ldots:\quad\{('d','e'):\{('c':1)\}\)}}
\]
\]
\[
\text{sage: \quad L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)}
\]
\]
\[
\text{sage: \quad L.is_nilpotent()}
\]
\[
\text{False}
\]
\]
\[
\text{sage: \quad L.is_solvable()}
\]
\]
\[
\text{True}
\]
\]
\[
\text{sage: \quad L.faithful_representation()}
\]
\]
\[
\text{Traceback (most recent call last):}
\]
\]
\[
\text{...}
\]
\[
\text{NotImplementedError: only implemented for nilpotent Lie algebras}
\]
\]
\[
\text{sage: \quad sl3 = LieAlgebra(QQ, cartan_type=['A', 2])}
\]
\]
\[
\text{sage: \quad sl3.is_semisimple()}
\]
\]
\[
\text{True}
\]
\]
\[
\text{sage: \quad sl3.faithful_representation()}
\]
\]
\[
\text{Traceback (most recent call last):}
\]
\]
\[
\text{...}
\]
\[
\text{NotImplementedError: only implemented for nilpotent Lie algebras}
\]
\]
\[
\text{>>> \quad \textbf{from sage.all import} *}
\]
\]
\[
\text{>>> \quad H2 = lie_algebras.Heisenberg(QQ, Integer(2))}
\]
\]
\[
\text{>>> \quad H2.is_nilpotent()}
\]
\]
\[
\text{True}
\]
\]
\[
\text{>>> \quad F = H2.faithful_representation(); F}
\]
\]
\[
\text{False}
\]
\]
\[
\text{>>> \quad M = H2.faithful_representation(algorithm="minimal"); M}
\]
\]
\[
\text{Minimal faithful representation of}
\]
\]
\[
\text{Heisenberg algebra of rank 2 over Rational Field}
\]
\[
\text{>>> \quad M.dimension()}
\]
\]
\[
\text{4}
\]
\]
\[
\text{>>> \quad H2.faithful_representation(algorithm="invalid")}
\]
\]
\[
\text{...}
\]
\[
\text{ValueError: invalid algorithm 'invalid'}
\]
\]
\[
\text{>>> \quad scoeffs = \{(\textbf{\texttt{\texttt{}}'a'}, \textbf{\texttt{\texttt{}}'d'}): \{\textbf{\texttt{\texttt{}}'a':Integer(1), \textbf{\texttt{\texttt{}}'a','e'): \{\textbf{\texttt{\texttt{}}'b':-Integer(1),}
\ldots
\quad\{('b','d'):\{('b':Integer(1)), ('b','e'): \{('a':Integer(1)),
\ldots
\quad\{('d','e'):\{('c':Integer(1))\}\}
\]
\]
\[
\text{L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b, ...
\text{\ldots}
\quad\{('d','e'):\{('c':Integer(1))\}\}
\]
\]
\[
\text{L.<a,b,c,d,e> = L._first_ngens(5)}
\]
\]
\[
\text{>>> \quad L.is_nilpotent()}
\]
\]
\[
\text{False}
\]
\]
\[
\text{sage: \quad L.is_solvable()}
\]
\]
\[
\text{True}
\]
\]
\[
\text{sage: \quad L.faithful_representation()}
\]
\]
\[
\text{Traceback (most recent call last):}
\]
\]
\[
\text{...}
\]
\[
\text{NotImplementedError: only implemented for nilpotent Lie algebras}
\]
\]
\[
\text{sage: \quad sl3 = LieAlgebra(QQ, cartan_type=['A', Integer(2)])}
\]
\]
\[
\text{sage: \quad sl3.is_semisimple()}
\]
\]
\[
\text{True}
\]
\]
\[
\text{sage: \quad sl3.faithful_representation()}
\]
\]
\[
\text{Traceback (most recent call last):}
\]
\]
\[
\text{...}
from_vector \(v, \text{order=None}\)

Return the element of \(self\) corresponding to the vector \(v\) in \(self\.module()\).

Implement this if you implement \(\text{module()}\); see the documentation of \(\text{sage.categories.lie_algebras.LieAlgebras().module()}\) for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u  # needs sage.modules
(1, 0, 0)
sage: parent(u) is L  # needs sage.modules
True
```

homology \((\text{deg=None, M=None, sparse=True, ncpus=None})\)

Return the Lie algebra homology of \(self\).

The Lie algebra homology is the homology of the Chevalley-Eilenberg chain complex.

INPUT:

- \(\text{deg}\) – the degree of the homology (optional)
- \(\text{M}\) – (default: the trivial module) a right module of \(self\)
- \(\text{sparse}\) – (default: True) whether to use sparse matrices for the Chevalley-Eilenberg chain complex
- \(\text{ncpus}\) – (optional) how many cpus to use when computing the Chevalley-Eilenberg chain complex

EXAMPLES:

```python
sage: L = lie_algebras.cross_product(QQ)
sage: L.homology()  # needs sage.combinat sage.modules
{0: Vector space of dimension 1 over Rational Field, 1: Vector space of dimension 0 over Rational Field, 2: Vector space of dimension 0 over Rational Field, 3: Vector space of dimension 1 over Rational Field}
sage: L = lie_algebras.pwitt(GF(5), 5)
sage: L.homology()  # needs sage.combinat sage.modules
{0: Vector space of dimension 1 over Finite Field of size 5, 1: Vector space of dimension 0 over Finite Field of size 5, 2: Vector space of dimension 0 over Finite Field of size 5, 3: Vector space of dimension 1 over Finite Field of size 5, 4: Vector space of dimension 0 over Finite Field of size 5, 5: Vector space of dimension 0 over Finite Field of size 5, 6: Vector space of dimension 0 over Finite Field of size 5, 7: Vector space of dimension 0 over Finite Field of size 5, 8: Vector space of dimension 1 over Finite Field of size 5}"
```
2: Vector space of dimension 1 over Finite Field of size 5,
3: Vector space of dimension 1 over Finite Field of size 5,
4: Vector space of dimension 0 over Finite Field of size 5,
5: Vector space of dimension 1 over Finite Field of size 5)

```
sage: # needs sage.combinat sage.modules
dsage: d = {('x', 'y'): {'y': 2}}
sage: L.<x,y> = LieAlgebra(ZZ, d)
sage: L.homology()
{0: Z, 1: Z x C2, 2: 0}
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = lie_algebras.cross_product(QQ)
>>> L.homology()
{0: Vector space of dimension 1 over Rational Field,
  1: Vector space of dimension 0 over Rational Field,
  2: Vector space of dimension 0 over Rational Field,
  3: Vector space of dimension 1 over Rational Field}
```

```
>>> # needs sage.combinat sage.modules
>>> L = lie_algebras.pwitt(GF(Integer(5)), Integer(5))
>>> L.homology()
{0: Vector space of dimension 1 over Finite Field of size 5,
  1: Vector space of dimension 0 over Finite Field of size 5,
  2: Vector space of dimension 1 over Finite Field of size 5,
  3: Vector space of dimension 1 over Finite Field of size 5,
  4: Vector space of dimension 0 over Finite Field of size 5,
  5: Vector space of dimension 1 over Finite Field of size 5}
```

```
>>> d = {('x', 'y'): {'y': Integer(2)}}
>>> L = LieAlgebra(ZZ, d, names=(x, y,)); (x, y,) = L._first_ngens(2)
>>> L.homology()
{0: Z, 1: Z x C2, 2: 0}
```

See also:

```
chevalley_eilenberg_complex()
```

```
hypercenter()
```

Return the hypercenter of self.

EXAMPLES:

```
sage: SGA3 = SymmetricGroup(3).algebra(QQ)
sage: L = LieAlgebra(associative=SGA3)
sage: L.hypercenter()
Ideal ((), (1,2,3) + (1,3,2), (2,3) + (1,2) + (1,3)) of
  Lie algebra of Symmetric group algebra of order 3
  over Rational Field
```

```
sage: L = lie_algebras.Heisenberg(QQ, 3)
sage: L.hypercenter()
Heisenberg algebra of rank 3 over Rational Field
```
>>> from sage.all import *
>>> SGA3 = SymmetricGroup(Integer(3)).algebra(QQ)
>>> L = LieAlgebra(associative=SGA3)
>>> L.hypercenter()
Ideal ((), (1,2,3) + (1,3,2), (2,3) + (1,2) + (1,3)) of
Lie algebra of Symmetric group algebra of order 3
over Rational Field

>>> L = lie_algebras.Heisenberg(QQ, Integer(3))
>>> L.hypercenter()
Heisenberg algebra of rank 3 over Rational Field

ideal (*gens, **kwds)

Return the ideal of self generated by gens.

INPUT:
• gens – a list of generators of the ideal
• category – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

EXAMPLES:
sage: # needs sage.combinat sage.modules
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: p1,p2,q1,q2,z = H.basis()
sage: I = H.ideal([p1 - p2, q1 - q2])
sage: I.basis().list()
[-p1 + p2, -q1 + q2, z]
sage: I.reduce(p1 + p2 + q1 + q2 + z)
2*p1 + 2*q1

Passing an extra category to an ideal:
sage: # needs sage.combinat sage.modules
sage: L.<x,y,z> = LieAlgebra(QQ, abelian=True)
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: C = C.Subobjects().Graded().Stratified()
sage: I = L.ideal(x, y, category=C)
sage: I.homogeneous_component_basis(1).list()
[x, y]
inner_derivations_basis()

Return a basis for the Lie algebra of inner derivations of self as matrices.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: H.inner_derivations_basis()
([0 0 0] [0 0 0]
[0 0 0] [0 0 0]
[1 0 0], [0 1 0]
)
```

is_abelian()

Return if self is an abelian Lie algebra.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_abelian()
True
```

```python
sage: # needs sage.combinat sage.modules
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})
sage: L.is_abelian()
False
```

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebra(QQ, {('x','y'): {'x':Integer(1)}}, names=('x', 'y',));
˓→(x, y,) = L._first_ngens(2)
sage: L.is_abelian()
False
```
is_ideal(A)

Return if self is an ideal of A.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: I = L.ideal([2*a - c, b + c])
sage: I.is_ideal(L)
True

sage: L.<x,y> = LieAlgebra(QQ, {{'x','y'}:{'x':1}})  # needs sage.combinat sage.modules
sage: L.is_ideal(L)  # needs sage.combinat sage.modules
True

sage: F = LieAlgebra(QQ, 'F', representation='polynomial')  # needs sage.combinat sage.modules
sage: L.is_ideal(F)  # needs sage.combinat sage.modules
Traceback (most recent call last):
... Not ImplementedError: A must be a finite dimensional Lie algebra with basis
```

is_nilpotent()

Return if self is a nilpotent Lie algebra.

A Lie algebra is nilpotent if the lower central series eventually becomes 0.

EXAMPLES:

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> I = L.ideal([Integer(2)*a - c, b + c])
>>> I.is_ideal(L)
True

>>> L = LieAlgebra(QQ, {{'x','y'}:{'x':Integer(1)}}, names=('x', 'y',));  
# needs sage.combinat sage.modules
>>> L.is_ideal(L)  # needs sage.combinat sage.modules
True

>>> F = LieAlgebra(QQ, 'F', representation='polynomial')  # needs sage.combinat sage.modules
>>> L.is_ideal(F)  # needs sage.combinat sage.modules
Traceback (most recent call last):
... NotImplementedError: A must be a finite dimensional Lie algebra with basis
```
is_semisimple()  
Return if self if a semisimple Lie algebra.  
A Lie algebra is semisimple if the solvable radical is zero. In characteristic 0, this is equivalent to saying the Killing form is non-degenerate.  

EXAMPLES:

Positive characteristic examples:

sage: sp4 = LieAlgebra(GF(Integer(3)), cartan_type=[C,Integer(2)])  
sage: sp4.killing_form_matrix().det()  
0  
sage: sp4.solvable_radical_basis() # long time  
()  
sage: sp4.is_semisimple() # long time  
True
sp4.is_semisimple()  # long time
True

is_solvable()

Return if self is a solvable Lie algebra.

A Lie algebra is solvable if the derived series eventually becomes 0.

EXAMPLES:
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_solvable()
True

>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()

>>> L.is_solvable()
True

L = LieAlgebras(QQ, {(x,y): {x:Integer(1)}}, names=('x', 'y',));
(continues on next page)

4.52. Finite Dimensional Lie Algebras With Basis 685
killing_form_matrix()

Return the matrix of the Killing form of self.

The rows and the columns of this matrix are indexed by the elements of the basis of self (in the order provided by basis()).

EXAMPLES:

```python
sage: # needs sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
```
```python
sage: L.killing_form_matrix()
```
```python
[0 0 0]
[0 0 0]
[0 0 0]
```

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example(0)
```
```python
sage: m = L.killing_form_matrix(); m
```
```python
[]
```
```python
sage: parent(m)
```
```python
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
```

killing_matrix(x, y)

Return the Killing matrix of x and y, where x and y are two elements of self.

The Killing matrix is defined as the matrix corresponding to the action of \( \text{ad}_x \circ \text{ad}_y \) in the basis of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #...
```
```python
sage: a, b, c = L.lie_algebra_generators()
```
```python
sage: L.killing_matrix(a, b)  #...
```
```python
[0 0 0]
[0 0 0]
[0 0 0]
```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
>>> a, b, c = L.lie_algebra_generators()  # needs sage.modules
>>> L.killing_matrix(a, b)  # needs sage.modules
[0 0 0]
[0 0 0]
[0 0 0]

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  # needs sage.combinat sage.modules
sage: L.killing_matrix(y, x)  # needs sage.combinat sage.modules
[ 0 -1]
[ 0  0]

lower_central_series (submodule=False)
Return the lower central series \((g_i)_i\) of \(self\) where the rightmost \(g_k = g_{k+1} = \cdots\).

INPUT:
- submodule – (default: False) if True, then the result is given as submodules of \(self\).

We define the lower central series of a Lie algebra \(g\) recursively by \(g_0 := g\) and

\[ g_{k+1} = [g, g_k] \]

and recall that \(g_k \supseteq g_{k+1}\). Alternatively we can express this as

\[ g \supseteq [g, g] \supseteq [[g, g], g] \supseteq \cdots. \]

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
(An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field,
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix: [])

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.derived_series()
(An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field,
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix: []

The lower central series as submodules:

```python
sage: # needs sage.combinat sage.modules
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {x:1}})
sage: L.lower_central_series(submodule=True)
(Sparse vector space of dimension 2 over Rational Field, Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0])
```

```python
>>> from sage.all import *
```
Ideal \((a, b)\) of Lie algebra on 5 generators \((a, b, c, d, e)\) over \(\mathbb{R}\) (Rational Field)

module \((R=None)\)

Return a dense free module associated to self over \(R\).

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L._dense_free_module()  # needs sage.modules
Vector space of dimension 3 over Rational Field
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
>>> L._dense_free_module()  # needs sage.modules
Vector space of dimension 3 over Rational Field
```

morphism \((on\_generators, codomain=None, base\_map=None, check=True)\)

Return a Lie algebra morphism defined by images of a Lie generating subset of self.

INPUT:
- `on_generators` – dictionary \(\{X: Y\}\) of the images \(Y\) in codomain of elements \(X\) of domain
- `codomain` – a Lie algebra (optional); this is inferred from the values of `on_generators` if not given
- `base_map` – a homomorphism from the base ring to something coercing into the codomain
- `check` – (default: True) boolean; if False the values on the Lie brackets implied by `on_generators` will not be checked for contradictory values

Note: The keys of `on_generators` need to generate domain as a Lie algebra.

See also:

`sage.algebras.lie_algebras.morphism.LieAlgebraMorphism_from_generators`

EXAMPLES:

A quotient type Lie algebra morphism

```python
sage: # needs sage.combinat sage.modules
sage: L.<X,Y,Z,W> = LieAlgebra(QQ, {\('X', 'Y'\): \{'Z': 1\}, ...
\('X', 'Z'\): \{'W': 1\}})
sage: K.<A,B> = LieAlgebra(QQ, abelian=True)
sage: L.morphism({X: A, Y: B})
Lie algebra morphism:
From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
To:   Abelian Lie algebra on 2 generators (A, B) over Rational Field
Defn: X |--> A
      Y |--> B
      Z |--> 0
      W |--> 0
```

4.52. Finite Dimensional Lie Algebras With Basis
The reverse map $A \mapsto X$, $B \mapsto Y$ does not define a Lie algebra morphism, since $[A, B] = 0$, but $[X, Y] \neq 0$:

```
sage: # needs sage.combinat sage.modules
sage: K.morphism({A:X, B: Y})
Traceback (most recent call last):
  ...
ValueError: this does not define a Lie algebra morphism; contradictory values for brackets of length 2
```

However, it is still possible to create a morphism that acts nontrivially on the coefficients, even though it's not a Lie algebra morphism (since it's not linear):

```
sage: # needs sage.combinat sage.modules sage.rings.number_fields
sage: R.<x> = ZZ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: cc = K.hom([-i])
sage: L.<X,Y,Z,W> = LieAlgebra(K, {{'X','Y'}: {Z: 1}, ...
  ...
                   {{'X','Z'}: {W: 1}}})
sage: M.<A,B> = LieAlgebra(K, abelian=True)
sage: phi = L.morphism({X: A, Y: B}, base_map=cc)
sage: phi(X)
A
sage: phi(i*X)
-i*A
```

(continues on next page)
L = LieAlgebra(K, {('X', 'Y') : {'Z': Integer(1)},
    ...
    ('X', 'Z') : {'W': Integer(1)}}, names=('X', 'Y', 'Z', 'W')); (X, Y, Z, W) = L._first_ngens(4)
M = LieAlgebra(K, abelian=True, names=('A', 'B')); (A, B) = M._first_ngens(2)
phi = L.morphism({X: A, Y: B}, base_map=cc)
phi(X)
A
phi(i*X)
-i*A

nilradical()

Return the nilradical of self.

The nilradical of a Lie algebra \( L \) is the largest nilpotent ideal of \( L \).

See also:

nilradical_basis()

EXAMPLES:

sage: scoeffs = {('a','d') : {'a':1}, ('a','e') : {'b':-1},
    ...
    ('b','d') : {'b':1}, ('b','e') : {'a':1},
    ...
    ('d','e') : {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.nilradical()
Ideal (a, b, c, d, e) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field

from sage.all import *

sage: scoeffs = {('a','d') : {'a':Integer(1)}, ('a','e') : {'b':-Integer(1)},
    ...
    ('b','d') : {'b':Integer(1)}, ('b','e') : {'a':Integer(1)},
    ...
    ('d','e') : {'c':Integer(1)}}

sage: L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e) = L._first_ngens(5)

sage: L.nilradical()
Ideal (a, b, c, d, e) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field

nilradical_basis()

Return a basis of the nilradical of self.

See also:

nilradical()

EXAMPLES:

sage: scoeffs = {('a','d') : {'a':1}, ('a','e') : {'b':-1},
    ...
    ('b','d') : {'b':1}, ('b','e') : {'a':1},
    ...
    ('d','e') : {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.nilradical_basis()
(a, b, c)
sage: L.is_nilpotent()
False

(continues on next page)
sage: sl3 = LieAlgebra(QQ, cartan_type=[‘A’, 2])
sage: sl3.nilradical_basis()
()
sage: scoeffs = {('a', 'e'): {'a': Integer(1)}, ('b', 'e'): {'b': -Integer(1)},
...: ('c', 'd'): {'a': Integer(1)}, ('c', 'e'): {'c': Integer(1)}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.nilradical_basis()
(a, b, c, d)
sage: L.is_solvable()
True
sage: L.is_nilpotent()
False

sage: K1 = L.quotient([a])
sage: K1.nilradical_basis()
(b, c, d)

sage: SL = L.subalgebra([a, b, c, d]); SL
Subalgebra generated by (a, b, c, d) of
Lie algebra on 5 generators (a, b, c, d, e) over Rational Field
sage: SL.nilradical_basis()
(a, b, c, d)
sage: scoeffs = {('x', 'z'): {'x': 1, 'y': 1}, ('y', 'z'): {'y': 1}}
sage: L.<x,y,z> = LieAlgebra(GF(3), scoeffs)
sage: L.nilradical_basis()
(x, y)

>>> from sage.all import *
>>> scoeffs = {('a', 'd'): {'a': Integer(1)}, ('a', 'e'): {'b': -Integer(1)},
...: ('b', 'd'): {'b': Integer(1)}, ('b', 'e'): {'a': Integer(1)},
...: ('d', 'e'): {'c': Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e) = L._first_ngens(5)
>>> L.nilradical_basis()
(a, b, c)
>>> L.is_nilpotent()
False

>>> sl3 = LieAlgebra(QQ, cartan_type=[‘A’, Integer(2)])
>>> sl3.nilradical_basis()
()

>>> scoeffs = {('a', 'e'): {'a': Integer(1)}, ('b', 'e'): {'a': Integer(1), 'b
...': -Integer(1)},
...: ('c', 'd'): {'a': Integer(1)}, ('c', 'e'): {'c': Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e) = L._first_ngens(5)
>>> L.nilradical_basis()
(a, b, c, d)
>>> L.is_solvable()
True
>>> L.is_nilpotent()
False

(continues on next page)
We check against the generic algorithm:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {(x,z): {x:1,y:1}, (y,z): {y:1}})
sage: L.nilradical_basis()
(x, y)
sage: dim = L.dimension()
sage: MS = MatrixSpace(L.base_ring(), dim)
sage: gens = [b.adjoint_matrix() for b in L.basis()]
sage: A = MS.subalgebra(gens)
sage: RB = A.radical_basis()
sage: mat = matrix(L.base_ring(),
    [g._vector_() for g in gens],
    + [A.lift(r)._vector_() for r in RB])
sage: tuple([L.from_vector(w) for v in mat.right_kernel().basis() if (w := v[:dim])])
(x, y)
```

A positive characteristic example:

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {(x,z): {x:Integer(1),y:Integer(1)}, (y,z): {y:Integer(1)}})
>>> L.nilradical_basis()
(x, y)
>>> dim = L.dimension()
>>> MS = MatrixSpace(L.base_ring(), dim)
>>> gens = [b.adjoint_matrix() for b in L.basis()]
>>> A = MS.subalgebra(gens)
>>> RB = A.radical_basis()
>>> mat = matrix(L.base_ring(),
    [g._vector_() for g in gens],
    + [A.lift(r)._vector_() for r in RB])
>>> tuple([L.from_vector(w) for v in mat.right_kernel().basis() if (w := v[:dim])])
(x, y)
```
Category Framework, Release 10.4

```python
sage: scoeffs = {('x','z'): {x:1, 'y':1}, ('y','z'): {y:1}}
sage: L.<x,y,z> = LieAlgebra(GF(3), scoeffs)
sage: L.nilradical_basis()
(x, y)
```

```python
>>> from sage.all import *

```sage```

```python
scoeffs = {(x,z): {x:Integer(1),y:Integer(1)}, (y,z): {y:
˓→Integer(1)}}

```sage```

```python
L = LieAlgebra(GF(Integer(3)), scoeffs, names=('x', 'y', 'z',)); (x,
˓→y, z,) = L._first_ngens(3)

```sage```

```python
L.nilradical_basis()
```

```
normalizer(S)
Return the normalizer of S in self.

INPUT:
• S — a subalgebra of self or a list of elements that represent generators for a subalgebra

See also:
normalizer_basis()

EXAMPLES:
```
```
sage: scoeffs = {(a,d): {a:1}, (a,e): {b:-1},
˓→(b,d): {b:1}, (b,e): {a:1},
˓→(d,e): {c:1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.normalizer([a, e])
Subalgebra generated by (b, c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field

sage: L.normalizer([a, c, e])
Subalgebra generated by (b, c, d) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field
```

```
>>> from sage.all import *

```sage```

```python
scoeffs = {(a,d): {a:Integer(1)}, (a,e): {b:-Integer(1)},
...
(b,d): {b:Integer(1)}, (b,e): {a:Integer(1)},
...
(d,e): {c:Integer(1)}}

```sage```

```python
L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e',)); (a, b,
˓→c, d, e,) = L._first_ngens(5)

```sage```

```python
L.normalizer([a, e])
Subalgebra generated by (b, c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field

>>> L.normalizer([a, c, e])
Subalgebra generated by (b, c, d) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field
```

```
normalizer_basis(S)
Return a basis of the normalizer of S in self.

INPUT:
• S — a subalgebra of self or a list of elements that represent generators for a subalgebra

See also:
normalizer()

EXAMPLES:
```
```python
sage: scoffs = {('a','d'): {'a':1}, ('a','e'): {'b':1},
        ('b','d'): {'b':1}, ('b','e'): {'a':1},
        ('d','e'): {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoffs)
sage: L.normalizer_basis([a, e])
[b, c]
sage: S = L.subalgebra([a, e])
sage: L.normalizer_basis(S)
[a, b, c, e]
```

When the subalgebra is the ambient Lie algebra, we return the basis of the ambient Lie algebra:

```python
sage: L.normalizer_basis(L)
Finite family {'a': a, 'b': b, 'c': c, 'd': d, 'e': e}
sage: L.normalizer_basis([a, b, c, a, d + e, a + e])
Finite family {'a': a, 'b': b, 'c': c, 'd': d, 'e': e}
```

**product_space** *(L, submodule=False)*

Return the product space `[self, L]`.

**INPUT:**
- `L` - a Lie subalgebra of `self`
- `submodule` - (default: `False`) if `True`, then the result is forced to be a submodule of `self`

**EXAMPLES:**

```python
sage: # needs sage.combinat.sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a,b,c = L.lie_algebra_generators()
sage: X = L.subalgebra([a, b + c])
sage: L.product_space(X)
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix: []
sage: Y = L.subalgebra([a, 2*b - c])
sage: X.product_space(Y)
An example of a finite dimensional Lie algebra with basis:
```

(continues on next page)
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix: []

```python
>>> from sage.all import *
```  
```python
# needs sage.combinat sage.modules
```

```python
L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()

>>> a,b,c = L.lie_algebra_generators()

>>> X = L.subalgebra([a, b + c])

>>> L.product_space(X)
```

An example of a finite dimensional Lie algebra with basis: the 0-dimensional abelian Lie algebra over Rational Field with basis matrix: []

```python
>>> Y = L.subalgebra([a, Integer(2)*b - c])

>>> X.product_space(Y)
```

An example of a finite dimensional Lie algebra with basis: the 0-dimensional abelian Lie algebra over Rational Field with basis matrix: []

```python
sage: H = lie_algebras.Heisenberg(ZZ, 4)

sage: Hp = H.product_space(H, submodule=True).basis()

sage: [H.from_vector(v) for v in Hp]
[z]
```

```python
sage: # needs sage.combinat sage.modules
```

```python
L.<x,y> = LieAlgebra(QQ, {('x','y'): {x:1}})

sage: Lp = L.product_space(L) # not implemented

sage: Lp
```

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: (x,)

```python
sage: Lp.product_space(L) # not implemented
```

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: (x,)

```python
sage: Lp.product_space(Lp) # not implemented
```

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: ()

```python
>>> from sage.all import *
```  
```python
# needs sage.combinat sage.modules
```

```python
L = LieAlgebra(QQ, {('x','y'): {x:Integer(1)}}, names=('x', 'y'),)

- (x, y) = L._first_ngens(2)
```
The quotient \( \mathfrak{g} / \mathfrak{h} \) is a Lie algebra.

INPUT:

- \( \mathfrak{g} \) – an ideal or a list of generators of the ideal
- \( \text{names} \) – (optional) a string or a list of strings; names for the basis elements of the quotient. If \( \text{names} \) is a string, the basis will be named \( \text{names}_1, \ldots, \text{names}_n \).

EXAMPLES:

The Engel Lie algebra as a quotient of the free nilpotent Lie algebra of step 3 with 2 generators:

```python
sage: L.<X,Y,Z,W,U> = LieAlgebra(QQ, 2, step=3)
sage: E = L.quotient(U); E
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X, Y, Z, W, U)
over Rational Field
I: Ideal (U)
sage: E.basis().list()
[X, Y, Z, W]
sage: E(X).bracket(E(Y))
Z
sage: E(Y).bracket(Z)
-U
sage: E(U)
0
```

```python
>>> from sage.all import *
```
Quotients when the base ring is not a field are not implemented:

```
sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.Heisenberg(ZZ, 1)
sage: L.quotient(L.an_element())
Traceback (most recent call last):
...  
NotImplementedError: quotients over non-fields not implemented
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = lie_algebras.Heisenberg(ZZ, Integer(1))
>>> L.quotient(L.an_element())
Traceback (most recent call last):
...  
NotImplementedError: quotients over non-fields not implemented
```

`solvable_radical()`

Return the solvable radical of `self`.

The **solvable radical** of a Lie algebra $L$ is the largest solvable ideal of $L$.

**See also:**

`solvable_radical_basis()`

**EXAMPLES:**

```
sage: scoeffs = {(a,d): {a:1}, (a,e): {b:-1}, ....: (b,d): {b:1}, (b,e): {a:1}, ....: (d,e): {c:1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.solvable_radical()
Ideal (a, b, c, d, e) of Lie algebra on 5 generators (a, b, c, d, e) over...
˓→ Rational Field
```

```
>>> from sage.all import *
>>> scoeffs = {(a',d'): {'a':Integer(1)}, ('a', 'e'): {'b':-Integer(1)}, ...
...: (b',d'): {b:1}, ('b', 'e'): {a:1}, ...
...: (d',e'): {'c':Integer(1)})
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b,...
˓→c, d, e) = L._first_ngens(5)
>>> L.solvable_radical()
Ideal (a, b, c, d, e) of Lie algebra on 5 generators (a, b, c, d, e) over...
˓→ Rational Field
```
\texttt{solvable_radical_basis()}

Return a basis of the solvable radical of \texttt{self}.

See also:

\texttt{solvable_radical()}

EXAMPLES:

\begin{verbatim}
sage: scoeffs = {('a', 'd'): {'a':1}, ('a', 'e'): {'b':-1},
    ...: ('b', 'd'): {'b':1}, ('b', 'e'): {'a':1},
    ...: ('d', 'e'): {'c':1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.solvable_radical_basis()
(a, b, c, d, e)
sage: L.is_solvable()
True

sage: sl3 = LieAlgebra(QQ, cartan_type=[A,2])
sage: sl3.solvable_radical_basis()
()

sage: L.<x,y,z> = LieAlgebra(QQ, scoeffs)

sage: S = L.subalgebra([x, y])
sage: S.solvable_radical_basis()
(x, y)
sage: S.is_solvable()
True

>>> from sage.all import *

>>> scoeffs = {('a', 'd'): {'a':Integer(1)}, ('a', 'e'): {'b':-Integer(1)},
    ...: ('b', 'd'): {'b':Integer(1)}, ('b', 'e'): {'a':Integer(1)},
    ...: ('d', 'e'): {'c':Integer(1)}}

>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e'))

>>> L.solvable_radical_basis()
(a, b, c, d, e)

>>> L.is_solvable()
True

>>> sl3 = LieAlgebra(QQ, cartan_type=[A,Integer(2)])

>>> sl3.solvable_radical_basis()
()

>>> L = LieAlgebra(QQ, {('x', 'z'): {'x':Integer(1), 'y':Integer(1)},
    ...: ('y', 'z'): {'y':Integer(1))}, names=('x', 'y', 'z'));

>>> S = L.subalgebra([x, y])

>>> S.is_solvable()
True

Positive characteristic examples:

\begin{verbatim}
sage: scoeffs = {('x', 'z'): {'x':1, 'y':1}, ('y', 'z'): {'y':1}}
sage: L.<x,y,z> = LieAlgebra(GF(3), scoeffs)

\end{verbatim}

(continues on next page)
structure_coefficients \( \text{(include_zeros=False)} \)

Return the structure coefficients of self.

INPUT:

• include_zeros – (default: False) if True, then include the \([x,y] = 0\) pairs in the output

OUTPUT:

A dictionary whose keys are pairs of basis indices \((i,j)\) with \(i < j\), and whose values are the corresponding elements \([b_i, b_j]\) in the Lie algebra.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L.structure_coefficients()  # needs sage.modules
Finite family {}
```

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L.structure_coefficients(True)  # needs sage.modules
Finite family {(0, 1): (0, 0, 0), (0, 2): (0, 0, 0), (1, 2): (0, 0, 0)}
```
[((1,2,3), (1,2)): (2,3) - (1,3),
((1,2,3), (1,3)): -(2,3) + (1,2),
((1,3,2), (2,3)): (1,2) - (1,3),
((1,3,2), (1,2)): -(2,3) + (1,3),
((1,3,2), (1,3)): (2,3) - (1,2),
((1,3), (1,2)): -(1,2,3) + (1,3,2)}

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups sage.modules
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S)
>>> L.structure_coefficients()
Finite family {((2,3), (1,2)): (1,2,3) - (1,3,2),
((2,3), (1,3)): -(1,2,3) + (1,3,2),
((1,2,3), (2,3)): -(1,2) + (1,3),
((1,2,3), (1,2)): (2,3) - (1,3),
((1,2,3), (1,3)): -(2,3) + (1,2),
((1,3,2), (2,3)): (1,2) - (1,3),
((1,3,2), (1,2)): -(2,3) + (1,3),
((1,3,2), (1,3)): (2,3) - (1,2),
((1,3), (1,2)): -(1,2,3) + (1,3,2)}
```

**subalgebra**(*gens, **kwds*)

Return the subalgebra of self generated by gens.

**INPUT:**

- **gens** – a list of generators of the subalgebra
- **category** – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: p1, p2, q1, q2, z = H.basis()
sage: S = H.subalgebra([p1, q1])
sage: S.basis().list()
[p1, q1, z]
sage: S.basis_matrix()
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 0 1]
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> H = lie_algebras.Heisenberg(QQ, Integer(2))
>>> p1, p2, q1, q2, z = H.basis()
>>> S = H.subalgebra([p1, q1])
>>> S.basis().list()
[p1, q1, z]
>>> S.basis_matrix()
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 0 1]
```

Passing an extra category to a subalgebra:
universal_commutative_algebra()

Return the universal commutative algebra associated to self.

Let $I$ be the index set of the basis of self. Let $P = \{P_{a,i,j}\}_{a,i,j \in I}$ denote the universal polynomials of a Lie algebra $L$. The universal commutative algebra associated to $L$ is the quotient ring $R[ X_{ij} ]_{i,j \in I} / (P)$.

EXAMPLES:

universal_polynomials()

Return the family of universal polynomials of self.

The universal polynomials of a Lie algebra $L$ with basis $\{ e_i \}_{i \in I}$ and structure coefficients $[ e_i, e_j ] = \tau_{ij}^a e_a$ is given by

$$ P_{aij} = \sum_{u \in I} \tau_{ij}^u X_{au} - \sum_{s,t \in I} \tau_{st}^a X_{si} X_{tj}, $$

where $a, i, j \in I$.

REFERENCES:
• [AM2020]

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: L.<x,y> = LieAlgebra(QQ, {'x':1}, 'y':1})
```
Category Framework, Release 10.4

>>> UP = L.universal_polynomials()
>>> len(UP)
450
>>> UP[a[Integer(2)], a[Integer(1)], -a[Integer(1)]]
X0_7*X4_1 - X0_1*X4_7 - 2*X0_7*X5_1 + 2*X0_1*X5_7 + X2_7*X7_1
- X2_1*X7_7 - X3_7*X8_1 + X3_1*X8_7 + X0_4

upper_central_series()

Return the upper central series \( (Z_i(g)) \) of self where the rightmost \( Z_k(g) = Z_{k+1}(g) = \cdots \).

The upper central series of a Lie algebra \( g \) is defined recursively by \( Z_0(g) := Z(g) \) and

\[
Z_k(g)/Z_{k+1}(g) = Z(g/Z_k(g)),
\]

and recall that \( Z(g) \) is the center() of \( g \).

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.upper_central_series()
[An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field]

sage: L.<x,y> = LieAlgebra(QQ, {(x,y): {x:1}})
sage: L.upper_central_series()
[Ideal () of Lie algebra on 2 generators (x, y) over Rational Field]

sage: scoeffs = {(a,d):{a:1}, (a,e): {b:-1},
...: (b,d):{b:1}, (b,e): {a:1},
...: (d,e):{c:1}}
sage: L.<a,b,c,d,e> = LieAlgebra(QQ, scoeffs)
sage: L.upper_central_series()
[Ideal (c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field]

sage: L = lie_algebras.Heisenberg(QQ, 3)
sage: L.upper_central_series()
[Ideal (z) of Heisenberg algebra of rank 3 over Rational Field,
Heisenberg algebra of rank 3 over Rational Field]

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.upper_central_series()
[An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field]

>>> L = LieAlgebra(QQ, {(x,y): {x:Integer(1)}}, names=(x, y,)); (x, y) = L._first_ngens(2)
>>> L.upper_central_series()
[Ideal () of Lie algebra on 2 generators (x, y) over Rational Field]

>>> scoeffs = {(a,d):{a:Integer(1)}, (a,e): {b:-Integer(1)},
...: (b,d):{b:Integer(1)}, (b,e): {a:Integer(1)},
...: (d,e):{c:Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e) = L._first_ngens(5)
>>> L.upper_central_series()
*** (Ideal (c) of Lie algebra on 5 generators (a, b, c, d, e) over Rational Field)

```python
>>> L = lie_algebras.Heisenberg(QQ, Integer(3))
>>> L.upper_central_series()
[Ideal (z) of Heisenberg algebra of rank 3 over Rational Field, Heisenberg algebra of rank 3 over Rational Field]
```

class Subobjects (category, *args)

Bases: SubobjectsCategory

A category for subalgebras of a finite dimensional Lie algebra with basis.

class ParentMethods

Bases: object

ambient()

Return the ambient Lie algebra of self.

EXAMPLES:

```python
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: L = C.example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a + b, b + c])
sage: S.ambient() == L
True
```

basis_matrix()

Return the basis matrix of self.

EXAMPLES:

```python
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: L = C.example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a + b, b + c])
sage: S.basis_matrix()
[ 1 0 -1/2]
[ 0 1 1]
```

(continues on next page)
reduce(\(X\))

Reduce an element of the ambient Lie algebra modulo the ideal \(\text{self}\).

**INPUT:**
- \(X\) – an element of the ambient Lie algebra

**OUTPUT:**
An element \(Y\) of the ambient Lie algebra that is contained in a fixed complementary submodule \(V\) to \(\text{self}\) such that \(X = Y \mod \text{self}\).

When the base ring of \(\text{self}\) is a field, the complementary submodule \(V\) is spanned by the elements of the basis that are not the leading supports of the basis of \(\text{self}\).

**EXAMPLES:**

An example reduction in a 6 dimensional Lie algebra:

```python
>>> from sage.all import *
>>> sc = {(a,b): {d: Integer(1)}, (a,c): {e: Integer(1)}, ...
>>> L = LieAlgebra(QQ, sc, names=(a, b, c, d, e, f)); (a, b, c, d, e, f) = L._first_ngens(6)
>>> I = L.ideal(c)
>>> I.reduce(a + b + c + d + e + f)
a + b + d
```

The reduction of an element is zero if and only if the element belongs to the subalgebra:

```python
sage: I.reduce(c + e)
0
sage: c + e in I
True
```

Over non-fields, the complementary submodule may not be spanned by a subset of the basis of the ambient Lie algebra:

```python
sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X': '3'})
sage: I = L.ideal(Y)
```
```python
sage: I.basis()
Family (Y, 3*Z)
sage: I.reduce(3*Z)
0
sage: I.reduce(Y + 14*Z)
2*Z
```

```python
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {('X','Y'): {'Z': Integer(3)}}, names=('X', 'Y', 'Z',))
>>> (X, Y, Z,) = L._first_ngens(3)
>>> I = L.ideal(Y)
>>> I.basis()
Family (Y, 3*Z)
>>> I.reduce(Integer(3)*Z)
0
>>> I.reduce(Y + Integer(14)*Z)
2*Z
```

### example \((n=3)\)

Return an example of a finite dimensional Lie algebra with basis as per `Category.example`.

**EXAMPLES:**

```python
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: C.example() # needs sage.modules
An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field
```

```python
>>> from sage.all import *
>>> C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
>>> C.example() # needs sage.modules
An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field
```

Other dimensions can be specified as an optional argument:

```python
sage: C.example(5) # needs sage.modules
An example of a finite dimensional Lie algebra with basis:
the 5-dimensional abelian Lie algebra over Rational Field
```

```python
>>> from sage.all import *
>>> C.example(Integer(5)) # needs sage.modules
An example of a finite dimensional Lie algebra with basis:
the 5-dimensional abelian Lie algebra over Rational Field
```
4.53 Finite dimensional modules with basis

class sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis

Bases: CategoryWithAxiom_over_base_ring

The category of finite dimensional modules with a distinguished basis

EXAMPLES:

```python
sage: C = FiniteDimensionalModulesWithBasis(ZZ); C
Category of finite dimensional modules with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of finite dimensional modules over Integer Ring,
 Category of modules with basis over Integer Ring]
sage: C is Modules(ZZ).WithBasis().FiniteDimensional()
True
```

class ElementMethods

Bases: object

dense_coefficient_list(order=None)

Return a list of all coefficients of self.

By default, this list is ordered in the same way as the indexing set of the basis of the parent of self.

INPUT:

• order – (optional) an ordering of the basis indexing set

EXAMPLES:

```python
sage: # needs sage.modules
sage: v = vector([0, -1, -3])
sage: v.dense_coefficient_list()
[0, -1, -3]
sage: v.dense_coefficient_list([2,1,0])
[-3, -1, 0]
sage: sorted(v.coefficients())
[-3, -1]
```

```python
>>> from sage.all import *
>>> # needs sage.modules

>>> v = vector([Integer(0), -Integer(1), -Integer(3)])

>>> v.dense_coefficient_list()
[0, -1, -3]
```

(continues on next page)
```python
>>> sorted(v.coefficients())
[-3, -1]
```

class **Homsets** *(category, *args)*

Bases: **HomsetsCategory**

class **Endset** *(base_category)*

Bases: **CategoryWithAxiom**

class **ElementMethods**

Bases: object

```
def characteristic_polynomial()
    return the characteristic polynomial of this endomorphism.

def characteristic_polynomial() and charpoly() are the same method.

INPUT:
* var - variable

EXAMPLES:
```
Return the characteristic polynomial of this endomorphism. 

\texttt{characteristic\_polynomial()} and \texttt{charpoly()} are the same method.

INPUT:
\begin{itemize}
  \item \texttt{var} – variable
\end{itemize}

EXAMPLES:

\begin{verbatim}
 sage: # needs sage.modules
 sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.characteristic_polynomial()
x^2 - 3*x + 2
 sage: phi.charpoly()
x^2 - 3*x + 2
 sage: phi.matrix().charpoly()
x^2 - 3*x + 2
 sage: phi.charpoly('T')
T^2 - 3*T + 2
 sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.charpoly()
x^2 - 3*x + 2
\end{verbatim}

Return the determinant of this endomorphism. 

\texttt{determinant()} and \texttt{det()} are the same method.

EXAMPLES:

\begin{verbatim}
 >>> from sage.all import *
 >>> # needs sage.modules
 >>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1),
 \rightarrow Integer(2)*V.gen(1)])
 >>> phi.characteristic_polynomial()
x^2 - 3*x + 2
 >>> phi.charpoly()
x^2 - 3*x + 2
 >>> phi.matrix().charpoly()
x^2 - 3*x + 2
 >>> phi.charpoly('T')
T^2 - 3*T + 2
 >>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
 >>> M = matrix(ZZ, [[Integer(1), Integer(0)],
 \rightarrow Integer(2)])
 >>> psi = W.module_morphism(matrix=M, codomain=W)
 >>> psi.charpoly()
x^2 - 3*x + 2
\end{verbatim}
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.det()
2

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1),
                         Integer(2)*V.gen(1)])
>>> phi.determinant()
2
>>> phi.det()
2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1),
                         Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.det()
2
```

*determinant()*  
Return the determinant of this endomorphism.  

*determinant()* and *det()* are the same method.  

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.determinant()
2
sage: phi.det()
2
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
```

```python
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.det()
2
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1),
                         Integer(2)*V.gen(1)])
>>> phi.determinant()
2
>>> phi.det()
2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
```

```python
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1),
                         Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.det()
2
```

*fcp()*  
Return the factorization of the characteristic polynomial.  

4.53. Finite dimensional modules with basis  

711
INPUT:

- `var` - variable

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.fcp()
   # needs sage.libs.pari
   (x - 2) * (x - 1)
sage: phi.fcp('T')
   # needs sage.libs.pari
   (T - 2) * (T - 1)
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.fcp()
   # needs sage.libs.pari
   (x - 2) * (x - 1)
```

```
from sage.all import *
# needs sage.modules
V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1),
   Integer(2)*V.gen(1)])
phi.fcp()
   # needs sage.libs.pari
   (x - 2) * (x - 1)
phi.fcp('T')
   # needs sage.libs.pari
   (T - 2) * (T - 1)
W = CombinatorialFreeModule(ZZ, ['x', 'y'])
M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1),
   Integer(2)]])
psi = W.module_morphism(matrix=M, codomain=W)
psi.fcp()
   # needs sage.libs.pari
   (x - 2) * (x - 1)
```

The `minimal_polynomial()` method returns the minimal polynomial of this endomorphism. `minimal_polynomial()` and `minpoly()` are the same method.

INPUT:

- `var` - string (default: 'x'); a variable name

EXAMPLES:

Compute the minimal polynomial, and check it.

```
sage: # needs sage.modules
sage: V = GF(7)^3
sage: H = V.Hom(V)([[0, 1, 2], [-1, 0, 3], [2, 4, 1]]); H
Vector space morphism represented by the matrix:
[0 1 2]
[6 0 3]
[2 4 1]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7
sage: H.minpoly()
```

(continues on next page)
→ # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
sage: H.minimal_polynomial()
→ # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
sage: H^3 + (H^2)*6 + H*6 + 1
Vector space morphism represented by the matrix:
[0 0 0]
[0 0 0]
[0 0 0]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7
sage: factor(phi.minpoly())
(x + 1) * (x + 2)^2
sage: A.minpoly()(A) == 0
True
sage: factor(phi.charpoly())
(x + 1)^2 * (x + 2)^2

>>> from sage.all import *
... # needs sage.modules
>>> V = GF(Integer(7))**Integer(3)
... H = V.Hom(V)([[Integer(0), Integer(1), Integer(2)],
... [-Integer(1), Integer(0), Integer(3)],
... [Integer(2), Integer(4), Integer(1)]]); H
Vector space morphism represented by the matrix:
[0 1 2]
[6 0 3]
[2 4 1]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7
>>> H.minpoly()
→ # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
... # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
... H.minimal_polynomial()
→ # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
... H**Integer(3) + (H**Integer(2))*Integer(6) + H*Integer(6) +...
... Integer(1)
Vector space morphism represented by the matrix:
[0 0 0]
[0 0 0]
[0 0 0]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7
... # needs sage.modules sage.rings.finite_rings
... k = GF(Integer(9), 'c')
... V = CombinatorialFreeModule(k, ['x', 'y', 'z', 'w'])
... A = matrix(k, Integer(4), [Integer(1),Integer(1),Integer(1),Integer(0),
... Integer(0), Integer(1),Integer(1),Integer(0),Integer(0),...

## 4.53. Finite dimensional modules with basis

713
\textbf{minpoly()}

Return the minimal polynomial of this endomorphism.

\texttt{minimal\_polynomial()} and \texttt{minpoly()} are the same method.

\textbf{INPUT:}

- \texttt{var} -- string (default: 'x'); a variable name

\textbf{EXAMPLES:}

Compute the minimal polynomial, and check it.

\begin{verbatim}
 sage: # needs sage.modules
 sage: V = GF(7)^3
 sage: H = V.Hom(V)([[0,1,2], [-1,0,3], [2,4,1]]); H
 Vector space morphism represented by the matrix:
 [0 1 2]
 [6 0 3]
 [2 4 1]
 Domain: Vector space of dimension 3 over Finite Field of size 7
codomain: Vector space of dimension 3 over Finite Field of size 7
 sage: H.minpoly()  # needs sage.libs.pari
 x^3 + 6*x^2 + 6*x + 1
 sage: H.minimal_polynomial()  # needs sage.libs.pari
 x^3 + 6*x^2 + 6*x + 1
 sage: H^3 + (H^2)*6 + H*6 + 1
 Vector space morphism represented by the matrix:
 [0 0 0]
 [0 0 0]
 [0 0 0]
 Domain: Vector space of dimension 3 over Finite Field of size 7
codomain: Vector space of dimension 3 over Finite Field of size 7

 sage: # needs sage.modules sage.rings.finite_rings
 sage: k = GF(9, 'c')
 sage: V = CombinatorialFreeModule(k, ['x', 'y', 'z', 'w'])
 sage: A = matrix(k, 4, [1,1,0,0, 0,1,0,0, 0,0,5,0, 0,0,0,5])
 sage: phi = V.module_morphism(matrix=A, codomain=V)
 sage: factor(phi.minpoly())
 (x + 1) * (x + 2)^2
 sage: A.minpoly()(A) == 0
 True
 sage: factor(phi.charpoly())
 (x + 1)^2 * (x + 2)^2
\end{verbatim}
>>> from sage.all import *
>>> # needs sage.modules
>>> V = GF(Integer(7))**Integer(3)
>>> H = V.Hom(V)([[Integer(0),Integer(1),Integer(2)],
        [-Integer(1),
        Integer(0),Integer(3)],[Integer(2),Integer(4),Integer(1)]]); H
Vector space morphism represented by the matrix:
[0 1 2]
[6 0 3]
[2 4 1]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7
>>> H.minpoly()  # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
>>> H.minimal_polynomial()  # needs sage.libs.pari
x^3 + 6*x^2 + 6*x + 1
>>> H**Integer(3) + (H**Integer(2))*Integer(6) + H*Integer(6) + Integer(1)
Vector space morphism represented by the matrix:
[0 0 0]
[0 0 0]
[0 0 0]
Domain: Vector space of dimension 3 over Finite Field of size 7
Codomain: Vector space of dimension 3 over Finite Field of size 7

trace()

Return the trace of this endomorphism.

EXAMPLES:

sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.trace()
3
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])

4.53. Finite dimensional modules with basis 715
```python
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.gen(1)])
>>> phi.trace()
3
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.trace()
3

class MorphismMethods
    Bases: object

    image()

    Return the image of self as a submodule of the codomain.

    EXAMPLES:

    sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # needs sage.groups sage.modules
    sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # needs sage.groups sage.modules
    sage: f.image()  # needs sage.groups sage.modules
    Free module generated by {0, 1, 2} over Rational Field

    >>> from sage.all import *
    >>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))  # needs sage.groups sage.modules
    >>> f = SGA.module_morphism(lambda x: SGA(x**Integer(2)), codomain=SGA)  # needs sage.groups sage.modules
    >>> f.image()  # needs sage.groups sage.modules
    Free module generated by {0, 1, 2} over Rational Field

    image_basis()

    Return a basis for the image of self in echelon form.

    EXAMPLES:

    sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # needs sage.groups sage.modules
    sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # needs sage.groups sage.modules
    sage: f.image_basis()  # needs sage.groups sage.modules
    (([1, 2, 3], [2, 3, 1], [3, 1, 2])

    >>> from sage.all import *
    >>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))  # needs sage.groups sage.modules
    >>> f = SGA.module_morphism(lambda x: SGA(x**Integer(2)), codomain=SGA)  # needs sage.groups sage.modules
    >>> f.image_basis()  # needs sage.groups sage.modules
```

(continues on next page)
kernel()

Return the kernel of \texttt{self} as a submodule of the domain.

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)
sage: K = f.kernel()
sage: K
Free module generated by \{0, 1, 2\} over \texttt{Rational Field}
```

```python
sage: K.ambient()
Symmetric group algebra of order 3 over \texttt{Rational Field}
```

```python
>>> from sage.all import *
```

```python
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(3))
```

```python
>>> f = SGA.module_morphism(lambda x: SGA(x**Integer(2)), codomain=SGA)
```

```python
>>> f.kernel_basis()
```

```
([[1, 2, 3] - [3, 2, 1], [1, 3, 2] - [3, 2, 1], [2, 1, 3] - [3, 2, 1])
```

matrix\((base\_ring=\text{None}, \text{side}=\text{’left’})\)

Return the matrix of this morphism in the distinguished bases of the domain and codomain.

INPUT:

- base\_ring -- a ring (default: \text{None}, meaning the base ring of the codomain)
- side -- “left” or “right” (default: “left”)
If `side` is “left”, this morphism is considered as acting on the left; i.e. each column of the matrix represents the image of an element of the basis of the domain.

The order of the rows and columns matches with the order in which the bases are enumerated.

See also:
Modules.WithBasis.ParentMethods.module_morphism()

EXAMPLES:

```sage
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(ZZ, [1,2]); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); y = Y.basis()
sage: phi = X.module_morphism(on_basis={1: y[3] + 3*y[4],
...: 2: 2*y[3] + 5*y[4]}.__getitem__,
...: codomain=Y)
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2)]); x = X.basis()
>>> Y = CombinatorialFreeModule(ZZ, [Integer(3),Integer(4)]); y = Y.basis()
>>> phi = X.module_morphism(on_basis={Integer(1): y[Integer(3)] +
...: Integer(3)*y[Integer(4)],
...: Integer(2):...
...: Integer(2)*y[Integer(3)] + Integer(5)*y[Integer(4)]}.__getitem__,
...: codomain=Y)
```

The resulting matrix is immutable:
Todo: Add support for morphisms where the codomain has a different base ring than the domain:

```python
sage: Y = CombinatorialFreeModule(QQ, [3,4]); y = Y.basis()  # needs sage.modules
sage: phi = X.module_morphism(on_basis={1: y[3] + 3*y[4],  # needs sage.modules
                                      2: 2*y[3] + 5/2*y[4]}.__getitem__,
                                      codomain=Y)
```

This currently does not work because, in this case, the morphism is just in the category of commutative additive groups (i.e. the intersection of the categories of modules over \( Z \) and over \( Q \)):

```python
sage: phi.parent().homset_category()  # not implemented, needs sage.modules
Category of commutative additive semigroups
```
class ParentMethods

    Bases: object

    annihilator (S, action=<built-in function mul>, side='right', category=None)

    Return the annihilator of a finite set.

    INPUT:
    • S — a finite set
    • action — a function (default: operator.mul)
    • side — 'left' or 'right' (default: 'right')
    • category — a category

    Assumptions:
    • action takes elements of self as first argument and elements of S as second argument;
    • The codomain is any vector space, and action is linear on its first argument; typically it is bilinear;
    • If side is 'left', this is reversed.

    OUTPUT:

    The subspace of the elements x of self such that action(x,s) = 0 for all s ∈ S. If side is 'left'
    replace the above equation by action(s,x) = 0.

    If self is a ring, action an action of self on a module M and S is a subset of M, we recover the
    Wikipedia article Annihilator_%28ring_theory%29. Similarly this can be used to compute torsion or
    orthogonals.

    See also:

    annihilator_basis() for lots of examples.

    EXAMPLES:

    sage: F = FiniteDimensionalAlgebrasWithBasis(QQ).example(); F
    An example of a finite dimensional algebra with basis:
    the path algebra of the Kronecker quiver
    (containing the arrows a:x->y and b:x->y) over Rational Field
    sage: x, y, a, b = F.basis()
    sage: A = F.annihilator([a + 3*b + 2*y]); A
    Free module generated by {0} over Rational Field
    sage: [b.lift() for b in A.basis()]
    [-1/2*a - 3/2*b + x]

    sage: # needs sage.modules
    sage: F = FiniteDimensionalAlgebrasWithBasis(QQ).example(); F
    An example of a finite dimensional algebra with basis:
    the path algebra of the Kronecker quiver
    (containing the arrows a:x->y and b:x->y) over Rational Field
    sage: x, y, a, b = F.basis()
    sage: A = F.annihilator([a + 3*b + 2*y]); A
    Free module generated by {0} over Rational Field
    sage: [b.lift() for b in A.basis()]
    [-1/2*a - 3/2*b + x]
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
```
>>> x, y, a, b = F.basis()
>>> A = F.annihilator([a + Integer(3)*b + Integer(2)*y]); A
Free module generated by (0) over Rational Field
```
```
>>> [b.lift() for b in A.basis()]
[-1/2*a - 3/2*b + x]
```

The category can be used to specify other properties of this subspace, like that this is a subalgebra:
```
sage: # needs sage.modules
sage: center = F.annihilator(F.basis(), F.bracket,
...: category=Algebras(QQ).Subobjects())
sage: (e,) = center.basis()
sage: e.lift()
x + y
sage: e * e == e
True
```
```
>>> from sage.all import *
>>> # needs sage.modules
>>> center = F.annihilator(F.basis(), F.bracket,
...: category=Algebras(QQ).Subobjects())
```
```
>>> (e,) = center.basis()
```
```
>>> e.lift()
x + y
>>> e * e == e
True
```

Taking annihilator is order reversing for inclusion:
```
sage: # needs sage.modules
sage: A = F.annihilator([]); A .rename("A")
sage: Ax = F.annihilator([x]); Ax .rename("Ax")
sage: Ay = F.annihilator([y]); Ay .rename("Ay")
sage: Axy = F.annihilator([x,y]); Axy.rename("Axy")
sage: P = Poset(\[[A, Ax, Ay, Axy], attrcall("is submodule")\])
```
```
```
```
```
```
```
```
```
```
```
```
Return a basis of the annihilator of a finite set of elements.

**INPUT:**
- \( S \) – a finite set of objects
- \( \text{action} \) – a function (default: `operator.mul`)
- \( \text{side} \) – ‘left’ or ‘right’ (default: ‘right’): on which side of \( \text{self} \) the elements of \( S \) acts.

See `annihilator()` for the assumptions and definition of the annihilator.

**EXAMPLES:**

By default, the action is the standard \( \ast \) operation. So our first example is about an algebra:

```python
sage: # needs sage.graphs sage.modules
sage: F = FiniteDimensionalAlgebrasWithBasis(QQ).example(); F
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: x,y,a,b = F.basis()
```

In this algebra, multiplication on the right by \( x \) annihilates all basis elements but \( x \):

```python
sage: x*x, y*x, a*x, b*x
˓→ needs sage.graphs sage.modules
(x, 0, 0, 0)

sage: from sage.all import *
˓→ needs sage.graphs sage.modules
sage: x*y, y*x, a*x, b*x
˓→ needs sage.graphs sage.modules
(x, 0, 0, 0)
```

So the annihilator is the subspace spanned by \( y, a, \) and \( b \):

```python
sage: F.annihilator_basis([x])
˓→ needs sage.graphs sage.modules
(y, a, b)

sage: from sage.all import *
˓→ needs sage.graphs sage.modules
sage: F.annihilator_basis([x])
˓→ needs sage.graphs sage.modules
(y, a, b)
```

The same holds for \( a \) and \( b \):

```python
sage: a*a, y*a, a*a, b*a
˓→ needs sage.graphs sage.modules
(a, 0, 0, 0)

sage: a*a, y*a, a*a, b*a
˓→ needs sage.graphs sage.modules
(a, 0, 0, 0)
```
On the other hand, \( y \) annihilates only \( x \):

\[
\text{sage: } F.\text{annihilator\_basis}([y])
\]

Here is a non trivial annihilator:

\[
\text{sage: } F.\text{annihilator\_basis}([a + 3\cdot b + 2\cdot y])
\]

Let's check it:

\[
\text{sage: } (-1/2\cdot a - 3/2\cdot b + x) \cdot (a + 3\cdot b + 2\cdot y)
\]

Doing the same calculations on the left exchanges the roles of \( x \) and \( y \):

\[
\text{sage: } F.\text{annihilator\_basis}([y], \text{side}="\text{left}")
\]

\[
\text{sage: } F.\text{annihilator\_basis}([a], \text{side}="\text{left}")
\]

\[
\text{sage: } F.\text{annihilator\_basis}([b], \text{side}="\text{left}")
\]

\[
\text{sage: } F.\text{annihilator\_basis}([x], \text{side}="\text{left}")
\]

\[
\text{sage: } F.\text{annihilator\_basis}([a + 3\cdot b + 2\cdot x], \text{side}="\text{left}")
\]
By specifying an inner product, this method can be used to compute the orthogonal of a subspace:

```python
sage: def scalar(u,v):
    ... return vector([sum(u[i]*v[i] for i in F.basis().keys())])
```

```python
sage: F.annihilator_basis([x + y, a + b], scalar)
(x - y, a - b)
```

By specifying the standard Lie bracket as action, one can compute the commutator of a subspace of $F$:

```python
sage: F.annihilator_basis([a + b], action=F.bracket)
(x + y, a, b)
```

In particular one can compute a basis of the center of the algebra. In our example, it is reduced to the identity:

```python
sage: F.annihilator_basis(F.algebra_generators(), action=F.bracket)
(x + y,)
```

But see also `FiniteDimensionalAlgebrasWithBasis.ParentMethods.center_basis()`. 
echelon_form(elements, row_reduced=False, order=None)

Return a basis in echelon form of the subspace spanned by a finite set of elements.

INPUT:
- elements – a list or finite iterable of elements of self
- row_reduced – (default: False) whether to compute the basis for the row reduced echelon form
- order – (optional) either something that can be converted into a tuple or a key function

OUTPUT:
A list of elements of self whose expressions as vectors form a matrix in echelon form. If base_ring is specified, then the calculation is achieved in this base ring.

EXAMPLES:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: V = X.echelon_form([x[0]-x[1], x[0]-x[2], x[1]-x[2]]); V
[x[0] - x[2], x[1] - x[2]]
sage: matrix(list(map(vector, V)))
[ 1 0 -1]
[ 0 1 -1]
```

```python
>>> from sage.all import *

>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix="x")
>>> x = X.basis()
>>> V = X.echelon_form([x[Integer(0)]-x[Integer(1)], x[Integer(0)]-x[Integer(2)], x[Integer(1)]-x[Integer(2)]]); V
[x[0] - x[2], x[1] - x[2]]
>>> matrix(list(map(vector, V)))
[ 1 0 -1]
[ 0 1 -1]
```

```python
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(ZZ, [1,2,3,4])
```

```python
>>> from sage.all import *

>>> # needs sage.modules
>>> F = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2),Integer(3), Integer(4)])
```

```python
sage: F.echelon_form(elements)
```

```python
>>> from sage.all import *

>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ["a","b","c"])
```

```python
sage: f = CombinatorialFreeModule(QQ, ['a','b','c'])
```
sage: F.echelon_form([8*a+b+10*c, -3*a+b-c, a-b-c])  # needs sage.modules
[B['a'] + B['c'], B['b'] + 2*B['c']]

>>> from sage.all import *

>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # needs sage.modules

>>> a,b,c = F.basis()  # needs sage.modules

>>> F.echelon_form([Integer(8)*a+b+Integer(10)*c, -Integer(3)*a+b-c, a-b-c])  # needs sage.modules
[B['a'] + B['c'], B['b'] + 2*B['c']]

sage: R.<x,y> = QQ[]

sage: C = CombinatorialFreeModule(R, range(3), prefix=x)  # needs sage.modules

sage: x = C.basis()  # needs sage.modules

sage: C.echelon_form([x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]])  # needs sage.modules sage.rings.function_field
[x[0] - x[2], x[1] - x[2]]

sage: M = MatrixSpace(QQ, 3, 3)  # needs sage.modules

sage: A = M([[0, 0, 2], [0, 0, 0], [0, 0, 0]])  # needs sage.modules

sage: M.echelon_form([A, A])  # needs sage.modules
[ 0 0 1
 0 0 0
 0 0 0 ]

>>> from sage.all import *

>>> M = MatrixSpace(QQ, Integer(3), Integer(3))  # needs sage.modules

>>> A = M([[Integer(0), Integer(0), Integer(2)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0), Integer(0)]])  # needs sage.modules

>>> M.echelon_form([A, A])  # needs sage.modules
[ 0 0 1 ]

(continues on next page)
from_vector(vector, order=None, coerce=True)

Build an element of self from a vector.

EXAMPLES:

```python
sage: # needs sage.modules
sage: p_mult = matrix([[0,0,0], [0,0,-1], [0,0,0]])
sage: q_mult = matrix([[0,0,1], [0,0,0], [0,0,0]])
sage: A = algebras.FiniteDimensional(QQ, [p_mult, q_mult, matrix(QQ, 3, 3)], 'p,q,z')
sage: A.from_vector(vector([1,0,2]))
p + 2*z
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> p_mult = matrix([[Integer(0),Integer(0),Integer(0)], [Integer(0),-Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(0)]])
>>> q_mult = matrix([[Integer(0),Integer(0),Integer(1)], [Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),Integer(0)]])
>>> A = algebras.FiniteDimensional(QQ, [p_mult, q_mult, matrix(QQ, Integer(3), Integer(3))], 'p,q,z')
>>> A.from_vector(vector([Integer(1),Integer(0),Integer(2)]))
p + 2*z
```

gens()

Return the generators of self.

OUTPUT:

A tuple containing the basis elements of self.

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
# needs sage.modules
sage: F.gens()
# needs sage.modules
(B['a'], B['b'], B['c'])
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
# needs sage.modules
>>> F.gens()
# needs sage.modules
(B['a'], B['b'], B['c'])
```

invariant_module(S, action=<built-in function mul>, action_on_basis=None, side='left', **kwargs)

Return the submodule of self invariant under the action of S.

For a semigroup S acting on a module M, the invariant submodule is given by

\[ M^S = \{ m \in M : s \cdot m = m, \forall s \in S \}. \]
INPUT:
- \( S \) – a finitely-generated semigroup
- \( \text{action} \) – a function (default: \texttt{operator.mul})
- \( \text{side} \) – 'left' or 'right' (default: 'right'); which side of self the elements of \( S \) acts
- \( \text{action_on_basis} \) – (optional) define the action of \( S \) on the basis of self

OUTPUT:
- \texttt{FiniteDimensionalInvariantModule}

EXAMPLES:
We build the invariant module of the permutation representation of the symmetric group:

```sage
sage: # needs sage.combinat sage.groups sage.modules
sage: G = SymmetricGroup(3); G.rename('S3')
```

```sage
sage: M = FreeModule(ZZ, [1,2,3], prefix='M'); M.rename('M')
```

```sage
sage: action = lambda g, x: M.term(g(x))
```

```sage
sage: I = M.invariant_module(G, action_on_basis=action); I
```

\((S3)\)-invariant submodule of \( M \)

```sage
sage: I.basis()
```

\{0: B[0]\}

```sage
sage: [I.lift(b) for b in I.basis()]
```


```sage
sage: G.rename(); M.rename()  # reset the names
```

We can construct the invariant module of any module that has an action of \( S \). In this example, we consider the dihedral group \( G = D_4 \) and the subgroup \( H < G \) of all rotations. We construct the \( H \)-invariant module of the group algebra \( \mathbb{Q}[G] \):

```sage
sage: # needs sage.combinat sage.groups sage.modules
sage: G = groups.permutation.Dihedral(4)
```

```sage
sage: H = G.subgroup(G.gen(0))
```

```sage
sage: H
```

Subgroup generated by [(1,2,3,4)]

```sage
sage: H.cardinality()
```

4

```sage
sage: A = G.algebra(QQ)
```

```sage
sage: I = A.invariant_module(H)
```

```sage
sage: [I.lift(b) for b in I.basis()]
```

\([() + (1,2,3,4) + (1,3)(2,4) + (1,4,3,2), (2,4) + (1,2)(3,4) + (1,3) + (1,4)(2,3)]\)

```sage
sage: all(h * I.lift(b) == I.lift(b)
```

(continues on next page)
for b in I.basis() for h in H)

True

```python
>>> from sage.all import *
>>> # needs sage.groups
>>> G = groups.permutation.Dihedral(Integer(4))
>>> H = G.subgroup(G.gen(Integer(0)))
>>> H
Subgroup generated by [(1,2,3,4)]
of (Dihedral group of order 8 as a permutation group)
>>> H.cardinality()
4

>>> # needs sage.groups sage.modules
>>> A = G.algebra(QQ)
>>> I = A.invariant_module(H)

>>> [I.lift(b) for b in I.basis()]
[(0 + (1,2,3,4) + (1,3)(2,4) + (1,4,3,2),
   (2,4) + (1,2)(3,4) + (1,3) + (1,4)(2,3)]

>>> all(h * I.lift(b) == I.lift(b) for b in I.basis() for h in H)
True
```

**twisted_invariant_module** *(G, chi, action=operator.mul, action_on_basis=None, side='left', **kwargs)*

Create the isotypical component of the action of G on self with irreducible character given by chi.

See also:

- **FiniteDimensionalTwistedInvariantModule**

**INPUT:**

- G – a finitely-generated group
- chi – a list/tuple of character values or an instance of ClassFunction_gap
- action – a function (default: operator.mul)
- action_on_basis – (optional) define the action of g on the basis of self
- side – 'left' or 'right' (default: 'right'); which side of self the elements of S acts

**OUTPUT:**

- **FiniteDimensionalTwistedInvariantModule**

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups sage.modules
sage: M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)])
sage: G = SymmetricGroup(Integer(3))
sage: def action(g,x):
    return (M.term(g(x))) # permute coordinates
sage: T = M.twisted_invariant_module(G, [2,0,-1],
        action=action)
```

(continues on next page)
class TensorProducts (category, *args)

Bases: TensorProductsCategory

e extra_super_categories ()

Implement the fact that a (finite) tensor product of finite dimensional modules is a finite dimensional module.

EXAMPLES:

```python
sage: C = ModulesWithBasis(ZZ).FiniteDimensional().TensorProducts()
sage: C.extra_super_categories()
[Category of finite dimensional modules with basis over Integer Ring]
sage: C.FiniteDimensional()
Category of tensor products of finite dimensional modules with basis over Integer Ring
```

>>> from sage.all import *

```python
>>> C = ModulesWithBasis(ZZ).FiniteDimensional().TensorProducts()
>>> C.extra_super_categories()
[Category of finite dimensional modules with basis over Integer Ring]
>>> C.FiniteDimensional()
Category of tensor products of finite dimensional modules with basis over Integer Ring
```

4.54 Finite Dimensional Nilpotent Lie Algebras With Basis

AUTHORS:

• Eero Hakavuori (2018-08-16): initial version

class sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.FiniteDimensionalNilpotentLieAlgebrasWithBasis

Bases: CategoryWithAxiom_over_base_ring

Category of finite dimensional nilpotent Lie algebras with basis.

class ParentMethods

Bases: object

is_nilpotent ()

Return True since self is nilpotent.

EXAMPLES:

```python
sage: L = LieAlgebra(QQ, {('x','y'): {('z'): 1}}, nilpotent=True)  # needs sage.combinat sage.modules
sage: L.is_nilpotent()  # needs sage.combinat sage.modules
True
```
```python
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'): {'z': Integer(1)}}, nilpotent=True)
# needs sage.combinat sage.modules
>>> L.is_nilpotent()
# needs sage.combinat sage.modules
True
```

```
lie_group(name='G', **kwds)
Return the Lie group associated to self.

INPUT:
  * name – string (default: 'G'); the name (symbol) given to the Lie group

EXAMPLES:

We define the Heisenberg group:
```
sage: L = lie_algebras.Heisenberg(QQ, 1)
# needs sage.combinat sage.modules
sage: G = L.lie_group('G'); G
# needs sage.combinat sage.modules sage.symbolic
Lie group G of Heisenberg algebra of rank 1 over Rational Field
```
```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
# needs sage.combinat sage.modules
>>> G = L.lie_group('G'); G
# needs sage.combinat sage.modules sage.symbolic
Lie group G of Heisenberg algebra of rank 1 over Rational Field
```
```
We test multiplying elements of the group:
```
sage: # needs sage.combinat sage.modules sage.symbolic
sage: p, q, z = L.basis()
sage: g = G.exp(p); g
exp(p1)
sage: h = G.exp(q); h
exp(q1)
sage: g * h
exp(p1 + q1 + 1/2*z)
```
```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules sage.symbolic
>>> p, q, z = L.basis()
>>> g = G.exp(p); g
exp(p1)
>>> h = G.exp(q); h
exp(q1)
>>> g * h
exp(p1 + q1 + 1/2*z)
```
```
We extend an element of the Lie algebra to a left-invariant vector field:
```
sage: X = G.left_invariant_extension(2*p + 3*q, name='X'); X
# needs sage.combinat sage.modules sage.symbolic
Vector field X on the Lie group G of Heisenberg algebra of rank 1 over Rational Field
sage: X.at(G.one()).display()
```
```
(continues on next page)
```
See also:

NilpotentLieGroup

step()

Return the nilpotency step of self.

EXAMPLES:

```
sage: # needs sage.combinat.sage.modules
sage: L = LieAlgebra(QQ, {('X', 'Y'): {'Z': 1}}, nilpotent=True)
sage: L.step()
2
sage: sc = {('X', 'Y'): {'Z': 1}, ('X', 'Z'): {'W': 1}}
sage: LieAlgebra(QQ, sc, nilpotent=True).step()
3
```

4.55 Finite dimensional semisimple algebras with basis

```
class sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis

Bases: CategoryWithAxiom_over_base_ring

The category of finite dimensional semisimple algebras with a distinguished basis

EXAMPLES:
```
This category is best constructed as:

```python
class Commutative(base_category):
    Bases:
    CategoryWithAxiom_over_base_ring

    class ParentMethods:
        Bases: object

        central_orthogonal_idempotents() :
        Return the central orthogonal idempotents of this semisimple commutative algebra.

        Those idempotents form a maximal decomposition of the identity into primitive orthogonal idempotents.

        OUTPUT:
        A list of orthogonal idempotents of self.

        EXAMPLES:
```

4.55. Finite dimensional semisimple algebras with basis 733
Lifting those idempotents from the center, we recognize among them the sum and alternating sum of all permutations:

```
sage: [e.lift() for e in idempotents]  # needs sage.combinat sage.groups sage.modules
[1/24*() + 1/24*(3,4) + 1/24*(2,3) + 1/24*(2,3,4) + 1/24*(2,4,3)
 + 1/24*(2,4) + 1/24*(1,2) + 1/24*(1,2,3) + 1/24*(1,2,3,4) + 1/24*(1,2,4)
 + 1/24*(1,3,2) + 1/24*(1,3,4) + 1/24*(1,4,3) + 1/24*(1,4,3,2) + 1/24*(1,4,4)
 + 1/24*(1,5) + 1/24*(1,6) + 1/24*(1,7) + 1/24*(1,7,2) + 1/24*(1,7,3)
 + 1/24*(1,7,4) + 1/24*(1,7,5) + 1/24*(1,7,6) + 1/24*(2,3,4,5) + 1/24*(2,3,5,4)
 + 1/24*(2,4,3,5) + 1/24*(2,5,3,4) + 1/24*(2,5,4,3) + 1/24*(2,6,3,4)
 + 1/24*(2,6,4,3) + 1/24*(3,4,5,6) + 1/24*(3,5,4,6) + 1/24*(3,5,6,4)
 + 1/24*(3,6,4,5) + 1/24*(4,5,3,6) + 1/24*(4,5,6,3) + 1/24*(4,6,3,5)
 + 1/24*(4,6,5,3) + 1/24*(5,6,3,4) + 1/24*(5,6,4,3) + 1/24*(6,5,3,4)]
```

We check that they indeed form a decomposition of the identity of $\mathbb{Z}_4$ into orthogonal idempotents:

```
sage: # needs sage.combinat sage.groups sage.modules
sage: Z4.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

```
>>> from sage.all import *

>>> # needs sage.combinat sage.groups sage.modules

>>> Z4.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```
class ParentMethods:
    Bases: object

    central_orthogonal_idempotents()

    Return a maximal list of central orthogonal idempotents of self.

    Central orthogonal idempotents of an algebra $A$ are idempotents $(e_1, \ldots, e_n)$ in the center of $A$ such that $e_i e_j = 0$ whenever $i \neq j$.

    With the maximality condition, they sum up to 1 and are uniquely determined (up to order).

    EXAMPLES:
    For the algebra of the (abelian) alternating group $A_3$, we recover three idempotents corresponding to the three one-dimensional representations $V_i$ on which $(1, 2, 3)$ acts on $V_i$ as multiplication by the $i$-th power of a cube root of unity:

    sage: # needs sage.groups sage.rings.number_field
    sage: R = CyclotomicField(3)
    sage: A3 = AlternatingGroup(3).algebra(R)
    sage: idempotents = A3.central_orthogonal_idempotents()
    sage: idempotents
    (1/3*() + 1/3*(1,2,3) + 1/3*(1,3,2),
     1/3*() - (1/3*zeta3+1/3)*(1,2,3) - (-1/3*zeta3)*(1,3,2),
     1/3*() - (-1/3*zeta3)*(1,2,3) - (1/3*zeta3+1/3)*(1,3,2))
    sage: A3.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
    True

    For the semisimple quotient of a quiver algebra, we recover the vertices of the quiver:

    sage: # needs sage.graphs sage.modules sage.rings.number_field
    sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
    An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver (containing the arrows a:x->y and b:x->y) over Rational Field
    sage: Aquo = A.semisimple_quotient()
    sage: Aquo.central_orthogonal_idempotents()
    (B['x'], B['y'])

    (continues on next page)
Aquo = A.semisimple_quotient()
Aquo.central_orthogonal_idempotents()

radical_basis(**keywords)

Return a basis of the Jacobson radical of this algebra.

• keywords – for compatibility; ignored.

OUTPUT: the empty list since this algebra is semisimple.

EXAMPLES:

```python
sage: A = SymmetricGroup(4).algebra(QQ)  # needs sage.combinat sage.groups sage.modules
sage: A.radical_basis()  # needs sage.combinat sage.groups sage.modules
()
```

4.56 Finite Enumerated Sets

```python
>>> from sage.all import *
>>> A = SymmetricGroup(Integer(4)).algebra(QQ)  # needs sage.combinat sage.groups sage.modules
>>> A.radical_basis()  # needs sage.combinat sage.groups sage.modules
()
```

Todo: sage.combinat.debruijn_sequence.DeBruijnSequences should not inherit from this class. If that is solved, then FiniteEnumeratedSets shall be turned into a subclass of Category_singleton.

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

class ParentMethods
    Bases: object
    cardinality()

        Return the cardinality of self.

    EXAMPLES:

        sage: E = FiniteEnumeratedSet([1,2,3])
        sage: C = cartesian_product([E, SymmetricGroup(4)])  
        # needs sage.groups
        sage: C.cardinality()  
        # needs sage.groups
        72

        sage: E = FiniteEnumeratedSet([])
        sage: C = cartesian_product([E, ZZ, QQ])
        sage: C.cardinality()  
        0

        sage: C = cartesian_product([ZZ, QQ])
        sage: C.cardinality()  
        +Infinity

        sage: cartesian_product([GF(5), Permutations(10)]).cardinality()  
        18144000
        sage: cartesian_product([GF(71)]*20).cardinality() == 71**20
        True

        >>> from sage.all import *
        >>> E = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])
        >>> C = cartesian_product([E, SymmetricGroup(Integer(4))])
        # needs sage.groups
        >>> C.cardinality()  
        # needs sage.groups
        72

        >>> E = FiniteEnumeratedSet([])
        >>> C = cartesian_product([E, ZZ, QQ])
        >>> C.cardinality()  
        0
```python
>>> C = cartesian_product([ZZ, QQ])
>>> C.cardinality()
+Infinity
```
```
>>> cartesian_product([GF(Integer(5)), Permutations(Integer(10))]).
˓→cardinality()
18144000
>>> cartesian_product([GF(Integer(71))]*Integer(20)).cardinality() == ˓→Integer(71)**Integer(20)
True
```

**last()**

Return the last element

**EXAMPLES:**

```python
sage: C = cartesian_product([Zmod(42), Partitions(10), ....: needs sage.combinat
....: IntegerRange(5)])
```
```
sage: C.last()
˓→needs sage.combinat
(41, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 4)
```
```
>>> from sage.all import *
>>>
>>> C = cartesian_product([Zmod(Integer(42)), Partitions(Integer(10)), ˓→# needs sage.combinat...
IntegerRange(Integer(5))])
>>>
>>> C.last()
˓→needs sage.combinat
(41, [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], 4)
```

**random_element(**args**)**

Return a random element of this Cartesian product.

The extra arguments are passed down to each of the factors of the Cartesian product.

**EXAMPLES:**

```python
sage: C = cartesian_product([Permutations(10)]*5)
sage: C.random_element()  # random
([2, 9, 4, 7, 1, 8, 6, 10, 5, 3],
 [8, 6, 5, 7, 1, 4, 9, 3, 10, 2],
 [5, 10, 3, 8, 2, 9, 1, 4, 7, 6],
 [9, 6, 10, 3, 2, 1, 5, 8, 7, 4],
 [8, 5, 2, 9, 10, 3, 7, 1, 4, 6])
sage: C = cartesian_product([ZZ]*10)
sage: c1 = C.random_element()
sage: c1  # random
(3, 1, 4, 1, 1, -3, 0, -4, -17, 2)
sage: c2 = C.random_element(4,7)
sage: c2  # random
(6, 5, 6, 4, 5, 6, 6, 4, 5, 5)
sage: all(4 <= i < 7 for i in c2)
True
```
rank \(x\)

Return the rank of an element of this Cartesian product.

The \textit{rank} of \(x\) is its position in the enumeration. It is an integer between 0 and \(n-1\) where \(n\) is the cardinality of this set.

See also:
- \texttt{EnumeratedSets.ParentMethods.rank()} 
- \texttt{unrank()}

EXAMPLES:

```
>>> from sage.all import *
>>> C = cartesian_product([Permutations(Integer(10))]**Integer(5))
>>> C.random_element() # random
([2, 9, 4, 7, 1, 8, 6, 10, 5, 3],
 [8, 6, 5, 7, 1, 4, 9, 3, 10, 2],
 [5, 10, 3, 8, 2, 9, 1, 4, 7, 6],
 [9, 6, 10, 3, 2, 1, 5, 8, 7, 4],
 [8, 5, 2, 9, 10, 3, 7, 1, 4, 6])

>>> C = cartesian_product([ZZ]**Integer(10))
>>> c1 = C.random_element()
>>> c1 # random
(3, 1, 4, 1, 1, -3, 0, -4, -17, 2)
>>> c2 = C.random_element(Integer(4),Integer(7))
>>> c2 # random
(6, 5, 6, 4, 5, 6, 6, 4, 5, 5)
>>> all(Integer(4) <= i < Integer(7) for i in c2)
True
```
54213

sage: C.unrank(54213)
('a', 86, [7, 5, 4, 4])

sage: from sage.all import *
sage: C = cartesian_product([GF(Integer(2)), GF(Integer(11)), GF(Integer(7))])
sage: C.rank(C((Integer(1), Integer(2), Integer(5))))
96
sage: C.rank(C((Integer(0), Integer(0), Integer(0))))
0

>>> for c in C: print(C.rank(c))

0
1
2
3
4
5...
150
151
152
153

>>> # needs sage.combinat
>>> F1 = FiniteEnumeratedSet('abcdefgh')
>>> F2 = IntegerRange(Integer(250))
>>> F3 = Partitions(Integer(20))
>>> C = cartesian_product([F1, F2, F3])
>>> c = C((a, Integer(86), [Integer(7), Integer(5), Integer(4), Integer(4)]))
>>> C.rank(c)
54213
>>> C.unrank(Integer(54213))
('a', 86, [7, 5, 4, 4])

unrank \((i)\)

Return the \(i\)-th element of this Cartesian product.

INPUT:

- \(i\) – integer between 0 and \(n-1\) where \(n\) is the cardinality of this set.

See also:

- EnumeratedSets.ParentMethods.unrank()
- rank()

EXAMPLES:

sage: C = cartesian_product([GF(3), GF(11), GF(7), GF(5)])
sage: c = C.unrank(123); c
(0, 3, 3, 3)
sage: C.rank(c)
123

sage: c = C.unrank(857); c
('a', 1, [Integer(7), Integer(5), Integer(4), Integer(4)])

sage: C.rank(c)
0

sage: C.unrank(Integer(0))
('a', 1, [Integer(7), Integer(5), Integer(4), Integer(4)])

(continues on next page)
```python
sage: C.rank(c)
857

sage: C.unrank(2500)
Traceback (most recent call last):
...
IndexError: index i (=2) is greater than the cardinality

>>> from sage.all import *

>>> C = cartesian_product([GF(Integer(3)), GF(Integer(11)),
→GF(Integer(7)), GF(Integer(5))])

>>> c = C.unrank(Integer(123)); c
(0, 3, 3, 3)

>>> C.rank(c)
123

>>> c = C.unrank(Integer(857)); c
(2, 2, 3, 2)

>>> C.rank(c)
857

>>> C.unrank(Integer(2500))
Traceback (most recent call last):
...
IndexError: index i (=2) is greater than the cardinality
```

**extra_super_categories()**

A Cartesian product of finite enumerated sets is a finite enumerated set.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().CartesianProducts()
sage: C.extra_super_categories()
[Category of finite enumerated sets]
```

**class IsomorphicObjects**(category, *args)**

Bases: IsomorphicObjectsCategory

**class ParentMethods**

Bases: object

**cardinality()**

Returns the cardinality of self which is the same as that of the ambient set self is isomorphic to.

EXAMPLES:

```python
sage: A = FiniteEnumeratedSets().IsomorphicObjects().example(); A
The image by some isomorphism of An example of a finite enumerated...
→set: {1,2,3}
```
sage: A.cardinality()
3

```python
>>> from sage.all import *
>>> A = FiniteEnumeratedSets().IsomorphicObjects().example(); A
The image by some isomorphism of An example of a finite enumerated...
→ set: {1,2,3}
```

```
>>> A.cardinality()
3
```

element ()

Returns an example of isomorphic object of a finite enumerated set, as per `Category.example`.

EXAMPLES:

```python
sage: FiniteEnumeratedSets().IsomorphicObjects().example()
The image by some isomorphism of An example of a finite enumerated set:
→{1,2,3}
```

```python
>>> from sage.all import *
>>> FiniteEnumeratedSets().IsomorphicObjects().example()
The image by some isomorphism of An example of a finite enumerated set:
→{1,2,3}
```

class ParentMethods

Bases: object

cardinality (*ignored_args, **ignored_kwds)

Return the cardinality of self.

This brute force implementation of `cardinality()` iterates through the elements of `self` to count them.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
sage: C._cardinality_from_iterator()
3
```

```python
>>> from sage.all import *
>>> C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
```

```
>>> C._cardinality_from_iterator()
3
```

iterator_range (start=None, stop=None, step=None)

Iterate over the range of elements of `self` starting at `start`, ending at `stop`, and stepping by `step`.

See also:

unrank(), unrank_range()

EXAMPLES:
sage: F = FiniteEnumeratedSet([1,2,3])
sage: list(F.iterator_range(1))
[2, 3]
sage: list(F.iterator_range(stop=2))
[1, 2]
sage: list(F.iterator_range(stop=2, step=2))
[1]
sage: list(F.iterator_range(start=1, step=2))
[2]
sage: list(F.iterator_range(start=1, stop=2))
[2]
sage: list(F.iterator_range(start=0, stop=1))
[1, 3]
sage: list(F.iterator_range(start=0, stop=3, step=2))
[1, 3]
sage: list(F.iterator_range(stop=-1))
[1, 2]
sage: F = FiniteEnumeratedSet([1,2,3,4])
sage: list(F.iterator_range(start=1, stop=3))
[2, 3]

>>> from sage.all import *

>>> F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])

>>> list(F.iterator_range(Integer(1)))
[2, 3]

>>> list(F.iterator_range(stop=Integer(2)))
[1, 2]

>>> list(F.iterator_range(stop=Integer(2), step=Integer(2)))
[1]

>>> list(F.iterator_range(start=Integer(1), stop=Integer(2)))
[2]

>>> list(F.iterator_range(start=Integer(1), stop=Integer(3), step=Integer(2)))
[1, 3]

>>> list(F.iterator_range(stop=-Integer(1)))
[1, 2]

>>> F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3),Integer(4)])

>>> list(F.iterator_range(start=Integer(1), stop=Integer(3)))
[2, 3]

>>> list(F.iterator_range(stop=Integer(10)))
[1, 2, 3, 4]

last()

The last element of self.

self.last() returns the last element of self.

This is the default (brute force) implementation from the category FiniteEnumeratedSet() which can be used when the method __iter__ is provided. Its complexity is \(O(n)\) where \(n\) is the size of self.
random_element()

A random element in self.

self.random_element() returns a random element in self with uniform probability.

This is the default implementation from the category EnumeratedSet() which uses the method unrank.

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().example()
sage: n = C.random_element()
sage: n in C
True

sage: n = C._random_element_from_unrank()
sage: n in C
True
```

TODO: implement _test_random which checks uniformness

tuple()

Return a tuple of the elements of ``self``.

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().example()
sage: C.tuple()
(1, 2, 3)
sage: C.tuple() is C.tuple()
True
```
```python
>>> from sage.all import *
>>> C = FiniteEnumeratedSets().example()
>>> C.tuple()
(1, 2, 3)
>>> C.tuple() is C.tuple()
True
```

**unrank_range**(start=None, stop=None, step=None)

Return the range of elements of `self` starting at `start`, ending at `stop`, and stepping by `step`.

See also `unrank()`.

EXAMPLES:

```python
sage: F = FiniteEnumeratedSet([1,2,3])
sage: F.unrank_range(1)
[2, 3]
sage: F.unrank_range(stop=2)
[1, 2]
sage: F.unrank_range(stop=2, step=2)
[1]
sage: F.unrank_range(start=1, step=2)
[2]
sage: F.unrank_range(stop=-1)
[1, 2]
sage: F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])
sage: F.unrank_range(Integer(1))
[2, 3]
```

```python
>>> from sage.all import *

>>> F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])
>>> F.unrank_range(Integer(1))
[2, 3]
>>> F.unrank_range(stop=Integer(2))
[1, 2]
>>> F.unrank_range(stop=Integer(2), step=Integer(2))
[1]
>>> F.unrank_range(start=Integer(1), step=Integer(2))
[2]
>>> F.unrank_range(stop=-Integer(1))
[1, 2]
```

```python
>>> F = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> F.unrank_range(stop=Integer(10))
[1, 2, 3, 4]
```
### 4.57 Finite fields

**class** `sage.categories.finite_fields.FiniteFields(base_category)`

Bases: `CategoryWithAxiom_singleton`

The category of finite fields.

**EXAMPLES:**

```
sage: K = FiniteFields(); K
Category of finite enumerated fields

>>> from sage.all import *
>>> K = FiniteFields(); K
Category of finite enumerated fields
```

A finite field is a finite monoid with the structure of a field; it is currently assumed to be enumerated:

```
sage: K.super_categories()
[Category of fields,
 Category of finite commutative rings,
 Category of finite enumerated sets]
```

Some examples of membership testing and coercion:

```
sage: FiniteField(17) in K
True
sage: RationalField() in K
False
sage: K(RationalField())
Traceback (most recent call last):
  ...TypeError: unable to canonically associate a finite field to Rational Field
```

```
>>> from sage.all import *
>>> FiniteField(Integer(17)) in K
True
>>> RationalField() in K
False
>>> K(RationalField())
Traceback (most recent call last):
  ...TypeError: unable to canonically associate a finite field to Rational Field
```

**class** `ElementMethods`

Bases: `object`

**class** `ParentMethods`

Bases: `object`
extra_super_categories()

Any finite field is assumed to be endowed with an enumeration.

4.58 Finite groups

class sage.categories.finite_groups.FiniteGroups(base_category)

    Bases: CategoryWithAxiom_singleton

    The category of finite (multiplicative) groups.

    EXAMPLES:

    sage: C = FiniteGroups(); C
    Category of finite groups
    sage: C.super_categories()
    [Category of finite monoids, Category of groups]
    sage: C.example()
    General Linear Group of degree 2 over Finite Field of size 3

    >>> from sage.all import *
    >>> C = FiniteGroups(); C
    Category of finite groups
    >>> C.super_categories()
    [Category of finite monoids, Category of groups]
    >>> C.example()
    General Linear Group of degree 2 over Finite Field of size 3

class Algebras(category, *args)

    Bases: AlgebrasCategory

    class ParentMethods

        Bases: object

        extra_super_categories()

            Implement Maschke's theorem.

            In characteristic 0 all finite group algebras are semisimple.

        EXAMPLES:

        sage: FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
        True
        sage: FiniteGroups().Algebras(FiniteField(7)).is_subcategory(Algebras(FiniteField(7)).Semisimple())
        False
        sage: FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
        False
        sage: FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
        False
        sage: Cat = CommutativeAdditiveGroups().Finite()
        sage: Cat.Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
        True

        (continues on next page)
sage: Cat.Algebras(GF(7)).is_subcategory(Algebras(GF(7)).Semisimple())
False
sage: Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False
sage: Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False

class ElementMethods
Bases: object

class ParentMethods
Bases: object

cardinality()

Returns the cardinality of self, as per EnumeratedSets.ParentMethods.cardinality().

This default implementation calls order() if available, and otherwise resorts to _cardinality_from_iterator(). This is for backward compatibility only. Finite groups should override this method instead of order().

EXAMPLES:

We need to use a finite group which uses this default implementation of cardinality:

sage: G = groups.misc.SemimonomialTransformation(GF(5), 3); G  # needs sage.rings.number_field
Semimonomial transformation group over Finite Field of size 5 of degree 3
sage: G.cardinality.__module__  # needs sage.rings.number_field
'sage.categories.finite_groups'
sage: G.cardinality()
needs sage.rings.number_field
384

```python
>>> from sage.all import *
>>> G = groups.misc.SemimonomialTransformation(GF(Integer(5)),
Integer(3)); G
Semimonomial transformation group over Finite Field of size 5 of degree 3
>>> G.cardinality.__module__ # needs sage.rings.number_field
'sage.categories.finite_groups'
>>> G.cardinality() # needs sage.rings.number_field
384
```

cayley_graph_disabled (connecting_set=None)

AUTHORS:
• Bobby Moretti (2007-08-10)
• Robert Miller (2008-05-01): editing

conjugacy_classes ()

Return a list with all the conjugacy classes of the group.

This will eventually be a fall-back method for groups not defined over GAP. Right now, it just raises a
NotImplementedError, until we include a non-GAP way of listing the conjugacy classes representa-
tives.

EXAMPLES:

```python
sage: from sage.groups.group import FiniteGroup
sage: G = FiniteGroup()
sage: G.conjugacy_classes() Traceback (most recent call last):
...
NotImplementedError: Listing the conjugacy classes for group <sage.groups.
~group.FiniteGroup object at ...> is not implemented
```

```python
>>> from sage.all import *
>>> from sage.groups.group import FiniteGroup
>>> G = FiniteGroup()
>>> G.conjugacy_classes() Traceback (most recent call last):
...
NotImplementedError: Listing the conjugacy classes for group <sage.groups.
~group.FiniteGroup object at ...> is not implemented
```

conjugacy_classes_representatives ()

Return a list of the conjugacy classes representatives of the group.

EXAMPLES:

```python
sage: G = SymmetricGroup(3)
sage: G.conjugacy_classes_representatives() # needs sage.combinat
[[], (1,2), (1,2,3)]
```
from sage.all import *
G = SymmetricGroup(Integer(3))
G.conjugacy_classes_representatives()
# needs sage.combinat
[(), (1,2), (1,2,3)]

monoid_generators()
Return monoid generators for self.
For finite groups, the group generators are also monoid generators. Hence, this default implementation calls group_generators().
EXAMPLES:
sage: A = AlternatingGroup(4)
sage: A.monoid_generators()
Family ((1,2,3), (2,3,4))

semigroup_generators()
Return semigroup generators for self.
For finite groups, the group generators are also semigroup generators. Hence, this default implementation calls group_generators().
EXAMPLES:
sage: A = AlternatingGroup(4)
sage: A.semigroup_generators()
Family ((1,2,3), (2,3,4))

some_elements()
Return some elements of self.
EXAMPLES:
sage: A = AlternatingGroup(4)
sage: A.some_elements()
Family ((1,2,3), (2,3,4))

type

example()
Return an example of finite group, as per Category.example().
EXAMPLES:
4.59 Finite lattice posets

class sage.categories.finite_lattice_posets.FiniteLatticePosets(base_category)
    Bases: CategoryWithAxiom

The category of finite lattices, i.e. finite partially ordered sets which are also lattices.

EXAMPLES:

sage: FiniteLatticePosets()
Category of finite lattice posets

sage: FiniteLatticePosets().super_categories()
[Category of lattice posets, Category of finite posets]

sage: FiniteLatticePosets().example()
NotImplemented

>>> from sage.all import *

>>> FiniteLatticePosets()
Category of finite lattice posets

>>> FiniteLatticePosets().super_categories()
[Category of lattice posets, Category of finite posets]

>>> FiniteLatticePosets().example()
NotImplemented

See also:

FinitePosets, LatticePosets, FiniteLatticePoset

class ParentMethods
    Bases: object

irreducibles_poset()

    Return the poset of meet- or join-irreducibles of the lattice.

    A *join-irreducible* element of a lattice is an element with exactly one lower cover. Dually a *meet-irreducible* element has exactly one upper cover.

    This is the smallest poset with completion by cuts being isomorphic to the lattice. As a special case this returns one-element poset from one-element lattice.

See also:

completion_by_cuts().

EXAMPLES:

sage: # needs sage.combinat sage.graphs sage.modules
sage: L = LatticePoset({1: [2, 3, 4], 2: [5, 6], 3: [5],
    ....: 4: [6], 5: [9, 7], 6: [9, 8], 7: [10],

(continues on next page)
is_lattice_morphism\( (f, \text{codomain}) \)  
Return whether \( f \) is a morphism of posets from \( \text{self} \) to \( \text{codomain} \).

A map \( f : P \to Q \) is a poset morphism if

\[
x \leq y \Rightarrow f(x) \leq f(y)
\]

for all \( x, y \in P \).

INPUT:

- \( f \) — a function from \( \text{self} \) to \( \text{codomain} \)
- \( \text{codomain} \) — a lattice

EXAMPLES:

We build the boolean lattice of \( \{2, 2, 3\} \) and the lattice of divisors of 60, and check that the map \( b \mapsto 5 \prod_{x \in b} x \) is a morphism of lattices:

\[
\text{sage: } D = \text{LatticePoset}((\text{divisors}(60), \text{attrcall("divides")))) \quad \#\text{ needs sage.graphs sage.modules}
\text{sage: } B = \text{LatticePoset}((\text{Subsets([2,2,3])}, \text{attrcall("issubset")})) \quad \#\text{ needs sage.graphs sage.modules}
\text{sage: } \text{def } f(b): \text{ return } D(5^*\text{prod}(b))
\text{sage: } B.\text{is_lattice_morphism}(f, D) \quad \#\text{ needs sage.graphs sage.modules}
\text{True}
\]
We construct the boolean lattice $B_2$:

```python
sage: B = posets.BooleanLattice(2)  # needs sage.graphs
sage: B.cover_relations()  # needs sage.graphs
[[0, 1], [0, 2], [1, 3], [2, 3]]
```

```python
>>> from sage.all import *
>>> B = posets.BooleanLattice(Integer(2))  # needs sage.graphs
>>> B.cover_relations()  # needs sage.graphs
[[0, 1], [0, 2], [1, 3], [2, 3]]
```

And the same lattice with new top and bottom elements numbered respectively $-1$ and 3:

```python
sage: G = DiGraph({-1:[0], 0:[1,2], 1:[3], 2:[3], 3:[4]})  # needs sage.graphs
sage: L = LatticePoset(G)  # needs sage.graphs sage.modules
sage: L.cover_relations()  # needs sage.graphs sage.modules
[[-1, 0], [0, 1], [0, 2], [1, 3], [2, 3], [3, 4]]

sage: f = {B(0): L(0), B(1): L(1), B(2): L(2), B(3): L(3)}.__getitem__  # needs sage.graphs sage.modules
sage: B.is_lattice_morphism(f, L)  # needs sage.graphs sage.modules
True

sage: f = {B(0): L(-1), B(1): L(1), B(2): L(2), B(3): L(3)}.__getitem__  # needs sage.graphs sage.modules
sage: B.is_lattice_morphism(f, L)  # needs sage.graphs sage.modules
False

sage: f = {B(0): L(0), B(1): L(1), B(2): L(2), B(3): L(4)}.__getitem__  # needs sage.graphs sage.modules
sage: B.is_lattice_morphism(f, L)  # needs sage.graphs sage.modules
False
```

```python
>>> f = {B(Integer(0)): L(Integer(0)), B(Integer(1)): L(Integer(1)),
>>>       B(Integer(2)): L(Integer(2)), B(Integer(3)): L(Integer(3))}.__getitem__
>>> # needs sage.graphs sage.modules
>>> B.is_lattice_morphism(f, L)
```

(continues on next page)
See also:

is_poset_morphism()

join_irreducibles()

Return the join-irreducible elements of this finite lattice.

A join-irreducible element of self is an element \( x \) that is not minimal and that cannot be written as the join of two elements different from \( x \).

EXAMPLES:

```python
sage: L = LatticePoset({0:[1,2],1:[3],2:[3,4],3:[5],4:[5]})
sage: L.join_irreducibles()
[1, 2, 4]
```

```python
>>> from sage.all import *
>>> L = LatticePoset({Integer(0):[Integer(1),Integer(2)],
                   Integer(1):[Integer(3)],Integer(2):[Integer(3),Integer(4)],
                   Integer(3):[Integer(5)],Integer(4):[Integer(5)]})
# needs sage.graphs sage.modules
>>> L.join_irreducibles()
[1, 2, 4]
```

See also:

- Dual function: meet_irreducibles()
- Other: double_irreducibles(), join_irreducibles_poset()

join_irreducibles_poset()

Return the poset of join-irreducible elements of this finite lattice.

A join-irreducible element of self is an element \( x \) that is not minimal and can not be written as the join of two elements different from \( x \).

EXAMPLES:
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})  # needs sage.graphs sage.modules
sage: L.join_irreducibles_poset()  # needs sage.graphs sage.modules
Finite poset containing 3 elements

>>> from sage.all import *
>>> L = LatticePoset({Integer(0):[Integer(1),Integer(2),Integer(3)],
Integer(1):[Integer(4)],Integer(2):[Integer(4)],Integer(3):[Integer(4)]})
>>> L.join_irreducibles_poset()  # needs sage.graphs sage.modules
Finite poset containing 3 elements

See also:

- Dual function: `meet_irreducibles_poset()`
- Other: `join_irreducibles()`

`meet_irreducibles()`

Return the meet-irreducible elements of this finite lattice.

A *meet-irreducible element* of `self` is an element `x` that is not maximal and that cannot be written as the meet of two elements different from `x`.

EXAMPLES:

sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})  # needs sage.graphs sage.modules
sage: L.join_irreducibles_poset()  # needs sage.graphs sage.modules
[1, 3, 4]

>>> from sage.all import *
>>> L = LatticePoset({Integer(0):[Integer(1),Integer(2),Integer(3)],
Integer(1):[Integer(4)],Integer(2):[Integer(4)],Integer(3):[Integer(4)]})
>>> L.join_irreducibles_poset()  # needs sage.graphs sage.modules
[1, 3, 4]

See also:

- Dual function: `join_irreducibles()`
- Other: `double_irreducibles()`, `meet_irreducibles_poset()`

`meet_irreducibles_poset()`

Return the poset of join-irreducible elements of this finite lattice.

A *meet-irreducible element* of `self` is an element `x` that is not maximal and cannot be written as the meet of two elements different from `x`.

EXAMPLES:

sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})  # needs sage.graphs sage.modules
sage: L.join_irreducibles_poset()  # needs sage.graphs sage.modules

(continues on next page)
4.60 Finite monoids

```python
from sage.all import *
L = LatticePoset({Integer(0): [Integer(1), Integer(2), Integer(3)],
                 Integer(1): [Integer(4)],
                 Integer(2): [Integer(4)],
                 Integer(3): [Integer(4)]})
```

See also:
- Dual function: `join_irreducibles_poset()`
- Other: `meet_irreducibles()`

```python
class sage.categories.finite_monoids.FiniteMonoids(base_category):
    Bases: CategoryWithAxiom_singleton
    
The category of finite (multiplicative) monoids.

    A finite monoid is a finite sets endowed with an associative unital binary operation `*`.

    EXAMPLES:
    ```
sage: FiniteMonoids()
    Category of finite monoids
    sage: FiniteMonoids().super_categories()
    [Category of monoids, Category of finite semigroups]
    ```
```

```python
class ElementMethods
    Bases: object
    
pseudo_order()
        Return the pair \([k, j]\) with \(k\) minimal and \(0 \leq j < k\) such that \(self^k == self^j\).

        Note that \(j\) is uniquely determined.

    EXAMPLES:
    ```
sage: M = FiniteMonoids().example(); M
    An example of a finite multiplicative monoid: the integers modulo 12
    sage: x = M(2)
    sage: [ x^i for i in range(7) ]
    [1, 2, 4, 8, 4, 8, 4]
    sage: x.pseudo_order()
    ```
```
```
sage: x = M(3)
sage: [ x^i for i in range(7) ]
[1, 3, 9, 3, 9, 3, 9]
sage: x.pseudo_order()
[3, 1]

sage: x = M(4)
sage: [ x^i for i in range(7) ]
[1, 4, 4, 4, 4, 4, 4]
sage: x.pseudo_order()
[2, 1]

sage: x = M(5)
sage: [ x^i for i in range(7) ]
[1, 5, 1, 5, 1, 5, 1]
sage: x.pseudo_order()
[2, 0]

Todo: more appropriate name? see, for example, Jean-Eric Pin’s lecture notes on semigroups.

class ParentMethods
Bases: object

nerve()

The nerve (classifying space) of this monoid.
the nerve $BG$ (if $G$ denotes this monoid), as a simplicial set. The $k$-dimensional simplices of this object are indexed by products of $k$ elements in the monoid:

$$a_1 \ast a_2 \ast \cdots \ast a_k$$

The 0th face of this is obtained by deleting $a_1$, and the $k$-th face is obtained by deleting $a_k$. The other faces are obtained by multiplying elements: the 1st face is

$$(a_1 \ast a_2) \ast \cdots \ast a_k$$

and so on. See Wikipedia article Nerve_(category_theory), which describes the construction of the nerve as a simplicial set.

A simplex in this simplicial set will be degenerate if in the corresponding product of $k$ elements, one of those elements is the identity. So we only need to keep track of the products of non-identity elements. Similarly, if a product $a_{i-1} a_i$ is the identity element, then the corresponding face of the simplex will be a degenerate simplex.

**EXAMPLES:**

The nerve (classifying space) of the cyclic group of order 2 is infinite-dimensional real projective space.

```python
sage: Sigma2 = groups.permutation.Cyclic(2)  # needs sage.groups
sage: BSigma2 = Sigma2.nerve()  # needs sage.graphs sage.groups
sage: BSigma2.cohomology(4, base_ring=GF(2))  # needs sage.graphs sage.groups sage.modules
Vector space of dimension 1 over Finite Field of size 2
```

The $k$-simplices of the nerve are named after the chains of $k$ non-unit elements to be multiplied. The group $\Sigma_2$ has two elements, written () (the identity element) and (1, 2) in Sage. So the 1-cells and 2-cells in $B\Sigma_2$ are:

```python
sage: BSigma2.n_cells(1)  # needs sage.graphs sage.groups
[(1,2)]
sage: BSigma2.n_cells(2)  # needs sage.graphs sage.groups
[(1,2) \ast (1,2)]
```

```python
>>> from sage.all import *

>>> Sigma2 = groups.permutation.Cyclic(Integer(2))  # needs sage.groups

>>> BSigma2 = Sigma2.nerve()  # needs sage.graphs sage.groups

>>> BSigma2.cohomology(Integer(4), base_ring=GF(Integer(2)))  # needs sage.graphs sage.groups sage.modules
Vector space of dimension 1 over Finite Field of size 2

>>> from sage.all import *

>>> BSigma2.n_cells(Integer(1))  # needs sage.graphs sage.groups
[(1,2)]

>>> BSigma2.n_cells(Integer(2))  # needs sage.graphs sage.groups
[(1,2) \ast (1,2)]
```
Another construction of the group, with different names for its elements:

```
sage: # needs sage.groups sage.rings.number_field
sage: C2 = groups.misc.MultiplicativeAbelian([2])
```

```
sage: BC2 = C2.nerve()
sage: BC2.n_cells(0)  # [1]
sage: BC2.n_cells(1)  # [f]
sage: BC2.n_cells(2)  # [f * f]
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> C2 = groups.misc.MultiplicativeAbelian([Integer(2)])
>>> BC2 = C2.nerve()
>>> BC2.n_cells(Integer(0))  # [1]
>>> BC2.n_cells(Integer(1))  # [f]
>>> BC2.n_cells(Integer(2))  # [f * f]
```

With mod \( p \) coefficients, \( B \Sigma_p \) should have its first nonvanishing homology group in dimension \( p \):

```
sage: Sigma3 = groups.permutation.Symmetric(3)  # needs sage.groups
```

```
sage: BSigma3 = Sigma3.nerve()  # needs sage.graphs sage.groups
sage: BSigma3.homology(range(4), base_ring=GF(3))  # needs sage.graphs sage.groups
{0: Vector space of dimension 0 over Finite Field of size 3, 1: Vector space of dimension 0 over Finite Field of size 3, 2: Vector space of dimension 0 over Finite Field of size 3, 3: Vector space of dimension 1 over Finite Field of size 3}
```

```
>>> from sage.all import *
>>> Sigma3 = groups.permutation.Symmetric(Integer(3))  # needs sage.groups
>>> BSigma3 = Sigma3.nerve()  # needs sage.graphs sage.groups
>>> BSigma3.homology(range(Integer(4)), base_ring=GF(Integer(3)))  # needs sage.graphs sage.groups
{0: Vector space of dimension 0 over Finite Field of size 3, 1: Vector space of dimension 0 over Finite Field of size 3, 2: Vector space of dimension 0 over Finite Field of size 3, 3: Vector space of dimension 1 over Finite Field of size 3}
```

Note that we can construct the \( n \)-skeleton for \( B \Sigma_2 \) for relatively large values of \( n \), while for \( B \Sigma_3 \), the complexes get large pretty quickly:

```
sage: # needs sage.graphs sage.groups
sage: Sigma2 = groups.permutation.Symmetric(3)
```

```
sage: Sigma2.nerve().n_skeleton(14)  # Simplicial set with 15 non-degenerate simplices
sage: BSigma3 = Sigma3.nerve()  # Simplicial set with 156 non-degenerate simplices
sage: BSigma3.n_skeleton(3)  # Simplicial set with 156 non-degenerate simplices
```

(continues on next page)
Finally, note that the classifying space of the order $p$ cyclic group is smaller than that of the symmetric group on $p$ letters, and its first homology group appears earlier:

```python
sage: # needs sage.graphs sage.groups sage.rings.number_field
sage: C3 = groups.misc.MultiplicativeAbelian([3])
```

```python
sage: list(C3)
[1, f, f^2]
```

```python
sage: BC3 = C3.nerve()
sage: BC3.n_cells(1)
[f, f^2]
```

```python
sage: len(BC3.n_cells(3))
25
```

```python
sage: BC5 = groups.permutation.Cyclic(5).nerve()
sage: BC5.homology(range(4), base_ring=GF(5))
{0: Vector space of dimension 0 over Finite Field of size 5,
 1: Vector space of dimension 1 over Finite Field of size 5,
 2: Vector space of dimension 1 over Finite Field of size 5,
 3: Vector space of dimension 1 over Finite Field of size 5}
```
rhodes_radical_congruence(base_ring=None)

Return the Rhodes radical congruence of the semigroup.

The Rhodes radical congruence is the congruence induced on S by the map $S \to kS \to kS/\text{rad}kS$ with $k$ a field.

INPUT:
- base_ring (default: $\mathbb{Q}$) a field

OUTPUT:
- A list of couples $(m, n)$ with $m \neq n$ in the lexicographic order for the enumeration of the monoid self.

EXAMPLES:

```python
sage: M = Monoids().Finite().example()
sage: M.rhodes_radical_congruence()  # needs sage.modules
[(0, 6), (2, 8), (4, 10)]
```

By Maschke’s theorem, every group algebra over $\mathbb{Q}$ is semisimple hence the Rhodes radical of a group must be trivial:
REFERENCES:
• [Rho69]

4.61 Finite Permutation Groups

```python
class sage.categories.finite_permutation_groups.FinitePermutationGroups(base_category)

Bases: CategoryWithAxiom

The category of finite permutation groups, i.e. groups concretely represented as groups of permutations acting on a finite set.

It is currently assumed that any finite permutation group comes endowed with a distinguished finite set of generators (method group_generators); this is the case for all the existing implementations in Sage.

EXAMPLES:
```
```
class ParentMethods
Bases: object

cycle_index (parent=None)
Return the cycle index of self.

INPUT:
• self – a permutation group G
• parent – a free module with basis indexed by partitions, or behave as such, with a term and sum
  method (default: the symmetric functions over the rational field in the p basis)
The cycle index of a permutation group G (Wikipedia article Cycle_index) is a gadget counting the
elements of G by cycle type, averaged over the group:

\[ P = \frac{1}{|G|} \sum_{g \in G} p_{\text{cycle type}(g)} \]

EXAMPLES:
Among the permutations of the symmetric group \( S_4 \), there is the identity, 6 cycles of length 2, 3 products
of two cycles of length 2, 8 cycles of length 3, and 6 cycles of length 4:

```python
sage: S4 = SymmetricGroup(4)
sage: P = S4.cycle_index()  # needs sage.combinat
sage: 24 * P  # needs sage.combinat
```

```python
>>> from sage.all import *
>>> S4 = SymmetricGroup(Integer(4))
>>> P = S4.cycle_index()  # needs sage.combinat
>>> Integer(24) * P  # needs sage.combinat
```

If \( l = (l_1, \ldots, l_k) \) is a partition, \(|G| \ P[l]\) is the number of elements of G with cycles of length
\((p_1, \ldots, p_k)\):

```python
sage: 24 * P[ Partition([3,1]) ]  # needs sage.combinat
8
```

```python
>>> from sage.all import *
>>> Integer(24) * P[ Partition([Integer(3),Integer(1)]) ]  # needs sage.combinat
8
```

The cycle index plays an important role in the enumeration of objects modulo the action of a group (Pólya
enumeration), via the use of symmetric functions and plethysms. It is therefore encoded as a symmetric
function, expressed in the powersum basis:

```python
sage: P.parent()  # needs sage.combinat
Symmetric Functions over Rational Field in the powersum basis
```
This symmetric function can have some nice properties; for example, for the symmetric group $S_n$, we get the complete symmetric function $h_n$:

```
sage: S = SymmetricFunctions(QQ); h = S.h()
sage: h( P )
```

Todo: Add some simple examples of Pólya enumeration, once it will be easy to expand symmetric functions on any alphabet.

Here are the cycle indices of some permutation groups:

```
sage: 6 * CyclicPermutationGroup(6).cycle_index()  # needs sage.combinat
sage: 60 * AlternatingGroup(5).cycle_index()  # needs sage.combinat
sage: for G in TransitiveGroups(5): # long time
    ....: G.cardinality() * G.cycle_index()  # needs sage.combinat
p[1, 1, 1, 1, 1] + 4*p[5]
p[1, 1, 1, 1, 1] + 5*p[2, 2, 1] + 4*p[5]
p[1, 1, 1, 1, 1] + 5*p[2, 2, 1] + 10*p[4, 1] + 4*p[5]
```

(continues on next page)
Permutation groups with arbitrary domains are supported (see Issue #22765):

```python
sage: G = PermutationGroup(['b','c','a'], domain=['a','b','c'])
sage: G.cycle_index()
1/3*p[1, 1, 1] + 2/3*p[3]
```

One may specify another parent for the result:

```python
sage: F = CombinatorialFreeModule(QQ, Partitions())
sage: P = CyclicPermutationGroup(6).cycle_index(parent=F)
sage: 6 * P
B[[1, 1, 1, 1, 1, 1]] + B[[2, 2, 2]] + 2*B[[3, 3]] + 2*B[[6]]
```

This parent should be a module with basis indexed by partitions:

```python
sage: CyclicPermutationGroup(6).cycle_index(parent=QQ)
Traceback (most recent call last):
  ... ValueError: parent should be a module with basis indexed by partitions
```

REFERENCES:

• [Ke1991]

AUTHORS:

Nicolas Borie and Nicolas M. Thiéry

profile (n, using_polya=True)

Return the value in n of the profile of the group self.
Optional argument \texttt{using\_polya} allows to change the default method.

**INPUT:**
- \(n\) – a nonnegative integer
- \texttt{using\_polya} (optional) – a boolean: if True (default), the computation uses Pólya enumeration (and all values of the profile are cached, so this should be the method used in case several of them are needed); if False, uses the GAP interface to compute the orbit.

**OUTPUT:**
- A nonnegative integer that is the number of orbits of \(n\)-subsets under the action induced by \texttt{self} on the subsets of its domain (i.e. the value of the profile of \texttt{self} in \(n\))

**See also:**
- \texttt{profile\_series()}

**EXAMPLES:**

```python
sage: C6 = CyclicPermutationGroup(6)
sage: C6.profile(2)  # needs sage.combinat
3
sage: C6.profile(3)  # needs sage.combinat
4
sage: D8 = DihedralGroup(8)
sage: D8.profile(4, using_polya=False)
8
```

\texttt{profile\_polynomial(\texttt{variable}='z')}

Return the (finite) generating series of the (finite) profile of the group.

The profile of a permutation group \(G\) is the counting function that maps each nonnegative integer \(n\) onto the number of orbits of the action induced by \(G\) on the \(n\)-subsets of its domain. If \(f\) is the profile of \(G\), \(f(n)\) is thus the number of orbits of \(n\)-subsets of \(G\).

**INPUT:**
- \texttt{variable} – a variable, or variable name as a string (default: ‘\(z\)’)

**OUTPUT:**
- A polynomial in \texttt{variable} with nonnegative integer coefficients. By default, a polynomial in \(z\) over ZZ.

**See also:**
- \texttt{profile()}

**EXAMPLES:**
sage: # needs sage.combinat
sage: C8 = CyclicPermutationGroup(8)
sage: C8.profile_series()
z^8 + z^7 + 4*z^6 + 7*z^5 + 10*z^4 + 7*z^3 + 4*z^2 + z + 1
sage: D8 = DihedralGroup(8)
sage: poly_D8 = D8.profile_series()
sage: poly_D8
z^8 + z^7 + 4*z^6 + 5*z^5 + 8*z^4 + 5*z^3 + 4*z^2 + z + 1
sage: poly_D8.parent()
Univariate Polynomial Ring in z over Rational Field
sage: D8.profile_series(variable=y)
y^8 + y^7 + 4*y^6 + 5*y^5 + 8*y^4 + 5*y^3 + 4*y^2 + y + 1
sage: u = var('u')
sage: D8.profile_series(u).parent()
Symbolic Ring

profile_series

Return the (finite) generating series of the (finite) profile of the group.

The profile of a permutation group G is the counting function that maps each nonnegative integer n onto the number of orbits of the action induced by G on the n-subsets of its domain. If f is the profile of G, f(n) is thus the number of orbits of n-subsets of G.

INPUT:
- variable – a variable, or variable name as a string (default: ’z’)

OUTPUT:
- A polynomial in variable with nonnegative integer coefficients. By default, a polynomial in z over ZZ.

See also:
- profile

EXAMPLES:

sage: # needs sage.combinat
sage: C8 = CyclicPermutationGroup(8)
sage: C8.profile_series()
z^8 + z^7 + 4*z^6 + 7*z^5 + 10*z^4 + 7*z^3 + 4*z^2 + z + 1
sage: D8 = DihedralGroup(8)
sage: poly_D8 = D8.profile_series()
sage: poly_D8
z^8 + z^7 + 4*z^6 + 5*z^5 + 8*z^4 + 5*z^3 + 4*z^2 + z + 1
sage: poly_D8.parent()
Univariate Polynomial Ring in z over Rational Field
sage: D8.profile_series(variable='y')
y^8 + y^7 + 4*y^6 + 5*y^5 + 8*y^4 + 5*y^3 + 4*y^2 + y + 1
sage: u = var('u')
sage: D8.profile_series(u).parent()
Symbolic Ring

```python
>>> from sage.all import *
>>> # needs sage.combinat
>>> C8 = CyclicPermutationGroup(Integer(8))
>>> C8.profile_series()
z^8 + z^7 + 4*z^6 + 7*z^5 + 10*z^4 + 7*z^3 + 4*z^2 + z + 1
>>> D8 = DihedralGroup(Integer(8))
>>> poly_D8 = D8.profile_series()
>>> poly_D8
z^8 + z^7 + 4*z^6 + 5*z^5 + 8*z^4 + 5*z^3 + 4*z^2 + z + 1
>>> poly_D8.parent()
Univariate Polynomial Ring in z over Rational Field
>>> D8.profile_series(variable='y')
y^8 + y^7 + 4*y^6 + 5*y^5 + 8*y^4 + 5*y^3 + 4*y^2 + y + 1
>>> u = var('u')
>>> D8.profile_series(u).parent()
Symbolic Ring
```

```python
example()

Returns an example of finite permutation group, as per \texttt{Category.example()}.

EXAMPLES:

```python
sage: G = FinitePermutationGroups().example(); G
Dihedral group of order 6 as a permutation group
```

```python
extra_super_categories()

Any permutation group is assumed to be endowed with a finite set of generators.

\section{4.62 Finite posets}

Here is some terminology used in this file:

- An order filter (or upper set) of a poset \( P \) is a subset \( S \) of \( P \) such that if \( x \leq y \) and \( x \in S \) then \( y \in S \).
- An order ideal (or lower set) of a poset \( P \) is a subset \( S \) of \( P \) such that if \( x \leq y \) and \( y \in S \) then \( x \in S \).

```

class sage.categories.finite_posets.FinitePosets(base_category)

Bases: \texttt{CategoryWithAxiom}

The category of finite posets i.e. finite sets with a partial order structure.

EXAMPLES:
sage: FinitePosets()
Category of finite posets
sage: FinitePosets().super_categories()
[Category of posets, Category of finite sets]
sage: FinitePosets().example()
NotImplemented

See also:
Posets, Poset()

class ParentMethods

Bases: object

antichains()

Return all antichains of self.

EXAMPLES:

sage: A = posets.PentagonPoset().antichains(); A
Set of antichains of Finite lattice containing 5 elements
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]

birational_free_labelling(linear_extension=None, prefix='x', base_field=None, reduced=False, addvars=None, labels=None, min_label=None, max_label=None)

Return the birational free labelling of self.

Let us hold back defining this, and introduce birational toggles and birational rowmotion first. These notions have been introduced in [EP2013] as generalizations of the notions of toggles (order_ideal_toggle()) and rowmotion on order ideals of a finite poset. They have been studied further in [GR2013].

Let $\mathbf{K}$ be a field, and $P$ be a finite poset. Let $\widehat{P}$ denote the poset obtained from $P$ by adding a new element 1 which is greater than all existing elements of $P$, and a new element 0 which is smaller than all existing elements of $P$ and 1. Now, a $\mathbf{K}$-labelling of $P$ will mean any function from $\widehat{P}$ to $\mathbf{K}$. The image of an element $v$ of $\widehat{P}$ under this labelling will be called the label of this labelling at $v$. The set of all $\mathbf{K}$-labellings of $P$ is clearly $\mathbf{K}^P$.
For any \( v \in P \), we now define a rational map \( T_v : \mathbf{K}^\hat{P} \rightarrow \mathbf{K}^\hat{P} \) as follows: For every \( f \in \mathbf{K}^\hat{P} \), the image \( T_v f \) should send every element \( u \in P \) distinct from \( v \) to \( f(u) \) (so the labels at all \( u \neq v \) don’t change), while \( v \) is sent to

\[
\frac{1}{f(v)} \sum_{u \leq v} \frac{f(u)}{f(u)}
\]

(both sums are over all \( u \in \hat{P} \) satisfying the respectively given conditions). Here, \(< \) and \( > \) mean (respectively) “covered by” and “covers”, interpreted with respect to the poset \( \hat{P} \). This rational map \( T_v \) is an involution and is called the \textit{(birational) }\textit{v-toggle}; see \texttt{birational\_toggle() }for its implementation.

Now, \textit{birational rowmotion} is defined as the composition \( T_{v_1} \circ T_{v_2} \circ \cdots \circ T_{v_n} \), where \( (v_1, v_2, \ldots, v_n) \) is a linear extension of \( P \) (written as a linear ordering of the elements of \( P \)). This is a rational map \( \mathbf{K}^P \rightarrow \mathbf{K}^P \) which does not depend on the choice of the linear extension; it is denoted by \( R \). See \texttt{birational\_rowmotion() }for its implementation.

The definitions of birational toggles and birational rowmotion extend to the case of \( \mathbf{K} \) being any semifield rather than necessarily a field (although it becomes less clear what constitutes a rational map in this generality). The most useful case is that of the \textit{tropical semiring}, in which case birational rowmotion relates to classical constructions such as promotion of rectangular semistandard Young tableaux (page 5 of [EP2013b] and future work, via the related notion of birational \textit{promotion} and rowmotion on order ideals of the poset ([EP2013]).

The \textit{birational free labelling} is a special labelling defined for every finite poset \( P \) and every linear extension \( (v_1, v_2, \ldots, v_n) \) of \( P \). It is given by sending every element \( v_i \) in \( P \) to \( x_i \), sending the element 0 of \( \hat{P} \) to \( a \), and sending the element 1 of \( \hat{P} \) to \( b \), where the ground field \( \mathbf{K} \) is the field of rational functions in \( n+2 \) indeterminates \( a, x_1, x_2, \ldots, x_n, b \) over \( \mathbb{Q} \).

In Sage, a labelling \( f \) of a poset \( P \) is encoded as a 4-tuple \((\mathbf{K}, d, u, v)\), where \( \mathbf{K} \) is the ground field of the labelling (i.e., its target), \( d \) is the dictionary containing the values of \( f \) at the elements of \( P \) (the keys being the respective elements of \( P \)), \( u \) is the label of \( f \) at 0, and \( v \) is the label of \( f \) at 1.

**Warning:** The dictionary \( d \) is labelled by the elements of \( P \). If \( P \) is a poset with \texttt{facade} option set to \texttt{False}, these might not be what they seem to be! (For instance, if \( P == \text{Poset}([1: [2, 3]], \text{facade}=\text{False}) \), then the value of \( d \) at 1 has to be accessed by \( d[P(1)] \), not by \( d[1] \).)

**Warning:** Dictionaries are mutable. They do compare correctly, but are not hashable and need to be cloned to avoid spooky action at a distance. Be careful!

**INPUT:**

- \texttt{linear\_extension=(default: the default linear extension of self) }a linear extension of \texttt{self} (as a linear extension or as a list), or more generally a list of all elements of all elements of \texttt{self} each occurring exactly once
- \texttt{prefix=(default: ‘x’) }the prefix to name the indeterminates corresponding to the elements of \texttt{self} in the labelling (so, setting it to ‘frog’ will result in these indeterminates being called \texttt{frog1, frog2, ..., frogn} rather than \texttt{x1, x2, ..., xn}).
- \texttt{base\_field=(default: QQ) }the base field to be used instead of \( \mathbf{Q} \) to define the rational function field over; this is not going to be the base field of the labelling, because the latter will have indeterminates adjoined!
- \texttt{reduced=(default: False) }if set to \texttt{True}, the result will be the \texttt{reduced} birational free labelling, which differs from the regular one by having 0 and 1 both sent to 1 instead of \( a \) and \( b \) (the indeterminates \( a \) and \( b \) then also won’t appear in the ground field)
• addvars – (default: '') a string containing names of extra variables to be adjoined to the ground field (these don’t have an effect on the labels)
• labels – (default: 'x') Either a function that takes an element of the poset and returns a name for the indeterminate corresponding to that element, or a string containing a comma-separated list of indeterminates that will be assigned to elements in the order of linear_extension. If the list contains more indeterminates than needed, the excess will be ignored. If it contains too few, then the needed indeterminates will be constructed from prefix.
• min_label – (default: 'a') a string to be used as the label for the element 0 of \( \hat{P} \)
• max_label – (default: 'b') a string to be used as the label for the element 1 of \( \hat{P} \)

OUTPUT:
The birational free labelling of the poset self and the linear extension linear_extension. Or, if reduced is set to True, the reduced birational free labelling.

EXAMPLES:
We construct the birational free labelling on a simple poset:

```
sage: P = Poset({1: [2, 3]})
sage: l = P.birational_free_labelling(); l
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b over Rational Field, {...}, a, b)
sage: sorted(l[1].items())
[(1, x1), (2, x2), (3, x3)]
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2]); l
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b over Rational Field, {...}, a, b)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], reduced=True, addvars="spam, eggs"); 1
(Fraction Field of Multivariate Polynomial Ring in x1, x2, x3, spam, eggs over Rational Field, {...}, 1, 1)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], prefix="wut", reduced=True, addvars="spam, eggs"); l
(Fraction Field of Multivariate Polynomial Ring in wut1, wut2, wut3, spam, eggs over Rational Field, {...}, 1, 1)
sage: sorted(l[1].items())
[(1, wut1), (2, wut3), (3, wut2)]
```
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2],
....: reduced=False, addvars="spam, eggs")
1 (Fraction Field of Multivariate Polynomial Ring
  in a, x1, x2, x3, b, spam, eggs over Rational Field,
  \{...\},
  a,
  b)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]
sage: l[1][2]
x3

>>> from sage.all import *

>>> P = Poset({1: [2, 3]})

>>> l = P.birational_free_labelling(); l
1 (Fraction Field of Multivariate Polynomial Ring
  in a, x1, x2, x3, b over Rational Field,
  \{...\},
  a,
  b)

>>> sorted(l[1].items())
[(1, x1), (2, x2), (3, x3)]

>>> l = P.birational_free_labelling(linear_extension=[1, 3, 2],
....: reduced=True, addvars="spam, eggs")
1 (Fraction Field of Multivariate Polynomial Ring
  in x1, x2, x3, spam, eggs over Rational Field,
  \{...\},
  1,
  1)

>>> sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]

>>> l = P.birational_free_labelling(linear_extension=[1, 3, 2],
....: prefix="wut", reduced=True,
....: addvars="spam, eggs")
1 (Fraction Field of Multivariate Polynomial Ring
  in wut1, wut2, wut3, spam, eggs over Rational Field,
  \{...\},
  1,
  1)

>>> sorted(l[1].items())
Illustrating labelling with a function:

```python
sage: P = posets.ChainPoset(2).product(posets.ChainPoset(2))  # needs sage.modules
sage: def x_label(e):
    ... return 'x_' + str(e[0]) + str(e[1])

sage: l = P.birational_free_labelling(labels=x_label)
sage: sorted(l[1].items())
[((0, 0), x_00), ((0, 1), x_01), ((1, 0), x_10), ((1, 1), x_11)]
sage: l[2]
lambda
sage: l[3]
mu
```

The same, but with `min_label` and `max_label` provided:

```python
sage: P = posets.ChainPoset(2).product(posets.ChainPoset(2))  # needs sage.modules
sage: l = P.birational_free_labelling(labels=x_label,
    ...: min_label="lambda", max_label="mu")

sage: sorted(l[1].items())
[((0, 0), x_00), ((0, 1), x_01), ((1, 0), x_10), ((1, 1), x_11)]
```

(continues on next page)
Illustrating labelling with a comma separated list of labels:

```sage
sage: l = P.birational_free_labelling(labels='w,x,y,z')
sage: sorted(l[1].items())
[((0, 0), w), ((0, 1), x), ((1, 0), y), ((1, 1), z)]
```

Illustrating the warning about facade:

```sage
sage: P = Poset({1: [2, 3]}, facade=False)
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2],
      reduced=False,
      addvars="spam, eggs"); l
(Fraction Field of Multivariate Polynomial Ring
  in a, x1, x2, x3, b, spam, eggs over Rational Field,
  \{\ldots\},
a, b)
sage: l[1][2]
Traceback (most recent call last):
  ... KeyErrors: 2
sage: l[1][P(2)]
x3
```
Another poset:

```python
sage: # needs sage.modules
sage: P = posets.SSTPoset([2,1])
sage: lext = sorted(P)
sage: l = P.birational_free_labelling(linear_extension=lext,
   ...: addvars="ohai"); l
(Fraction Field of Multivariate Polynomial Ring
   in a, x1, x2, x3, x4, x5, x6, x7, x8, b, ohai over Rational Field,
   {...},
   a,
   b)
sage: sorted(l[1].items())
[['([1, 1], [2]), x1], '([1, 1], [3]), x2),
  ('([1, 2], [2]), x3), '([1, 2], [3]), x4),
  ('([1, 3], [2]), x5), '([1, 3], [3]), x6),
  ('([2, 2], [3]), x7), '([2, 3], [3]), x8]]
```

See `birational_rowmotion()`, `birational_toggle()` and `birational_toggles()` for more substantial examples of what one can do with the birational free labelling.

**birational_rowmotion** (labelling)

Return the result of applying birational rowmotion to the \(K\)-labelling `labelling` of the poset `self`.

See the documentation of `birational_free_labelling()` for a definition of birational rowmotion and \(K\)-labellings and for an explanation of how \(K\)-labellings are to be encoded to be understood by Sage. This implementation allows \(K\) to be a semifield, not just a field. Birational rowmotion is only a rational map, so an exception (most likely, `ZeroDivisionError`) will be thrown if the denominator is zero.
INPUT:
• labelling – a \( K \)-labelling of self in the sense as defined in the documentation of \texttt{birational_free_labelling()}

OUTPUT:
The image of the \( K \)-labelling \( f \) under birational rowmotion.

EXAMPLES:

```sage
P = Poset({1: [2, 3], 2: [4], 3: [4]})
lex = [1, 2, 3, 4]
t = P.birational_free_labelling(linear_extension=lex)
print(t)
```

```sage
sage: P = Poset({1: [2, 3], 2: [4], 3: [4]})
sage: lex = [1, 2, 3, 4]
sage: t = P.birational_free_labelling(linear_extension=lex)
sage: sorted(t[1].items())
```

```sage
P = Poset({1: [2, 3], 2: [4], 3: [4]})
lex = [1, 2, 3, 4]
t = P.birational_free_labelling(linear_extension=lex)
print(t)
```

```sage
sage: P = Poset({1: [2, 3], 2: [4], 3: [4]})
sage: lex = [1, 2, 3, 4]
sage: t = P.birational_rowmotion(t)
sage: sorted(t[1].items())
```

A result of [GR2013] states that applying birational rowmotion \( n + m \) times to a \( K \)-labelling \( f \) of the poset \([n] \times [m]\) gives back \( f \). Let us check this:

```sage
def test_rectangle_periodicity(n, m, k):
    P = posets.ChainPoset(n).product(posets.ChainPoset(m))
t0 = P.birational_free_labelling(P)
t = t0
```

(continues on next page)
...:     for i in range(k):
...:         t = P.birational_rowmotion(t)
...:     return t == t0
sage: test_rectangle_periodicity(2, 2, 4)  # needs sage.modules
True
sage: test_rectangle_periodicity(2, 2, 2)  # needs sage.modules
False
sage: test_rectangle_periodicity(2, 3, 5)  # long time  # needs sage.modules
True

While computations with the birational free labelling quickly run out of memory due to the complexity of the rational functions involved, it is computationally cheap to check properties of birational rowmotion on examples in the tropical semiring:

sage: def test_rectangle_periodicity_tropical(n, m, k):
...:     P = posets.ChainPoset(n).product(posets.ChainPoset(m))
...:     TT = TropicalSemiring(ZZ)
...:     t0 = (TT, {v: TT(randint(0, 99)) for v in P}, TT(0), TT(124))
...:     t = t0
...:     for i in range(k):
...:         t = P.birational_rowmotion(t)
...:     return t == t0
sage: test_rectangle_periodicity_tropical(7, 6, 13)  # needs sage.modules
True

>>> from sage.all import *
>>> def test_rectangle_periodicity_tropical(n, m, k):
...     P = posets.ChainPoset(n).product(posets.ChainPoset(m))
...     TT = TropicalSemiring(ZZ)
...     t0 = (TT, {v: TT(randint(Integer(0), Integer(99))) for v in P},
...     TT(Integer(0)), TT(Integer(124)))
...     t = t0
...     for i in range(k):
...         t = P.birational_rowmotion(t)
...     return t == t0

Tropicalization is also what relates birational rowmotion to classical rowmotion on order ideals. In fact, if $T$ denotes the tropical semiring of $\mathbb{Z}$ and $P$ is a finite poset, then we can define an embedding $\phi$ from the set $J(P)$ of all order ideals of $P$ into the set $T^P$ of all $T$-labellings of $P$ by sending every $I \in J(P)$ to the indicator function of $I$ extended by the value 1 at the element 0 and the value 0 at the element 1. This map $\phi$ has the property that $R\circ\phi = \phi\circ r$, where $R$ denotes birational rowmotion, and $r$ denotes classical rowmotion on $J(P)$. An example:

```
sage: P = posets.IntegerPartitions(5)
sage: TT = TropicalSemiring(ZZ)
sage: def indicator_labelling(I):
...    # send order ideal AIA to a ATA-labelling of APA.
...    dct = {v: TT(v in I) for v in P}
...    return (TT, dct, TT(Integer(1)), TT(Integer(0))

sage: all(indicator_labelling(P.rowmotion(I))
...    == P.birational_rowmotion(indicator_labelling(I))
...    for I in P.order_ideals_lattice(facade=True))
True
```

The result of applying the birational $v$-toggle $T_v$ to the $K$-labelling $\phi$ of the poset $P$.

See the documentation of `birational_free_labelling()` for a definition of this toggle and of $K$-labellings as well as an explanation of how $K$-labellings are to be encoded to be understood by Sage. This implementation allows $K$ to be a semifield, not just a field. The birational $v$-toggle is only a rational map, so an exception (most likely, `ZeroDivisionError`) will be thrown if the denominator is zero.

**INPUT:**
- $v$ - an element of $P$.
- $\phi$ - a $K$-labelling of $P$.

**OUTPUT:**
The $K$-labelling $T_v\phi$ of $P$. 

**EXAMPLES:**
Let us start with the birational free labelling of the “V”-poset (the three-element poset with Hasse diagram...
looking like a “V”):

```
sage: V = Poset({1: [2, 3]})
sage: s = V.birational_free_labelling(); s
(Fraction Field of Multivariate Polynomial Ring
in a, x1, x2, x3, b over Rational Field,
{...},
a, b)
sage: sorted(s[1].items())
[(1, x1), (2, x2), (3, x3)]
```

```
>>> from sage.all import *

>>> V = Poset({Integer(1): [Integer(2), Integer(3)]})

>>> s = V.birational_free_labelling(); s
(Fraction Field of Multivariate Polynomial Ring
in a, x1, x2, x3, b over Rational Field,
{...},
a, b)

>>> sorted(s[Integer(1)].items())
[(1, x1), (2, x2), (3, x3)]
```

The image of $s$ under the 1-toggle $T_1$ is:

```
sage: s1 = V.birational_toggle(1, s); s1
(Fraction Field of Multivariate Polynomial Ring
in a, x1, x2, x3, b over Rational Field,
{...},
a, b)
sage: sorted(s1[1].items())
[(1, a*x2*x3/(x1*x2 + x1*x3)), (2, x2), (3, x3)]
```

```
>>> from sage.all import *

>>> s1 = V.birational_toggle(Integer(1), s); s1
(Fraction Field of Multivariate Polynomial Ring
in a, x1, x2, x3, b over Rational Field,
{...},
a, b)

>>> sorted(s1[Integer(1)].items())
[(1, a*x2*x3/(x1*x2 + x1*x3)), (2, x2), (3, x3)]
```

Now let us apply the 2-toggle $T_2$ (to the old $s$):

```
sage: s2 = V.birational_toggle(2, s); s2
(Fraction Field of Multivariate Polynomial Ring
in a, x1, x2, x3, b over Rational Field,
{...},
a, b)
sage: sorted(s2[1].items())
[(1, x1), (2, x1*b/x2), (3, x3)]
```

```
>>> from sage.all import *

>>> s2 = V.birational_toggle(Integer(2), s); s2
```

(continues on next page)
On the other hand, we can also apply $T_2$ to the image of $s$ under $T_1$:

```sage
sage: s12 = V.birational_toggle(2, s1); s12
(Fraction Field of Multivariate Polynomial Ring
 in a, x1, x2, x3, b over Rational Field,
 {...},
a,
b)
sage: sorted(s12[1].items())
[[(1, a*x2*x3/(x1*x2 + x1*x3)), (2, a*x3*b/(x1*x2 + x1*x3)), (3, x3)]
```

Each toggle is an involution:

```sage
sage: all( V.birational_toggle(i, V.birational_toggle(i, s)) == s
       ...: for i in V )
True
```

We can also start with a less generic labelling:

```sage
sage: t = (QQ, {1: 3, 2: 6, 3: 7}, 2, 10)
sage: t1 = V.birational_toggle(1, t); t1
(Rational Field, {...}, 2, 10)
sage: sorted(t1[1].items())
[(1, 28/13), (2, 6), (3, 7)]
sage: t13 = V.birational_toggle(3, t1); t13
(Rational Field, {...}, 2, 10)
sage: sorted(t13[1].items())
[(1, 28/13), (2, 6), (3, 40/13)]
```

>>> from sage.all import *
>>> t = (QQ, {Integer(1): Integer(3), Integer(2): Integer(6), Integer(3):
˓→Integer(7)}, Integer(2), Integer(10))
However, labellings have to be sufficiently generic, lest denominators vanish:

```
sage: t = (QQ, {1: 3, 2: 5, 3: -5}, 1, 15)
sage: t1 = V.birational_toggle(1, t)
Traceback (most recent call last):
  ... 
ZeroDivisionError: rational division by zero
```

```
>>> from sage.all import *
```

```
sage: t1 = V.birational_toggle(Integer(1), t)
Traceback (most recent call last):
  ... 
ZeroDivisionError: rational division by zero
```

We don’t get into zero-division issues in the tropical semiring (unless the zero of the tropical semiring appears in the labelling):

```
sage: TT = TropicalSemiring(QQ)
sage: t = (TT, {1: TT(Integer(2)), 2: TT(4), 3: TT(1)}, TT(6), TT(0))
sage: t1 = V.birational_toggle(1, t); t1
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t1[1].items())
[(1, 8), (2, 4), (3, 1)]
sage: t12 = V.birational_toggle(2, t1); t12
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t12[1].items())
[(1, 8), (2, 4), (3, 1)]
sage: t123 = V.birational_toggle(3, t12); t123
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t123[1].items())
[(1, 8), (2, 4), (3, 7)]
```

```
>>> from sage.all import *
```

```
sage: TT = TropicalSemiring(QQ)
sage: t = (TT, {Integer(1): TT(Integer(2)), Integer(2): TT(Integer(4)), Integer(3): TT(Integer(1))}, TT(6), TT(0))
sage: t1 = V.birational_toggle(Integer(1), t); t1
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t1[Integer(1)].items())
[(1, 8), (2, 4), (3, 1)]
sage: t12 = V.birational_toggle(Integer(2), t1); t12
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t12[Integer(1)].items())
[(1, 8), (2, 4), (3, 1)]
sage: t123 = V.birational_toggle(Integer(3), t12); t123
```

(continues on next page)
We turn to more interesting posets. Here is the 6-element poset arising from the weak order on $S_3$:

```python
>>> sorted(t123[1].items())
[(1, 8), (2, 4), (3, 7)]
```

Let us verify on this example some basic properties of toggles. First of all, again let us check that $T_v$ is an involution for every $v$:

```python
sage: all( P.birational_toggle(v, P.birational_toggle(v, t)) == t
....: for v in P )
True
```

Furthermore, two toggles $T_v$ and $T_w$ commute unless one of $v$ or $w$ covers the other:

```python
sage: all( P.covers(v, w) or P.covers(w, v)
....: or P.birational_toggle(v, P.birational_toggle(w, t))
....: == P.birational_toggle(w, P.birational_toggle(v, t))
....: for v in P for w in P )
True
```
>>> from sage.all import *

>>> all( P.covers(v, w) 
... or P.covers(w, v) 
... or 
... P.birational_toggle(v, P.birational_toggle(w, t)) 
... == P.birational_toggle(w, P.birational_toggle(v, t)) 
... for v in P for w in P )
True

birational_toggles((vs, labelling))

Return the result of applying a sequence of birational toggles (specified by vs) to the K-labelling labelling of the poset self.

See the documentation of birational_free_labelling() for a definition of birational toggles and K-labellings and for an explanation of how K-labellings are to be encoded to be understood by Sage. This implementation allows K to be a semifield, not just a field. The birational v-toggle is only a rational map, so an exception (most likely, ZeroDivisionError) will be thrown if the denominator is zero.

INPUT:

- vs – an iterable comprising elements of self (which must have self as parent if self is a facade=False poset)
- labelling – a K-labelling of self in the sense as defined in the documentation of birational_free_labelling()

OUTPUT:

The K-labelling \( T_{v_n} T_{v_{n-1}} \cdots T_{v_1} f \) of self, where \( f \) is labelling and \((v_1, v_2, \ldots, v_n)\) is vs (written as list).

EXAMPLES:

sage: P = posets.SymmetricGroupBruhatOrderPoset(3)
sage: sorted(list(P))
[123, 132, 213, 231, 312, 321]
sage: TT = TropicalSemiring(ZZ)
sage: t = (TT, {123: TT(4), 132: TT(2), 213: TT(3), 
...: 231: TT(1), 312: TT(1), 321: TT(2)}, TT(7), TT(1))
sage: tA = P.birational_toggles([123, 231, 312], t); tA
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tA[1].items())
[(123, 6), (132, 2), (213, 3), (231, 2), (312, 1), (321, 1)]
sage: tAB = P.birational_toggles([132, 213, 321], tA); tAB
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tAB[1].items())
[(123, 6), (132, 6), (213, 5), (231, 2), (312, 1), (321, 1)]

sage: P = Poset({1: [2, 3], 2: [4], 3: [4]})
sage: Qx = PolynomialRing(QQ, 'x').fraction_field()
sage: x = Qx.gen()
sage: t = (Qx, {1: 1, 2: x, 3: (x+1)/x, 4: x^2}, 1, 1)
sage: t1 = P.birational_toggles((i for i in range(1, 5)), t); t1
(Fraction Field of Univariate Polynomial Ring in x over Rational Field, {...}, 1, 1)
sage: sorted(t1[1].items())
[(1, (x+1)/(x^2 + x + 1)), 
(2, (x^3 + x^2)/(x^2 + x + 1)), 
(3, x^4/(x^2 + x + 1)), (4, 1)]
sage: t2 = P.birational_toggles(reversed(range(1, 5)), t)
sage: sorted(t2[1].items())
(continues on next page)
from sage.all import *

P = posets.SymmetricGroupBruhatOrderPoset(Integer(3))

sorted(list(P))

TT = TropicalSemiring(ZZ)
t = (TT, {'123': TT(Integer(4)), '132': TT(Integer(2)), '213':
    TT(Integer(3)),
    '231': TT(Integer(1)), '321': TT(Integer(1)), '312':
    TT(Integer(2)), TT(Integer(7)), TT(Integer(1)))
tA = P.birational_toggles(['123', '132', '312'], t); tA
(Tropical semiring over Integer Ring, {...}, 7, 1)

sorted(tA[Integer(1)].items())
[(123, 6), (132, 2), (213, 3), (231, 2), (312, 1), (321, 1)]

tAB = P.birational_toggles(['132', '213', '321'], tA); tAB
(Tropical semiring over Integer Ring, {...}, 7, 1)

sorted(tAB[Integer(1)].items())
[(123, 6), (132, 6), (213, 5), (231, 2), (312, 1), (321, 1)]

P = Poset({Integer(1): [Integer(2), Integer(3)], Integer(2):
    [Integer(4)], Integer(3): [Integer(4)]})

Qx = PolynomialRing(QQ, 'x').fraction_field()
x = Qx.gen()
t = (Qx, {Integer(1): Integer(1), Integer(2): x, Integer(3):
    x**Integer(2), Integer(4): Integer(1), Integer(1)})
t1 = P.birational_toggles((i for i in range(Integer(1), Integer(5))),...
    t); t1
(Fraction Field of Univariate Polynomial Ring in x over Rational Field,
    {...}, 1, 1)

sorted(t1[Integer(1)].items())
[(1, (x^2 + x)/(x^2 + x + 1)),
(2, (x^3 + x^2)/(x^2 + x + 1)),
(3, x^4/(x^2 + x + 1)), (4, 1)]

t2 = P.birational_toggles(reversed(range(Integer(1), Integer(5))), t)

sorted(t2[Integer(1)].items())
[(1, 1/x^2), (2, (x^2 + x + 1)/x^4), (3, (x^2 + x + 1)/(x^3 + x^2)),
(4, (x^2 + x + 1)/x^3)]
directed_subsets (direction)

Return the order filters (resp. order ideals) of self, as lists.

If direction is ‘up’, returns the order filters (upper sets).
If direction is ‘down’, returns the order ideals (lower sets).

INPUT:
  - direction – ‘up’ or ‘down’

EXAMPLES:

```python
sage: P = Poset((divisors(12), attrcall("divides")), facade=True)
sage: A = P.directed_subsets('up')  # needs sage.modules
sage: sorted(list(A))

[[], [1, 2, 4, 3, 6, 12], [2, 4, 3, 6, 12], [2, 4, 6, 12], [3, 6, 12],
 [4, 3, 6, 12], [4, 6, 12], [4, 12], [6, 12], [12]]
```

is_lattice()

Return whether the poset is a lattice.

A poset is a lattice if all pairs of elements have both a least upper bound (“join”) and a greatest lower bound (“meet”) in the poset.

EXAMPLES:

```python
sage: P = Poset([[1, 3, 2], [4], [4, 5, 6], [6], [7], [7], [7], []])
sage: P.is_lattice()  # needs sage.modules
True
sage: P = Poset([[1, 2], [3], [3], []])
sage: P.is_lattice()  # needs sage.modules
True
sage: P = Poset({0: [2, 3], 1: [2, 3]})
sage: P.is_lattice()  # needs sage.modules
False
sage: P = Poset({1: [2, 3, 4], 2: [5, 6], 3: [5, 7], 4: [6, 7], 5: [8, 9],
 ... 6: [8, 10], 7: [9, 10], 8: [11], 9: [11], 10: [11]})
sage: P.is_lattice()  # needs sage.modules
False
```
>>> from sage.all import *

>>> P = Poset([[Integer(1), Integer(3), Integer(2)], [Integer(4)],
...
<Integer(7)], [Integer(7)], [Integer(7)], []])
>>> P.is_lattice()    # needs sage.modules
True

>>> P = Poset([[Integer(1), Integer(2)], [Integer(3)], [Integer(3)], []])
>>> P.is_lattice()    # needs sage.modules
True

>>> P = Poset({Integer(0): [Integer(2), Integer(3)], Integer(1):
...
<Integer(2), Integer(3)]})
>>> P.is_lattice()    # needs sage.modules
False

>>> P = Poset({Integer(1): [Integer(2), Integer(3), Integer(4)],
...
<Integer(2): [Integer(5), Integer(6)], Integer(3): [Integer(5),
...
<Integer(7)], Integer(4): [Integer(6), Integer(7)], Integer(5):
...
<Integer(8), Integer(9)], ...
<Integer(6): [Integer(8), Integer(10)], Integer(7):
...
<Integer(9), Integer(10)], Integer(8): [Integer(11)], Integer(9):
...
<Integer(11), Integer(10): [Integer(11)]})
>>> P.is_lattice()    # needs sage.modules
False

See also:

- Weaker properties: is_join_semilattice(), is_meet_semilattice()

**is_poset_isomorphism**(f, codomain)

Return whether \( f \) is an isomorphism of posets from \( \text{self} \) to \( \text{codomain} \).

**INPUT:**

- \( f \) – a function from \( \text{self} \) to \( \text{codomain} \)
- \( \text{codomain} \) – a poset

**EXAMPLES:**

We build the poset \( D \) of divisors of 30, and check that it is isomorphic to the boolean lattice \( B \) of the subsets of \( \{2, 3, 5\} \) ordered by inclusion, via the reverse function \( f : B \to D, b \mapsto \prod_{x \in b} x \):

```python
code:
sage: D = Poset((divisors(30), attrcall("divides")))
sage: B = Poset((\[frozenset(s) for s in Subsets([2,3,5])\],
...
attrcall("issubset")))
sage: def f(b):
    return D(prod(b))
sage: B.is_poset_isomorphism(f, D)
True
```

(continues on next page)
On the other hand, \( f \) is not an isomorphism to the chain of divisors of 30, ordered by usual comparison:

```python
sage: P = Poset((divisors(30), operator.le))
sage: def f(b):
    return P(prod(b))
sage: B.is_poset_isomorphism(f, P)
False
```

A non surjective case:

```python
sage: B = Poset([[frozenset(s) for s in Subsets([2,3])],
              ...: attrcall("issubset")])
sage: def f(b):
    return D(prod(b))
sage: B.is_poset_isomorphism(f, D)
False
```

A non injective case:

```python
sage: B = Poset([[frozenset(s) for s in Subsets([2,3,5,6])],
              ...: attrcall("issubset")])
sage: def f(b):
    return D(gcd(prod(b), 30))
sage: B.is_poset_isomorphism(f, D)
False
```

Note: since \( D \) and \( B \) are not facade posets, \( f \) is responsible for the conversions between integers and subsets to elements of \( D \) and \( B \) and back.

See also:

\( \text{FiniteLatticePosets.ParentMethods.is_lattice_morphism()} \)
**is_poset_morphism** \((f, \text{codomain})\)

Return whether \(f\) is a morphism of posets from \(\text{self}\) to \(\text{codomain}\), that is

\[ x \leq y \implies f(x) \leq f(y) \]

for all \(x\) and \(y\) in \(\text{self}\).

**INPUT:**
- \(f\) – a function from \(\text{self}\) to \(\text{codomain}\)
- \(\text{codomain}\) – a poset

**EXAMPLES:**

We build the boolean lattice of the subsets of \(\{2, 3, 5, 6\}\) and the lattice of divisors of 30, and check that the map \(b \mapsto \gcd(\prod_{x \in b} x, 30)\) is a morphism of posets:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: B = Poset(([frozenset(s) for s in Subsets([2,3,5,6])], ....: attrcall("issubset")))
sage: def f(b): return D(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, D)
True
```

```from sage.all import *```  
```D = Poset((divisors(Integer(30)), attrcall("divides")))```  
```B = Poset(([frozenset(s) for s in Subsets([Integer(2),Integer(3), Integer(5),Integer(6)])], ....: attrcall("issubset")))```  
```def f(b): return D(gcd(prod(b), Integer(30)))```  
```B.is_poset_morphism(f, D)```  
```True```  

**Note:** since \(D\) and \(B\) are not facade posets, \(f\) is responsible for the conversions between integers and subsets to elements of \(D\) and \(B\) and back.

\(f\) is also a morphism of posets to the chain of divisors of 30, ordered by usual comparison:

```
sage: P = Poset((divisors(30), operator.le))
sage: def f(b): return P(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, P)
True
```

```from sage.all import *```  
```P = Poset((divisors(Integer(30)), operator.le))```  
```def f(b): return P(gcd(prod(b), Integer(30)))```  
```B.is_poset_morphism(f, P)```  
```True```  

**FIXME:** should this be \(\text{is\_order\_preserving\_morphism}\)?

**See also:**

- \(\text{is_poset_isomorphism()}\)
- \(\text{is_self\_dual()}\)

Return whether the poset is \(\text{self\_dual}\).

A poset is self-dual if it is isomorphic to its dual poset.
EXAMPLES:

```
sage: P = Poset({1: [3, 4], 2: [3, 4]})
sage: P.is_self_dual()
True

sage: P = Poset({1: [2, 3]})
sage: P.is_self_dual()
False
```

```>> from sage.all import *
>> P = Poset({Integer(1): [Integer(3), Integer(4)], Integer(2):
˓→[Integer(3), Integer(4)]})
>> P.is_self_dual()
True

>> P = Poset({Integer(1): [Integer(2), Integer(3)]})
>> P.is_self_dual()
False
``` 

See also:

- Stronger properties: `is_orthocomplemented()` (for lattices)
- Other: `dual()`

**order_filter_generators** *(filter)*
Generators for an order filter

**INPUT:**
- `filter` – an order filter of `self`, as a list (or iterable)

**EXAMPLES:**

```
sage: P = Poset((Subsets([1,2,3]), attrcall("issubset")))
sage: I = P.order_filter([Set([1,2]), Set([2,3]), Set([1])])
sage: sorted(sorted(p) for p in I)
[[1], [1, 2], [1, 2, 3], [1, 3], [2, 3]]
sage: gen = P.order_filter_generators(I)
sage: sorted(sorted(p) for p in gen)
[[1], [2, 3]]
```

```>> from sage.all import *
>> P = Poset((Subsets([Integer(1),Integer(2),Integer(3)]), attrcall("issubset")))
>> I = P.order_filter([Set([Integer(1),Integer(2)]), Set([Integer(2),
˓→Integer(3)]), Set([Integer(1)])])
>> sorted(sorted(p) for p in I)
[[1], [1, 2], [1, 2, 3], [1, 3], [2, 3]]
>> gen = P.order_filter_generators(I)
>> sorted(sorted(p) for p in gen)
[[1], [2, 3]]
``` 

See also:

- `order_ideal_generators()`

- `order_ideal_complement_generators` *(antichain, direction='up')*
  Return the Panyushev complement of the antichain `antichain`.
Given an antichain $A$ of a poset $P$, the Panyushev complement of $A$ is defined to be the antichain consisting of the minimal elements of the order filter $B$, where $B$ is the (set-theoretic) complement of the order ideal of $P$ generated by $A$.

Setting the optional keyword variable `direction` to 'down' leads to the inverse Panyushev complement being computed instead of the Panyushev complement. The inverse Panyushev complement of an antichain $A$ is the antichain whose Panyushev complement is $A$. It can be found as the antichain consisting of the maximal elements of the order ideal $C$, where $C$ is the (set-theoretic) complement of the order filter of $P$ generated by $A$.

`panyushev_complement()` is an alias for this method.

Panyushev complementation is related (actually, isomorphic) to rowmotion (`rowmotion()`).

**INPUT:**
- `antichain` – an antichain of `self`, as a list (or iterable), or, more generally, generators of an order ideal (resp. order filter)
- `direction` – 'up' or 'down' (default: 'up')

**OUTPUT:**
- the generating antichain of the complement order filter (resp. order ideal) of the order ideal (resp. order filter) generated by the antichain `antichain`

**EXAMPLES:**

```python
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: P.order_ideal_complement_generators([1])
{2}
sage: P.order_ideal_complement_generators([3])
set()
sage: P.order_ideal_complement_generators([1,2])
{3}
sage: P.order_ideal_complement_generators([1,2,3])
set()

sage: P.order_ideal_complement_generators([1], direction="down")
{2}
sage: P.order_ideal_complement_generators([3], direction="down")
{1, 2}
sage: P.order_ideal_complement_generators([1,2], direction="down")
set()
sage: P.order_ideal_complement_generators([1,2,3], direction="down")
set()
```

```python
>> from sage.all import *
>> P = Poset( ( [Integer(1),Integer(2),Integer(3)], [ [Integer(1),
   \rightarrow Integer(3)], [Integer(2),Integer(3)] ] ) )
>> P.order_ideal_complement_generators([Integer(1)])
{2}
>> P.order_ideal_complement_generators([Integer(3)])
set()
>> P.order_ideal_complement_generators([Integer(1),Integer(2)])
{3}
>> P.order_ideal_complement_generators([Integer(1),Integer(2),
   \rightarrow Integer(3)])
set()

>> P.order_ideal_complement_generators([Integer(1)], direction="down")
{2}
>> P.order_ideal_complement_generators([Integer(3)], direction="down")
```

(continues on next page)
{1, 2}

```python
>>> P.order_ideal_complement_generators([Integer(1), Integer(2)], direction="down")
set()
```

```python
>>> P.order_ideal_complement_generators([Integer(1), Integer(2), Integer(3)], direction="down")
set()
```

**Warning:** This is a brute force implementation, building the order ideal generated by the antichain, and searching for order filter generators of its complement.

```python
order_ideal_generators (ideal, direction='down')
```

Return the antichain of (minimal) generators of the order ideal (resp. order filter) `ideal`.

**INPUT:**
- `ideal` – an order ideal \( I \) (resp. order filter) of `self`, as a list (or iterable); this should be an order ideal if `direction` is set to 'down', and an order filter if `direction` is set to 'up'.
- `direction` – 'up' or 'down' (default: 'down').

The antichain of (minimal) generators of an order ideal \( I \) in a poset \( P \) is the set of all minimal elements of \( P \). In the case of an order filter, the definition is similar, but with "maximal" used instead of "minimal".

**EXAMPLES:**

We build the boolean lattice of all subsets of \{1, 2, 3\} ordered by inclusion, and compute an order ideal there:

```python
sage: P = Poset((Subsets([1,2,3]), attrcall("issubset")))
sage: I = P.order_ideal([Set([1,2]), Set([2,3]), Set([1])])
sage: sorted(sorted(p) for p in I)
[[], [1], [1, 2], [2], [2, 3], [3]]
```

```python
>>> from sage.all import *
```

```python
P = Poset((Subsets([Integer(1),Integer(2),Integer(3)]), attrcall("issubset")))
```

```python
I = P.order_ideal([Set([Integer(1),Integer(2)]), Set([Integer(2), Integer(3)]), Set([Integer(1)])])
```

```python
>>> sorted(sorted(p) for p in I)
[[], [1], [1, 2], [2], [2, 3], [3]]
```

Then, we retrieve the generators of this ideal:

```python
sage: gen = P.order_ideal_generators(I)
sage: sorted(sorted(p) for p in gen)
[[1, 2], [2, 3]]
```

```python
>>> from sage.all import *
```

```python
gen = P.order_ideal_generators(I)
```

```python
>>> sorted(sorted(p) for p in gen)
[[1, 2], [2, 3]]
```

If `direction` is ‘up’, then this instead computes the minimal generators for an order filter:

```python
sage: I = P.order_filter([Set([1,2]), Set([2,3]), Set([1])])
sage: sorted(sorted(p) for p in I)
```

(continues on next page)
Complexity: $O(n + m)$ where $n$ is the cardinality of $I$, and $m$ the number of upper covers of elements of $I$.

**order_ideals_lattice** *(as_ideals=True, facade=None)*

Return the lattice of order ideals of a poset `self`, ordered by inclusion.

The lattice of order ideals of a poset $P$ is usually denoted by $J(P)$. Its underlying set is the set of order ideals of $P$, and its partial order is given by inclusion.

The order ideals of $P$ are in a canonical bijection with the antichains of $P$. The bijection maps every order ideal to the antichain formed by its maximal elements. By setting the `as_ideals` keyword variable to `False`, one can make this method apply this bijection before returning the lattice.

**INPUT:**

- `as_ideals` – Boolean, if `True` (default) returns a poset on the set of order ideals, otherwise on the set of antichains
- `facade` – Boolean or `None` (default). Whether to return a facade lattice or not. By default return facade lattice if the poset is a facade poset.

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: P = posets.PentagonPoset()
sage: P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
sage: J = P.order_ideals_lattice(); J
Finite lattice containing 8 elements
sage: sorted(sorted(e) for e in J)
[[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [1, 2, 3, 4], [0, 2], [0, 3], [0, 2, 3], [0, 2, 3, 4]]
```

As a lattice on antichains:

```python
>>> from sage.all import *
>>> P = posets.PentagonPoset()
>>> P.cover_relations()
[[0, 1], [0, 2], [1, 4], [2, 3], [3, 4]]
>>> J = P.order_ideals_lattice(); J
Finite lattice containing 8 elements
>>> sorted(sorted(e) for e in J)
[[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [0, 2], [0, 3], [0, 2, 3], [0, 2, 3, 4]]
```
panyushev_complement (antichain, direction='up')

Return the Panyushev complement of the antichain antichain.

Given an antichain $A$ of a poset $P$, the Panyushev complement of $A$ is defined to be the antichain consisting of the minimal elements of the order filter $B$, where $B$ is the (set-theoretic) complement of the order ideal of $P$ generated by $A$.

Setting the optional keyword variable direction to 'down' leads to the inverse Panyushev complement being computed instead of the Panyushev complement. The inverse Panyushev complement of an antichain $A$ is the antichain whose Panyushev complement is $A$. It can be found as the antichain consisting of the maximal elements of the order ideal $C$, where $C$ is the (set-theoretic) complement of the order filter of $P$ generated by $A$.

panyushev_complement() is an alias for this method.

Panyushev complementation is related (actually, isomorphic) to rowmotion (rowmotion()).

INPUT:
- antichain – an antichain of self, as a list (or iterable), or, more generally, generators of an order ideal (resp. order filter)
- direction – ‘up’ or ‘down’ (default: ‘up’)

OUTPUT:
- the generating antichain of the complement order filter (resp. order ideal) of the order ideal (resp. order filter) generated by the antichain antichain

EXAMPLES:

```python
sage: P = Poset( ([1,2,3], [[1,3], [2,3]] ) )
sage: P.order_ideal_complement_generators([1])
{2}
sage: P.order_ideal_complement_generators([3])
set()
sage: P.order_ideal_complement_generators([1,2])
{3}
sage: P.order_ideal_complement_generators([1,2,3])
set()
sage: P.order_ideal_complement_generators([1], direction="down")
{2}
sage: P.order_ideal_complement_generators([3], direction="down")
{1, 2}
sage: P.order_ideal_complement_generators([1,2], direction="down")
set()
```
Warning: This is a brute force implementation, building the order ideal generated by the antichain, and searching for order filter generators of its complement

panyushev_orbit_iter (antichain, element_constructor=<class 'set'>, stop=True, check=True)

Iterate over the Panyushev orbit of an antichain antichain of self.

The Panyushev orbit of an antichain is its orbit under Panyushev complementation (see panyushev_complement()).

INPUT:
• antichain – an antichain of self, given as an iterable.
• element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the antichains before they are yielded.
• stop – a Boolean (default: True) determining whether the iterator should stop once it completes its cycle (this happens when it is set to True) or go on forever (this happens when it is set to False).
• check – a Boolean (default: True) determining whether antichain should be checked for being an antichain.

OUTPUT:
• an iterator over the orbit of the antichain antichain under Panyushev complementation.

This iterator \( I \) has the property that \( I[0] == \text{antichain} \) and each \( i \) satisfies \( \text{self.}
\text{order_ideal_complement_generators}(I[i]) == I[i+1], \) where \( I[i+1] \) has to be understood as \( I[0] \) if it is undefined. The entries \( I[i] \) are sets by default, but depending on the optional keyword variable element_constructors they can also be tuples, lists etc.

EXAMPLES:
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.panyushev_orbit_iter(set([1, 2])))
[(1, 2), (3), set()]
sage: list(P.panyushev_orbit_iter([1, 2]))
[(1, 2), (3), set()]
sage: list(P.panyushev_orbit_iter([2, 1]))
[(1, 2), (3), set()]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=list))
[[1, 2], [3], []]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=frozenset))
[frozenset({1, 2}), frozenset({3}), frozenset()]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=tuple))
[(1, 2), (3,), ()]
sage: P = Poset( {} )
sage: list(P.panyushev_orbit_iter([]))
[set()]
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.panyushev_orbit_iter([2], stop=False)
sage: next(Piter)
{2}
sage: next(Piter)
{3}
sage: next(Piter)
{2}
sage: next(Piter)
{3}

>>> from sage.all import *

```python
>>> P = Poset( ( [Integer(1),Integer(2),Integer(3)], [ [Integer(1),
Integer(3)], [Integer(2),Integer(3)] ] ) )
>>> list(P.panyushev_orbit_iter(set([Integer(1), Integer(2)])))
[(1, 2), (3), set()]
>>> list(P.panyushev_orbit_iter([Integer(1), Integer(2)]))
[(1, 2), (3), set()]
>>> list(P.panyushev_orbit_iter([Integer(2), Integer(1)]))
[(1, 2), (3), set()]
>>> list(P.panyushev_orbit_iter(set([Integer(1), Integer(2)]), element_  
constructor=list))
[[1, 2], [3], []]
>>> list(P.panyushev_orbit_iter(set([Integer(1), Integer(2)]), element_  
constructor=frozenset))
[frozenset({1, 2}), frozenset({3}), frozenset()]
>>> list(P.panyushev_orbit_iter(set([Integer(1), Integer(2)]), element_  
constructor=tuple))
[(1, 2), (3,), ()]
```
panyushev_orbits (element_constructor=<class 'set'>)

Return the Panyushev orbits of antichains in self.

The Panyushev orbit of an antichain is its orbit under Panyushev complementation (see panyushev_complement()).

INPUT:
- element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the antichains before they are returned.

OUTPUT:
- the partition of the set of all antichains of self into orbits under Panyushev complementation. This is returned as a list of lists $L$ such that for each $L$ and $i$, cyclically: self.order_ideal_complement_generators($L[i]$) == $L[i+1]$. The entries $L[i]$ are sets by default, but depending on the optional keyword variable element_constructors they can also be tuples, lists etc.

EXAMPLES:

```
sage: # needs sage.modules
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: orb = P.panyushev_orbits()
sage: sorted(sorted(o) for o in orb)  
[([], [1, 2], [3]), [[1], [2]]]
sage: orb = P.panyushev_orbits(element_constructor=list)
 sage: sorted(sorted(o) for o in orb)  
[[], [1, 2], [3]], [[1], [2]]
sage: orb = P.panyushev_orbits(element_constructor=frozenset)
 sage: sorted(sorted(o) for o in orb)  
[[frozenset(), frozenset({1, 2}), frozenset({3})],
 [frozenset({2}), frozenset({1})]]
sage: orb = P.panyushev_orbits(element_constructor=tuple)
 sage: sorted(sorted(o) for o in orb)  
[(), (1, 2), (3,)], [(1,), (2,)]
sage: P = Poset( {} )
sage: P.panyushev_orbits()
[[set()]]
```
rowmotion (order_ideal)
The image of the order ideal order_ideal under rowmotion in self.

Rowmotion on a finite poset $P$ is an automorphism of the set $J(P)$ of all order ideals of $P$. One way to define it is as follows: Given an order ideal $I \in J(P)$, we let $F$ be the set-theoretic complement of $I$ in $P$. Furthermore we let $A$ be the antichain consisting of all minimal elements of $F$. Then, the rowmotion of $I$ is defined to be the order ideal of $P$ generated by the antichain $A$ (that is, the order ideal consisting of each element of $P$ which has some element of $A$ above it).

Rowmotion is related (actually, isomorphic) to Panyushev complementation (panyushev_complement()).

INPUT:
• order_ideal – an order ideal of self, as a set

OUTPUT:
• the image of order_ideal under rowmotion, as a set again

EXAMPLES:

```python
sage: P = Poset({1: [2, 3], 2: [], 3: [], 4: [8],
      ....: 5: [], 6: [5], 7: [1, 4], 8: []})
sage: I = Set({2, 6, 1, 7})
sage: P.rowmotion(I)
{1, 3, 4, 5, 6, 7}

sage: P = Poset({})
sage: I = Set({})
sage: P.rowmotion(I)
{ }
```

rowmotion_orbit_iter (oideal, element_constructor=<class 'set'>, stop=True, check=True)
Iterate over the rowmotion orbit of an order ideal oideal of self.

The rowmotion orbit of an order ideal is its orbit under rowmotion (see rowmotion()).
INPUT:
• oideal – an order ideal of self, given as an iterable.
• element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the order ideals before they are yielded.
• stop – a Boolean (default: True) determining whether the iterator should stop once it completes its cycle (this happens when it is set to True) or go on forever (this happens when it is set to False).
• check – a Boolean (default: True) determining whether oideal should be checked for being an order ideal.

OUTPUT:
• an iterator over the orbit of the order ideal oideal under rowmotion. This iterator \( I \) has the property that \( I[0] == \text{oideal} \) and that every \( i \) satisfies \( \text{self.rowmotion}(I[i]) == I[i+1] \), where \( I[i+1] \) has to be understood as \( I[0] \) if it is undefined. The entries \( I[i] \) are sets by default, but depending on the optional keyword variable element_constructors they can also be tuples, lists etc.

EXAMPLES:

```python
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.rowmotion_orbit_iter(set([1, 2])))
[(1, 2), {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter([1, 2]))
[(1, 2), {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter([2, 1]))
[(1, 2), {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=list))
[[1, 2], [1, 2, 3], []]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=frozenset))
[frozenset({1, 2}), frozenset({1, 2, 3}), frozenset({})]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=tuple))
[(1, 2), (1, 2, 3), ()]
sage: P = Poset( {} )
sage: list(P.rowmotion_orbit_iter([]))
[set()]
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.rowmotion_orbit_iter([1, 2, 3], stop=False)
sage: next(Piter)
{1, 2, 3}
sage: next(Piter)
{1, 2, 3, 4}
sage: next(Piter)
set()
sage: next(Piter)
{1}
sage: next(Piter)
{1, 2, 3}
sage: P = Poset({ 1: [4], 2: [4, 5], 3: [5] })
sage: list(P.rowmotion_orbit_iter((1, 2), element_constructor=list))
[(1, 2), [1, 2, 3, 4], [2, 3, 5], [1, 2, 3, 5], [1, 2, 4, 3], [1]]
```

```python
>>> from sage.all import *
>>> P = Poset( ( [Integer(1),Integer(2),Integer(3)], [ [Integer(1), Integer(3)], [Integer(2),Integer(3)] ] ) )
```

(continues on next page)
list(P.rowmotion_orbit_iter(set([Integer(1), Integer(2)])))
[[1, 2], {1, 2, 3}, set()]
list(P.rowmotion_orbit_iter([Integer(1), Integer(2)]))
[[1, 2], {1, 2, 3}, set()]
list(P.rowmotion_orbit_iter([Integer(2), Integer(1)]))
[[1, 2], {1, 2, 3}, set()]
list(P.rowmotion_orbit_iter(set([Integer(1), Integer(2)]), element_constructor=list))
[[1, 2], [1, 2, 3], []]
list(P.rowmotion_orbit_iter(set([Integer(1), Integer(2)]), element_constructor=frozendict))
[frozendict((1, 2)), frozendict((1, 2, 3)), frozendict()]
list(P.rowmotion_orbit_iter(set([Integer(1), Integer(2)]), element_constructor=tuple))
[(1, 2), (1, 2, 3), ()]

P = Poset( {} )
list(P.rowmotion_orbit_iter([]))
[set()]


Piter = P.rowmotion_orbit_iter([Integer(1), Integer(2), Integer(3)], stop=False)
next(Piter)
{1, 2, 3}
next(Piter)
{1, 2, 3, 4}
next(Piter)
set()
next(Piter)
{1}
next(Piter)
{1, 2, 3}

P = Poset({ Integer(1): [Integer(2), Integer(3)], Integer(2): [Integer(4), Integer(5)], Integer(3): [Integer(5)] })
list(P.rowmotion_orbit_iter([Integer(1), Integer(2)]), element_constructor=list))
[[1, 2], [1, 2, 3, 4], [2, 3, 5], [1], [2, 3], [1, 2, 3, 5], [1, 2, 4], [3]]

rowmotion_orbits(element_constructor=<class 'set'>)
Return the rowmotion orbits of order ideals in self.

The rowmotion orbit of an order ideal is its orbit under rowmotion (see rowmotion()).

INPUT:
• element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the antichains before they are returned.

OUTPUT:
• the partition of the set of all order ideals of self into orbits under rowmotion. This is returned as a list of lists L such that for each L and i, cyclically: self.rowmotion(L[i]) == L[i+1]. The entries L[i] are sets by default, but depending on the optional keyword variable element_constructors they can also be tuples, lists etc.

EXAMPLES:
```python
sage: # needs sage.modules
sage: P = Poset( {1: [2, 3], 2: [], 3: [], 4: [2]} )
sage: sorted(len(o) for o in P.rowmotion_orbits())
[3, 5]
sage: orb = P.rowmotion_orbits(element_constructor=list)
sage: sorted(sorted(e) for e in orb)
[[[], [4, 1], [4, 1, 2, 3]], [[1], [1, 3], [4], [4, 1, 2], [4, 1, 3]]]
sage: orb = P.rowmotion_orbits(element_constructor=tuple)
sage: sorted(sorted(e) for e in orb)
[[(), (4, 1), (4, 1, 2, 3)], [(1,), (1, 3), (4,), (4, 1, 2), (4, 1, 3)]]
sage: P = Poset({})
sage: P.rowmotion_orbits(element_constructor=tuple)
[()]```
vs[0], vs[1], ... in this order.

See order_ideal_toggle() for a definition of toggling.

**Warning:** The orbit is that under the composition of toggles, not under the single toggles themselves. Thus, for example, if vs == [1, 2], then the orbit has the form (I, T_2 T_1 I, T_2 T_1 T_1 I, ...) (where I denotes oideal and T_i means toggling at i) rather than (I, T_1 I, T_1 T_2 I, T_1 T_2 T_1 I, ...).

**INPUT:**
- vs: a list (or other iterable) of elements of self (but since the output depends on the order, sets should not be used as vs).
- oideal – an order ideal of self, given as an iterable.
- element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the order ideals before they are yielded.
- stop – a Boolean (default: True) determining whether the iterator should stop once it completes its cycle (this happens when it is set to True) or go on forever (this happens when it is set to False).
- check – a Boolean (default: True) determining whether oideal should be checked for being an order ideal.

**OUTPUT:**
- an iterator over the orbit of the order ideal oideal under toggling the vertices in the list vs in this order. This iterator I has the property that I[0] == oideal and that every i satisfies self.order_ideal_toggles(I[i], vs) == I[i+1], where I[i+1] has to be understood as I[0] if it is undefined. The entries I[i] are sets by default, but depending on the optional keyword variable element_constructors they can also be tuples, lists etc.

**EXAMPLES:**

```python
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.toggling_orbit_iter([1, 3, 1], set([1, 2])))
[(1, 2)]
sage: list(P.toggling_orbit_iter([1, 2, 3], set([1, 2])))
[(1, 2), (1, 2, 3)]
sage: list(P.toggling_orbit_iter([3, 2, 1], set([1, 2])))
[(1, 2), (1, 2, 3), ()]
sage: list(P.toggling_orbit_iter([3, 2, 1], set([1, 2]), element_constructor=list))
[(1, 2), (1, 2, 3), []]
sage: list(P.toggling_orbit_iter([3, 2, 1], set([1, 2]), element_constructor=frozenset))
[(1, 2), (1, 2, 3), ()]
sage: list(P.toggling_orbit_iter([3, 2, 1], [2, 1], element_constructor=tuple))
[(1, 2), (1, 2, 3), ()]
sage: P = Poset( {} )
sage: list(P.toggling_orbit_iter([], []))
[set()]
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.toggling_orbit_iter([1, 2, 4, 3], [1, 2, 3], stop=False)
sage: next(Piter)
{1, 2, 3}
sage: next(Piter)
```

(continues on next page)
\[
\begin{align*}
\text{sage: } & \text{next}(Piter) \\
\text{set()} & \\
\text{sage: } & \text{next}(Piter) \\
\{1, 2, 3\} & \\
\text{sage: } & \text{next}(Piter) \\
\{1\} & \\
\end{align*}
\]

\[
\begin{align*}
>>> \text{from sage.all import } * \\
>>> P = \text{Poset}((\text{[Integer(1), Integer(2), Integer(3)]}, [\text{[Integer(1),
\text{Integer(3)]}, [\text{Integer(2), Integer(3)] \} \})}) \\
>>> \text{list}(P.\text{toggling_orbit}._\text{iter}([\text{Integer(1), Integer(3), Integer(1)]}, \text{set\{Integer(1), Integer(2)\} })) \\
\{1, 2\} & \\
>>> \text{list}(P.\text{toggling_orbit}._\text{iter}([\text{Integer(1), Integer(2), Integer(3)]}, \text{set\{Integer(1), Integer(2)\} })) \\
\{1, 2\}, \{1, 2, 3\} & \\
>>> \text{list}(P.\text{toggling_orbit}._\text{iter}([\text{Integer(3), Integer(2), Integer(1)]}, \text{set\{Integer(1), Integer(2)\} })) \\
\{1, 2\}, \{1, 2, 3\}, \text{set()} & \\
>>> \text{list}(P.\text{toggling_orbit}._\text{iter}([\text{Integer(3), Integer(2), Integer(1)]}, \text{set\{Integer(1), Integer(2)\} })) \\
\{1, 2\}, \{1, 2, 3\}, \{\}
\end{align*}
\]

\[
\begin{align*}
>>> P = \text{Poset}(\{\}) \\
>>> \text{list}(P.\text{toggling_orbit}._\text{iter}([], [])) \\
\{\text{set()}\} & \\
\end{align*}
\]

\[
\begin{align*}
>>> P = \text{Poset}\{\text{Integer(1): [Integer(2), Integer(3), Integer(2):}} \\
\text{[Integer(4), Integer(3): [Integer(4), Integer(4): [)}} \\
>>> \text{Piter} = P.\text{toggling_orbit}._\text{iter}([\text{Integer(1), Integer(2), Integer(4), Integer(3), [Integer(1), Integer(2), Integer(3), stop=False}} \\
>>> \text{next}(Piter) \\
\{1, 2, 3\} & \\
>>> \text{next}(Piter) \\
\{1\} & \\
>>> \text{next}(Piter) \\
\text{set()} & \\
>>> \text{next}(Piter) \\
\{1, 2, 3\} & \\
>>> \text{next}(Piter) \\
\{1\} & \\
\end{align*}
\]

toggling_orbits(vs, element_constructor=<class set>)}
Return the orbits of order ideals in self under the operation of toggling the vertices \(vs[0], vs[1], \ldots\) in this order.

See \texttt{order_ideal_toggle()} for a definition of toggling.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Warning:} The orbits are those under the composition of toggles, \textit{not} under the single toggles themselves. Thus, for example, if \(vs == [1, 2]\), then the orbits have the form \((I, T_2 T_1 I, T_2 T_1 T_2 T_1 I, \ldots)\) (where \(I\) denotes an order ideal and \(T_i\) means toggling at \(i\)) rather than \((I, T_1 I, T_2 T_1 I, T_1 T_2 T_1 I, \ldots)\).
\hline
\end{tabular}
\end{center}

INPUT:
\begin{itemize}
\item \(vs\): a list (or other iterable) of elements of self (but since the output depends on the order, sets should not be used as vs).
\end{itemize}

OUTPUT:
\begin{itemize}
\item a partition of the order ideals of self, as a list of sets \(L\) such that for each \(L\) and \(i\), cyclically:
\begin{center}
\texttt{self.order_ideal_toggles(L[i], vs) == L[i+1]}.
\end{center}
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: P = Poset( {1: [2, 4], 2: [], 3: [4], 4: []} )
sage: sorted(len(o) for o in P.toggling_orbits([1, 2]))
[2, 3, 3]
sage: P = Poset( {1: [3], 2: [1, 4], 3: [], 4: [3]} )
sage: sorted(len(o) for o in P.toggling_orbits((1, 2, 4, 3)))
[3, 3]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> P = Poset( {Integer(1): [Integer(2), Integer(4)], Integer(2): [],
Integer(3): [Integer(4)], Integer(4): []} )
>>> sorted(len(o) for o in P.toggling_orbits((Integer(1), Integer(2)))
[2, 3, 3]
>>> P = Poset( {Integer(1): [Integer(3)], Integer(2): [Integer(1),
Integer(4)], Integer(3): [], Integer(4): [Integer(3)]} )
>>> sorted(len(o) for o in P.toggling_orbits((Integer(1), Integer(2),
Integer(4), Integer(3))))
[3, 3]
\end{verbatim}

toggling_orbits_plots(vs)

Return plots of the orbits of order ideals in self under the operation of toggling the vertices \(vs[0], vs[1], \ldots\) in this order.

See \texttt{toggling_orbits()} for more information.

EXAMPLES:

\begin{verbatim}
sage: P = Poset( {1: [2, 3], 2: [], 3: [4], 4: []} )
sage: P.toggling_orbits_plots([1,2,3,4])
Graphics Array of size 2 x 5
sage: P = Poset({})
sage: P.toggling_orbits_plots([])
Graphics Array of size 1 x 1
\end{verbatim}
4.63 Finite semigroups

class sage.categories.finite_semigroups.FiniteSemigroups(base_category)

Bases: CategoryWithAxiom_singleton

The category of finite (multiplicative) semigroups.

A finite semigroup is a finite set endowed with an associative binary operation *.

Warning: Finite semigroups in Sage used to be automatically endowed with an enumerated set structure; the default enumeration is then obtained by iteratively multiplying the semigroup generators. This forced any finite semigroup to either implement an enumeration, or provide semigroup generators; this was often inconvenient.

Instead, finite semigroups that provide a distinguished finite set of generators with semigroup_generators() should now explicitly declare themselves in the category of finitely generated semigroups:

```python
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
```

This is a backward incompatible change.

EXAMPLES:

```python
sage: C = FiniteSemigroups(); C
Category of finite semigroups
sage: C.super_categories()

[Category of semigroups, Category of finite sets]
sage: sorted(C.axioms())
['Associative', 'Finite']
sage: C.example()
An example of a finite semigroup:
the left regular band generated by ('a', 'b', 'c', 'd')
```

(continues on next page)
An example of a finite semigroup:
the left regular band generated by ('a', 'b', 'c', 'd')

class ParentMethods

    idempotents()

    Returns the idempotents of the semigroup

    EXAMPLES:

    sage: S = FiniteSemigroups().example(alphabet=('x', 'y'))
    sage: sorted(S.idempotents())
    ['x', 'xy', 'y', 'yx']

    j_classes()

    Returns the J-classes of the semigroup.

    Two elements \( u \) and \( v \) of a monoid are in the same J-class if \( u \) divides \( v \) and \( v \) divides \( u \).

    OUTPUT:
    All the J-classes of self, as a list of lists.

    EXAMPLES:

    sage: S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
    sage: sorted(map(sorted, S.j_classes()))
    [[a], [ab, ba], [abc, acb, bac, bca, cab, cba], [ac, ca], [b], [bc, cb], [c]]

    j_classes_of_idempotents()

    Returns all the idempotents of self, grouped by J-class.

    OUTPUT:
    a list of lists.

    EXAMPLES:

    sage: S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
    sage: sorted(map(sorted, S.j_classes_of_idempotents()))
    [[a], [ab, ba], [abc, acb, bac, bca, cab, cba], [ac, ca], [b], [bc, cb], [c]]
>>> from sage.all import *
>>> S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
# needs sage.graphs
>>> sorted(map(sorted, S.j_classes_of_idempotents()))
[['a'], ['ab', 'ba'], ['abc', 'acb', 'bac', 'bca', 'cab', 'cba'], ['ac', 'ca'], ['b'], ['bc', 'cb'], ['c']]

**j_transversal_of_idempotents()**

Returns a list of one idempotent per regular J-class

EXAMPLES:

```python
sage: S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
```

```python
>>> from sage.all import *
>>> S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
# needs sage.graphs
```

The chosen elements depend on the order of each J-class, and that order is random when using Python 3.

```python
sage: sorted(S.j_transversal_of_idempotents())  # random
# needs sage.graphs
```

```python
>>> from sage.all import *
>>> sorted(S.j_transversal_of_idempotents())  # random
# needs sage.graphs
```

### 4.64 Finite sets

**class** `sage.categories.finite_sets.FiniteSets(base_category)`

Bases: `CategoryWithAxiom_singleton`

The category of finite sets.

EXAMPLES:

```python
sage: C = FiniteSets(); C
Category of finite sets
sage: C.super_categories()
[Category of sets]
sage: C.all_super_categories()
[Category of finite sets, Category of sets, Category of sets with partial maps, Category of objects]
sage: C.example()
NotImplemented
```
```python
>>> from sage.all import *
>>> C = FiniteSets(); C
Category of finite sets
>>> C.super_categories()
[Category of sets]
>>> C.all_super_categories()
[Category of finite sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
>>> C.example()
NotImplemented
```

```python
class Algebras(category, *args)
    Bases: AlgebrasCategory

eextra_super_categories()
    EXAMPLES:

    sage: FiniteSets().Algebras(QQ).extra_super_categories()
    [Category of finite dimensional vector spaces with basis over Rational Field]

>>> from sage.all import *
>>> FiniteSets().Algebras(QQ).extra_super_categories()
[Category of finite dimensional vector spaces with basis over Rational Field]

This implements the fact that the algebra of a finite set is finite dimensional:

    sage: FiniteMonoids().Algebras(QQ).is_subcategory(AlgebrasWithBasis(QQ).FiniteDimensional())
    True

>>> from sage.all import *
>>> FiniteMonoids().Algebras(QQ).is_subcategory(AlgebrasWithBasis(QQ).FiniteDimensional())
    True
```

```python
class ParentMethods
    Bases: object

    is_finite()
    Return True since self is finite.

    EXAMPLES:

    sage: C = FiniteEnumeratedSets().example()
    sage: C.is_finite()
    True

>>> from sage.all import *
>>> C = FiniteEnumeratedSets().example()
>>> C.is_finite()
    True
```

4.64. Finite sets
class Subquotients(category, *args)
    Bases: SubquotientsCategory

    extra_super_categories()

    EXAMPLES:

    sage: FiniteSets().Subquotients().extra_super_categories()
    [Category of finite sets]

    >>> from sage.all import *
    >>> FiniteSets().Subquotients().extra_super_categories()
    [Category of finite sets]

This implements the fact that a subquotient (and therefore a quotient or subobject) of a finite set is finite:

    sage: FiniteSets().Subquotients().is_subcategory(FiniteSets())
    True
    sage: FiniteSets().Quotients().is_subcategory(FiniteSets())
    True
    sage: FiniteSets().Subobjects().is_subcategory(FiniteSets())
    True

>>> from sage.all import *
>>> FiniteSets().Subquotients().is_subcategory(FiniteSets())
True
>>> FiniteSets().Quotients().is_subcategory(FiniteSets())
True
>>> FiniteSets().Subobjects().is_subcategory(FiniteSets())
True

4.65 Finite Weyl Groups

class sage.categories.finite_weyl_groups.FiniteWeylGroups(base_category)
    Bases: CategoryWithAxiom

    The category of finite Weyl groups.

    EXAMPLES:

    sage: C = FiniteWeylGroups()
    sage: C
    Category of finite Weyl groups
    sage: C.super_categories()
    [Category of finite Coxeter groups, Category of Weyl groups]
    sage: C.example()
    The symmetric group on {0, ..., 3}

>>> from sage.all import *
>>> C = FiniteWeylGroups()
>>> C
Category of finite Weyl groups
>>> C.super_categories()
[Category of finite Coxeter groups, Category of Weyl groups]
>>> C.example()
The symmetric group on {0, ..., 3}
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

4.66 Finitely Generated Lambda bracket Algebras

AUTHORS:

- Reimundo Heluani (2020-08-21): Initial implementation.

class sage.categories.finitely_generated_lambda_bracket_algebras.FinitelyGeneratedLambdaBracketAlgebras(base_category)
    Bases: CategoryWithAxiom_over_base_ring

The category of finitely generated lambda bracket algebras.

EXAMPLES:

sage: from sage.categories.lambda_bracket_algebras import LambdaBracketAlgebras
sage: LambdaBracketAlgebras(QQbar).FinitelyGenerated()
# Category of finitely generated lambda bracket algebras over Algebraic Field

>>> from sage.all import *
>>> from sage.categories.lambda_bracket_algebras import LambdaBracketAlgebras
>>> LambdaBracketAlgebras(QQbar).FinitelyGenerated()
# Category of finitely generated lambda bracket algebras over Algebraic Field

class Graded(base_category)
    Bases: GradedModulesCategory

The category of H-graded finitely generated Lie conformal algebras.

EXAMPLES:

sage: LieConformalAlgebras(QQbar).FinitelyGenerated().Graded()
# Category of H-graded finitely generated Lie conformal algebras over Algebraic Field

>>> from sage.all import *
>>> LieConformalAlgebras(QQbar).FinitelyGenerated().Graded()
# Category of H-graded finitely generated Lie conformal algebras over Algebraic Field

class ParentMethods
    Bases: object

    gen(i)
        The i-th generator of this Lie conformal algebra.

    EXAMPLES:
sage: # needs sage.combinat sage.modules
sage: V = lie_conformal_algebras.Affine(QQ, 'A1')
sage: V.gens()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])
sage: V.gen(0)
B[alpha[1]]
sage: V.1
B[alphacheck[1]]

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> V = lie_conformal_algebras.Affine(QQ, 'A1')
>>> V.gens()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])
>>> V.gen(Integer(0))
B[alpha[1]]
>>> V.gen(1)
B[alphacheck[1]]

ngens()

The number of generators of this Lie conformal algebra.

EXAMPLES:

<table>
<thead>
<tr>
<th>Python Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: Vir = lie_conformal_algebras.Virasoro(QQ)</td>
<td># needs sage.combinat sage.modules</td>
</tr>
<tr>
<td>sage: Vir.ngens()</td>
<td># needs sage.combinat sage.modules</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>sage: V = lie_conformal_algebras.Affine(QQ, 'A2')</td>
<td># needs sage.combinat sage.modules</td>
</tr>
<tr>
<td>sage: V.ngens()</td>
<td># needs sage.combinat sage.modules</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

some_elements()

Some elements of this Lie conformal algebra.

This method returns a list with elements containing at least the generators.

EXAMPLES:
4.67 Finitely Generated Lie Conformal Algebras

AUTHORS:


class sage.categories.finitely_generated_lie_conformal_algebras.FinitelyGeneratedLieConformalAlgebras

Bases: CategoryWithAxiom_over_base_ring

The category of finitely generated Lie conformal algebras.

EXAMPLES:

class Graded(base_category)

Bases: GradedModulesCategory

The category of H-graded finitely generated Lie conformal algebras.

EXAMPLES:
>>> from sage.all import *

>>> LieConformalAlgebras(QQbar).FinitelyGenerated().Graded() # needs sage.rings.number_field
Category of H-graded finitely generated Lie conformal algebras over Algebraic Field

class ParentMethods

Bases: object

some_elements()

Some elements of this Lie conformal algebra.

Returns a list with elements containing at least the generators.

EXAMPLES:

sage: V = lie_conformal_algebras.Affine(QQ, 'A1', 
# needs sage.combinat sage.modules
...: names=('e', 'h', 'f'))

sage: V.some_elements()  
# needs sage.combinat sage.modules
[e, h, f, K, ...]

sage: all(v.parent() is V for v in V.some_elements())  
# needs sage.combinat sage.modules
True

>>> from sage.all import *

>>> LieConformalAlgebras(AA).FinitelyGenerated().Super() # needs sage.rings.number_field
Category of super finitely generated Lie conformal algebras over Algebraic Real Field

>>> from sage.all import *

>>> LieConformalAlgebras(AA).FinitelyGenerated().Super() # needs sage.rings.number_field
Category of super finitely generated Lie conformal algebras over Algebraic Real Field

class Super(base_category)

Bases: SuperModulesCategory

The category of super finitely generated Lie conformal algebras.

EXAMPLES:

sage: LieConformalAlgebras(AA).FinitelyGenerated().Super()  
# needs sage.rings.number_field
Category of super finitely generated Lie conformal algebras over Algebraic Real Field

>>> from sage.all import *

>>> LieConformalAlgebras(AA).FinitelyGenerated().Super()  
# needs sage.rings.number_field
Category of super finitely generated Lie conformal algebras over Algebraic Real Field

class Graded(base_category)

Bases: GradedModulesCategory
The category of $H$-graded super finitely generated Lie conformal algebras.

EXAMPLES:

```python
sage: LieConformalAlgebras(QQbar).FinitelyGenerated().Super().Graded()  # Needs sage.rings.number_field
Category of $H$-graded super finitely generated Lie conformal algebras
over Algebraic Field
```

4.68 Finitely generated magmas

```python
>>> from sage.all import *

>>> C = Magmas().FinitelyGeneratedAsMagma(); C
Category of finitely generated magmas

>>> C.super_categories()
[Category of magmas]

>>> sorted(C.axioms())
['FinitelyGeneratedAsMagma']
```

```python
class ParentMethods
    Bases: object

    magma_generators()  
        Return distinguished magma generators for self.  
        OUTPUT: a finite family  
        This method should be implemented by all finitely generated magmas.

        EXAMPLES:
        sage: S = FiniteSemigroups().example()
        sage: S.magma_generators()
        Family ('a', 'b', 'c', 'd')
```
4.69 Finitely generated semigroups

```python
>>> from sage.all import *
>>> S = FiniteSemigroups().example()
>>> S.magma_generators()
Family (a, b, c, d)
```

A **finitely generated semigroup** is a semigroup endowed with a distinguished finite set of generators (see `FinitelyGeneratedSemigroups.ParentMethods.semigroup_generators()`). This makes it into an **enumerated set**.

**EXAMPLES:**

```python
sage: C = Semigroups().FinitelyGenerated(); C
Category of finitely generated semigroups
sage: C.super_categories()
[Category of semigroups, 
 Category of finitely generated magmas, 
 Category of enumerated sets]
sage: sorted(C.axioms())
['Associative', 'Enumerated', 'FinitelyGeneratedAsMagma']
```

```python
sage: C.example()
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
```

```python
>>> from sage.all import *
>>> C = Semigroups().FinitelyGenerated(); C
Category of finitely generated semigroups
>>> C.super_categories()
[Category of semigroups, 
 Category of finitely generated magmas, 
 Category of enumerated sets]
>>> sorted(C.axioms())
['Associative', 'Enumerated', 'FinitelyGeneratedAsMagma']
```

```python
>>> C.example()
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
```

**Class** `sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups(base_category)`

**Bases:** `CategoryWithAxiom_singleton`

The category of finitely generated (multiplicative) semigroups.

**class** `Finite(base_category)`

**Bases:** `CategoryWithAxiom_singleton`

**class** `ParentMethods`

**Bases:** `object`

**some_elements()**

Return an iterable containing some elements of the semigroup.

**OUTPUT:** the ten first elements of the semigroup, if they exist.
EXAMPLES:

```python
sage: S = FiniteSemigroups().example(alphabet=('x', 'y'))
sage: sorted(S.some_elements())
['x', 'xy', 'y', 'yx']
sage: S = FiniteSemigroups().example(alphabet=('x', 'y', 'z'))
sage: X = S.some_elements()
sage: len(X)
10
sage: all(x in S for x in X)
True
```

```python
>>> from sage.all import *
>>> S = FiniteSemigroups().example(alphabet=('x', 'y'))
>>> sorted(S.some_elements())
['x', 'xy', 'y', 'yx']
>>> S = FiniteSemigroups().example(alphabet=('x', 'y', 'z'))
>>> X = S.some_elements()
>>> len(X)
10
>>> all(x in S for x in X)
True
```

class ParentMethods
Bases: object

```python
ideal (gens, side='twosided')
```

Return the side-sided ideal generated by gens.

This brute force implementation recursively multiplies the elements of gens by the distinguished generators of this semigroup.

See also:

```python
semigroup_generators()
```

INPUT:

- gens -- a list (or iterable) of elements of self
- side -- [default: “twosided”] “left”, “right” or “twosided”

EXAMPLES:

```python
sage: S = FiniteSemigroups().example()
sage: sorted(S.ideal([S('cab')], side="left"))
['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
 'adcb', 'bac', 'bacd', 'badc', 'bca', 'bcad', 'bcda',
 'bdac', 'bdca', 'cab', 'cabd', 'cadb', 'cba', 'cbad',
 'cbda', 'cdab', 'cdba', 'dabc', 'dacb', 'dbac', 'dbca',
 'dcab', 'dcba']
sage: list(S.ideal([S('cab')], side="right"))
['cab', 'cabd']
sage: sorted(S.ideal([S('cab')], side="twosided"))
['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
 'adcb', 'bac', 'bacd', 'badc', 'bca', 'bcad', 'bcda',
 'bdac', 'bdca', 'cab', 'cabd', 'cadb', 'cba', 'cbad',
 'cbda', 'cdab', 'cdba', 'dabc', 'dacb', 'dbac', 'dbca',
 'dcab', 'dcba']
sage: sorted(S.ideal([S('cab')])))
['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
```
semigroup_generators()

Return distinguished semigroup generators for self.

OUTPUT: a finite family

This method should be implemented by all semigroups in FinitelyGeneratedSemigroups.

EXAMPLES:

```python
sage: S = FiniteSemigroups().example()
sage: S.semigroup_generators()
Family ("a", "b", "c", "d")
```

succ_generators(side='twosided')

Return the successor function of the side-sided Cayley graph of self.

This is a function that maps an element of self to all the products of x by a generator of this semigroup, where the product is taken on the left, right, or both sides.

INPUT:

- side: “left”, “right”, or “twosided”

Todo: Design choice:
- find a better name for this method
• should we return a set? a family?

EXAMPIES:

```python
sage: S = FiniteSemigroups().example()
sage: S.succGenerators("left")(S('ca')) ('ac', 'bca', 'ca', 'dca')
sage: S.succGenerators("right") (S('ca'))
('ca', 'cab', 'ca', 'cad')
sage: S.succGenerators("twosided") (S('ca'))
('ac', 'bca', 'ca', 'dca', 'ca', 'cab', 'ca', 'cad')
```

```python
>>> from sage.all import *
>>> S = FiniteSemigroups().example()
>>> S.succGenerators("left") (S('ca'))
('ac', 'bca', 'ca', 'dca')
>>> S.succGenerators("right") (S('ca'))
('ca', 'cab', 'ca', 'cad')
>>> S.succGenerators("twosided") (S('ca'))
('ac', 'bca', 'ca', 'dca', 'ca', 'cab', 'ca', 'cad')
```

`example()`

EXAMPIES:

```python
sage: Semigroups().FinitelyGenerated().example()
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
```

```python
>>> from sage.all import *
>>> Semigroups().FinitelyGenerated().example()
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
```

`extraSuper_categories()`

State that a finitely generated semigroup is endowed with a default enumeration.

EXAMPIES:

```python
sage: Semigroups().FinitelyGenerated().extraSuper_categories()
[Category of enumerated sets]
```

```python
>>> from sage.all import *
>>> Semigroups().FinitelyGenerated().extraSuper_categories()
[Category of enumerated sets]
```
4.70 Function fields

class sage.categories.function_fields.FunctionFields
    Bases: Category
    The category of function fields.

    EXAMPLES:
    We create the category of function fields:

    sage: C = FunctionFields()
sage: C
    Category of function fields

>>> from sage.all import *
>>> C = FunctionFields()
>>> C
    Category of function fields

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

    super_categories()
        Returns the Category of which this is a direct sub-Category For a list off all super categories see all_super_categories

        EXAMPLES:

        sage: FunctionFields().super_categories()
        [Category of fields]

>>> from sage.all import *
>>> FunctionFields().super_categories()
[Category of fields]

4.71 G-Sets

class sage.categories.g_sets.GSets(G)
    Bases: Category
    The category of G-sets, for a group G.

    EXAMPLES:

    sage: S = SymmetricGroup(3)  #...
    → needs sage.groups
    sage: GSets(S)  #...
    → needs sage.groups
    Category of G-sets for Symmetric group of order 3! as a permutation group
>>> from sage.all import *
>>> S = SymmetricGroup(Integer(3))  # needs sage.groups
>>> GSets(S)  # needs sage.groups
Category of G-sets for Symmetric group of order 3! as a permutation group

TODO: should this derive from Category_over_base?

**classmethod an_instance()**

Returns an instance of this class.

**EXAMPLES:**

```python
sage: GSets.an_instance()  # indirect doctest
# needs sage.groups
Category of G-sets for Symmetric group of order 3! as a permutation group
```

```python
>>> from sage.all import *
>>> GSets.an_instance()  # indirect doctest
# needs sage.groups
Category of G-sets for Symmetric group of order 3! as a permutation group
```

**super_categories()**

**EXAMPLES:**

```python
sage: GSets(SymmetricGroup(8)).super_categories()  # need sage.groups
[Category of sets]
```

```python
>>> from sage.all import *
>>> GSets(SymmetricGroup(Integer(8))).super_categories()  # need sage.groups
[Category of sets]
```

### 4.72 Gcd domains

class sage.categories.gcd_domains.GcdDomains

**Bases:** Category_singleton

The category of gcd domains domains where gcd can be computed but where there is no guarantee of factorisation into irreducibles

**EXAMPLES:**

```python
sage: GcdDomains()
Category of gcd domains
```

```python
sage: GcdDomains().super_categories()
[Category of integral domains]
```

```python
>>> from sage.all import *
>>> GcdDomains()
Category of gcd domains
```

(continues on next page)
>>> GcdDomains().super_categories()
[Category of integral domains]

class ElementMethods
    Bases: object
class ParentMethods
    Bases: object
additional_structure()
    Return None.
    Indeed, the category of gcd domains defines no additional structure: a ring morphism between two gcd
domains is a gcd domain morphism.
    See also:
    Category.additional_structure()
    EXAMPLES:
    sage: GcdDomains().additional_structure()

super_categories()
    EXAMPLES:
    sage: GcdDomains().super_categories()
[Category of integral domains]

4.73 Generalized Coxeter Groups

class sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups
    Bases: Category_singleton
    The category of generalized Coxeter groups.
    A generalized Coxeter group is a group with a presentation of the following form:
    \[ \langle s_i \mid s_i^{p_i}, s_i s_j \cdots = s_j s_i \cdots \rangle, \]
    where \( p_i > 1, i \in I \), and the factors in the braid relation occur \( m_{ij} = m_{ji} \) times for all \( i \neq j \in I \).
    EXAMPLES:
    sage: from sage.categories.generalized_coxeter_groups import...
    ~GeneralizedCoxeterGroups
    sage: C = GeneralizedCoxeterGroups(); C
    Category of generalized Coxeter groups
>>> from sage.all import *
>>> from sage.categories.generalized_coxeter_groups import...
...GeneralizedCoxeterGroups
>>> C = GeneralizedCoxeterGroups(); C
Category of generalized Coxeter groups

class Finite(base_category)
    Bases: CategoryWithAxiom_singleton

    The category of finite generalized Coxeter groups.

    extra_super_categories()
        Implement that a finite generalized Coxeter group is a well-generated complex reflection group.

        EXAMPLES:

        sage: from sage.categories.generalized_coxeter_groups import...
...GeneralizedCoxeterGroups
        sage: from sage.categories.complex_reflection_groups import...
...ComplexReflectionGroups
        sage: Cat = GeneralizedCoxeterGroups().Finite()
        sage: Cat.extra_super_categories()
        [Category of well generated finite complex reflection groups]
        sage: Cat.is_subcategory(ComplexReflectionGroups().Finite().WellGenerated())
        True

additional_structure()
    Return None.

    Indeed, all the structure generalized Coxeter groups have in addition to groups (simple reflections, …) is
    already defined in the super category.

    See also:
    Category.additional_structure()

    EXAMPLES:

    sage: from sage.categories.generalized_coxeter_groups import...
    ...GeneralizedCoxeterGroups
    sage: GeneralizedCoxeterGroups().additional_structure()

(continues on next page)
super_categories()

EXAMPLES:

```python
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: GeneralizedCoxeterGroups().super_categories()
[Category of complex reflection or generalized Coxeter groups]
```

```python
>>> from sage.all import *
>>> from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
>>> GeneralizedCoxeterGroups().super_categories()
[Category of complex reflection or generalized Coxeter groups]
```

### 4.74 Graded Algebras

**class** `sage.categories.graded_algebras.GradedAlgebras(base_category)`

**Bases:** `GradedModulesCategory`

The category of graded algebras

**EXAMPLES:**

```python
sage: GradedAlgebras(ZZ)
Category of graded algebras over Integer Ring
sage: GradedAlgebras(ZZ).super_categories()
[Category of filtered algebras over Integer Ring, Category of graded modules over Integer Ring]
```

```python
>>> from sage.all import *
>>> GradedAlgebras(ZZ)
Category of graded algebras over Integer Ring
>>> GradedAlgebras(ZZ).super_categories()
[Category of filtered algebras over Integer Ring, Category of graded modules over Integer Ring]
```

**class** `ElementMethods`

**Bases:** `object`

**class** `ParentMethods`

**Bases:** `object`

**graded_algebra()**

Return the associated graded algebra to `self`.

Since `self` is already graded, this just returns `self`.

**EXAMPLES:**
class SignedTensorProducts(category, *args)

Bases: SignedTensorProductsCategory

extra_super_categories()

EXAMPLES:

```python
sage: Algebras(QQ).Graded().SignedTensorProducts().extra_super_categories()
[Category of graded algebras over Rational Field]
sage: Algebras(QQ).Graded().SignedTensorProducts().super_categories()
[Category of graded algebras over Rational Field]
```

Meaning: a signed tensor product of algebras is an algebra

class SubcategoryMethods

Bases: object

SignedTensorProducts()

Return the full subcategory of objects of self constructed as signed tensor products.

See also:

- SignedTensorProductsCategory
- CovariantFunctorialConstruction

EXAMPLES:

```python
sage: AlgebrasWithBasis(QQ).Graded().SignedTensorProducts()
Category of signed tensor products of graded algebras with basis over Rational Field
```

```python
>>> from sage.all import *
>>> AlgebrasWithBasis(QQ).Graded().SignedTensorProducts()
Category of signed tensor products of graded algebras with basis over Rational Field
```
### 4.75 Graded algebras with basis

**class** `sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis(base_category)`

**Bases:** `GradedModulesCategory`

The category of graded algebras with a distinguished basis

**EXAMPLES:**

```
sage: C = GradedAlgebrasWithBasis(ZZ); C
Category of graded algebras with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of filtered algebras with basis over Integer Ring,
 Category of graded algebras over Integer Ring,
 Category of graded modules with basis over Integer Ring]
```

```
>>> from sage.all import *
>>> C = GradedAlgebrasWithBasis(ZZ); C
Category of graded algebras with basis over Integer Ring
>>> sorted(C.super_categories(), key=str)
[Category of filtered algebras with basis over Integer Ring,
 Category of graded algebras over Integer Ring,
 Category of graded modules with basis over Integer Ring]
```

**class** `ElementMethods`

**Bases:** `object`

**class** `ParentMethods`

**Bases:** `object`

**completion()**

Return the completion of all formal linear combinations of `self` with finite linear combinations in each homogeneous degree (computed lazily).

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete()
sage: L = S.formal_series_ring()
sage: L
Lazy completion of Non-Commutative Symmetric Functions over
the Rational Field in the Complete basis
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> S = NCSF.Complete()
>>> L = S.formal_series_ring()
>>> L
Lazy completion of Non-Commutative Symmetric Functions over
the Rational Field in the Complete basis
```

**formal_series_ring()**

Return the completion of all formal linear combinations of `self` with finite linear combinations in each homogeneous degree (computed lazily).
EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete()
sage: L = S.formal_series_ring()
sage: L
Lazy completion of Non-Commutative Symmetric Functions over
the Rational Field in the Complete basis
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> NCSF = NonCommutativeSymmetricFunctions(QQ)
>>> S = NCSF.Complete()
>>> L = S.formal_series_ring()
>>> L
Lazy completion of Non-Commutative Symmetric Functions over
the Rational Field in the Complete basis
```

**free_graded_module** *(generator_degrees, names=None)*

Create a finitely generated free graded module over self

**INPUT:**

- `generator_degrees` – tuple of integers defining the number of generators of the module and their degrees
- `names` – (optional) the names of the generators. If names is a comma-separated string like 'a, b, c', then those will be the names. Otherwise, for example if names is abc, then the names will be abc[d,i].

By default, if all generators are in distinct degrees, then the names of the generators will have the form g[d] where d is the degree of the generator. If the degrees are not distinct, then the generators will be called g[d,i] where d is the degree and i is its index in the list of generators in that degree.

See sage.modules.fp_graded.free_module for more examples and details.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])
sage: Cl = CliffordAlgebra(Q)
sage: M = Cl.free_graded_module((0, 2, 3))
sage: M.gens()
(g[0], g[2], g[3])
sage: N.<xy, z> = Cl.free_graded_module((1, 2)); (xy, z) = N._first_ngens(2)
sage: N.generators()
(xy, z)
```
graded_algebra()

Return the associated graded algebra to self.

This is self, because self is already graded. See graded_algebra() for the general behavior of this method, and see AssociatedGradedAlgebra for the definition and properties of associated graded algebras.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
    # needs sage.combinat sage.modules
sage: m.graded_algebra() is m
    # needs sage.combinat sage.modules
True
```

```
>>> from sage.all import *
>>> m = SymmetricFunctions(QQ).m()
    # needs sage.combinat sage.modules
>>> m.graded_algebra() is m
    # needs sage.combinat sage.modules
True
```

class SignedTensorProducts (category, *args)

Bases: SignedTensorProductsCategory

The category of algebras with basis constructed by signed tensor product of algebras with basis.

class ParentMethods

Bases: object

Implements operations on tensor products of super algebras with basis.

one_basis()

Return the index of the one of this signed tensor product of algebras, as per
AlgebrasWithBasis.ParentMethods.one_basis.

It is the tuple whose operands are the indices of the ones of the operands, as returned by their one_basis() methods.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: A.<x,y> = ExteriorAlgebra(QQ)
sage: A.one_basis()
0
sage: B = tensor((A, A, A))
sage: B.one_basis()
(0, 0, 0)
sage: B.one()
1 # 1 # 1
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = ExteriorAlgebra(QQ, names=('x', 'y'))
>>> (x, y) = A._first_  
   # needs sage.combinat sage.modules
>>> A.one_basis()
0
>>> B = tensor((A, A, A))
```

(continues on next page)
product_on_basis($t_0, t_1$)

The product of the algebra on the basis, as per `AlgebrasWithBasis.ParentMethods.product_on_basis`.

EXAMPLES:

Test the sign in the super tensor product:

```python
sage: # needs sage.combinat sage.modules
sage: A = SteenrodAlgebra(3)
sage: x = A.Q(0)
sage: y = x.coproduct()
sage: y^2
0
```

TODO: optimize this implementation!

extra_super_categories()

EXAMPLES:

```python
sage: Cat = AlgebrasWithBasis(QQ).Graded()
sage: Cat.SignedTensorProducts().extra_super_categories()
[Category of graded algebras with basis over Rational Field]
sage: Cat.SignedTensorProducts().super_categories()
[Category of graded algebras with basis over Rational Field,
  Category of signed tensor products of graded algebras over Rational Field]
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.combinat sage.modules
>>> A = SteenrodAlgebra(Integer(3))
>>> x = A.Q(Integer(0))
>>> y = x.coproduct()
>>> y**Integer(2)
0
```
4.76 Graded bialgebras

`sage.categories.graded_bialgebras.GradedBialgebras(base_ring)`

The category of graded bialgebras

```python
sage: C = GradedBialgebras(QQ); C
Join of Category of graded algebras over Rational Field
    and Category of bialgebras over Rational Field
    and Category of graded coalgebras over Rational Field
sage: C is Bialgebras(QQ).Graded()
True
```

```python
>>> from sage.all import *
>>> C = GradedBialgebras(QQ); C
Join of Category of graded algebras over Rational Field
    and Category of bialgebras over Rational Field
    and Category of graded coalgebras over Rational Field
>>> C is Bialgebras(QQ).Graded()
True
```

4.77 Graded bialgebras with basis

`sage.categories.graded_bialgebras_with_basis.GradedBialgebrasWithBasis(base_ring)`

The category of graded bialgebras with a distinguished basis

```python
sage: C = GradedBialgebrasWithBasis(QQ); C
Join of Category of graded algebras with basis over Rational Field
    and Category of bialgebras over Rational Field
    and Category of graded coalgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
 Category of graded algebras with basis over Rational Field,
 Category of graded coalgebras with basis over Rational Field]
```

```python
>>> from sage.all import *
>>> C = GradedBialgebrasWithBasis(QQ); C
Join of Category of graded algebras with basis over Rational Field
    and Category of bialgebras over Rational Field
    and Category of graded coalgebras with basis over Rational Field
>>> sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
 Category of graded algebras with basis over Rational Field,
 Category of graded coalgebras with basis over Rational Field]
```
4.78 Graded Coalgebras

class sage.categories.graded_coalgebras.GradedCoalgebras(base_category)

Bases: GradedModulesCategory

The category of graded coalgebras

EXAMPLES:

sage: C = GradedCoalgebras(QQ); C
Category of graded coalgebras over Rational Field
sage: C is Coalgebras(QQ).Graded()
True

class SignedTensorProducts(category, *args)

Bases: SignedTensorProductsCategory

eextra_super_categories()

EXAMPLES:

sage: Coalgebras(QQ).Graded().SignedTensorProducts().extra_super_categories()
[Category of graded coalgebras over Rational Field]
sage: Coalgebras(QQ).Graded().SignedTensorProducts().super_categories()
[Category of graded coalgebras over Rational Field]

Meaning: a signed tensor product of coalgebras is a coalgebra

class SubcategoryMethods

Bases: object

SignedTensorProducts()

Return the full subcategory of objects of self constructed as signed tensor products.

See also:

• SignedTensorProductsCategory
  • CovariantFunctorialConstruction

EXAMPLES:

sage: CoalgebrasWithBasis(QQ).Graded().SignedTensorProducts()
Category of signed tensor products of graded coalgebras with basis over Rational Field
4.79 Graded coalgebras with basis

class sage.categories.graded_coalgebras_with_basis.GradedCoalgebrasWithBasis(base_category)

Bases: GradedModulesCategory

The category of graded coalgebras with a distinguished basis.

EXAMPLES:

sage: C = GradedCoalgebrasWithBasis(QQ); C
Category of graded coalgebras with basis over Rational Field
sage: C is Coalgebras(QQ).WithBasis().Graded()
True

class SignedTensorProducts(category, *args)

Bases: SignedTensorProductsCategory

The category of coalgebras with basis constructed by signed tensor product of coalgebras with basis.

extra_super_categories()

EXAMPLES:

sage: Cat = CoalgebrasWithBasis(QQ).Graded()
sage: Cat.SignedTensorProducts().extra_super_categories() [Category of graded coalgebras with basis over Rational Field, Category of signed tensor products of graded coalgebras over Rational Field]
4.80 Graded Hopf algebras

sage.categories.graded_hopf_algebras.GradedHopfAlgebras(base_ring)

The category of graded Hopf algebras.

EXAMPLES:

```
sage: C = GradedHopfAlgebras(QQ); C
Join of Category of Hopf algebras over Rational Field and Category of graded algebras over Rational Field and Category of graded coalgebras over Rational Field
sage: C is HopfAlgebras(QQ).Graded()
True
```

Note: This is not a graded Hopf algebra as is typically defined in algebraic topology as the product in the tensor square \((x \otimes y)(a \otimes b) = (xa) \otimes (yb)\) does not carry an additional sign. For this, instead use super Hopf algebras.

4.81 Graded Hopf algebras with basis

class sage.categories.graded_hopf_algebras_with_basis.GradedHopfAlgebrasWithBasis(base_category)

Bases: GradedModulesCategory

The category of graded Hopf algebras with a distinguished basis.

EXAMPLES:

```
sage: C = GradedHopfAlgebrasWithBasis(ZZ); C
Category of graded Hopf algebras with basis over Integer Ring
sage: C.super_categories()
[Category of filtered Hopf algebras with basis over Integer Ring, Category of graded algebras with basis over Integer Ring, Category of graded coalgebras with basis over Integer Ring]
sage: C is HopfAlgebras(ZZ).WithBasis().Graded()
True
sage: C is HopfAlgebras(ZZ).Graded().WithBasis()
False
```

```
Category of graded Hopf algebras with basis over Integer Ring

```python
>>> C.super_categories()
[Category of filtered Hopf algebras with basis over Integer Ring,
 Category of graded algebras with basis over Integer Ring,
 Category of graded coalgebras with basis over Integer Ring]
```

```python
>>> C is HopfAlgebras(ZZ).WithBasis().Graded()
True
>>> C is HopfAlgebras(ZZ).Graded().WithBasis()
False
```

class **Connected** *(base_category)*

**Bases:** `CategoryWithAxiom_over_base_ring`

class **ElementMethods**

**Bases:** `object`

class **ParentMethods**

**Bases:** `object`

**antipode_on_basis** *(index)*

The antipode on the basis element indexed by `index`.

**INPUT:**

- `index` – an element of the index set

For a graded connected Hopf algebra, we can define an antipode recursively by

\[ S(x) := - \sum_{x \neq x'} S(x') \times x'' \]

when \(|x| > 0\), and by \(S(x) = x\) when \(|x| = 0\).

**counit_on_basis** *(i)*

The default counit of a graded connected Hopf algebra.

**INPUT:**

- `i` – an element of the index set

**OUTPUT:**

- an element of the base ring

\[ c(i) := \begin{cases} 
1 & \text{if } i \text{ indexes the 1 of the algebra} \\
0 & \text{otherwise.} 
\end{cases} \]

**EXAMPLES:**

```python
sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()  # needs sage.modules
sage: H.monomial(4).counit()  # indirect doctest  # needs sage.modules
0
sage: H.monomial(0).counit()  # indirect doctest  # needs sage.modules
1
```

```python
>>> from sage.all import *
>>> H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()  # needs sage.modules
```

(continues on next page)
example()

Return an example of a graded connected Hopf algebra with a distinguished basis.

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

class WithRealizations(category, *args)
    Bases: WithRealizationsCategory

    super_categories()

    EXAMPLES:

    sage: GradedHopfAlgebrasWithBasis(QQ).WithRealizations().super_categories()
    [Join of Category of Hopf algebras over Rational Field
    and Category of graded algebras over Rational Field
    and Category of graded coalgebras over Rational Field]

example()

Return an example of a graded Hopf algebra with a distinguished basis.

4.82 Graded Lie Algebras

AUTHORS:

• Eero Hakavuori (2018-08-16): initial version

class sage.categories.graded_lie_algebras.GradedLieAlgebras(base_category)
    Bases: GradedModulesCategory

Category of graded Lie algebras.

class Stratified(base_category)
    Bases: CategoryWithAxiom_over_base_ring

Category of stratified Lie algebras.

A graded Lie algebra \( L = \bigoplus_{k \geq 1} L_k \) (where possibly \( M = \infty \)) is called \textit{stratified} if it is generated by \( L_1 \); in other words, we have \( L_{k+1} = [L_1, L_k] \).
**class FiniteDimensional** *(base_category)*

**Bases:** CategoryWithAxiom_over_base_ring

Category of finite dimensional stratified Lie algebras.

**EXAMPLES:**

```
sage: LieAlgebras(QQ).Graded().Stratified().FiniteDimensional()
Category of finite dimensional stratified Lie algebras over Rational Field
```

**extra_super_categories()**

Implements the fact that a finite dimensional stratified Lie algebra is nilpotent.

**EXAMPLES:**

```
sage: C = LieAlgebras(QQ).Graded().Stratified().FiniteDimensional()
sage: C.extra_super_categories()
[Category of nilpotent Lie algebras over Rational Field]
sage: C is C.Nilpotent()
True
sage: C.is_subcategory(LieAlgebras(QQ).Nilpotent())
True
```

**class SubcategoryMethods**

**Bases:** object

**Stratified()**

Return the full subcategory of stratified objects of self.

A Lie algebra is stratified if it is graded and generated as a Lie algebra by its component of degree one.

**EXAMPLES:**

```
sage: LieAlgebras(QQ).Graded().Stratified()
Category of stratified Lie algebras over Rational Field
```

```
4.83 Graded Lie Algebras With Basis

class sage.categories.graded_lie_algebras_with_basis.GradedLieAlgebrasWithBasis(base_category):

    Bases: GradedModulesCategory

    The category of graded Lie algebras with a distinguished basis.

    EXAMPLES:

    sage: C = LieAlgebras(ZZ).WithBasis().Graded(); C
    Category of graded Lie algebras with basis over Integer Ring
    sage: C.super_categories()
    [Category of graded modules with basis over Integer Ring,
     Category of Lie algebras with basis over Integer Ring,
     Category of graded Lie algebras over Integer Ring]
    sage: C is LieAlgebras(ZZ).WithBasis().Graded()
    True
    sage: C is LieAlgebras(ZZ).Graded().WithBasis()
    False

>>> from sage.all import *

>>> C = LieAlgebras(ZZ).WithBasis().Graded(); C
Category of graded Lie algebras with basis over Integer Ring
>>> C.super_categories()
[Category of graded modules with basis over Integer Ring,
 Category of Lie algebras with basis over Integer Ring,
 Category of graded Lie algebras over Integer Ring]

>>> C is LieAlgebras(ZZ).WithBasis().Graded()
True
>>> C is LieAlgebras(ZZ).Graded().WithBasis()
False

FiniteDimensional
    alias of FiniteDimensionalGradedLieAlgebrasWithBasis

4.84 Graded Lie Conformal Algebras

AUTHORS:


class sage.categories.graded_lie_conformal_algebras.GradedLieConformalAlgebras(base_category):

    Bases: GradedLieConformalAlgebrasCategory

    The category of graded Lie conformal algebras.

    EXAMPLES:
class sage.categories.graded_lie_conformal_algebras.GradedLieConformalAlgebrasCategory(base_category)

Bases: GradedModulesCategory

Super (base_ring=None)

Return the super-analogue category of self.

INPUT:

• base_ring – this is ignored

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: C = LieConformalAlgebras(QQbar).Graded(); C
Category of H-graded Lie conformal algebras over Algebraic Field

sage: C is LieConformalAlgebras(QQ).Super().Graded()
True
```

```python
>>> from sage.all import *

>>> C = LieConformalAlgebras(QQbar).Graded(); C
Category of H-graded Lie conformal algebras over Algebraic Field

>>> C is LieConformalAlgebras(QQ).Super().Graded()
True
```

```python
>>> from sage.all import *

>>> C = LieConformalAlgebras(QQ).Super().Graded()
```

```python
>>> C is LieConformalAlgebras(QQ).Super().Graded()
True
```
4.85 Graded modules

class sage.categories.graded_modules.GradedModules(base_category)
   Bases: GradedModulesCategory

   The category of graded modules.
   We consider every graded module $M = \bigoplus_i M_i$ as a filtered module under the (natural) filtration given by
   $$F_i = \bigoplus_{j<i} M_j.$$ 

   EXAMPLES:

   sage: GradedModules(ZZ)
   Category of graded modules over Integer Ring
   sage: GradedModules(ZZ).super_categories()
   [Category of filtered modules over Integer Ring]

   The category of graded modules defines the graded structure which shall be preserved by morphisms:

   sage: Modules(ZZ).Graded().additional_structure()
   Category of graded modules over Integer Ring

   >>> from sage.all import *
   >>> Modules(ZZ).Graded().additional_structure()
   Category of graded modules over Integer Ring

   class ElementMethods
       Bases: object

   class ParentMethods
       Bases: object

class sage.categories.graded_modules.GradedModulesCategory(base_category)
   Bases: RegressiveCovariantConstructionCategory, Category_over_base_ring

   EXAMPLES:

   sage: C = GradedAlgebras(QQ)
   sage: C
   Category of graded algebras over Rational Field
   sage: C.base_category()
   Category of algebras over Rational Field
   sage: sorted(C.super_categories(), key=str)
   [Category of filtered algebras over Rational Field, Category of graded vector spaces over Rational Field]
   sage: AlgebrasWithBasis(QQ).Graded().base_ring()
   Rational Field
   sage: GradedHopfAlgebrasWithBasis(QQ).base_ring()
   Rational Field
```python
>>> from sage.all import *
>>> C = GradedAlgebras(QQ)
>>> C
Category of graded algebras over Rational Field
>>> C.base_category()
Category of algebras over Rational Field
>>> sorted(C.super_categories(), key=str)
[Category of filtered algebras over Rational Field,
 Category of graded vector spaces over Rational Field]
```

```python
class method default_super_categories(category, *args)

Return the default super categories of category.Graded().

Mathematical meaning: every graded object (module, algebra, etc.) is a filtered object with the (implicit) filtration defined by $F_i = \bigoplus_{j \leq i} G_j$.

INPUT:

- cls – the class GradedModulesCategory
- category – a category

OUTPUT: a (join) category

In practice, this returns category.Filtered(), joined together with the result of the method RegressiveCovariantConstructionCategory.default_super_categories() (that is the join of category.Filtered() and cat for each cat in the super categories of category).

EXAMPLES:

Consider category=Algebras(), which has cat=Modules() as super category. Then, a grading of an algebra $G$ is also a filtration of $G$:

```python
sage: Algebras(QQ).Graded().super_categories()
[Category of filtered algebras over Rational Field,
 Category of graded vector spaces over Rational Field]
```

This resulted from the following call:

```python
sage: sage.categories.graded_modules.GradedModulesCategory.default_super_categories().
Join of Category of filtered algebras over Rational Field
and Category of graded vector spaces over Rational Field
```

4.86 Graded modules with basis

**class** sage.categories.graded_modules_with_basis.GradedModulesWithBasis(base_category)

**Bases:** GradedModulesCategory

The category of graded modules with a distinguished basis.

**EXAMPLES:**

```python
sage: C = GradedModulesWithBasis(ZZ); C
Category of graded modules with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of filtered modules with basis over Integer Ring,
 Category of graded modules over Integer Ring]
sage: C is ModulesWithBasis(ZZ).Graded()
True
```

**class** ElementMethods

**Bases:** object

**degree_negation()**

Return the image of self under the degree negation automorphism of the graded module to which self belongs.

The degree negation is the module automorphism which scales every homogeneous element of degree \( k \) by \( (-1)^k \) (for all \( k \)). This assumes that the module to which self belongs (that is, the module self.parent()) is \( \mathbb{Z} \)-graded.

**EXAMPLES:**

```python
sage: E.<a,b> = ExteriorAlgebra(QQ)  # needs sage.modules
sage: ((1 + a) * (1 + b)).degree_negation()  # needs sage.modules
a*b - a - b + 1
sage: E.zero().degree_negation()  # needs sage.modules
0
```

```
sage: P = GradedModulesWithBasis(ZZ).example(); P  # needs sage.combinat sage.modules
An example of a graded module with basis:
the free module on partitions over Integer Ring
sage: pbp = lambda x: P.basis()[Partition(list(x))]
sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])  # needs sage.combinat sage.modules
sage: p.degree_negation()  # needs sage.modules
```

(continues on next page)
### Class `ParentMethods`

**Bases:** object

**degree_negation**(element)

Return the image of element under the degree negation automorphism of the graded module self.

The degree negation is the module automorphism which scales every homogeneous element of degree \( k \) by \((-1)^k\) (for all \( k \)). This assumes that the module self is \( \mathbb{Z} \)-graded.

**INPUT:**
- element - element of the module self

**EXAMPLES:**

```python
sage: E.<a,b> = ExteriorAlgebra(QQ)  # needs sage.combinat sage.modules
sage: E.degree_negation((1 + a) * (1 + b))  # needs sage.combinat sage.modules
a*b - a - b + 1
sage: E.degree_negation(E.zero())  # needs sage.combinat sage.modules
0
sage: P = GradedModulesWithBasis(ZZ).example(); P  # needs sage.combinat sage.modules
An example of a graded module with basis:
the free module on partitions over Integer Ring
sage: pbp = lambda x: P.basis()[Partition(list(x))]  # needs sage.combinat sage.modules
sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])  # needs sage.combinat sage.modules
sage: P.degree_negation(p)  # needs sage.combinat sage.modules
```
```python
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('a', 'b', )); (a, b,) = E._first_ngens(2)  # needs sage.modules
>>> E.degree_negation((Integer(1) + a) * (Integer(1) + b))  # needs sage.modules
a*b - a - b + 1
>>> E.degree_negation(E.zero());  # needs sage.modules
0

>>> P = GradedModulesWithBasis(ZZ).example(); P  # needs sage.combinat sage.modules
An example of a graded module with basis:
the free module on partitions over Integer Ring

>>> pbp = lambda x: P.basis()[Partition(list(x))]

>>> p = pbp([Integer(3),Integer(1)]) - Integer(2) * pbp([Integer(2)]) + Integer(4) * pbp([Integer(1)])  # needs sage.combinat sage.modules

>>> P.degree_negation(p);  # needs sage.combinat sage.modules
```

**quotient_module** *(submodule, check=True, already_echelonized=False, category=None)*

Construct the quotient module $self / submodule$.

**INPUT:**

- **submodule** — a submodule with basis of $self$, or something that can be turned into one via $self.submodule(submodule)$
- **check, already_echelonized** — passed down to $ModulesWithBasis.ParentMethods.submodule()$
- **category** — (optional) the category of the quotient module

**Warning:** At this point, this only supports quotients by free submodules admitting a basis in unitriangular echelon form. In this case, the quotient is also a free module, with a basis consisting of the retract of a subset of the basis of $self$.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: S = E.submodule([x + y, x*y - y*z, y])
sage: Q = E.quotient_module(S)
sage: Q.category()
Join of
Category of quotients of graded modules with basis over Rational Field

                and
Category of graded vector spaces with basis over Rational Field and
Category of finite dimensional filtered modules with basis over Rational
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z', )); (x, y, z,) = E._first_ngens(3)
>>> S = E.submodule([x + y, x*y - y*z, y])
```

(continues on next page)
>>> Q = E.quotient_module(S)

Join of
Category of quotients of graded modules with basis over Rational Field
-> and
Category of graded vector spaces with basis over Rational Field and
Category of finite dimensional filtered modules with basis over Rational Field

See also:
• Modules.WithBasis.ParentMethods.submodule()
• Rings.ParentMethods.quotient()
• sage.modules.with_basis.subquotient.QuotientModuleWithBasis

submodule(gens, check=True, already_echelonized=False, unitriangular=False, support_order=None, category=None, *args, **opts)

Return the submodule spanned by a finite set of elements.

INPUT:
• gens – a list or family of elements of self
• check – (default: True) whether to verify that the
elements of gens are in self
• already_echelonized – (default: False) whether
  the elements of gens are already in (not necessarily reduced) echelon form
• unitriangular – (default: False) whether the lift morphism is unitriangular
• support_order – (optional) either something that can be converted into a tuple or a key function
• category – (optional) the category of the submodule

If already_echelonized is False, then the generators are put in reduced echelon form using
echelonize(), and reindexed by 0, 1, ...

Warning: At this point, this method only works for finite dimensional submodules and if matrices
can be echelonized over the base ring.

If in addition unitriangular is True, then the generators are made such that the coefficients of the
pivots are 1, so that lifting map is unitriangular.

The basis of the submodule uses the same index set as the generators, and the lifting map sends $y_i$ to
gens[i].

See also:
  quotient_module()
• sage.modules.with_basis.subquotient.SubmoduleWithBasis

EXAMPLES:
A graded submodule of a graded module generated by homogeneous elements is naturally graded:

sage: # needs sage.combinat sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: S = E.submodule([x + y, x*y - y*z])
sage: S.category()
Join of
Category of graded vector spaces with basis over Rational Field and

(continues on next page)
Category of subobjects of filtered modules with basis over Rational
→ Field and
Category of finite dimensional filtered modules with basis over Rational
→ Field
sage: S.basis()[0].degree()
1
sage: S.basis()[1].degree()
2

We check on the echelonized basis:

sage: Sp = E.submodule([1, x + y + 5, x*y - y*z + x + y - 2])  # needs sage.combinat sage.modules
sage: Sp.category()  # needs sage.combinat sage.modules
Join of
Category of graded vector spaces with basis over Rational Field and
Category of subobjects of filtered modules with basis over Rational
→ Field and
Category of finite dimensional filtered modules with basis over Rational
→ Field

If it is generated by inhomogeneous elements, then it is filtered by default:

sage: F = E.submodule([x + y*z, x*z + y*x])  # needs sage.combinat sage.modules
sage: F.category()  # needs sage.combinat sage.modules
(continues on next page)
Join of Category of subobjects of filtered modules with basis over Rational Field and Category of finite dimensional filtered modules with basis over Rational Field and Category of filtered vector spaces with basis over Rational Field

If `category` is specified, then it does not give any extra structure to the submodule (we can think of this as applying the forgetful functor):

```python
sage: # needs sage.combinat sage.modules
sage: SM = E.submodule([x + y, x*y - y*z],
...:                     category=ModulesWithBasis(QQ))
sage: SM.category()
Join of Category of finite dimensional vector spaces with basis over Rational Field and Category of subobjects of sets
```

```python
sage: FM = E.submodule([x + Integer(1), x*y - x*y*z],
...:                     category=ModulesWithBasis(QQ))
sage: FM.category()
Join of Category of finite dimensional vector spaces with basis over Rational Field and Category of subobjects of sets
```

If we have specified that this is a graded submodule of a graded module, then the echelonized elements
must be homogeneous:

```python
sage: Cat = ModulesWithBasis(QQ).Graded().Subobjects()
sage: E.submodule([x + y, x*y - 1], category=Cat)  # needs sage.combinat sage.modules
Traceback (most recent call last):
... ValueError: all of the generators must be homogeneous
sage: E.submodule([x + y, x*y - x - y], category=Cat)  # needs sage.combinat sage.modules
Free module generated by {0, 1} over Rational Field
```

```python
>>> from sage.all import *

>>> Cat = ModulesWithBasis(QQ).Graded().Subobjects()
>>> E.submodule([x + y, x*y - Integer(1)], category=Cat)  # needs sage.combinat sage.modules
Traceback (most recent call last):
... ValueError: all of the generators must be homogeneous
>>> E.submodule([x + y, x*y - x - y], category=Cat)  # needs sage.combinat sage.modules
Free module generated by {0, 1} over Rational Field
```

### class `Quotients` (category, *args)

**Bases:** `QuotientsCategory`

#### class `ElementMethods`

**Bases:** object

**degree()**

Return the degree of `self`.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: S = E.submodule([x + y, x*y - y*z, y])
sage: Q = E.quotient_module(S)
sage: B = Q.basis()

sage: [B[i].lift() for i in Q.indices()]
[1, z, x*z, y*z, x*y*z]

sage: [B[i].degree() for i in Q.indices()]
[0, 1, 2, 2, 3]
```

```python
>>> from sage.all import *

>>> # needs sage.combinat sage.modules

>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z'))
>>> (x, y, z,) = E._first_ngens(3)

>>> S = E.submodule([x + y, x*y - y*z, y])

>>> Q = E.quotient_module(S)

>>> B = Q.basis()

>>> [B[i].lift() for i in Q.indices()]
[1, z, x*z, y*z, x*y*z]

>>> [B[i].degree() for i in Q.indices()]
[0, 1, 2, 2, 3]
```

### class `ParentMethods`

**Bases:** object

4.86. Graded modules with basis
degree_on_basis(m)

Return the degree of the basis element indexed by m in self.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
```

```python
sage: S = E.submodule([x + y, x*y - y*z, y])
```

```python
sage: Q = E.quotient_module(S)
```

```python
sage: B = Q.basis()
```

```python
sage: [B[i].lift() for i in Q.indices()]
```

```python
[1, z, x*z, y*z, x*y*z]
```

```python
sage: [Q.degree_on_basis(i) for i in Q.indices()]
```

```python
[0, 1, 2, 2, 3]
```

4.87 Graphs

```python
>>> from sage.all import *
```

```python
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E.
```

```python
>>> S = E.submodule([x + y, x*y - y*z, y])
```

```python
>>> Q = E.quotient_module(S)
```

```python
>>> B = Q.basis()
```

```python
>>> [B[i].lift() for i in Q.indices()]
```

```python
[1, z, x*z, y*z, x*y*z]
```

```python
>>> [Q.degree_on_basis(i) for i in Q.indices()]
```

```python
[0, 1, 2, 2, 3]
```

class sage.categories.graphs.Graphs

Bases: Category_singleton

The category of graphs.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs(); C
Category of graphs
```

```python
>>> from sage.all import *
```

```python
>>> from sage.categories.graphs import Graphs
```

```python
>>> C = Graphs(); C
Category of graphs
```

class Connected(base_category)

Bases: CategoryWithAxiom

The category of connected graphs.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().Connected()
```

```python
sage: TestSuite(C).run()
```
```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().Connected()
>>> TestSuite(C).run()

```extra_super_categories()

Return the extra super categories of self.

A connected graph is also a metric space.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: Graphs().Connected().super_categories()  # indirect doctest
(Category of connected topological spaces,
 Category of connected simplicial complexes,
 Category of graphs,
 Category of metric spaces)

```class ParentMethods

Bases: object

dimension()

Return the dimension of self as a CW complex.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.dimension()
1

```edges()

Return the edges of self.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.edges()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```
```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().example()
>>> C.edges()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

**faces()**

Return the faces of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: sorted(C.faces(), key=lambda x: (x.dimension(), x.value))
[0, 1, 2, 3, 4, (0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().example()
>>> sorted(C.faces(), key=lambda x: (x.dimension(), x.value))
[0, 1, 2, 3, 4, (0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

**facets()**

Return the facets of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.facets()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().example()
>>> C.facets()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

**vertices()**

Return the vertices of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.vertices()
[0, 1, 2, 3, 4]
```

```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().example()
>>> C.vertices()
[0, 1, 2, 3, 4]
```

**super_categories()**

**EXAMPLES:**
4.88 Group Algebras

This module implements the category of group algebras for arbitrary groups over arbitrary commutative rings. For details, see `sage.categories.algebra_functor`.

AUTHOR:

- David Loeffler (2008-08-24): initial version
- John Palmieri (2011-07): more updates to coercion, categories, etc., group algebras constructed using CombinatorialFreeModule – see Issue #6670.
- Nicolas M. Thiéry (2010-2017), Travis Scrimshaw (2017): generalization to a covariant functorial construction for monoid algebras, and beyond – see e.g. Issue #18700.

```python
class sage.categories.group_algebras.GroupAlgebras(category, *args)
```

Bases: `AlgebrasCategory`

The category of group algebras over a given base ring.

EXAMPLES:

```python
sage: C = Groups().Algebras(ZZ); C
Category of group algebras over Integer Ring
sage: C.super_categories()
(Category of Hopf algebras with basis over Integer Ring,
 Category of monoid algebras over Integer Ring)
```

We can also construct this category with:

```python
sage: C is GroupAlgebras(ZZ)
True
```

Here is how to create the group algebra of a group $G$: 
sage: G = DihedralGroup(5)  # needs sage.groups
sage: QG = G.algebra(QQ); QG  # needs sage.groups sage.modules
Algebra of Dihedral group of order 10 as a permutation group over Rational Field

and an example of computation:

sage: g = G.an_element(); g  # needs sage.groups sage.modules
(1,4)(2,3)
sage: (QG.term(g) + 1)**3  # needs sage.groups sage.modules
4*() + 4*(1,4)(2,3)

Todo:

• Check which methods would be better located in Monoid.Algebras or Groups.Finite.
  Algebras.

class ElementMethods
    Bases: object

central_form()
    Return self expressed in the canonical basis of the center of the group algebra.
    INPUT:
    • self – an element of the center of the group algebra
    OUTPUT:
    • A formal linear combination of the conjugacy class representatives representing its coordi-
      center_basis() for details.

Warning:

• This method requires the underlying group to have a method conjugacy_classes_repre-
  sentatives (every permutation group has one, thanks GAP!).
• This method does not check that the element is indeed central. Use the method `Monoids.Algebras.ElementMethods.is_central()` for this purpose.

• This function has a complexity linear in the number of conjugacy classes of the group. One could easily implement a function whose complexity is linear in the size of the support of `self`.

EXAMPLES:

```python
sage: # needs sage.combinat sage.groups sage.modules
sage: QS3 = SymmetricGroup(3).algebra(QQ)
sage: A = QS3([2,3,1]) + QS3([3,1,2])
sage: A.central_form()
B[(1,2,3)]
sage: QS4 = SymmetricGroup(4).algebra(QQ)
sage: B = sum(len(s.cycle_type()) * QS4(s) for s in Permutations(4))
sage: B.central_form()
4*B[()] + 3*B[(1,2)] + 2*B[(1,2)(3,4)] + 2*B[(1,2,3)] + B[(1,2,3,4)]
```

The following test fails due to a bug involving combinatorial free modules and the coercion system (see Issue #28544):

```python
sage: # needs sage.groups sage.modules
sage: G = PermutationGroup([(1,2,3),(4,5)], [(3,4)])
sage: QG = GroupAlgebras(QQ).example(G)
sage: s = sum(QG.basis())
sage: s.central_form()  # not tested
B[()] + B[(4,5)] + B[(3,4,5)] + B[(2,3)(4,5)] + B[(2,3,4,5)] + B[(1,2)(3,4,5)] + B[(1,2,3,4,5)]
```

```python
>>> from sage.all import *

sage: G = PermutationGroup([(Integer(1),Integer(2),Integer(3)), (Integer(4), Integer(5))])
sage: QG = GroupAlgebras(QQ).example(G)
sage: s = sum(QG.basis())
sage: s.central_form()  # not tested
B[()] + B[(4,5)] + B[(3,4,5)] + B[(2,3)(4,5)] + B[(2,3,4,5)] + B[(1,2)(3,4,5)] + B[(1,2,3,4,5)]
```

See also:

• `Groups.Algebras.ParentMethods.center_basis()`
• `Monoids.Algebras.ElementMethods.is_central()`
class ParentMethods
Bases: object

antipode_on_basis(g)

Return the antipode of the element g of the basis.

Each basis element g is group-like, and so has antipode \( g^{-1} \). This method is used to compute the antipode of any element.

EXAMPLES:

```python
sage: # needs sage.groups sage.modules
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
Algebra of Cyclic group of order 6 as a permutation group over Integer Ring
sage: g = CyclicPermutationGroup(6).an_element(); g
(1,2,3,4,5,6)
```

```python
sage: A.antipode_on_basis(g)
(1,6,5,4,3,2)
```

```python
sage: a = A.an_element(); a
() + 3*(1,2,3,4,5,6) + 3*(1,3,5)(2,4,6)
```

```python
sage: a.antipode()
() + 3*(1,5,3)(2,6,4) + 3*(1,6,5,4,3,2)
```

center_basis()

Return a basis of the center of the group algebra.

The canonical basis of the center of the group algebra is the family \( (f_\sigma)_{\sigma \in C} \), where \( C \) is any collection of representatives of the conjugacy classes of the group, and \( f_\sigma \) is the sum of the elements in the conjugacy class of \( \sigma \).

OUTPUT:

• tuple of elements of self

Warning:

• This method requires the underlying group to have a method `conjugacy_classes` (every permutation group has one, thanks GAP!).

EXAMPLES:

```python
sage: SymmetricGroup(3).algebra(QQ).center_basis()  # needs sage.combinat sage.groups sage.modules
((), (2,3) + (1,2) + (1,3), (1,2,3) + (1,3,2))
```
See also:

- Monoids.Algebras.ElementMethods.is_central()

**coproduct_on_basis(g)**

Return the coproduct of the element \( g \) of the basis.

Each basis element \( g \) is group-like. This method is used to compute the coproduct of any element.

**EXAMPLES:**

```python
sage: # needs sage.groups sage.modules
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
Algebra of
  Cyclic group of order 6 as a permutation group over Integer Ring
sage: g = CyclicPermutationGroup(6).an_element(); g
(1,2,3,4,5,6)
sage: A.coproduct_on_basis(g)
(1,2,3,4,5,6) # (1,2,3,4,5,6)
sage: a = A.an_element(); a
() + 3*(1,2,3,4,5,6) + 3*(1,3,5)(2,4,6)
sage: a.coproduct()
() # () + 3*(1,2,3,4,5,6) # (1,2,3,4,5,6) + 3*(1,3,5)(2,4,6) # (1,3,5)(2,4,6)
```

**counit(x)**

Return the counit of the element \( x \) of the group algebra.

This is the sum of all coefficients of \( x \) with respect to the standard basis of the group algebra.

**EXAMPLES:**

```python
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
# needs sage.groups sage.modules
sage: g = CyclicPermutationGroup(6).an_element(); g
(1,2,3,4,5,6)
sage: A.coproduct_on_basis(g)
(1,2,3,4,5,6) # (1,2,3,4,5,6)
sage: a = A.an_element(); a
() + 3*(1,2,3,4,5,6) + 3*(1,3,5)(2,4,6)
sage: a.coproduct()
() # () + 3*(1,2,3,4,5,6) # (1,2,3,4,5,6) + 3*(1,3,5)(2,4,6) # (1,3,5)(2,4,6)
```

(continues on next page)
counit_on_basis(g)

Return the counit of the element \( g \) of the basis.

Each basis element \( g \) is group-like, and so has counit 1. This method is used to compute the counit of any element.

EXAMPLES:

```
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
# needs sage.groups sage.modules
Algebra of
Cyclic group of order 6 as a permutation group over Integer Ring
sage: g = CyclicPermutationGroup(6).an_element(); g
# needs sage.groups sage.modules
(1,2,3,4,5,6)
sage: A.counit_on_basis(g)
# needs sage.groups sage.modules
1
```

group()

Return the underlying group of the group algebra.

EXAMPLES:

```
sage: GroupAlgebras(QQ).example(GL(3, GF(11))).group()
# needs sage.groups sage.modules
```

(continues on next page)
General Linear Group of degree 3 over Finite Field of size 11

```sage
sage: SymmetricGroup(10).algebra(QQ).group()  # needs sage.combinat sage.groups sage.modules
Symmetric group of order 10! as a permutation group
```

```python
>>> from sage.all import *

>>> GroupAlgebras(QQ).example(GL(Integer(3), GF(Integer(11)))).group()  # needs sage.groups sage.modules
General Linear Group of degree 3 over Finite Field of size 11
```

```sage
>>> SymmetricGroup(Integer(10)).algebra(QQ).group()  # needs sage.combinat sage.groups sage.modules
Symmetric group of order 10! as a permutation group
```

**is_integral_domain** *(proof=True)*

Return `True` if `self` is an integral domain.

This is false unless `self.base_ring()` is an integral domain, and even then it is false unless `self.group()` has nontrivial elements of finite order. I don’t know if this condition suffices, but it obviously does if the group is abelian and finitely generated.

**EXAMPLES:**

```sage
sage: # needs sage.groups sage.modules
sage: S2 = SymmetricGroup(2)
sage: GroupAlgebra(S2).is_integral_domain()  # needs sage.combinat
False
sage: S1 = SymmetricGroup(1)
sage: GroupAlgebra(S1).is_integral_domain()  # needs sage.combinat
True
sage: GroupAlgebra(S1, IntegerModRing(4)).is_integral_domain()  # needs sage.combinat
False
sage: GroupAlgebra(AbelianGroup(1)).is_integral_domain()  # not implemented
True
sage: GroupAlgebra(AbelianGroup(2, [0,2])).is_integral_domain()  # not implemented
False
sage: GroupAlgebra(GL(2, ZZ)).is_integral_domain()  # not implemented
False
```

```python
>>> from sage.all import *

>>> S2 = SymmetricGroup(Integer(2))

>>> GroupAlgebra(S2).is_integral_domain()  # needs sage.combinat
False

>>> S1 = SymmetricGroup(Integer(1))

>>> GroupAlgebra(S1).is_integral_domain()  # needs sage.combinat
True

>>> GroupAlgebra(S1, IntegerModRing(Integer(4))).is_integral_domain()  # needs sage.combinat
False
```

```sage
>>> GroupAlgebra(AbelianGroup(Integer(1))).is_integral_domain()  # not implemented
True
```

(continues on next page)
example \((G=\text{None})\)

Return an example of group algebra.

EXAMPLES:

```python
sage: GroupAlgebras(QQ['x']).example()  
Algebra of Dihedral group of order 8 as a permutation group over Univariate Polynomial Ring in x over Rational Field
```

An other group can be specified as optional argument:

```python
sage: GroupAlgebras(QQ).example(AlternatingGroup(4))  
Algebra of Alternating group of order 4!/2 as a permutation group over Rational Field
```

extra_super_categories()

Implement the fact that the algebra of a group is a Hopf algebra.

EXAMPLES:

```python
sage: C = Groups().Algebras(QQ)
sage: C.extra_super_categories()  
[Category of Hopf algebras over Rational Field]
sage: sorted(C.super_categories(), key=str)  
[Category of Hopf algebras with basis over Rational Field,  
Category of monoid algebras over Rational Field]
```
4.89 Groupoid

class sage.categories.groupoid.Groupoid(G=None)

    Bases: CategoryWithParameters

    The category of groupoids, for a set (usually a group) \( G \).

    FIXME:
    
    • Groupoid or Groupoids?
    • definition and link with Wikipedia article Groupoid
    • Should Groupoid inherit from Category_over_base?

    EXAMPLES:

    sage: Groupoid(DihedralGroup(3))
    Groupoid with underlying set Dihedral group of order 6 as a permutation group

    >>> from sage.all import *
    >>> Groupoid(DihedralGroup(Integer(3)))
    Groupoid with underlying set Dihedral group of order 6 as a permutation group

    @classmethod
    def an_instance(self)
        
        Returns an instance of this class.

        EXAMPLES:

        sage: Groupoid.an_instance()
        # indirect doctest
        Groupoid with underlying set Symmetric group of order 8! as a permutation group

        >>> from sage.all import *
        >>> Groupoid.an_instance()
        # indirect doctest
        Groupoid with underlying set Symmetric group of order 8! as a permutation group

    super_categories(self)

        EXAMPLES:

        sage: Groupoid(DihedralGroup(3)).super_categories()
        [Category of sets]

        >>> from sage.all import *
        >>> Groupoid(DihedralGroup(Integer(3))).super_categories()
        [Category of sets]
4.90 Groups

class sage.categories.groups.Groups(base_category)
    Bases: CategoryWithAxiom_singleton

The category of (multiplicative) groups, i.e. monoids with inverses.

EXAMPLES:

sage: Groups()
Category of groups
sage: Groups().super_categories()
[Category of monoids, Category of inverse unital magmas]

Algebras
    alias of GroupAlgebras

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

The category of groups constructed as Cartesian products of groups.

This construction gives the direct product of groups. See Wikipedia article Direct_product and Wikipedia article Direct_product_of_groups for more information.

class ParentMethods
    Bases: object

    group_generators()

    Return the group generators of self.

    EXAMPLES:

sage: # needs sage.groups
sage: C5 = CyclicPermutationGroup(5)
sage: C4 = CyclicPermutationGroup(4)
sage: S4 = SymmetricGroup(3)
sage: C = cartesian_product([C5, C4, S4])
sage: C.group_generators()
Family (((1,2,3,4,5), (), ()),
          ((), (1,2,3,4), ()),
          ((), (), (1,2)),
          ((), (), (2,3))))

>>> from sage.all import *
>>> # needs sage.groups
>>> C5 = CyclicPermutationGroup(Integer(5))
>>> C4 = CyclicPermutationGroup(Integer(4))
>>> S4 = SymmetricGroup(Integer(3))
>>> C = cartesian_product([C5, C4, S4])
>>> C.group_generators()
(continues on next page)
We check the other portion of Issue #16718 is fixed:

```python
sage: len(C.j_classes())
needs sage.graphs sage.groups
1
```

```python
>>> from sage.all import *
```

```python
>>> len(C.j_classes())
needs sage.graphs sage.groups
1
```

An example with an infinitely generated group (a better output is needed):

```python
sage: # needs sage.combinat sage.groups
sage: G = Groups.free([1,2])
```

```python
sage: H = Groups.free(ZZ)
```

```python
sage: C = cartesian_product([G, H])
```

```python
sage: C.monoid_generators()
Lazy family (gen(i))_{i in The Cartesian product of (...)}
```

```python
>>> from sage.all import *
```

```python
>>> G = Groups.free([Integer(1),Integer(2)])
```

```python
>>> H = Groups.free(ZZ)
```

```python
>>> C = cartesian_product([G, H])
```

```python
>>> C.monoid_generators()
Lazy family (gen(i))_{i in The Cartesian product of (...)}
```

```python
order()
```

Returns the cardinality of self.

EXAMPLES:

```python
sage: C = cartesian_product([SymmetricGroup(10), SL(2, GF(3))])
needs sage.groups sage.modules
```

```python
sage: C.order()
needs sage.groups sage.modules
87091200
```

```python
>>> from sage.all import *
```

```python
>>> C = cartesian_product([SymmetricGroup(Integer(10)), SL(Integer(2), GF(Integer(3)))]
needs sage.groups sage.modules
```

```python
>>> C.order()
needs sage.groups sage.modules
87091200
```

Todo: this method is just here to prevent `FiniteGroups.ParentMethods` to call
`_cardinality_from_iterator`. 

4.90. Groups
**extra_super_categories()**

A Cartesian product of groups is endowed with a natural group structure.

EXAMPLES:

```
sage: C = Groups().CartesianProducts()
sage: C.extra_super_categories()  
[Category of groups]
sage: sorted(C.super_categories(), key=str)  
[Category of Cartesian products of inverse unital magmas,  
Category of Cartesian products of monoids,  
Category of groups]
```

```
>>> from sage.all import *
>>> C = Groups().CartesianProducts()
>>> C.extra_super_categories()  
[Category of groups]
```

---

```python
class Commutative(base_category):

Bases: CategoryWithAxiom

Category of commutative (abelian) groups.

A group $G$ is commutative if $xy = yx$ for all $x, y \in G$.

**static free** *(index_set=None, names=None, **kwds)*

Return the free commutative group.

**INPUT:**

- `index_set` – (optional) an index set for the generators; if an integer, then this represents \{0, 1, ..., $n-1$\}
- `names` – a string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix

**EXAMPLES:**

```
sage: # needs sage.combinat sage.groups
sage: Groups().Commutative().free()  
Free abelian group
sage: Groups().Commutative().free(index_set=ZZ)  
Free abelian group indexed by Integer Ring
sage: Groups().Commutative().free(5)  
Multiplicative Abelian group isomorphic to $Z \times Z \times Z \times Z \times Z$
```

```
>>> from sage.all import *
>>> Groups().Commutative().free()  
Free abelian group
```

```
>>> Groups().Commutative().free(index_set=ZZ)  
Free abelian group indexed by Integer Ring
```

```
>>> Groups().Commutative().free(5)  
Multiplicative Abelian group isomorphic to $Z \times Z \times Z \times Z \times Z$
```

```
>>> F.<x, y, z> = Groups().Commutative().free(); F  
Multiplicative Abelian group isomorphic to $Z \times Z \times Z$
```

(continues on next page)
class ElementMethods

Bases: object

conjugacy_class()

Return the conjugacy class of self.

EXAMPLES:

sage: D = DihedralGroup(5)  # needs sage.groups
sage: g = D((1,3,5,2,4))  # needs sage.groups
sage: g.conjugacy_class()  # needs sage.groups
Conjugacy class of (1,3,5,2,4)
in Dihedral group of order 10 as a permutation group

sage: H = MatrixGroup([matrix(GF(5), 2, [1,2, -1,1]),
                   matrix(GF(5), 2, [1,1, 0,1])])
sage: h = H(matrix(GF(5), 2, [1,2, -1,1]))
sage: h.conjugacy_class()  # needs sage.groups sage.modules
Conjugacy class of [1 2]
[4 1]
in Matrix group over Finite Field of size 5 with 2 generators ( [1 2] [1 1]
[4 1], [0 1] )

sage: G = SL(2, GF(2))  # needs sage.modules
sage: g = G.gens()[0]  # needs sage.groups sage.modules
sage: g.conjugacy_class()  # needs sage.groups sage.modules
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Finite Field of size 2

sage: G = SL(2, QQ)  # needs sage.modules
sage: g = G([[1,1], [0,1]])  # needs sage.modules
sage: g.conjugacy_class()  # needs sage.groups sage.modules
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Rational Field

>>> from sage.all import *
>>> D = DihedralGroup(Integer(5))  # needs sage.groups
>>> g = D((Integer(1),Integer(3),Integer(5),Integer(2),Integer(4)))  # needs sage.groups sage.modules
(continues on next page)
# needs sage.groups

```python
>>> g.conjugacy_class()
Conjugacy class of (1,3,5,2,4)
in Dihedral group of order 10 as a permutation group
```

```python
>>> H = MatrixGroup([matrix(GF(Integer(5)), Integer(2), [Integer(1),
-Integer(1), Integer(1), Integer(1)]),
-integer(2), -integer(1), Integer(1)])
>>> h = H(matrix(GF(Integer(5)), Integer(2), [Integer(1),
-Integer(1), Integer(0), Integer(1)]))
```

```python
>>> h.conjugacy_class()
Conjugacy class of [1 2]
[4 1]
in Matrix group over Finite Field of size 5 with 2 generators ( [1 2] [1 1] [4 1], [0 1] )
```

```python
>>> G = SL(Integer(2), GF(Integer(2)))
```

```python
>>> g = G.gens()[Integer(0)]
```

```python
>>> g.conjugacy_class()
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Finite Field of size 2
```

```python
>>> G = SL(Integer(2), QQ)
```

```python
>>> g = G([[Integer(1),Integer(1)], [Integer(0),Integer(1)])
```

```python
>>> g.conjugacy_class()
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Rational Field
```

### Finite

alias of `FiniteGroups`

### Lie

alias of `LieGroups`

#### class ParentMethods

Bases: object

`cayley_table(names='letters', elements=None)`

Return the “multiplication” table of this multiplicative group, which is also known as the “Cayley table”.

**Note:** The order of the elements in the row and column headings is equal to the order given by the table’s `column_keys()` method. The association between the actual elements and the names/symbols used
in the table can also be retrieved as a dictionary with the `translation()` method.

For groups, this routine should behave identically to the `multiplication_table()` method for magmas, which applies in greater generality.

**INPUT:**
- **names** – the type of names used, values are:
  - 'letters' – lowercase ASCII letters are used for a base 26 representation of the elements’ positions in the list given by `list()`, padded to a common width with leading ‘a’s.
  - 'digits' – base 10 representation of the elements’ positions in the list given by `column_keys()`, padded to a common width with leading zeros.
  - 'elements' – the string representations of the elements themselves.
  - a list - a list of strings, where the length of the list equals the number of elements.
- **elements** – (default = None); A list of elements of the group, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering is provided by the the group, which is reported by the `column_keys()` method. Or the elements can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.

**OUTPUT:**
An object representing the multiplication table. This is an `OperationTable` object and even more documentation can be found there.

**EXAMPLES:**
Permutation groups, matrix groups and abelian groups can all compute their multiplication tables.

```
sage: # needs sage.groups
sage: G = DiCyclicGroup(3)
sage: T = G.cayley_table()
sage: T.column_keys()
(((), (5,6,7), ..., (1,4,2,3)(5,7))
sage: T
* a b c d e f g h i j k l
+------------------------
a| a b c d e f g h i j k l
b| b c a e f d i g h l j k
| c a b f d e h i g k l j
d| d e f a b c j k l g h i
e| e f d b c a l j k i g
f| f d e c a b k l j h i g
g| g h i j k l d e f a b c
h| h i g k l j f d e c a
i| i g h l j k e f d b c a
j| j k l g h i a b c d e
k| k l j h i g c a b f d
e| e f d b c a l j k i g
l| l j k i g h b c a e f d
```
sage: M = SL(2, 2)  # needs sage.modules
sage: M.cayley_table()  # needs sage.libs.gap sage.modules
* a b c d e f
t---------
a| a b c d e f
b| b a d c f e
c| c e a f b d
d| d f b e a c
e| e c f a d b
f| f d e b c a

>>> from sage.all import *
>>> M = SL(Integer(2), Integer(2))  # needs sage.modules
>>> M.cayley_table()  # needs sage.libs.gap sage.modules
* a b c d e f
t---------
a| a b c d e f
b| b c a e f d
"...
Lowercase ASCII letters are the default symbols used for the table, but you can also specify the use of decimal digit strings, or provide your own strings (in the proper order if they have meaning). Also, if the elements themselves are not too complex, you can choose to just use the string representations of the elements themselves.
The \texttt{change_names()} routine behaves similarly, but changes an existing table "in-place."
For an infinite group, you can still work with finite sets of elements, provided the set is closed under multiplication. Elements will be coerced into the group as part of setting up the table.

```python
sage: # needs sage.modules
sage: G = SL(2,ZZ); G
Special Linear Group of degree 2 over Integer Ring
sage: identity = matrix(ZZ, [[1,0], [0,1]])
sage: G.cayley_table(elements=[identity, -identity])
```

```plaintext
* a b 
+----
a| a b 
b| b a 
```

The `OperationTable` class provides even greater flexibility, including changing the operation. Here is one such example, illustrating the computation of commutators. The commutator is defined as a function of two variables, before being used to build the table. From this, the commutator subgroup seems obvious, and creating a Cayley table with just these three elements confirms that they form a closed subset in the group.

```python
sage: # needs sage.groups sage.modules
sage: from sage.matrix.operation_table import OperationTable
```
(continues on next page)
```python
sage: G = DiCyclicGroup(3)
sage: commutator = lambda x, y: x*y*x^-1*y^-1
sage: T = OperationTable(G, commutator); T
   .  a  b  c  d  e  f  g  h  i  j  k  l
   +------------------------
   a| a  a  a  a  a  a  a  a  a  a  a  a  a  a
   b| a  a  a  a  a  a  c  c  c  c  c  c  c  c
   c| a  a  a  a  a  a  b  b  b  b  b  b  b  b
   d| a  a  a  a  a  a  a  a  a  a  a  a  a  a
   e| a  a  a  a  a  a  c  c  c  c  c  c  c  c
   f| a  a  a  a  a  a  b  b  b  b  b  b  b  b
   g| a  b  c  a  b  c  a  c  b  a  c  b  a  c
   h| a  b  c  a  b  c  b  a  c  b  a  c  b  a
   i| a  b  c  a  b  c  c  b  a  c  b  a  c  b
   j| a  b  c  a  b  c  a  c  b  a  c  b  a  c
   k| a  b  c  a  b  c  b  a  c  b  a  c  b  a
   l| a  b  c  a  b  c  c  b  a  c  b  a  c  b
sage: trans = T.translation()
sage: comm = [trans['a'], trans['b'], trans['c']]
sage: comm
[(), (5,6,7), (5,7,6)]
sage: P = G.cayley_table(elements=comm)
sage: P
   *  a  b  c
   +------
   a| a  b  c
   b| b  c  a
   c| c  a  b
```

(continues from previous page)
Todo: Arrange an ordering of elements into cosets of a normal subgroup close to size $\sqrt{n}$. Then the quotient group structure is often apparent in the table. See comments on Issue #7555.

AUTHOR:
• Rob Beezer (2010-03-15)

conjugacy_class($g$)
Return the conjugacy class of the element $g$.
This is a fall-back method for groups not defined over GAP.

EXAMPLES:

```
sage: A = AbelianGroup([2, 2])
    # needs sage.groups
sage: c = A.conjugacy_class(A.an_element())
    # needs sage.groups
sage: type(c)
    # needs sage.groups
<class 'sage.groups.conjugacy_classes.ConjugacyClass_with_category'>
```

```
>>> from sage.all import *
>>> A = AbelianGroup([Integer(2), Integer(2)])
    # needs sage.groups
>>> c = A.conjugacy_class(A.an_element())
    # needs sage.groups
>>> type(c)
    # needs sage.groups
<class 'sage.groups.conjugacy_classes.ConjugacyClass_with_category'>
```

group_generators() Return group generators for self.
This default implementation calls gens(), for backward compatibility.

EXAMPLES:

```
sage: A = AlternatingGroup(4)
    # needs sage.groups
sage: A.group_generators()
    # needs sage.groups
Family ((1,2,3), (2,3,4))
```

```
>>> from sage.all import *
>>> A = AlternatingGroup(Integer(4))
    # needs sage.groups
>>> A.group_generators()
    # needs sage.groups
Family ((1,2,3), (2,3,4))
```
holomorph()

The holomorph of a group

The holomorph of a group $G$ is the semidirect product $G \rtimes_{id} Aut(G)$, where $id$ is the identity function on $Aut(G)$, the automorphism group of $G$.

See Wikipedia article Holomorph (mathematics)

EXAMPLES:

```python
sage: G = Groups().example()  # needs sage.modules
sage: G.holomorph()  # needs sage.modules
Traceback (most recent call last):
... NotImplementedError: holomorph of General Linear Group of degree 4 over Rational Field not yet implemented
```

monoid_generators()

Return the generators of self as a monoid.

Let $G$ be a group with generating set $X$. In general, the generating set of $G$ as a monoid is given by $X \cup X^{-1}$, where $X^{-1}$ is the set of inverses of $X$. If $G$ is a finite group, then the generating set as a monoid is $X$.

EXAMPLES:

```python
sage: # needs sage.groups
sage: A = AlternatingGroup(4)
sage: A.monoid_generators()
Family ((1,2,3), (2,3,4))
sage: F.<x,y> = FreeGroup()
sage: F.monoid_generators()
Family (x, y, x^-1, y^-1)
```

semidirect_product($N$, $mapping$, $check=True$)

The semi-direct product of two groups

EXAMPLES:
sage: G = Groups().example()  # needs sage.modules
sage: G.semidirect_product(G, Morphism(G, G))  # needs sage.modules
Traceback (most recent call last):
... Not ImplementedError: semidirect product of General Linear Group of degree 4 over Rational Field and General Linear Group of degree 4 over Rational Field not yet implemented

>>> from sage.all import *

>>> G = Groups().example()  # needs sage.modules
>>> G.semidirect_product(G, Morphism(G, G))  # needs sage.modules
Traceback (most recent call last):
... Not ImplementedError: semidirect product of General Linear Group of degree 4 over Rational Field and General Linear Group of degree 4 over Rational Field not yet implemented

class Topological(category, *args)
    Bases: TopologicalSpacesCategory

    Category of topological groups.

    A topological group $G$ is a group which has a topology such that multiplication and taking inverses are continuous functions.

    REFERENCES:
    • Wikipedia article Topological_group

    example()

    EXAMPLES:

    sage: Groups().example()  # needs sage.modules
    General Linear Group of degree 4 over Rational Field

    >>> from sage.all import *
    >>> Groups().example()  # needs sage.modules
    General Linear Group of degree 4 over Rational Field

    static free(index_set=None, names=None, **kwds)
        Return the free group.

        INPUT:
        • index_set – (optional) an index set for the generators; if an integer, then this represents \{0, 1, \ldots, n - 1\}
        • names – a string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix
When the index set is an integer or only variable names are given, this returns `FreeGroup_class`, which currently has more features due to the interface with GAP than `IndexedFreeGroup`.

**EXAMPLES:**

```
sage: # needs sage.combinat sage.groups
sage: Groups().free(index_set=ZZ)
Free group indexed by Integer Ring
sage: Groups().free(ZZ)
Free group indexed by Integer Ring
sage: Groups().free(Integer(5))
Free Group on generators \{x0, x1, x2, x3, x4\}
sage: F.<x,y,z> = Groups().free(); F
Free group on generators \{x, y, z\}
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> Groups().free(index_set=ZZ)
Free group indexed by Integer Ring
>>> Groups().free(ZZ)
Free group indexed by Integer Ring
>>> Groups().free(Integer(5))
Free Group on generators \{x0, x1, x2, x3, x4\}
>>> F = Groups().free(names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3);
```

4.91 Hecke modules

```python
class sage.categories.hecke_modules.HeckeModules(R)
Bases: Category_module

The category of Hecke modules.

A Hecke module is a module \( M \) over the emph{anemic} Hecke algebra, i.e., the Hecke algebra generated by Hecke operators \( T_n \) with \( n \) coprime to the level of \( M \). (Every Hecke module defines a level function, which is a positive integer.) The reason we require that \( M \) only be a module over the anemic Hecke algebra is that many natural maps, e.g., degeneracy maps, Atkin-Lehner operators, etc., are \( T \)-module homomorphisms; but they are homomorphisms over the anemic Hecke algebra.

**EXAMPLES:**

We create the category of Hecke modules over \( \mathbb{Q} \):

```
sage: C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
```

```python
>>> from sage.all import *
>>> C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
```

TODO: check that this is what we want:

```
sage: C.super_categories()
[Category of vector spaces with basis over Rational Field]
```
```
>>> from sage.all import *
>>> C.super_categories()
[Category of vector spaces with basis over Rational Field]
# [Category of vector spaces over Rational Field]
Note that the base ring can be an arbitrary commutative ring:

```
sage: HeckeModules(IntegerRing())
Category of Hecke modules over Integer Ring
sage: HeckeModules(FiniteField(5))
Category of Hecke modules over Finite Field of size 5
```

The base ring doesn’t have to be a principal ideal domain:

```
sage: HeckeModules(PolynomialRing(IntegerRing(), 'x'))
Category of Hecke modules over Univariate Polynomial Ring in x over Integer Ring
```

```
class Homsets (category, *args)
    Bases: HomsetsCategory
class ParentMethods
    Bases: object
    extra_super_categories()

class ParentMethods
    Bases: object
    super_categories()
    EXAMPLES:
    ...
```

```
>>> from sage.all import *
>>> HeckeModules(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```

```
>>> from sage.all import *
>>> HeckeModules(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```
4.92 Highest Weight Crystals

class sage.categories.highest_weight_crystals.HighestWeightCrystalHomset(X, Y, category=None)

Bases: CrystalHomset

The set of crystal morphisms from a highest weight crystal to another crystal.

See also:

See sage.categories.crystals.CrystalHomset for more information.

Element

type: HighestWeightCrystalMorphism

class sage.categories.highest_weight_crystals.HighestWeightCrystalMorphism(parent, on_gens, cartan_type=None, virtualization=None, scaling_factors=None, gens=None, check=True)

Bases: CrystalMorphismByGenerators

A virtual crystal morphism whose domain is a highest weight crystal.

INPUT:

• parent -- a homset
• on_gens -- a function or list that determines the image of the generators (if given a list, then this uses the order of the generators of the domain) of the domain under self
• cartan_type -- (optional) a Cartan type; the default is the Cartan type of the domain
• virtualization -- (optional) a dictionary whose keys are in the index set of the domain and whose values are lists of entries in the index set of the codomain
• scaling_factors -- (optional) a dictionary whose keys are in the index set of the domain and whose values are scaling factors for the weight, \( \varepsilon \) and \( \varphi \)
• gens -- (optional) a list of generators to define the morphism; the default is to use the highest weight vectors of the crystal
• check -- (default: True) check if the crystal morphism is valid

class sage.categories.highest_weight_crystals.HighestWeightCrystals

Bases: Category_singleton

The category of highest weight crystals.

A crystal is highest weight if it is acyclic; in particular, every connected component has a unique highest weight element, and that element generate the component.
EXAMPLES:

```
sage: C = HighestWeightCrystals()
sage: C
Category of highest weight crystals
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

```python
>>> from sage.all import *
>>> C = HighestWeightCrystals()
>>> C
Category of highest weight crystals
>>> C.super_categories()
[Category of crystals]
>>> C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

class ElementMethods

   Bases: object

   **string_parameters**(word=None)

   Return the string parameters of self corresponding to the reduced word word.

   Given a reduced expression \( w = s_1 \cdots s_k \), the string parameters of \( b \in B \) corresponding to \( w \) are \((a_1, \ldots, a_k)\) such that

   \[
   e_i^{a_m} \cdots e_i^{a_1} b \neq 0 \\
   e_i^{a_m+1} \cdots e_i^{a_1} b = 0
   \]

   for all \( 1 \leq m \leq k \).

   For connected components isomorphic to \( B(\lambda) \) or \( B(\infty) \), if \( w = w_0 \) is the longest element of the Weyl group, then the path determined by the string parametrization terminates at the highest weight vector.

   **INPUT:**
   
   - word – a word in the alphabet of the index set; if not specified and we are in finite type, then this will be some reduced expression for the long element determined by the Weyl group

   **EXAMPLES:**

```
sage: B = crystals.infinity.NakajimaMonomials(['A',3])
sage: mg = B.highest_weight_vector()
sage: w0 = [1,2,1,3,2,1]
sage: mg.string_parameters(w0)
[0, 0, 0, 0, 0, 0]
sage: mg.f_string([1]).string_parameters(w0)
[1, 0, 0, 0, 0, 0]
sage: mg.f_string([1,1]).string_parameters(w0)
[3, 0, 0, 0, 0, 0]
sage: mg.f_string([1,1,1]).string_parameters(w0)
[1, 2, 2, 0, 0, 0]
sage: mg.f_string([1,1,1,2,2]).string_parameters(w0)
[1, 2, 2, 0, 0, 0]
sage: mg.f_string([1,1,1,2,2]) == mg.f_string([1,1,2,2,1])
True
sage: x = mg.f_string([1,1,1,2,2,1,3,3,2,1,1])
sage: x.string_parameters(w0)
[4, 1, 1, 2, 2, 2]
sage: x.string_parameters([3,2,1,3,2,3])
```

(continues on next page)
\begin{verbatim}
from sage.all import *
B = crystals.infinity.NakajimaMonomials(['A', Integer(3)])
mg = B.highest_weight_vector()
w0 = [Integer(1), Integer(2), Integer(1), Integer(3), Integer(2),
     Integer(1)]
mg.string_parameters(w0)
[0, 0, 0, 0, 0, 0]
mg.f_string([Integer(1)]).string_parameters(w0)
[1, 0, 0, 0, 0, 0]
mg.f_string([Integer(1), Integer(1), Integer(1)]).string_parameters(w0)
[3, 0, 0, 0, 0, 0]
mg.f_string([Integer(1), Integer(1), Integer(1), Integer(2), Integer(2)]).
     string_parameters(w0)
[1, 2, 2, 0, 0, 0]
mg.f_string([Integer(1), Integer(1), Integer(1), Integer(2), Integer(2),
     Integer(1), Integer(1), Integer(1), Integer(1)])
mg.f_string([Integer(1), Integer(1), Integer(1), Integer(2),
     Integer(1), Integer(3), Integer(3), Integer(2), Integer(1),
     Integer(1), Integer(1)])
mg.f_string([Integer(1), Integer(1), Integer(1), Integer(2),
     Integer(1), Integer(3), Integer(3), Integer(2), Integer(1),
     Integer(1), Integer(1)])
mg.f_string([Integer(1)]).string_parameters(w0)
[1, 0, 0, 0, 0, 0]
mf = lambda x: x == mg.f_string([Integer(1)]*Integer(7) + [Integer(2)]*Integer(3) +
    [Integer(3)]*Integer(2))
True

crystals.infinity.Tableaux("A5")
[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,6,6,6,6,6,6],
 ....:  [2,2,2,2,2,2,2,2,2,4,5,5,5,6],
 ....:  [3,3,3,3,3,3,3,3],
 ....:  [4,4,4,6,6,6],[
 ....:  [5,6]]]
b.string_parameters([[1,2,1,3,2,1,4,3,2,1,5,4,3,2,1]])
[0, 1, 1, 1, 0, 4, 4, 3, 0, 11, 10, 7, 7, 6]

sage: B = crystals.infinity.Tableaux("G2")
sage: b = B(rows= [[1,1,1,1,1,3,3,0,-3,-3,-2,-2,-1,-1,-1,-1],
     [2,3,3,3]])
sage: b.string_parameters([2,1,2,1,2,1])
[5, 13, 11, 15, 4, 4]
sage: b.string_parameters([1,2,1,2,1,2])
[7, 12, 15, 8, 10, 0]

from sage.all import *
B = crystals.infinity.Tableaux("A5")
b = B(rows=[[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,3,6,6,6,6,6,6],
 ....:  [2,2,2,2,2,2,2,2,2,4,5,5,5,6],
from sage.all import *

C = crystals.Tableaux(['C',2], shape=[2,1])
mw = C.highest_weight_vector()
lw = C.lowest_weight_vectors()[0]
lw.string_parameters([1,2,1,2])
[1, 2, 3, 1]
lw.string_parameters([2,1,2,1])
[1, 3, 2, 1]
lw.e_string([2,1,1,2,2,1]) == mw
True
lw.e_string([1,2,2,1,1,1,2]) == mw
True

class ParentMethods
    Bases: object
    connected_components_generators()

    Returns the highest weight vectors of self
This default implementation selects among the module generators those that are highest weight, and caches the result. A crystal element \( b \) is highest weight if \( e_i(b) = 0 \) for all \( i \) in the index set.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C.highest_weight_vectors()
(1,)

>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(5)])
>>> C.highest_weight_vectors()
(1,)

sage: C = crystals.Letters(['A', 2])
sage: T = crystals.TensorProduct(C, C, generators=[[C(2),C(1),C(1)],
...[C(1),C(2),C(1)]])
sage: T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])

>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(2)])
>>> T = crystals.TensorProduct(C, C, generators=[[C(Integer(2)),
...C(Integer(1)),C(Integer(1))],
...[C(Integer(1)),C(Integer(2)),C(Integer(1))]])
>>> T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

digraph (subset=None, index_set=None, depth=None)

Return the DiGraph associated to self.

INPUT:

- `subset` – (optional) a subset of vertices for which the digraph should be constructed
- `index_set` – (optional) the index set to draw arrows
- `depth` – the depth to draw; optional only for finite crystals

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: T.digraph()
Digraph on 8 vertices
sage: S = T.subcrystal(max_depth=2)
sage: len(S)
5
sage: G = T.digraph(subset=list(S))
sage: G.is_isomorphic(T.digraph(depth=2), edge_labels=True)
True

>>> from sage.all import *
>>> T = crystals.Tableaux(['A', Integer(2)], shape=[Integer(2),Integer(1)])
>>> T.digraph()
Digraph on 8 vertices
>>> S = T.subcrystal(max_depth=Integer(2))
>>> len(S)
5
>>> G = T.digraph(subset=list(S))
```

(continues on next page)
G.is_isomorphic(T.digraph(depth=Integer(2)), edge_labels=True)
True

**highest_weight_vector()**

Returns the highest weight vector if there is a single one; otherwise, raises an error.

Caveat: this assumes that `highest_weight_vectors()` returns a list or tuple.

**EXAMPLES:**

```
sage: C = crystals.Letters(['A',5])
sage: C.highest_weight_vector()
1
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C.highest_weight_vector()
1
```

**highest_weight_vectors()**

Returns the highest weight vectors of `self`

This default implementation selects among the module generators those that are highest weight, and caches the result. A crystal element $b$ is highest weight if $e_i(b) = 0$ for all $i$ in the index set.

**EXAMPLES:**

```
sage: C = crystals.Letters(['A',5])
sage: C.highest_weight_vectors()
(1,)
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(5)])
>>> C.highest_weight_vectors()
(1,)
```

```
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C, C, C, generators=[[C(Integer(2)),C(Integer(1)),C(Integer(1))],
....:            [C(Integer(1)),C(Integer(2)),C(Integer(1))]])
sage: T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

```
>>> from sage.all import *
>>> C = crystals.Letters(['A',Integer(2)])
>>> T = crystals.TensorProduct(C, C, C, generators=[[C(Integer(2)),
....:            C(Integer(1)),C(Integer(1))],
....:            [C(Integer(1)),
....:            C(Integer(2)),C(Integer(1))]])
>>> T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

**lowest_weight_vectors()**

Return the lowest weight vectors of `self`.

This default implementation selects among all elements of the crystal those that are lowest weight, and cache the result. A crystal element $b$ is lowest weight if $f_i(b) = 0$ for all $i$ in the index set.

4.92. Highest Weight Crystals
EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C.lowest_weight_vectors()
(6,)
```

```python
>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(5)])
>>> C.lowest_weight_vectors()
(6,)
```

```python
sage: C = crystals.Letters(['A', 2])
sage: T = crystals.TensorProduct(C, C, generators=[[C(2), C(1), C(1)],
....: [C(1), C(2), C(1)]])
sage: T.lowest_weight_vectors()
([3, 2, 3], [3, 3, 2])
>>> from sage.all import *
>>> C = crystals.Letters(['A', Integer(2)])
>>> T = crystals.TensorProduct(C, C, C, generators=[[C(Integer(2)),
....: C(Integer(1)), C(Integer(1))],
....: [C(Integer(1)),
... C(Integer(2)), C(Integer(1))]])
>>> T.lowest_weight_vectors()
([3, 2, 3], [3, 3, 2])
```

```
q_dimension(q=None, prec=None, use_product=False)
```

Return the \(q\)-dimension of \(self\).

Let \(B(\lambda)\) denote a highest weight crystal. Recall that the degree of the \(\mu\)-weight space of \(B(\lambda)\) (under the principal gradation) is equal to \(\langle \rho^\vee, \lambda - \mu \rangle\) where \(\langle \rho^\vee, \alpha_i \rangle = 1\) for all \(i \in I\) (in particular, take \(\rho^\vee = \sum_{i \in I} h_i\)).

The \(q\)-dimension of a highest weight crystal \(B(\lambda)\) is defined as

\[
\dim_q B(\lambda) := \sum_{j \geq 0} \dim(B_j)q^j,
\]

where \(B_j\) denotes the degree \(j\) portion of \(B(\lambda)\). This can be expressed as the product

\[
\dim_q B(\lambda) = \prod_{\alpha^\vee \in \Delta_+^\vee} \left( \frac{1 - q^{\langle \lambda + \rho, \alpha^\vee \rangle}}{1 - q^{\langle \rho, \alpha^\vee \rangle}} \right)^{\text{mult} \alpha},
\]

where \(\Delta_+^\vee\) denotes the set of positive coroots. Taking the limit as \(q \to 1\) gives the dimension of \(B(\lambda)\).

For more information, see [Ka1990] Section 10.10.

INPUT:
- \(q\) – the (generic) parameter \(q\)
- \(\text{prec}\) – (default: None) The precision of the power series ring to use if the crystal is not known to be finite (i.e. the number of terms returned). If None, then the result is returned as a lazy power series.
- \(\text{use_product}\) – (default: False) if we have a finite crystal and True, use the product formula

EXAMPLES:
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: qdim = C.q_dimension(); qdim
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
sage: qdim(1)
8
sage: len(C) == qdim(1)
True
sage: C.q_dimension(use_product=True) == qdim
True
sage: C.q_dimension(prec=20)
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
sage: C.q_dimension(prec=2)
2*q + 1
sage: R.<t> = QQ[]
sage: C.q_dimension(q=t^2)
t^8 + 2*t^6 + 2*t^4 + 2*t^2 + 1
sage: C = crystals.Tableaux(['A',2], shape=[5,2])
sage: C.q_dimension()
q^10 + 2*q^9 + 4*q^8 + 5*q^7 + 6*q^6 + 6*q^5 + 6*q^4 + 5*q^3 + 4*q^2 + 2*q + 1
sage: C = crystals.Tableaux(['B',2], shape=[2,1])
sage: qdim = C.q_dimension(); qdim
q^10 + 2*q^9 + 3*q^8 + 4*q^7 + 5*q^6 + 5*q^5 + 5*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
sage: qdim == C.q_dimension(use_product=True)
True
sage: C = crystals.Tableaux(['D',4], shape=[2,1])
sage: C.q_dimension()
q^16 + 2*q^15 + 4*q^14 + 7*q^13 + 10*q^12 + 13*q^11 + 16*q^10 + 18*q^9 + 18*q^8 + 18*q^7 + 16*q^6 + 13*q^5 + 10*q^4 + 7*q^3 + 4*q^2 + 2*q + 1

>>> from sage.all import *
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])
>>> qdim = C.q_dimension(); qdim
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
>>> qdim(Integer(1))
8
>>> len(C) == qdim(Integer(1))
True
>>> C.q_dimension(use_product=True) == qdim
True
>>> C.q_dimension(prec=Integer(20))
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
>>> C.q_dimension(prec=Integer(2))
2*q + 1
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> C.q_dimension(q=t**Integer(2))
t^8 + 2*t^6 + 2*t^4 + 2*t^2 + 1
>>> C = crystals.Tableaux(['A',Integer(2)], shape=[Integer(5),Integer(2)])
\[ q^{10} + 2q^9 + 4q^8 + 5q^7 + 6q^6 + 6q^5 + 6q^4 + 5q^3 + 4q^2 + 2q + 1 \]

```python
>>> C = crystals.Tableaux(['B', Integer(2)], shape=[Integer(2), Integer(1)])
```

\[ q^{10} + 2q^9 + 3q^8 + 4q^7 + 5q^6 + 5q^5 + 5q^4 + 4q^3 + 3q^2 + 2q + 1 \]

```python
>>> C.q_dimension() == C.q_dimension(use_product=True)
```

True

```python
>>> C = crystals.Tableaux(['D', Integer(4)], shape=[Integer(2), Integer(1)])
```

\[ q^{16} + 2q^{15} + 4q^{14} + 7q^{13} + 10q^{12} + 13q^{11} + 16q^{10} + 18q^9 + 18q^8 + 18q^7 + 16q^6 + 13q^5 + 10q^4 + 7q^3 + 4q^2 + 2q + 1 \]

We check with a finite tensor product:

```python
sage: TP = crystals.TensorProduct(C, C)
sage: TP.cardinality()
25600
```

\[ q^{32} + 2q^{31} + 8q^{30} + 15q^{29} + 34q^{28} + 63q^{27} + 110q^{26} + 175q^{25} + 276q^{24} + 389q^{23} + 550q^{22} + 725q^{21} + 930q^{20} + 1131q^{19} + 1362q^{18} + 1548q^{17} + 1736q^{16} + 1858q^{15} + 1947q^{14} + 1944q^{13} + 1918q^{12} + 1777q^{11} + 1628q^{10} + 1407q^9 + 1186q^8 + 928q^7 + 720q^6 + 498q^5 + 342q^4 + 201q^3 + 117q^2 + 48q + 26 \]

```python
sage: qdim = TP.q_dimension(use_product=True)
```

```
>> from sage.all import *
```

```python
>>> TP = crystals.TensorProduct(C, C)
```

25600

```
>>> qdim(Integer(1))
25600
```

The \(q\)-dimensions of infinite crystals are returned as formal power series:

```python
sage: C = crystals.LSPaths(['A', 2, 1], [1, 0, 0])
sage: C.q_dimension(prec=5)
1 + q + 2q^2 + 2q^3 + 4q^4 + 0(q^5)
sage: C.q_dimension(prec=10)
1 + q + 2q^2 + 2q^3 + 4q^4 + 5q^5 + 7q^6
```

(continues on next page)
class TensorProducts (category, *args)

Bases: TensorProductsCategory

The category of highest weight crystals constructed by tensor product of highest weight crystals.

class ParentMethods

Bases: object

Implements operations on tensor products of crystals.

highest_weight_vectors()

Return the highest weight vectors of self.

This works by using a backtracing algorithm since if $b_2 \otimes b_1$ is highest weight then $b_1$ is highest weight.

EXAMPLES:

sage: C = crystals.Tableaux(['D',4], shape=[2,2])
sage: D = crystals.Tableaux('D',4, shape=[1])
sage: T = crystals.TensorProduct(D, C)
sage: T.highest_weight_vectors()
([[1]], [[1, 1], [2, 2]]),
([[3]], [[1, 1], [2, 2]]),
([[2]], [[1, 1], [2, 2]])
sage: L = filter(lambda x: x.is_highest_weight(), T)
sage: tuple(L) == T.highest_weight_vectors()
True

>>> from sage.all import *
>>> C = crystals.Tableaux('D',4, shape=[Integer(2), Integer(2)])
>>> D = crystals.Tableaux('D',4, shape=[Integer(1)])
>>> T = crystals.TensorProduct(D, C)
>>> T.highest_weight_vectors()
([[1]], [[1, 1], [2, 2]]),
([[3]], [[1, 1], [2, 2]]),
([[2]], [[1, 1], [2, 2]])
L = filter(lambda x: x.is_highest_weight(), T)

>>> tuple(L) == T.highest_weight_vectors()
True

highest_weight_vectors_iterator()
Iterate over the highest weight vectors of self.

This works by using a backtracing algorithm since if \( b_2 \otimes b_1 \) is highest weight then \( b_1 \) is highest weight.

EXAMPLES:

```python
sage: C = crystals.Tableaux(['D',4], shape=[2,2])
sage: D = crystals.Tableaux(['D',4], shape=[1])
sage: T = crystals.TensorProduct(D, C)
sage: tuple(T.highest_weight_vectors_iterator())
([(1, 1), (2, 2)],
 [(3, 1), (2, 2)],
 [(-2, 1), (2, 2)])
```

```python
L = filter(lambda x: x.is_highest_weight(), T)
>>> tuple(L) == tuple(T.highest_weight_vectors_iterator())
True
```

extra_super_categories()

EXAMPLES:

```python
sage: HighestWeightCrystals().TensorProducts().extra_super_categories()
[Category of highest weight crystals]
```

addional_structure()
Return None.

Indeed, the category of highest weight crystals defines no additional structure: it only guarantees the existence of a unique highest weight element in each component.

See also:

```
Category.addional_structure()
```
Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

```
sage: HighestWeightCrystals().additional_structure()
```

```
>>> from sage.all import *

>>> HighestWeightCrystals().additional_structure()
```

dexample()

Returns an example of highest weight crystals, as per `Category.example()`.

EXAMPLES:

```
sage: B = HighestWeightCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1
```

```
>>> from sage.all import *

>>> B = HighestWeightCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1
```

```
sage: HighestWeightCrystals().super_categories()

[Category of crystals]
```

```
>>> from sage.all import *

>>> HighestWeightCrystals().super_categories()

[Category of crystals]
```

4.93 Hopf algebras

class `sage.categories.hopf_algebras.HopfAlgebras`(`base`, `name=None`)

   The category of Hopf algebras.

   EXAMPLES:

   ```sage
   sage: HopfAlgebras(QQ)
   Category of Hopf algebras over Rational Field
   sage: HopfAlgebras(QQ).super_categories()
   [Category of bialgebras over Rational Field]
   ```

   ```sage
   >>> from sage.all import *

   >>> HopfAlgebras(QQ)
   Category of Hopf algebras over Rational Field
   >>> HopfAlgebras(QQ).super_categories()
   [Category of bialgebras over Rational Field]
   ```

4.93. Hopf algebras
class DualCategory(base, name=None)
    Bases: Category_over_base_ring

    The category of Hopf algebras constructed as dual of a Hopf algebra

class ParentMethods
    Bases: object

class ElementMethods
    Bases: object

    antipode()

        Return the antipode of self

    EXAMPLES:

    sage: # needs sage.groups
    sage: A = HopfAlgebrasWithBasis(QQ).example(); A
    An example of Hopf algebra with basis: the group algebra of the
    Dihedral group of order 6 as a permutation group over Rational Field
    sage: [a,b] = A.algebra_generators()
    sage: a, a.antipode()
    (B[(1,2,3)], B[(1,3,2)])
    sage: b, b.antipode()
    (B[(1,3)], B[(1,3)])

>>> from sage.all import *
    # needs sage.groups
    >>> A = HopfAlgebrasWithBasis(QQ).example(); A
    An example of Hopf algebra with basis: the group algebra of the
    Dihedral group of order 6 as a permutation group over Rational Field
    >>> [a,b] = A.algebra_generators()
    >>> a, a.antipode()
    (B[(1,2,3)], B[(1,3,2)])
    >>> b, b.antipode()
    (B[(1,3)], B[(1,3)])

class Morphism
    Bases: Category

    The category of Hopf algebra morphisms.

class ParentMethods
    Bases: object

class Realizations(category, *args)
    Bases: RealizationsCategory

class ParentMethods
    Bases: object

    antipode_by_coercion(x)

        Returns the image of x by the antipode

        This default implementation coerces to the default realization, computes the antipode there, and
        coerces the result back.

    EXAMPLES:
```python
sage: from sage.all import *
>>> from sage.all import *
```

```python
with sage.combinat sage.modules
with sage.combinat sage.modules
with sage.combinat sage.modules
```

```python
N = NonCommutativeSymmetricFunctions(QQ)
R = N.ribbon()
R.antipode_by_coercion(R[1,3,1])
- R[2, 1, 2]
```

```python
>>> from sage.all import *
>>> from sage.all import *
```

```python
N = NonCommutativeSymmetricFunctions(QQ)
R = N.ribbon()
R.antipode_by_coercion(R[1,3,1])
- R[2, 1, 2]
```

### class Super

**Bases:** SuperModulesCategory

The category of super Hopf algebras.

**Note:** A super Hopf algebra is not simply a Hopf algebra with a Z/2Z grading due to the signed bialgebra compatibility conditions.

### class ElementMethods

**Bases:** object

**antipode()**

Return the antipode of self.

**EXAMPLES:**

```python
sage: A = SteenrodAlgebra(3)
# needs sage.combinat sage.modules

# needs sage.combinat sage.modules

# needs sage.combinat sage.modules

# needs sage.combinat sage.modules
```

```python
a, a.antipode()

(2 Q_1 Q_3 P(2,1), Q_1 Q_3 P(2,1))
```

```python
>>> from sage.all import *
```

```python
A = SteenrodAlgebra(Integer(3))
```

```python
da, a.antipode()

(2 Q_1 Q_3 P(2,1), Q_1 Q_3 P(2,1))
```

**dual()**

Return the dual category.

**EXAMPLES:**

The category of super Hopf algebras over any field is self dual:
class TensorProducts (category, *args)

Bases: TensorProductsCategory

The category of Hopf algebras constructed by tensor product of Hopf algebras

class ElementMethods

Bases: object

class ParentMethods

Bases: object

element super_categories ()

EXAMPLES:

```python
sage: C = HopfAlgebras(QQ).Super()
sage: C.dual()
Category of super Hopf algebras over Rational Field

>>> from sage.all import *
>>> C = HopfAlgebras(QQ).Super()
>>> C.dual()
Category of super Hopf algebras over Rational Field
```

WithBasis

alias of HopfAlgebrasWithBasis
dual ()

Return the dual category

EXAMPLES:

The category of Hopf algebras over any field is self dual:

```python
sage: C = HopfAlgebras(QQ)
sage: C.dual()
Category of Hopf algebras over Rational Field

>>> from sage.all import *
>>> C = HopfAlgebras(QQ)
(continues on next page)```
super_categories()

EXAMPLES:

```python
sage: HopfAlgebras(QQ).super_categories()
[Category of bialgebras over Rational Field]
```

```
>>> from sage.all import *

>>> HopfAlgebras(QQ).super_categories()
[Category of bialgebras over Rational Field]
```

## 4.94 Hopf algebras with basis

**class** `sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis` *(base_category)*

**Bases:** `CategoryWithAxiom_over_base_ring`

The category of Hopf algebras with a distinguished basis

**EXAMPLES:**

```python
sage: C = HopfAlgebrasWithBasis(QQ)
sage: C
Category of Hopf algebras with basis over Rational Field
sage: C.super_categories()
[Category of Hopf algebras over Rational Field,
 Category of bialgebras with basis over Rational Field]
```

```
>>> from sage.all import *

>>> C = HopfAlgebrasWithBasis(QQ)

>>> C
Category of Hopf algebras with basis over Rational Field

>>> C.super_categories()
[Category of Hopf algebras over Rational Field,
 Category of bialgebras with basis over Rational Field]
```

We now show how to use a simple Hopf algebra, namely the group algebra of the dihedral group (see also AlgebrasWithBasis):

```python
sage: A = C.example(); A
#...

An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a permutation group over Rational Field
```

```python
sage: A.rename("A")
#...

# needs sage.groups
sage: A.category()
#...

# needs sage.groups
Category of finite dimensional Hopf algebras with basis over Rational Field
```

```python
c sage: A.one_basis()
#...
```

(continues on next page)
sage: A.one()  # needs sage.groups
B[()]

sage: A.base_ring()  # needs sage.groups
Rational Field

sage: A.basis().keys()  # needs sage.groups
Dihedral group of order 6 as a permutation group

sage: # needs sage.groups
sage: [a, b] = A.algebra_generators()
sage: a, b
(B[(1,2,3)], B[(1,3)])
sage: a^3, b^2
(B[()], B[()])
sage: a*b
B[(1,2)]

sage: A.product  # todo: not quite ...  # needs sage.groups
<bound method MagmaticAlgebras.WithBasis.ParentMethods._product_from_product_on_ _basis_multiply of A>

sage: A.product(b, b)  # needs sage.groups
B[()]

sage: A.zero().coproduct()  # needs sage.groups
0

sage: A.zero().coproduct().parent()  # needs sage.groups
A # A

sage: a.coproduct()  # needs sage.groups
B[(1,2,3)] # B[(1,2,3)]

sage: TestSuite(A).run(verbosetrue)  # needs sage.groups
Running the test suite of self.an_element()
Running _test_additive_associativity() ... pass
running _test_an_element() ... pass
running _test_antipode() ... pass
running _test_associativity() ... pass
running _test_cardinality() ... pass
running _test_category() ... pass
running _test_characteristic() ... pass
running _test_construction() ... pass
running _test_distributivity() ... pass
running _test_elements() ... Running the test suite of self.an_element() running _test_category() ... pass
running _test_eq() ... pass
running _test_monomial_coefficients() ... pass
running _test_new() ... pass
running _test_nonzero_equal() ... pass

(continues on next page)
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass

sage: A.__class__
# needs sage.groups
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_
category'>

sage: A.element_class
# needs sage.groups
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_
category.element_class'>

>>> from sage.all import *

>>> A = C.example(); A
# needs sage.groups
An example of Hopf algebra with basis: the group algebra of the
Dihedral group of order 6 as a permutation group over Rational Field

>>> A.rename("A")
# needs sage.groups

>>> A.category()
# needs sage.groups
Category of finite dimensional Hopf algebras with basis over Rational Field

>>> A.one_basis()
# needs sage.groups
()

>>> A.one()
# needs sage.groups
B[()]

>>> A.base_ring()
# needs sage.groups
Rational Field

>>> A.basis().keys()
# needs sage.groups
Dihedral group of order 6 as a permutation group

>>> # needs sage.groups

>>> [a,b] = A.algebra_generators()

>>> a, b
(B[(1,2,3)], B[(1,3)])

>>> a**Integer(3), b**Integer(2)
(B[()], B[()])

>>> a*b

(continues on next page)
B[(1,2)]

>>> A.product  # todo: not quite ...  
→ needs sage.groups 
<bound method MagmaticAlgebras.WithBasis.ParentMethods._product_from_product_on_basis_multiply of A>

>>> A.product(b, b)  
→ needs sage.groups 
B[]

>>> A.zero().coproduct()  
→ needs sage.groups 
0

>>> A.zero().coproduct().parent()  
→ needs sage.groups 
A # A

>>> a.coproduct()  
→ needs sage.groups 
B[(1,2,3)] # B[(1,2,3)]

>>> TestSuite(A).run(verbosity=True)  
→ needs sage.groups 
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_antipode() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_construction() . . . pass
running ._test_distributivity() . . . pass
running ._test_elements() . . .
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_monomial_coefficients() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass

>>> A.__class__  
→ needs sage.groups 
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_category'>

(continues on next page)
Let us look at the code for implementing A:

```python
from sage.all import *

A = HopfAlgebrasWithBasis(ZZ).example(); A
An example of Hopf algebra with basis: the group algebra of the
Dihedral group of order 6 as a permutation group over Integer Ring
```

```python
[a, b] = A.algebra_generators()
a, A.antipode(a)
(B[(1,2,3)], B[(1,3,2)])
b, A.antipode(b)
(B[(1,3)], B[(1,3)])
```

```python
>>> from sage.all import *
```
antipode_on_basis(x)
The antipode of the Hopf algebra on the basis (optional)

INPUT:
• x – an index of an element of the basis of self

Returns the antipode of the basis element indexed by x.

If this method is implemented, then antipode() is defined from this by linearity.

EXAMPLES:

```python
sage: # needs sage.groups
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: W = A.basis().keys(); W
Dihedral group of order 6 as a permutation group
sage: w = W.gen(0); w
(1,2,3)
sage: A.antipode_on_basis(w)
B[(1,3,2)]
```

Super
alias of SuperHopfAlgebrasWithBasis
class TensorProducts(category, *args)
Bases: TensorProductsCategory

The category of Hopf algebras with basis constructed by tensor product of Hopf algebras with basis
class ElementMethods
Bases: object
class ParentMethods
Bases: object

extra_super_categories()

EXAMPLES:

```python
sage: C = HopfAlgebrasWithBasis(QQ).TensorProducts()
sage: C.extra_super_categories()
[Category of Hopf algebras with basis over Rational Field]
sage: sorted(C.extra_super_categories(), key=str)
[Category of Hopf algebras with basis over Rational Field,
 Category of tensor products of Hopf algebras over Rational Field,
 Category of tensor products of algebras with basis over Rational Field]
```
example \((G=\text{None})\)

Returns an example of algebra with basis:

```
sage: HopfAlgebrasWithBasis(QQ['x']).example()  # needs sage.groups
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a permutation group over Univariate Polynomial Ring in x over Rational Field
```

An other group can be specified as optional argument:

```
sage: HopfAlgebrasWithBasis(QQ).example(SymmetricGroup(4))  # needs sage.groups
An example of Hopf algebra with basis: the group algebra of the Symmetric group of order 4! as a permutation group over Rational Field
```

4.95 H-trivial semigroups

```
class sage.categories.h_trivial_semigroups.HTrivialSemigroups(base_category):
    Bases: CategoryWithAxiom

    Finite_extra_super_categories()

    Implement the fact that a finite \(H\)-trivial is aperiodic

    EXAMPLES:

    sage: Semigroups().HTrivial().Finite_extra_super_categories()
    [Category of aperiodic semigroups]
    sage: Semigroups().HTrivial().Finite() is Semigroups().Aperiodic().Finite()
    True
```
Inverse_extra_super_categories()

Implement the fact that an $H$-trivial inverse semigroup is $J$-trivial.

Todo: Generalization for inverse semigroups.

Recall that there are two invertibility axioms for a semigroup $S$:

- One stating the existence, for all $x$, of a local inverse $y$ satisfying $x = xyx$ and $y = yxy$;
- One stating the existence, for all $x$, of a global inverse $y$ satisfying $xy = yx = 1$, where 1 is the unit of $S$ (which must of course exist).

It is sufficient to have local inverses for $H$-triviality to imply $J$-triviality. However, at this stage, only the second axiom is implemented in Sage (see `Magmas.Unital.SubcategoryMethods.Inverse()`). Therefore this fact is only implemented for semigroups with global inverses, that is groups. However the trivial group is the unique $H$-trivial group, so this is rather boring.

EXAMPLES:

```sage
def _init_(self):
    super(_init_, self).subcategory()

```

4.96 Infinite Enumerated Sets

AUTHORS:


class sage.categories.infinite Enumerated Sets.InfiniteEnumeratedSets (base_category)

Bases: CategoryWithAxiom singleton

The category of infinite enumerated sets

An infinite enumerated sets is a countable set together with a canonical enumeration of its elements.

EXAMPLES:

```sage: InfiniteEnumeratedSets()
Category of infinite enumerated sets
sage: InfiniteEnumeratedSets().super_categories()
```
[Category of enumerated sets, Category of infinite sets]
sage: InfiniteEnumeratedSets().all_super_categories()
[Category of infinite enumerated sets,
Category of enumerated sets,
Category of infinite sets,
Category of sets,
Category of sets with partial maps,
Category of objects]

>>> from sage.all import *
>>> InfiniteEnumeratedSets()
Category of infinite enumerated sets
>>> InfiniteEnumeratedSets().super_categories()
[Category of enumerated sets, Category of infinite sets]
>>> InfiniteEnumeratedSets().all_super_categories()
[Category of infinite enumerated sets,
Category of enumerated sets,
Category of infinite sets,
Category of sets,
Category of sets with partial maps,
Category of objects]

class ParentMethods

Bases: object

list()

Raise an error because self is an infinite enumerated set.

EXAMPLES:

sage: NN = InfiniteEnumeratedSets().example()
sage: NN.list()
Traceback (most recent call last):
... NotImplementedError: cannot list an infinite set

random_element()

Raise an error because self is an infinite enumerated set.

EXAMPLES:

sage: NN = InfiniteEnumeratedSets().example()
sage: NN.random_element()
Traceback (most recent call last):
... NotImplementedError: infinite set
>>> NN.random_element()
Traceback (most recent call last):
...  
NotImplementedError: infinite set

TODO: should this be an optional abstract_method instead?

tuple()
Raise an error because self is an infinite enumerated set.

EXAMPLES:

sage: NN = InfiniteEnumeratedSets().example()
sage: NN.tuple()
Traceback (most recent call last):
...  
NotImplementedError: cannot list an infinite set

>>> from sage.all import *
>>> NN = InfiniteEnumeratedSets().example()
>>> NN.tuple()
Traceback (most recent call last):
...  
NotImplementedError: cannot list an infinite set

4.97 Integral domains

class sage.categories.integral_domains.IntegralDomains(base_category)

Bases: CategoryWithAxiom_singleton

The category of integral domains

An integral domain is commutative ring with no zero divisors, or equivalently a commutative domain.

EXAMPLES:

sage: C = IntegralDomains(); C
Category of integral domains
sage: sorted(C.super_categories(), key=str)
[Category of commutative rings, Category of domains]
sage: C is Domains().Commutative()
True
sage: C is Rings().Commutative().NoZeroDivisors()
True

>>> from sage.all import *
>>> C = IntegralDomains(); C
Category of integral domains
>>> sorted(C.super_categories(), key=str)
[Category of commutative rings, Category of domains]
>>> C is Domains().Commutative()
True
>>> C is Rings().Commutative().NoZeroDivisors()
True
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

    is_integral_domain(proof=True)
    Return True, since this in an object of the category of integral domains.

    EXAMPLES:

    sage: ZZ.is_integral_domain()
    True
    sage: QQ.is_integral_domain()
    True
    sage: Parent(QQ, category=IntegralDomains()).is_integral_domain()
    True

    sage: L.<z> = LazyLaurentSeriesRing(QQ)
    # needs sage.combinat
    sage: L.is_integral_domain()
    # needs sage.combinat
    True
    sage: L.is_integral_domain(proof=True)
    # needs sage.combinat
    True

    sage: ZZ['x'].is_integral_domain()
    True

    >>> from sage.all import *
    >>> ZZ.is_integral_domain()
    True
    >>> QQ.is_integral_domain()
    True
    >>> Parent(QQ, category=IntegralDomains()).is_integral_domain()
    True

    >>> L = LazyLaurentSeriesRing(QQ, names=('z',)); (z,) = L._first_ngens(1)
    # needs sage.combinat
    >>> L.is_integral_domain()
    # needs sage.combinat
    True
    >>> L.is_integral_domain(proof=True)
    # needs sage.combinat
    True

    >>> ZZ['x'].is_integral_domain()
    True
4.98 J-trivial semigroups

class sage.categories.j_trivial_semigroups.JTrivialSemigroups(base_category):
    Bases: CategoryWithAxiom
    extra_super_categories()
    Implement the fact that a $J$-trivial semigroup is $L$ and $R$-trivial.

    EXAMPLES:

    sage: Semigroups().JTrivial().extra_super_categories()
    [Category of l trivial semigroups, Category of r trivial semigroups]

    >>> from sage.all import *
    >>> Semigroups().JTrivial().extra_super_categories()
    [Category of l trivial semigroups, Category of r trivial semigroups]

4.99 Kac-Moody Algebras

AUTHORS:

• Travis Scrimshaw (07-15-2017): Initial implementation

class sage.categories.kac_moody_algebras.KacMoodyAlgebras(base, name=None):
    Bases: Category_over_base_ring

    Category of Kac-Moody algebras.

class ParentMethods:
    Bases: object
    cartan_type()
    Return the Cartan type of self.

    EXAMPLES:

    sage: L = LieAlgebra(QQ, cartan_type=['A', 2])  # needs sage.combinat sage.modules
    sage: L.cartan_type()  # needs sage.combinat sage.modules
    ['A', 2]

    >>> from sage.all import *
    >>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)])  # needs sage.combinat sage.modules
    >>> L.cartan_type()  # needs sage.combinat sage.modules
    ['A', 2]

    weyl_group()
    Return the Weyl group of self.

    EXAMPLES:
### Example \((n=2)\)

Return an example of a Kac-Moody algebra as per `Category.example`.

**EXAMPLES:**

```python
sage: from sage.categories.kac_moody_algebras import KacMoodyAlgebras
sage: KacMoodyAlgebras(QQ).example()  # needs sage.combinat sage.modules
Lie algebra of ['A', 2] in the Chevalley basis
```

We can specify the rank of the example:

```python
sage: KacMoodyAlgebras(QQ).example(4)  # needs sage.combinat sage.modules
Lie algebra of ['A', 4] in the Chevalley basis
```

### super_categories()

**EXAMPLES:**

```python
sage: from sage.categories.kac_moody_algebras import KacMoodyAlgebras
sage: KacMoodyAlgebras(QQ).super_categories()
[Category of Lie algebras over Rational Field]
```
### 4.100 Lambda Bracket Algebras

**AUTHORS:**

**class** `sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras` *(base, name=None)*

* **Bases:** `Category_over_base_ring`
* The category of Lambda bracket algebras.

This is an abstract base category for Lie conformal algebras and super Lie conformal algebras.

**class** `ElementMethods`

* **Bases:** `object`

  `T(n=1)`
  
  The n-th derivative of `self`.

  **INPUT:**
  - `n` -- integer (default: 1); how many times to apply `T` to this element
  **OUTPUT:**
  
  `T^n a` where `a` is this element. Notice that we use the *divided powers* notation `T^{(j)} = T^j / j!`.

**EXAMPLES:**

```sage
# needs sage.combinat sage.modules
sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.inject_variables()
Defining L, C
tsage: L.T()  
TL
sage: L.T(3)  
6*T^(3)L
sage: C.T()  
0
```

```python
>>> from sage.all import *

>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.inject_variables()
Defining L, C
>>> L.T()
TL
>>> L.T(Integer(3))
6*T^(3)L
>>> C.T()
0
```

**bracket** *(rhs)*

The $\lambda$-bracket of these two elements.

**EXAMPLES:**

The brackets of the Virasoro Lie conformal algebra:
\begin{verbatim}
sage: Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0
#-- needs sage.combinat sage.modules
sage: L.bracket(L)  #-- needs sage.combinat sage.modules
{0: TL, 1: 2*L, 3: 1/2*C}

>>> from sage.all import *

>>> Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0
#-- needs sage.combinat sage.modules
>>> L.bracket(L)  #-- needs sage.combinat sage.modules
{0: TL, 1: 2*L, 3: 1/2*C}

>>> L.bracket(L.T())  #-- needs sage.combinat sage.modules
{0: 2*T^(2)L, 1: 3*TL, 2: 4*L, 4: 2*C}
\end{verbatim}

Now with a current algebra:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
sage: V = lie_conformal_algebras.Affine(QQ, 'A1')
sage: V.gens()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])
sage: E = V.0; H = V.1; F = V.2
sage: H.bracket(H)  #-- needs sage.combinat sage.modules
{1: 2*B['K']}
sage: E.bracket(F)  #-- needs sage.combinat sage.modules
{0: B[alphacheck[1]], 1: B['K']}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> V = lie_conformal_algebras.Affine(QQ, 'A1')

>>> V.gens()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])

>>> E = V.gen(0); H = V.gen(1); F = V.gen(2)

>>> H.bracket(H)  #-- needs sage.combinat sage.modules
{1: 2*B['K']}

>>> E.bracket(F)  #-- needs sage.combinat sage.modules
{0: B[alphacheck[1]], 1: B['K']}
\end{verbatim}

\texttt{nproduct} \((rhs, n)\)

The \(n\)-th product of these two elements.

EXAMPLES:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
sage: Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0
sage: L.nproduct(L, 1/2*C)  #-- needs sage.combinat sage.modules
1/2*C

sage: L.nproduct(L.T(), 2*T^(2)L)  #-- needs sage.combinat sage.modules
2*T^(2)L

sage: V = lie_conformal_algebras.Affine(QQ, 'A1')

sage: E = V.0; H = V.1; F = V.2

sage: E.nproduct(H, 0) == - 2*E
True
\end{verbatim}

(continues on next page)
sage: E.nproduct(F, 1)
B['K']

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.gen(0)
>>> L.nproduct(L, Integer(3))
1/2*C
>>> L.nproduct(L.T(), Integer(0))
2*T^2*L
>>> V = lie_conformal_algebras.Affine(QQ, 'A1')
>>> E = V.gen(0); H = V.gen(1); F = V.gen(2)
>>> E.nproduct(H, Integer(0)) == - Integer(2) * E
True
>>> E.nproduct(F, Integer(1))
B['K']

FinitelyGeneratedAsLambdaBracketAlgebra
alias of FinitelyGeneratedLambdaBracketAlgebras
class ParentMethods
    Bases: object

    ideal(*gens, **kwds)
    The ideal of this Lambda bracket algebra generated by gens.

    Todo: Ideals of Lie Conformal Algebras are not implemented yet.

EXAMPLES:

sage: Vir = lie_conformal_algebras.Virasoro(QQ)  #...
˓→ needs sage.combinat sage.modules
sage: Vir.ideal()  #...
˓→ needs sage.combinat sage.modules
Traceback (most recent call last):
    ...
NotImplementedError: ideals of Lie Conformal algebras are not implemented... ˓→yet

>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)  #...
˓→ needs sage.combinat sage.modules
>>> Vir.ideal()  #...
˓→ needs sage.combinat sage.modules
Traceback (most recent call last):
    ...
NotImplementedError: ideals of Lie Conformal algebras are not implemented... ˓→yet

class SubcategoryMethods
    Bases: object

    FinitelyGenerated()
    The category of finitely generated Lambda bracket algebras.
### Category Framework, Release 10.4

**EXAMPLES:**

```python
sage: LieConformalAlgebras(QQ).FinitelyGenerated()
Category of finitely generated Lie conformal algebras over Rational Field
```

```python
>>> from sage.all import *

>>> LieConformalAlgebras(QQ).FinitelyGenerated()
Category of finitely generated Lie conformal algebras over Rational Field
```

**FinitelyGeneratedAsLambdaBracketAlgebra()**

The category of finitely generated Lambda bracket algebras.

**EXAMPLES:**

```python
sage: LieConformalAlgebras(QQ).FinitelyGenerated()
Category of finitely generated Lie conformal algebras over Rational Field
```

```python
>>> from sage.all import *

>>> LieConformalAlgebras(QQ).FinitelyGenerated()
Category of finitely generated Lie conformal algebras over Rational Field
```

**WithBasis**

alias of `LambdaBracketAlgebrasWithBasis`

**super_categories()**

The list of super categories of this category.

**EXAMPLES:**

```python
sage: from sage.categories.lambda_bracket_algebras import LambdaBracketAlgebras
sage: LambdaBracketAlgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]
```

```python
>>> from sage.all import *

>>> from sage.categories.lambda_bracket_algebras import LambdaBracketAlgebras

>>> LambdaBracketAlgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]
```

### 4.101 Lambda Bracket Algebras With Basis

**AUTHORS:**

- Reimundo Heluani (2020-08-21): Initial implementation.

**class** `sage.categories.lambda_bracket_algebras_with_basis.LambdaBracketAlgebrasWithBasis`

The category of Lambda bracket algebras with basis.

**EXAMPLES:**

```python
class sage.categories.lambda_bracket_algebras_with_basis.LambdaBracketAlgebrasWithBasis:

Bases: CategoryWithAxiom_over_base_ring

The category of Lambda bracket algebras with basis.

**EXAMPLES:**
```
```python
sage: LieConformalAlgebras(QQbar).WithBasis()  # needs sage.rings.number_field
Category of Lie conformal algebras with basis over Algebraic Field

>>> from sage.all import *
>>> LieConformalAlgebras(QQbar).WithBasis()  # needs sage.rings.number_field
Category of Lie conformal algebras with basis over Algebraic Field
```

class ElementMethods

    Bases: object

    index()

        The index of this basis element.

        EXAMPLES:

        ```python
sage: # needs sage.combinat sage.modules
sage: V = lie_conformal_algebras.NeveuSchwarz(QQ)
sage: V.inject_variables()
Defining L, G, C
sage: v = V.an_element(); v
L + G + C
sage: v.index()
Traceback (most recent call last):
... ValueError: index can only be computed for monomials, got L + G + C
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> V = lie_conformal_algebras.NeveuSchwarz(QQ)
>>> V.inject_variables()
Defining L, G, C
>>> G.T(Integer(3)).index()
('G', 3)
>>> v = V.an_element(); v
L + G + C
>>> v.index()
Traceback (most recent call last):
... ValueError: index can only be computed for monomials, got L + G + C
```

class FinitelyGeneratedAsLambdaBracketAlgebra(base_category)

    Bases: CategoryWithAxiom_over_base_ring

    The category of finitely generated lambda bracket algebras with basis.

    EXAMPLES:

    ```python
sage: # needs sage.rings.number_field
sage: C = LieConformalAlgebras(QQbar)
sage: C1 = C.WithBasis().FinitelyGenerated(); C1
Category of finitely generated Lie conformal algebras with basis
over Algebraic Field
sage: C2 = C.FinitelyGenerated().WithBasis(); C2
```
```
Category of finitely generated Lie conformal algebras with basis over Algebraic Field

```sage
C1 is C2
True
```

```sage
from sage.all import *
# needs sage.rings.number_field
C = LieConformalAlgebras(QQbar)
C1 = C.WithBasis().FinitelyGenerated(); C1
Category of finitely generated Lie conformal algebras with basis over Algebraic Field
C2 = C.FinitelyGenerated().WithBasis(); C2
Category of finitely generated Lie conformal algebras with basis over Algebraic Field
C1 is C2
True
```

```class Graded(base_category)
    Bases: GradedModulesCategory

    The category of H-graded finitely generated lambda bracket algebras with basis.

    EXAMPLES:
```

```sage
V = lie_conformal_algebras.Virasoro(QQ)
V.degree_on_basis((L, 2))
4
```

```class ParentMethods
    Bases: object

    degree_on_basis(m)

    Return the degree of the basis element indexed by m in self.

    EXAMPLES:
```

```sage
V = lie_conformal_algebras.Virasoro(QQ)
V.degree_on_basis((L, 2))
4
```

(continues on next page)
4.102 Lattice posets

class sage.categories.lattice_posets.LatticePosets

Bases: Category

The category of lattices, i.e. partially ordered sets in which any two elements have a unique supremum (the elements’ least upper bound; called their join) and a unique infimum (greatest lower bound; called their meet).

EXAMPLES:

```python
sage: LatticePosets()
Category of lattice posets
sage: LatticePosets().super_categories()
[Category of posets]
sage: LatticePosets().example()
NotImplemented
```

See also:

Posets, FiniteLatticePosets, LatticePoset()

Finite

alias of FiniteLatticePosets

class ParentMethods

Bases: object

join(x, y)

Returns the join of x and y in this lattice

INPUT:
• x, y – elements of self

EXAMPLES:

```python
sage: D = LatticePoset((divisors(60), attrcall("divides"))) #...
needs sage.graphs sage.modules
sage: D.join( D(6), D(10) ) #...
needs sage.graphs sage.modules
30
```
Hyperlink to the content of the document.
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

super_categories()

EXAMPLES:

sage: LeftModules(QQ).super_categories()
[Category of commutative additive groups]

4.104 Lie Algebras

AUTHORS:

• Travis Scrimshaw (07-15-2013): Initial implementation

class sage.categories.lie_algebras.LieAlgebras(base, name=None)
    Bases: Category_over_base_ring

The category of Lie algebras.

EXAMPLES:

sage: C = LieAlgebras(QQ); C
Category of Lie algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of vector spaces over Rational Field]

We construct a typical parent in this category, and do some computations with it:

sage: # needs sage.combinat sage.groups sage.modules
sage: A = C.example(); A
An example of a Lie algebra: the Lie algebra from the associative algebra Symmetric group algebra of order 3 over Rational Field generated by ([2, 1, 3], [2, 3, 1])

sage: A.category()
Category of Lie algebras over Rational Field

(continues on next page)
sage: A.base_ring()
Rational Field
sage: a, b = A.lie_algebra_generators()
sage: a.bracket(b)
-\[1, 3, 2\] + \[3, 2, 1\]
sage: b.bracket(2*a + b)
2*[1, 3, 2] - 2*[3, 2, 1]
sage: A.bracket(a, b)
-\[1, 3, 2\] + \[3, 2, 1\]

>>> from sage.all import *
>>> # needs sage.combinat sage.groups sage.modules
>>> A = C.example(); A
An example of a Lie algebra: the Lie algebra from the associative algebra Symmetric group algebra of order 3 over Rational Field generated by [[2, 1, 3], [2, 3, 1]]
>>> A.category()
Category of Lie algebras over Rational Field
>>> A.base_ring()
Rational Field
>>> a, b = A.lie_algebra_generators()
>>> a.bracket(b)
-\[1, 3, 2\] + \[3, 2, 1\]
>>> b.bracket(Integer(2)*a + b)
2*[1, 3, 2] - 2*[3, 2, 1]
>>> A.bracket(a, b)
-\[1, 3, 2\] + \[3, 2, 1\]

Please see the source code of $A$ (with $A??$) for how to implement other Lie algebras.

**Todo:** Many of these tests should use Lie algebras that are not the minimal example and need to be added after Issue #16820 (and Issue #16823).

class ElementMethods
Bases: object

bracket(rhs)

Return the Lie bracket $[self, rhs]$.

EXAMPLES:

```python
sage: # needs sage.combinat sage.groups sage.modules
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: x.bracket(y)
-\[1, 3, 2\] + \[3, 2, 1\]
sage: x.bracket(0)
0
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups sage.modules
>>> L = LieAlgebras(QQ).example()
>>> x, y = L.lie_algebra_generators()
>>> x.bracket(y)
```
\[-[1, 3, 2] + [3, 2, 1] \]

```python
>>> x.bracket(Integer(0))
0
```

**exp** *(lie_group=None)*

Return the exponential of *self* in *lie_group*.

**INPUT:**

- *lie_group* – (optional) the Lie group to map into; If *lie_group* is not given, the Lie group associated to the parent Lie algebra of *self* is used.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules sage.symbolic
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: g = (X + Y + Z).exp(); g
exp(X + Y + Z)
sage: h = X.exp(); h
exp(X)
sage: g.parent()  # is h.parent()
True
```

The Lie group can be specified explicitly:

```python
sage: # needs sage.combinat sage.modules sage.symbolic
sage: H = L.lie_group(H)
sage: k = Z.exp(lie_group=H); k
exp(Z)
sage: k.parent() == L.lie_group()
False
```

(continues on next page)

912 Chapter 4. Individual Categories
Lie group $H$ of Free Nilpotent Lie algebra on 3 generators $(X, Y, Z)$ over Rational Field

```python
>>> g.parent() == k.parent()
False
```

**killing_form($x$)**

Return the Killing form of `self` and $x$.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: a, b, c = L.lie_algebra_generators()  # needs sage.modules
sage: a.killing_form(b)  # needs sage.modules
0
```

**lift()**

Return the image of `self` under the canonical lift from the Lie algebra to its universal enveloping algebra.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.libs.singular sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
```

```python
sage: a, b, c = L.lie_algebra_generators()
```

```python
sage: elt = 3*a + b - c
```

```python
sage: elt.lift()
3*b0 + b1 - b2
```

```python
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)  # needs sage.combinat sage.modules
sage: x.lift()  # needs sage.combinat sage.modules
x
```
from sage.all import *

L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_ngens(2) # needs sage.combinat sage.modules
x.lift() # needs sage.combinat sage.modules

.. automethod:: to_vector

Return the vector in g.module() corresponding to the element self of g (where g is the parent of self).

Implement this if you implement g.module(). See LieAlgebras.module() for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example() # needs sage.modules
sage: u = L((1, 0, 0)).to_vector(); u
(1, 0, 0)
sage: parent(u) # needs sage.modules
Vector space of dimension 3 over Rational Field
```

### class FiniteDimensional(base_category)

**Bases:** :class:`CategoryWithAxiom_over_base_ring`

**WithBasis**

alias of :class:`FiniteDimensionalLieAlgebrasWithBasis`

**extra_super_categories()**

Implements the fact that a finite dimensional Lie algebra over a finite ring is finite.

**EXAMPLES:**

```python
c\[
sage: LieAlgebras(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
[Category of finite sets]
esage: LieAlgebras(ZZ).FiniteDimensional().extra_super_categories()
[]
esage: C = LieAlgebras(GF(5)).FiniteDimensional()
esage: C.is_subcategory(Sets().Finite())
True
esage: C = LieAlgebras(ZZ).FiniteDimensional()
esage: C.is_subcategory(Sets().Finite())
False
\]
```
Graded

alias of GradedLieAlgebras

class Nilpotent (base_category)

Bases: CategoryWithAxiom_over_base_ring
Category of nilpotent Lie algebras.

class ParentMethods

Bases: object

is_nilpotent ()

Return True since self is nilpotent.

EXAMPLES:

```
sage: h = lie_algebras.Heisenberg(ZZ, oo)  # needs sage.combinat sage.modules
sage: h.is_nilpotent()                       # needs sage.combinat sage.modules
True
```

step ()

Return the nilpotency step of self.

EXAMPLES:

```
sage: h = lie_algebras.Heisenberg(ZZ, oo)  # needs sage.combinat sage.modules
sage: h.is_nilpotent()                       # needs sage.combinat sage.modules
True
```

(continues on next page)
class ParentMethods

Bases: object

\texttt{baker\_c\_hau\_s\_dorff}(X, Y, \texttt{prec=\texttt{None}})

Return the element \(\log(\exp(X)\exp(Y))\).

The BCH formula is an expression for \(\log(\exp(X)\exp(Y))\) as a sum of Lie brackets of \(X\) and \(Y\) with rational coefficients. It is only defined if the base ring of \texttt{self} has a coercion from the rationals.

**INPUT:**
- \(X\) – an element of \texttt{self}
- \(Y\) – an element of \texttt{self}
- \(\texttt{prec}\) – an integer; the maximum length of Lie brackets to be considered in the formula

**EXAMPLES:**

The BCH formula for the generators of a free nilpotent Lie algebra of step 4:

\begin{verbatim}
sage: L = LieAlgebra(QQ, 2, step=4)  # needs sage.combinat sage.modules
sage: L.inject_variables()  # needs sage.combinat sage.modules
Defining X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222
sage: L.bch(X_1, X_2)  # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122
\end{verbatim}

An example of the BCH formula in a quotient:

\begin{verbatim}
sage: Q = L.quotient(X_112 + X_122)  # needs sage.combinat sage.modules
sage: x, y = Q.basis().list()[:2]  # needs sage.combinat sage.modules
sage: Q.bch(x, y)  # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122
\end{verbatim}
The BCH formula for a non-nilpotent Lie algebra requires the precision to be explicitly stated:

```
sage: L.<X,Y> = LieAlgebra(QQ)
# needs sage.combinat sage.modules
sage: L.bch(X, Y)
# needs sage.combinat sage.modules
Traceback (most recent call last):
...  
ValueError: the Lie algebra is not known to be nilpotent,
so you must specify the precision
```

```
>>> from sage.all import *  
>>> L = LieAlgebra(QQ, names=('X', 'Y')); (X, Y,) = L._first_ngens(2)#
# needs sage.combinat sage.modules
>>> L.bch(X, Y)
# needs sage.combinat sage.modules
Traceback (most recent call last):
...  
ValueError: the Lie algebra is not known to be nilpotent,
so you must specify the precision
```

The BCH formula requires a coercion from the rationals:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2)  
# needs sage.combinat sage.modules
sage: L.bch(X, Y)
# needs sage.combinat sage.modules
Traceback (most recent call last):
...  
TypeError: the BCH formula is not well defined
since Integer Ring has no coercion from Rational Field
```

(continues on next page)
The BCH formula is not well defined since Integer Ring has no coercion from Rational Field

\( \text{bch} \ (X, Y, \text{prec=None}) \)

Return the element \( \log(\exp(X) \exp(Y)) \).

The BCH formula is an expression for \( \log(\exp(X) \exp(Y)) \) as a sum of Lie brackets of \( X \) and \( Y \) with rational coefficients. It is only defined if the base ring of \( \text{self} \) has a coercion from the rationals.

**INPUT:**
- \( X \) – an element of \( \text{self} \)
- \( Y \) – an element of \( \text{self} \)
- \( \text{prec} \) – an integer; the maximum length of Lie brackets to be considered in the formula

**EXAMPLES:**

The BCH formula for the generators of a free nilpotent Lie algebra of step 4:

```
sage: L = LieAlgebra(QQ, 2, step=4) # needs sage.combinat sage.modules
sage: L.inject_variables() # needs sage.combinat sage.modules
Defining X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222
sage: L.bch(X_1, X_2) # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122
```

An example of the BCH formula in a quotient:

```
sage: Q = L.quotient(X_112 + X_122) # needs sage.combinat sage.modules
sage: x, y = Q.basis().list()[:2] # needs sage.combinat sage.modules
sage: Q.bch(x, y) # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 - 1/24*X_1112
```

The BCH formula for a non-nilpotent Lie algebra requires the precision to be explicitly stated:

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4)) # needs sage.combinat sage.modules
>>> L.inject_variables() # needs sage.combinat sage.modules
Defining X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222
>>> L.bch(X_1, X_2) # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122
```

```
>>> from sage.all import *
>>> Q = L.quotient(X_112 + X_122) # needs sage.combinat sage.modules
>>> x, y = Q.basis().list()[:Integer(2)] # needs sage.combinat sage.modules
>>> Q.bch(x, y) # needs sage.combinat sage.modules
X_1 + X_2 + 1/2*X_12 - 1/24*X_1112
```
The BCH formula requires a coercion from the rationals:

```sage
sage: L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2) #--
                     --needs sage.combinat sage.modules
sage: L.bch(X, Y) #--
                     --needs sage.combinat sage.modules
Traceback (most recent call last):
...
TypeError: the BCH formula is not well defined
```

```sage```
>>> from sage.all import *

```sage```

**bracket** *(lhs, rhs)*

Return the Lie bracket `[lhs, rhs] after coercing lhs and rhs into elements of self.*

If `lhs` and `rhs` are Lie algebras, then this constructs the product space, and if only one of them is a Lie algebra, then it constructs the corresponding ideal.

**EXAMPLES:**
Constructing the product space:

```python
sage: L = lie_algebras.Heisenberg(QQ, 1)  # needs sage.combinat sage.modules
sage: Z = L.bracket(L, L); Z  # needs sage.combinat sage.modules
Ideal (z) of Heisenberg algebra of rank 1 over Rational Field
sage: L.bracket(L, Z)  # needs sage.combinat sage.modules
Ideal () of Heisenberg algebra of rank 1 over Rational Field
```

Constructing ideals:

```python
sage: p, q, z = L.basis(); p, q, z  # needs sage.combinat sage.modules
(p1, q1, z)
sage: L.bracket(3*p, L)  # needs sage.combinat sage.modules
Ideal (3*p1) of Heisenberg algebra of rank 1 over Rational Field
sage: L.bracket(L, q + p)  # needs sage.combinat sage.modules
Ideal (p1 + q1) of Heisenberg algebra of rank 1 over Rational Field
```

(continues on next page)
center_universal_enveloping_algebra ($UEA=None$)  

Return the center of the universal enveloping algebra of self.

EXAMPLES:

```python
sage: L = LieAlgebra(QQ, 3, 'x', abelian=True)
sage: L.center_universal_enveloping_algebra()
Center of Universal enveloping algebra of Abelian Lie algebra on 3 generators (x0, x1, x2) over Rational Field in the Poincare-Birkhoff-Witt basis
sage: PBW = L.pbw_basis()
sage: L.center_universal_enveloping_algebra(PBW)
Center of Universal enveloping algebra of Abelian Lie algebra on 3 generators (x0, x1, x2) over Rational Field in the Poincare-Birkhoff-Witt basis
```

from_vector ($v$, order=None, coerce=False)  

Return the element of self corresponding to the vector $v$ in self.module().

Implement this if you implement module(); see the documentation of the latter for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  
# needs sage.combinat sage.modules
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
sage: parent(u) is L  
# needs sage.combinat sage.modules
True
```
ideal(*gens, **kwds)

Return the ideal of self generated by gens.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: a, b, c = L.lie_algebra_generators()  # needs sage.modules
sage: L.ideal([2*a - c, b + c])  # needs sage.modules
An example of a finite dimensional Lie algebra with basis: the 2-dimensional abelian Lie algebra over Rational Field with basis matrix:
[ 1 0 -1/2]
[ 0 1  1]
```

```
>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
>>> a, b, c = L.lie_algebra_generators()  # needs sage.modules
>>> L.ideal([Integer(2)*a - c, b + c])  # needs sage.modules
An example of a finite dimensional Lie algebra with basis: the 2-dimensional abelian Lie algebra over Rational Field with basis matrix:
[ 1 0 -1/2]
[ 0 1  1]
```

```
sage: # needs sage.combinat sage.groups sage.modules
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: L.ideal([x + y])
Traceback (most recent call last):
...
NotImplementedError: ideals not yet implemented: see #16824
```

```
>>> from sage.all import *

>>> # needs sage.combinat sage.groups sage.modules
>>> L = LieAlgebras(QQ).example()
>>> x, y = L.lie_algebra_generators()
>>> L.ideal([x + y])
Traceback (most recent call last):
...
NotImplementedError: ideals not yet implemented: see #16824
```

is_abelian()
Return `True` if this Lie algebra is abelian.

A Lie algebra $\mathfrak{g}$ is abelian if $[x, y] = 0$ for all $x, y \in \mathfrak{g}$.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).example()  # needs sage.groups
sage: L.is_abelian()  # needs sage.groups
False
sage: R = QQ['x', 'y']
```

```python
sage: L = LieAlgebras(QQ).example(R.gens())
```

```python
sage: L.is_abelian()
```

```
True
```

```python
sage: # not implemented, needs sage.combinat sage.modules
sage: L.<x> = LieAlgebra(QQ, 1)
```

```python
sage: L.is_abelian()
```

```
True
```

```python
sage: L.<x,y> = LieAlgebra(QQ, 2)
```

```python
sage: L.is_abelian()
```

```
False
```

`is_commutative()`

Return if `self` is commutative. This is equivalent to `self` being abelian.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).example()  # needs sage.combinat sage.groups sage.modules
sage: L.is_commutative()  # needs sage.combinat sage.groups sage.modules
False
```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).example() # needs sage.combinat sage.groups sage.modules
>>> L.is_commutative() # needs sage.combinat sage.groups sage.modules
False

sage: L.<x> = LieAlgebra(QQ, 1) # not implemented # needs sage.combinat sage.modules
sage: L.is_commutative() # not implemented # needs sage.combinat sage.modules
True

>>> from sage.all import *
>>> L = LieAlgebras(QQ).example() # needs sage.combinat sage.groups sage.modules
>>> L.is_ideal(L) # needs sage.combinat sage.groups sage.modules
True

is_ideal(A)
Return if self is an ideal of A.

EXAMPLES:

sage: L = LieAlgebras(QQ).example() # needs sage.combinat sage.modules
sage: L.is_ideal(L) # needs sage.combinat sage.modules
True

>>> from sage.all import *
>>> L = LieAlgebras(QQ).example() # needs sage.combinat sage.groups sage.modules
>>> L.is_nilpotent() # needs sage.combinat sage.groups sage.modules
True

is_nilpotent()
Return if self is a nilpotent Lie algebra.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example() # needs sage.modules
sage: L.is_nilpotent() # needs sage.modules
True

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example() # needs sage.modules
>>> L.is_nilpotent() # needs sage.modules
True
**is_solvable()**

Return if `self` is a solvable Lie algebra.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L.is_solvable()  # needs sage.modules
True
```

```
>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
>>> L.is_solvable()  # needs sage.modules
True
```

**killing_form(x, y)**

Return the Killing form of `x` and `y`.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: a, b, c = L.lie_algebra_generators()

sage: L.killing_form(a, b + c)  # needs sage.modules
0
```

```
>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules

>>> a, b, c = L.lie_algebra_generators()

>>> L.killing_form(a, b + c)  # needs sage.modules
0
```

**lie_group(name='G', **kwds)**

Return the simply connected Lie group related to `self`.

**INPUT:**

- `name` - string (default: `'G'`); the name (symbol) given to the Lie group

**EXAMPLES:**

```
sage: L = lie_algebras.Heisenberg(QQ, 1)  # needs sage.combinat, sage.modules
sage: G = L.lie_group('G'); G
Lie group G of Heisenberg algebra of rank 1 over Rational Field
```

```
>>> from sage.all import *

>>> L = lie_algebras.Heisenberg(QQ, Integer(1))  # needs sage.combinat, sage.modules

>>> G = L.lie_group('G'); G
```

(continues on next page)
Construct the lift morphism from self to the universal enveloping algebra of self (the latter is implemented as `universal_enveloping_algebra()`).

This is a Lie algebra homomorphism. It is injective if self is a free module over its base ring, or if the base ring is a $\mathbb{Q}$-algebra.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: lifted = L.lift(2*a + b - c); lifted
2*b0 + b1 - b2
sage: lifted.parent() is L.universal_enveloping_algebra()
True
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> lifted = L.lift(Integer(2)*a + b - c); lifted
2*b0 + b1 - b2
>>> lifted.parent() is L.universal_enveloping_algebra()
True
```

Return an $R$-module which is isomorphic to the underlying $R$-module of self.

The rationale behind this method is to enable linear algebraic functionality on self (such as computing the span of a list of vectors in self) via an isomorphism from self to an $R$-module (typically, although not always, an $R$-module of the form $R^n$ for an $n \in \mathbb{N}$) on which such functionality already exists. For this method to be of any use, it should return an $R$-module which has linear algebraic functionality that self does not have.

For instance, if self has ordered basis $(e, f, h)$, then self.module() will be the $R$-module $R^3$, and the elements $e$, $f$ and $h$ of self will correspond to the basis vectors $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ of self.module().

This method `module()` needs to be set whenever a finite-dimensional Lie algebra with basis is intended to support linear algebra (which is, e.g., used in the computation of centralizers and lower central series). One then needs to also implement the $R$-module isomorphism from self to self.module() in both directions; that is, implement:

- a `to_vector` `ElementMethod` which sends every element of self to the corresponding element of self.module();
- a `from_vector` `ParentMethod` which sends every element of self.module() to an element of self.

The `from_vector` method will automatically serve as an element constructor of self (that is, `self(v)` for any `v` in self.module() will return self.from_vector(v)).

**Todo:** Ensure that this is actually so.
EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # Needs sage.modules
sage: L.module()  # Needs sage.modules
Vector space of dimension 3 over Rational Field
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # Needs sage.modules
>>> L.module()  # Needs sage.modules
Vector space of dimension 3 over Rational Field
```

representation ($f=\text{None}, \text{index\_set}=\text{None}, \text{on\_basis}=\text{False}, \text{**kwargs}$)

Return a representation of self.

If no arguments are given, then this returns the trivial representation.

Currently the only implemented method of constructing a representation is by explicitly specifying the action of

- the elements of self by matrices;
- the basis elements of self using a dict or a Family();
- a function on basis elements (either passed as on\_basis or setting on\_basis=True).

INPUT:

- $f$ – the function that defines the action
- index\_set – the index set of the representation
- on\_basis – (optional) see above

See also:

RepresentationByMorphism

EXAMPLES:

```python
sage: L.<x,y> = LieAlgebra(QQ, {'x','y'}: {'y':1})
sage: f = {x: Matrix([[1,0],[0,0]]), y: Matrix([[0,1],[0,0]])}
sage: L.representation(f)
Representation of Lie algebra on 2 generators (x, y) over Rational Field defined by:
[1 0]
|--> [0 0]
[0 1]
y |--> [0 0]
sage: L.representation()
Trivial representation of Lie algebra on 2 generators (x, y) over Rational Field
```

```python
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y'}: {'y':Integer(1)}, names=('x', 'y',));
(x, y,) = L._first_ngens(2)
>>> f = {x: Matrix([[Integer(1),Integer(0)],[Integer(0),Integer(0)]]), y: Matrix([[Integer(0),Integer(1)],[Integer(0),Integer(0)]]))
>>> L.representation(f)
Representation of Lie algebra on 2 generators (x, y) over Rational Field defined by:
[1 0]
x |--> [0 0]
(continues on next page)"
trivial_representation()  

Return the trivial representation of self.
EXAMPLES:

```
sage: L = lie_algebras.strictly_upper_triangular_matrices(QQ, 4)
sage: L.trivial_representation()
```
Trivial representation of Lie algebra of 4-dimensional strictly upper triangular matrices over Rational Field

```
>>> from sage.all import *
>>> L = lie_algebras.strictly_upper_triangular_matrices(QQ, Integer(4))
>>> L.trivial_representation()
```
Trivial representation of Lie algebra of 4-dimensional strictly upper triangular matrices over Rational Field

```
universal_enveloping_algebra()
```
Return the universal enveloping algebra of self.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L.universal_enveloping_algebra()  # needs sage.combinat sage.libs.singular sage.modules
Noncommutative Multivariate Polynomial Ring in b0, b1, b2 over Rational Field, nc-relations: {}
```

```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
>>> L.universal_enveloping_algebra()  # needs sage.combinat sage.libs.singular sage.modules
Noncommutative Multivariate Polynomial Ring in b0, b1, b2 over Rational Field, nc-relations: {}
```

```
sage: L = LieAlgebra(QQ, 3, x, abelian=True)  # needs sage.combinat sage.modules
sage: L.universal_enveloping_algebra()  # needs sage.combinat sage.libs.singular sage.modules
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x', abelian=True)  # needs sage.combinat sage.modules
>>> L.universal_enveloping_algebra()  # needs sage.combinat sage.libs.singular sage.modules
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

See also:

```
lift()
```

class SubcategoryMethods

Bases: object

```
Nilpotent()
```

Return the full subcategory of nilpotent objects of self.

A Lie algebra $L$ is nilpotent if there exist an integer $s$ such that all iterated brackets of $L$ of length more than $s$ vanish. The integer $s$ is called the nilpotency step. For instance any abelian Lie algebra is nilpotent.
of step 1.

EXAMPLES:

```
sage: LieAlgebras(QQ).Nilpotent()
Category of nilpotent Lie algebras over Rational Field
sage: LieAlgebras(QQ).WithBasis().Nilpotent()
Category of nilpotent Lie algebras with basis over Rational Field
```

```
>>> from sage.all import *
>>> LieAlgebras(QQ).Nilpotent()
Category of nilpotent Lie algebras over Rational Field
>>> LieAlgebras(QQ).WithBasis().Nilpotent()
Category of nilpotent Lie algebras with basis over Rational Field
```

**WithBasis**

alias of :class:`LieAlgebrasWithBasis`

**example**(gens=None)

Return an example of a Lie algebra as per :class:`Category.example`.

EXAMPLES:

```
sage: LieAlgebras(QQ).example()  # needs sage.combinat sage.groups sage.modules
An example of a Lie algebra: the Lie algebra from the associative algebra
Symmetric group algebra of order 3 over Rational Field
generated by ([2, 1, 3], [2, 3, 1])
```

```
>>> from sage.all import *
>>> LieAlgebras(QQ).example()  # needs sage.combinat sage.groups sage.modules
An example of a Lie algebra: the Lie algebra from the associative algebra
Symmetric group algebra of order 3 over Rational Field
generated by ([2, 1, 3], [2, 3, 1])
```

Another set of generators can be specified as an optional argument:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)  # needs sage.combinat sage.modules
sage: LieAlgebras(QQ).example(F.gens())  # needs sage.combinat sage.modules
An example of a Lie algebra: the Lie algebra from the associative algebra
Free Algebra on 3 generators (x, y, z) over Rational Field
generated by (x, y, z)
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
    # needs sage.combinat sage.modules
>>> LieAlgebras(QQ).example(F.gens())  # needs sage.combinat sage.modules
An example of a Lie algebra: the Lie algebra from the associative algebra
Free Algebra on 3 generators (x, y, z) over Rational Field
generated by (x, y, z)
```

**super_categories**()

EXAMPLES:
sage: LieAlgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

>>> from sage.all import *
>>> LieAlgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

class sage.categories.lie_algebras.LiftMorphism(domain, codomain)
    Bases: Morphism

    The natural lifting morphism from a Lie algebra to its enveloping algebra.

4.105 Lie Algebras With Basis

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

class sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis(base_category)
    Bases: CategoryWithAxiom_over_base_ring

    Category of Lie algebras with a basis.

class ElementMethods
    Bases: object

    lift()
    
    Lift self to the universal enveloping algebra.

    EXAMPLES:

sage: # needs sage.combinat sage.groups
sage: S = SymmetricGroup(3).algebra(QQ)
sage: L = LieAlgebra(associative=S)
sage: x = L.gen(3)
sage: y = L.gen(1)
sage: x.lift()
b3
sage: y.lift()
b1
sage: x * y
b1*b3 + b4 - b5

>>> # needs sage.combinat sage.groups
>>> S = SymmetricGroup(Integer(3)).algebra(QQ)
>>> L = LieAlgebra(associative=S)
>>> x = L.gen(Integer(3))
>>> y = L.gen(Integer(1))
>>> x.lift()
b3
>>> y.lift()
b1
>>> x * y
b1*b3 + b4 - b5

4.105. Lie Algebras With Basis 931
to_vector(order=None)

Return the vector in g.module() corresponding to the element self of g (where g is the parent of self).

Implement this if you implement g.module(). See sage.categories.lie_algebras.LieAlgebras.module() for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.modules
sage: L.an_element().to_vector()  # needs sage.modules
(0, 0, 0)
```

Todo: Doctest this implementation on an example not overshadowed.

Graded

alias of GradedLieAlgebrasWithBasis

class ParentMethods

Bases: object

bracket_on_basis(x, y)

Return the bracket of basis elements indexed by x and y where x < y. If this is not implemented, then the method _bracket_() for the elements must be overwritten.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()  # needs sage.combinat sage.modules
sage: L.bracket_on_basis(Partition([3, 1]), Partition([2, 2, 1, 1]))  # needs sage.combinat sage.modules
0
```

>>> from sage.all import *

>>> L = LieAlgebras(QQ).WithBasis().example()  # needs sage.combinat sage.modules
>>> L.bracket_on_basis(Partition([Integer(3), Integer(1)]), Partition([Integer(2), Integer(2), Integer(1), Integer(1)]))  # needs sage.combinat sage.modules
0

dimension()

Return the dimension of self.

EXAMPLES:
from sage.all import *

L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #
-> needs sage.modules

L.dimension()  #
-> needs sage.modules

L = LieAlgebra(QQ, 'x,y', {('x', 'y'): {x:1}})  #
-> needs sage.combinat sage.modules

L.dimension()  #
-> needs sage.combinat sage.modules

from_vector(v, order=None, coerce=False)

Return the element of self corresponding to the vector v in self.module().

Implement this if you implement module(); see the documentation of sage.categories.lie_algebras.LieAlgebras.module() for how this is to be done.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #
-> needs sage.modules

sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u  #
-> needs sage.modules
(1, 0, 0)

sage: parent(u) is L  #
-> needs sage.modules
True

module()

Return an $R$-module which is isomorphic to the underlying $R$-module of self.

See sage.categories.lie_algebras.LieAlgebras.module() for an explanation.
EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
# needs sage.combinat sage.modules
sage: L.module()
# needs sage.combinat sage.modules
Free module generated by Partitions over Rational Field
```

```python
>>> from sage.all import *

>>> L = LieAlgebras(QQ).WithBasis().example()
# needs sage.combinat sage.modules
>>> L.module()
# needs sage.combinat sage.modules
Free module generated by Partitions over Rational Field
```

```
pbw_basis (basis_key=None, **kwds)
```

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to `self`.

EXAMPLES:

```python
sage: L = lie_algebras.sl(QQ, 2)
# needs sage.combinat sage.modules
sage: PBW = L.pbw_basis()
# needs sage.combinat sage.modules
```

```python
>>> from sage.all import *

>>> L = lie_algebras.sl(QQ, Integer(2))
# needs sage.combinat sage.modules
>>> PBW = L.pbw_basis()
# needs sage.combinat sage.modules
```

```
poincare_birkhoff_witt_basis (basis_key=None, **kwds)
```

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to `self`.

EXAMPLES:

```python
sage: L = lie_algebras.sl(QQ, 2)
# needs sage.combinat sage.modules
sage: PBW = L.pbw_basis()
# needs sage.combinat sage.modules
```

```python
>>> from sage.all import *

>>> L = lie_algebras.sl(QQ, Integer(2))
# needs sage.combinat sage.modules
>>> PBW = L.pbw_basis()
# needs sage.combinat sage.modules
```

```
example (gens=None)
```

Return an example of a Lie algebra as per `Category.example`.

EXAMPLES:

```python
sage: LieAlgebras(QQ).WithBasis().example()
# needs sage.combinat sage.modules
```

An example of a Lie algebra: the abelian Lie algebra on the generators indexed by Partitions over Rational Field
```python
>>> from sage.all import *

>>> LieAlgebras(QQ).WithBasis().example()  # needs sage.combinat sage.modules
An example of a lie algebra: the abelian Lie algebra on the generators indexed by Partitions over Rational Field

A different set of generators can be specified as an optional argument:

```python
sage: LieAlgebras(QQ).WithBasis().example(Compositions())  # needs sage.combinat sage.modules
An example of a lie algebra: the abelian Lie algebra on the generators indexed by Compositions of non-negative integers over Rational Field

```python
>>> from sage.all import *

>>> LieAlgebras(QQ).WithBasis().example(Compositions())  # needs sage.combinat sage.modules
An example of a lie algebra: the abelian Lie algebra on the generators indexed by Compositions of non-negative integers over Rational Field
```

## 4.106 Lie Conformal Algebras

Let $R$ be a commutative ring, a super Lie conformal algebra [Kac1997] over $R$ (also known as a vertex Lie algebra) is an $R[T]$ super module $L$ together with a $\mathbb{Z}/2\mathbb{Z}$-graded $R$-bilinear operation (called the $\lambda$-bracket) $L \otimes L \to L[\lambda]$ (polynomials in $\lambda$ with coefficients in $L$), $a \otimes b \mapsto [a_\lambda b]$ satisfying

1. Sesquilinearity:

   
   
   \[
   [Ta_\lambda b] = -\lambda [a_\lambda b], \quad [a_\lambda Tb] = (\lambda + T)[a_\lambda b].
   \]

2. Skew-Symmetry:

   
   \[
   [a_\lambda b] = -(-1)^{p(a)p(b)}[b_{-\lambda - T}a],
   \]

   where $p(a)$ is 0 if $a$ is even and 1 if $a$ is odd. The bracket in the RHS is computed as follows. First we evaluate $[b_\mu a]$ with the formal parameter $\mu$ to the left, then replace each appearance of the formal variable $\mu$ by $-\lambda - T$. Finally apply $T$ to the coefficients in $L$.

3. Jacobi identity:

   
   \[
   [a_\lambda [b_\mu c]] = [[a_\lambda+\mu b_\mu c] + (-1)^{p(a)p(b)}[b_\mu [a_\lambda c]]],
   \]

   which is understood as an equality in $L[\lambda, \mu]$.

$T$ is usually called the translation operation or the derivative. For an element $a \in L$ we will say that $Ta$ is the derivative of $a$. We define the $n$-th products $a_{(n)}b$ for $a, b \in L$ by

\[
[a_\lambda b] = \sum_{n \geq 0} \frac{\lambda^n}{n!} a_{(n)} b.
\]

A Lie conformal algebra is called $H$-Graded [DSK2006] if there exists a decomposition $L = \oplus L_n$ such that the $\lambda$-bracket becomes graded of degree $-1$, that is:

\[
a_{(n)} b \in L_{p+q-n-1} \quad a \in L_p, \quad b \in L_q, \quad n \geq 0.
\]

In particular this implies that the action of $T$ increases degree by 1.
Note: In the literature arbitrary gradings are allowed. In this implementation we only support non-negative rational gradings.

EXAMPLES:

1. The Virasoro Lie conformal algebra $Vir$ over a ring $R$ where $12$ is invertible has two generators $L, C$ as an $R[T]$-module. It is the direct sum of a free module of rank 1 generated by $L$, and a free rank one $R$ module generated by $C$ satisfying $TC = 0$. $C$ is central (the $\lambda$-bracket of $C$ with any other vector vanishes). The remaining $\lambda$-bracket is given by

$$[L_\lambda L] = TL + 2\lambda L + \frac{\lambda^3}{12} C.$$

2. The affine or current Lie conformal algebra $L(\mathfrak{g})$ associated to a finite dimensional Lie algebra $\mathfrak{g}$ with non-degenerate, invariant $R$-bilinear form $(,)$ is given as a central extension of the free $R[T]$ module generated by $\mathfrak{g}$ by a central element $K$. The $\lambda$-bracket of generators is given by

$$[a_\lambda b] = [a, b] + \lambda(a, b)K, \quad a, b \in \mathfrak{g}$$

3. The Weyl Lie conformal algebra, or $\beta - \gamma$ system is given as the central extension of a free $R[T]$ module with two generators $\beta$ and $\gamma$, by a central element $K$. The only non-trivial brackets among generators are

$$[\beta_\lambda \gamma] = -[\gamma_\lambda \beta] = K$$

4. The Neveu-Schwarz super Lie conformal algebra is a super Lie conformal algebra which is an extension of the Virasoro Lie conformal algebra. It consists of a Virasoro generator $L$ as in example 1 above and an odd generator $G$. The remaining brackets are given by:

$$[L_\lambda G] = \left( T + \frac{3}{2} \lambda \right) G \quad [G_\lambda G] = 2L + \frac{\lambda^2}{3} C$$

See also:

- sage.algebras.lie_conformal_algebras.lie_conformal_algebra
- sage.algebras.lie_conformal_algebras.examples

AUTHORS:


class sage.categories.lie_conformal_algebras.LieConformalAlgebras (base, name=None)

    Bases: Category_over_base_ring

    The category of Lie conformal algebras.

    This is the base category for all Lie conformal algebras. Subcategories with axioms are FinitelyGenerated and WithBasis. A finitely generated Lie conformal algebra is a Lie conformal algebra over $R$ which is finitely generated as an $R[T]$-module. A Lie conformal algebra with basis is one with a preferred basis as an $R$-module.

    EXAMPLES:

    The base category:
sage: C = LieConformalAlgebras(QQ); C
Category of Lie conformal algebras over Rational Field
sage: C.is_subcategory(VectorSpaces(QQ))
True

>>> from sage.all import *
>>> C = LieConformalAlgebras(QQ); C
Category of Lie conformal algebras over Rational Field
>>> C.is_subcategory(VectorSpaces(QQ))
True

Some subcategories:

sage: LieConformalAlgebras(QQbar).FinitelyGenerated().WithBasis() #...

needs sage.rings.number_field
Category of finitely generated Lie conformal algebras with basis over Algebraic Field

>>> from sage.all import *
>>> LieConformalAlgebras(QQbar).FinitelyGenerated().WithBasis() #...

needs sage.rings.number_field
Category of finitely generated Lie conformal algebras with basis over Algebraic Field

In addition we support functorial constructions Graded and Super. These functors commute:

sage: CGS = LieConformalAlgebras(AA).Graded().Super(); CGS #...

needs sage.rings.number_field
Category of H-graded super Lie conformal algebras over Algebraic Real Field
sage: CGS is LieConformalAlgebras(AA).Super().Graded() #...

needs sage.rings.number_field
True

>>> from sage.all import *
>>> CGS = LieConformalAlgebras(AA).Graded().Super(); CGS #...

needs sage.rings.number_field
Category of H-graded super Lie conformal algebras over Algebraic Real Field
>>> CGS is LieConformalAlgebras(AA).Super().Graded() #...

needs sage.rings.number_field
True

That is, we only consider gradings on super Lie conformal algebras that are compatible with the $\mathbb{Z}/2\mathbb{Z}$ grading.

The base ring needs to be a commutative ring:

sage: LieConformalAlgebras(QuaternionAlgebra(2)) #...

needs sage.combinat sage.modules
Traceback (most recent call last):
  File "", line 1, in ...
ValueError: base must be a commutative ring
got Quaternion Algebra (-1, -1) with base ring Rational Field

>>> from sage.all import *
>>> LieConformalAlgebras(QuaternionAlgebra(Integer(2))) ...

# needs sage.combinat sage.modules
Traceback (most recent call last):
  File "", line 1, in ...
ValueError: base must be a commutative ring
got Quaternion Algebra (-1, -1) with base ring Rational Field
class ElementMethods
Bases: object

is_even_odd()
    Return 0 if this element is even and 1 if it is odd.

Note: This method returns 0 by default since every Lie conformal algebra can be thought as a purely even Lie conformal algebra. In order to implement a super Lie conformal algebra, the user needs to implement this method.

EXAMPLES:

sage: R = lie_conformal_algebras.NeveuSchwarz(QQ)  # ...
    ← needs sage.combinat sage.modules
sage: R.inject_variables()  # ...
    ← needs sage.combinat sage.modules
Defining L, G, C
sage: G.is_even_odd()  # ...
    ← needs sage.combinat sage.modules
1

>>> from sage.all import *
>>> R = lie_conformal_algebras.NeveuSchwarz(QQ)  # ...
    ← needs sage.combinat sage.modules
>>> R.inject_variables()  # ...
    ← needs sage.combinat sage.modules
Defining L, G, C
>>> G.is_even_odd()  # ...
    ← needs sage.combinat sage.modules
1

FinitelyGeneratedAsLambdaBracketAlgebra
    alias of FinitelyGeneratedLieConformalAlgebras

Graded
    alias of GradedLieConformalAlgebras

class ParentMethods
Bases: object

Super
    alias of SuperLieConformalAlgebras

WithBasis
    alias of LieConformalAlgebrasWithBasis
e
e

example()
    An example of parent in this category.

EXAMPLES:

sage: LieConformalAlgebras(QQ).example()  # ...
    ← needs sage.combinat sage.modules
The Virasoro Lie conformal algebra over Rational Field
The Virasoro Lie conformal algebra over Rational Field

\textbf{super_categories()}

The list of super categories of this category.

\textbf{EXAMPLES:}

```
>>> from sage.all import *
>>> LieConformalAlgebras(QQ).example()  # needs sage.combinat sage.modules
The Virasoro Lie conformal algebra over Rational Field

sage: C = LieConformalAlgebras(QQ)
sage: C.super_categories()
(Category of Lambda bracket algebras over Rational Field)

sage: C = LieConformalAlgebras(QQ).FinitelyGenerated(); C
Category of finitely generated Lie conformal algebras over Rational Field

sage: C.super_categories()
(Category of finitely generated lambda bracket algebras over Rational Field,
 Category of Lie conformal algebras over Rational Field)

sage: C.all_super_categories()
(Category of finitely generated Lie conformal algebras over Rational Field,
 Category of finitely generated lambda bracket algebras over Rational Field,
 Category of Lie conformal algebras over Rational Field,
 Category of Lambda bracket algebras over Rational Field,
 Category of vector spaces over Rational Field,
 Category of modules over Rational Field,
 Category of bimodules over Rational Field on the left and Rational Field on... the right,
 Category of right modules over Rational Field,
 Category of left modules over Rational Field,
 Category of commutative additive groups,
 Category of additive groups,
 Category of additive inverse unital additive magmas,
 Category of commutative additive monoids,
 Category of additive monoids,
 Category of additive unital additive magmas,
 Category of commutative additive semigroups,
 Category of additive semigroups,
 Category of additive magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects)
```

(continues on next page)
Category of vector spaces over Rational Field,
Category of modules over Rational Field,
Category of bimodules over Rational Field on the left and Rational Field on the right,
Category of right modules over Rational Field,
Category of left modules over Rational Field,
Category of commutative additive groups,
Category of additive groups,
Category of additive inverse additive unital additive magmas,
Category of commutative additive monoids,
Category of additive monoids,
Category of additive unital additive magmas,
Category of commutative additive semigroups,
Category of additive commutative additive magmas,
Category of additive semigroups,
Category of additive magmas,
Category of sets,
Category of sets with partial maps,
Category of objects]

4.107 Lie Conformal Algebras With Basis

AUTHORS:

• Reimundo Heluani (2019-10-05): Initial implementation.

class sage.categories.lie_conformal_algebras_with_basis.LieConformalAlgebrasWithBasis(base_category)

Bases: CategoryWithAxiom_over_base_ring

The category of Lie conformal algebras with basis.

EXAMPLES:

```python
sage: LieConformalAlgebras(QQbar).WithBasis()
# needs sage.rings.number_field
Category of Lie conformal algebras with basis over Algebraic Field
```

```python
>>> from sage.all import *
>>> LieConformalAlgebras(QQbar).WithBasis()
# needs sage.rings.number_field
Category of Lie conformal algebras with basis over Algebraic Field
```

class FinitelyGeneratedAsLambdaBracketAlgebra(base_category)

Bases: CategoryWithAxiom_over_base_ring

The category of finitely generated Lie conformal algebras with basis.

EXAMPLES:

```python
sage: C = LieConformalAlgebras(QQbar)
# needs sage.rings.number_field
```

```python
sage: CWF = C.WithBasis().FinitelyGenerated(); CWF
# needs sage.rings.number_field
```

(continues on next page)
Category of finitely generated Lie conformal algebras with basis over Algebraic Field

```
sage: CWF is C.FinitelyGenerated().WithBasis()  # needs sage.rings.number_field
True
```

```python
>>> from sage.all import *

>>> C = LieConformalAlgebras(QQbar)  # needs sage.rings.number_field

>>> CWF = C.WithBasis().FinitelyGenerated(); CWF
Category of finitely generated Lie conformal algebras with basis over Algebraic Field

>>> CWF is C.FinitelyGenerated().WithBasis()  # needs sage.rings.number_field
True
```

class Graded(base_category):
    Bases: GradedLieConformalAlgebrasCategory

The category of $H$-graded finitely generated Lie conformal algebras with basis.

EXAMPLES:

```
sage: LieConformalAlgebras(QQbar).WithBasis().FinitelyGenerated().Graded()  # needs sage.rings.number_field
Category of $H$-graded finitely generated Lie conformal algebras with basis over Algebraic Field
```

class Super(base_category):
    Bases: SuperModulesCategory

The category of super finitely generated Lie conformal algebras with basis.

EXAMPLES:

```
sage: LieConformalAlgebras(AA).WithBasis().FinitelyGenerated().Super()  # needs sage.rings.number_field
Category of super finitely generated Lie conformal algebras with basis over Algebraic Real Field
```

class Graded(base_category):
    Bases: GradedModulesCategory

The category of $H$-graded super finitely generated Lie conformal algebras with basis.
EXAMPLES:

```python
sage: C = LieConformalAlgebras(QQbar).WithBasis().FinitelyGenerated()  # needs sage.rings.number_field
sage: C.Graded().Super()  # needs sage.rings.number_field
Category of H-graded super finitely generated Lie conformal algebras with basis over Algebraic Field
sage: C.Graded().Super() is C.Super().Graded()  # needs sage.rings.number_field
True
```

```python
>>> from sage.all import *
>>> C = LieConformalAlgebras(QQbar).WithBasis().FinitelyGenerated()  # needs sage.rings.number_field
>>> C.Graded().Super()  # needs sage.rings.number_field
Category of H-graded super finitely generated Lie conformal algebras with basis over Algebraic Field

```
class Graded(base_category)
    Bases: GradedLieConformalAlgebrasCategory
    The category of H-graded super Lie conformal algebras with basis.
    EXAMPLES:

    sage: LieConformalAlgebras(QQbar).WithBasis().Super().Graded()  # needs sage.rings.number_field
    Category of H-graded super Lie conformal algebras with basis over Algebraic Field

    >>> from sage.all import *
    >>> LieConformalAlgebras(QQbar).WithBasis().Super().Graded()  # needs sage.rings.number_field
    Category of H-graded super Lie conformal algebras with basis over Algebraic Field

class ParentMethods
    Bases: object

4.108 Lie Groups

class sage.categories.lie_groups.LieGroups(base, name=None)
    Bases: Category_over_base_ring
    The category of Lie groups.
    A Lie group is a topological group with a smooth manifold structure.
    EXAMPLES:

    sage: from sage.categories.lie_groups import LieGroups
    sage: C = LieGroups(QQ); C
    Category of Lie groups over Rational Field

    >>> from sage.all import *
    >>> from sage.categories.lie_groups import LieGroups
    >>> C = LieGroups(QQ); C
    Category of Lie groups over Rational Field

    additional_structure()
    Return None.
    Indeed, the category of Lie groups defines no new structure: a morphism of topological spaces and of smooth
    manifolds is a morphism as Lie groups.
    See also:
    Category.additional_structure()
    EXAMPLES:

    sage: from sage.categories.lie_groups import LieGroups
    sage: LieGroups(QQ).additional_structure()
### 4.109 Loop Crystals

**class** `sage.categories.loop_crystals.KirillovReshetikhinCrystals`

Bases: `Category_singleton`

Category of Kirillov-Reshetikhin crystals.

**class** `ElementMethods`

Bases: `object`

**energy_function()**

Return the energy function of `self`.

Let $B$ be a KR crystal. Let $b^\dagger$ denote the unique element such that $\varphi(b^\dagger) = \ell \Lambda_0$ with $\ell = \min\{c, \varphi(b) | b \in B\}$. Let $u_B$ denote the maximal element of $B$. The energy of $b \in B$ is given by

$$D(b) = H(b \otimes b^\dagger) - H(u_B \otimes b^\dagger),$$

where $H$ is the local energy function.

**EXAMPLES:**

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: for x in K.classically_highest_weight_vectors():
    ....:     x, x.energy_function()
([], 1)
([[1], [2]], 0)
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,2)
sage: for x in K.classically_highest_weight_vectors():
    ....:     x, x.energy_function()
([], 2)
([[1]], 1)
([[1, 1]], 0)
```
```python
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],
                                           Integer(2), Integer(1))
>>> for x in K.classically_highest_weight_vectors():
    ...   x, x.energy_function()
    ([], 1)
    ([(1), [2]], 0)
>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(3)],
                                           Integer(1), Integer(2))
>>> for x in K.classically_highest_weight_vectors():
    ...   x, x.energy_function()
    ([], 2)
    ([(1)], 1)
    ([(1, 1)], 0)
```

```python
lusztig_involution()
```

Return the result of the classical Lusztig involution on `self`.

EXAMPLES:

```python
sage: KRT = crystals.KirillovReshetikhin(['D', 4, 1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]
sage: mg.lusztig_involution()
[[0, -2, 1], [1, -1, 2]]
sage: elt = mg.f_string([2, 1, 3, 2]); elt
[[0, -3, 1], [2, -1, 2]]
sage: elt.lusztig_involution()
[[0, -4, 1], [1, -2, 2]]
```

```python
class ParentMethods
Bases: object

R_matrix(K)
```

Return the combinatorial $R$-matrix of `self` to `K`.

The combinatorial $R$-matrix is the affine crystal isomorphism $R : L \otimes K \rightarrow K \otimes L$ which maps $u_L \otimes u_K$ to $u_K \otimes u_L$, where $u_K$ is the unique element in $K = B_{r,s}$ of weight $s\Lambda_r - sc\Lambda_0$ (see maximal_vector()).

INPUT:

- `self` – a crystal $L$
- `K` – a Kirillov-Reshetikhin crystal of the same type as $L$

EXAMPLES:

```python
>>> from sage.all import *
>>> KRT = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],
                                           Integer(2), Integer(1))
>>> mg = KRT.module_generators[Integer(1)]
>>> mg.lusztig_involution()
[[0, -2, 1], [1, -1, 2]]
>>> elt = mg.f_string([Integer(2), Integer(1), Integer(3), Integer(2)]); elt
[[0, -3, 1], [2, -1, 2]]
>>> elt.lusztig_involution()
[[0, -4, 1], [1, -2, 2]]
```
```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: L = crystals.KirillovReshetikhin(['A',2,1],1,2)
sage: f = K.R_matrix(L)
sage: [[b, f(b)] for b in crystals.TensorProduct(K, L)]
```

```
[[[[[1]], [[1, 1]]], [[[1, 1]], [[1]]]],
 [[[1]], [[1, 2]]], [[[1, 1]], [[2]]]],
 [[[1]], [[2, 2]]], [[[1, 1]], [[2, 1]]],
 [[[1]], [[1, 3]]], [[[1, 1]], [[3]]]],
 [[[1]], [[2, 3]]], [[[1, 2]], [[2]]]],
 [[[1]], [[3, 3]]], [[[1, 3]], [[3]]]],
 [[[2]], [[1, 1]]], [[[2, 1]], [[1]]]],
 [[[2]], [[1, 2]]], [[[2, 1]], [[2]]]],
 [[[2]], [[2, 2]]], [[[2, 1]], [[2, 2]]],
 [[[2]], [[1, 3]]], [[[2, 1]], [[3]]]],
 [[[2]], [[2, 3]]], [[[2, 2]], [[2]]]],
 [[[2]], [[3, 3]]], [[[2, 3]], [[3]]]],
 [[[3]], [[1, 1]]], [[[3, 1]], [[1]]]],
 [[[3]], [[1, 2]]], [[[3, 1]], [[2]]]],
 [[[3]], [[2, 2]]], [[[3, 1]], [[2, 2]]],
 [[[3]], [[1, 3]]], [[[3, 2]], [[2]]]],
 [[[3]], [[2, 3]]], [[[3, 2]], [[3]]]],
 [[[3]], [[3, 3]]], [[[3, 2]], [[3]]]],
 [[[3]], [[3, 3]]], [[[3, 3]], [[3]]]]
```

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],1,1)
sage: L = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: f = K.R_matrix(L)
sage: T = crystals.TensorProduct(K,L)
sage: b = T( K(rows=[[1]]), L(rows=[]) )
sage: f(b)
```

```
[[[2], [-2]], [[1]]]
```

```
>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
˓→Integer(1),Integer(1))
>>> L = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
˓→Integer(1),Integer(2))
>>> f = K.R_matrix(L)
>>> [[b, f(b)] for b in crystals.TensorProduct(K, L)]
```

```
[[[[[1]], [[1, 1]]], [[[1, 1]], [[1]]]],
 [[[1]], [[1, 2]]], [[[1, 1]], [[2]]]],
 [[[1]], [[2, 2]]], [[[1, 1]], [[2, 1]]],
 [[[1]], [[1, 3]]], [[[1, 1]], [[3]]]],
 [[[1]], [[2, 3]]], [[[1, 2]], [[2]]]],
 [[[1]], [[3, 3]]], [[[1, 3]], [[3]]]],
 [[[2]], [[1, 1]]], [[[2, 1]], [[1]]]],
 [[[2]], [[1, 2]]], [[[2, 1]], [[2]]]],
 [[[2]], [[2, 2]]], [[[2, 1]], [[2, 2]]],
 [[[2]], [[1, 3]]], [[[2, 1]], [[3]]]],
 [[[2]], [[2, 3]]], [[[2, 2]], [[2]]]],
 [[[2]], [[3, 3]]], [[[2, 3]], [[3]]]],
 [[[3]], [[1, 1]]], [[[3, 1]], [[1]]]],
 [[[3]], [[1, 2]]], [[[3, 1]], [[2]]]],
 [[[3]], [[2, 2]]], [[[3, 1]], [[2, 2]]],
 [[[3]], [[1, 3]]], [[[3, 2]], [[2]]]],
 [[[3]], [[2, 3]]], [[[3, 2]], [[3]]]],
 [[[3]], [[3, 3]]], [[[3, 2]], [[3]]]],
 [[[3]], [[3, 3]]], [[[3, 3]], [[3]]]]
```

(continues on next page)
| K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(1), Integer(1)) |
| L = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)], Integer(2), Integer(1)) |
| f = K.R_matrix(L) |
| T = crystals.TensorProduct(K, L) |
| b = T( K(rows=[Integer(1)]), L(rows=[]) ) |
| f(b) |

```python
[[[2], [-2]], [[1]]]
```

Alternatively, one can compute the combinatorial $R$-matrix using the isomorphism method of digraphs:

```python
sage: K1 = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K2 = crystals.KirillovReshetikhin(['A',2,1], 2, 1)
sage: T1 = crystals.TensorProduct(K1,K2)
sage: T2 = crystals.TensorProduct(K2,K1)
sage: T1.digraph().is_isomorphic(T2.digraph(), edge_labels=True, # todo: not implemented (see #10904 and #10549)
(continues on next page)
```

**affinization()**

Return the corresponding affinization crystal of self.

**EXAMPLES:**

```python
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.affinization()
```

(continues on next page)
Affinization of Kirillov-Reshetikhin crystal of type $[\text{A'}, 2, 1]$ with $(r, s) = (1, 1)$

\begin{verbatim}
sage: K = crystals.KirillovReshetikhin(['A', 2, 1], 1, 1, model='KR')
sage: K.affinization()
Affinization of Kirillov-Reshetikhin tableaux of type ['A', 2, 1] and shape (1, 1)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *


>>> K = crystals.KirillovReshetikhin(['A', Integer(2), Integer(1)],
     Integer(1), Integer(1))

>>> K.affinization()
Affinization of Kirillov-Reshetikhin tableaux of type ['A', 2, 1] and shape (1, 1)

>>> K = crystals.KirillovReshetikhin(['A', Integer(2), Integer(1)],
     Integer(1), Integer(1), model='KR')

>>> K.affinization()
Affinization of Kirillov-Reshetikhin tableaux of type ['A', 2, 1] and shape (1, 1)
\end{verbatim}

$b_{\text{sharp}}()$

Return the element $b^\sharp$ of self.

Let $B$ be a KR crystal. The element $b^\sharp$ is the unique element such that $\varphi(b^\sharp) = \ell\Lambda_0$ with $\ell = \min\{\langle c, \varphi(b) \rangle \mid b \in B\}$.

EXAMPLES:

\begin{verbatim}
sage: K = crystals.KirillovReshetikhin(['A', 6, 2], 2,1)
sage: K.b_sharp()
[]
sage: K.b_sharp().Phi()
Lambda[0]

sage: K = crystals.KirillovReshetikhin(['C', 3, 1], 1,3)
sage: K.b_sharp()
[-1]
sage: K.b_sharp().Phi()
2*Lambda[0]

sage: K = crystals.KirillovReshetikhin(['D', 6, 2], 2,2)
sage: K.b_sharp() # long time
[]
sage: K.b_sharp().Phi() # long time
2*Lambda[0]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin(['A', Integer(6), Integer(2)],
     Integer(2), Integer(1))

>>> K.b_sharp()
[]

>>> K.b_sharp().Phi()
Lambda[0]

>>> K = crystals.KirillovReshetikhin(['C', Integer(3), Integer(1)],
     Integer(1), Integer(1))


(continues on next page)
cardinality()

Return the cardinality of self.

EXAMPLES:

    sage: K = crystals.KirillovReshetikhin(['E', 6, 1], 1, 1)
    sage: K.cardinality()
    27
    sage: K = crystals.KirillovReshetikhin(['C', 6, 1], 4, 3)
    sage: K.cardinality()
    4736732

    >>> from sage.all import *
    >>> K = crystals.KirillovReshetikhin(['E', Integer(6), Integer(1)], Integer(1), Integer(1))
    >>> K.cardinality()
    27
    >>> K = crystals.KirillovReshetikhin(['C', Integer(6), Integer(1)], Integer(4), Integer(3))
    >>> K.cardinality()
    4736732

classical_decomposition()

Return the classical decomposition of self.

EXAMPLES:

    sage: K = crystals.KirillovReshetikhin(['A', 3, 1], 2, 2)
    sage: K.classical_decomposition()
    The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]

    >>> from sage.all import *
    >>> K = crystals.KirillovReshetikhin(['A', Integer(3), Integer(1)], Integer(2), Integer(2))
    >>> K.classical_decomposition()
    The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]

classically_highest_weight_vectors()

Return the classically highest weight elements of self.

EXAMPLES:
```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.classically_highest_weight_vectors()
([(1,)],)
```

```
>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin(['E',Integer(6),Integer(1)],
      -Integer(1),Integer(1))

>>> K.classically_highest_weight_vectors()
([(1,)],)
```

`is_perfect(ell=None)`

Check if `self` is a perfect crystal of level `ell`.

A crystal $\mathcal{B}$ is perfect of level $\ell$ if:
1. $\mathcal{B}$ is isomorphic to the crystal graph of a finite-dimensional $U'_q(\mathfrak{g})$-module.
2. $\mathcal{B} \otimes \mathcal{B}$ is connected.
3. There exists a $\lambda \in X$, such that wt($\mathcal{B}$) $\subset \lambda + \sum_{i \in I} \mathbb{Z}_{\leq 0} \alpha_i$ and there is a unique element in $\mathcal{B}$ of classical weight $\lambda$.
4. For all $b \in \mathcal{B}$, level($\varepsilon(b)$) $\geq \ell$.
5. For all $\Lambda$ dominant weights of level $\ell$, there exist unique elements $b_\Lambda, b_\Lambda^\Lambda \in \mathcal{B}$, such that $\varepsilon(b_\Lambda) = \Lambda = \varphi(b_\Lambda^\Lambda)$.

Points (1)-(3) are known to hold. This method checks points (4) and (5).

If `self` is the Kirillov-Reshetikhin crystal $B_{r,s}$, then it was proven for non-exceptional types in [FOS2010] that it is perfect if and only if $s/c_r$ is an integer (where $c_r$ is a constant related to the type of the crystal).

It is conjectured this is true for all affine types.

INPUT:

- `ell` - (default: $s/c_r$) integer; the level

REFERENCES:

[FOS2010]

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.is_perfect()
True

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: K.is_perfect()
False

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: K.is_perfect()
True

sage: K = crystals.KirillovReshetikhin(['E',6,1], 1, 3)
sage: K.is_perfect()
True
```

```
>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
      -Integer(1), Integer(1))

>>> K.is_perfect()
```

(continues on next page)
Todo: Implement a version for tensor products of KR crystals.

level()

Return the level of self when self is a perfect crystal.

See also:

is_perfect()

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A', 2, [1]], 1, 1)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['C', 2, [1]], 1, 2)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['D', 4, [1]], 1, 3)
sage: K.level()
3
sage: K = crystals.KirillovReshetikhin(['C', 2, [1]], 1, 1)
sage: K.level()
Traceback (most recent call last):
...
ValueError: this crystal is not perfect
```
local_energy_function(B)

Return the local energy function of self and B.

See LocalEnergyFunction for a definition.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,1)
sage: Kp = crystals.KirillovReshetikhin(['A',6,2], 1,1)
sage: H = K.local_energy_function(Kp); H
Local energy function of
Kirillov-Reshetikhin crystal of type ['BC', 3, 2] with (r,s)=(2,1)
tensor
Kirillov-Reshetikhin crystal of type ['BC', 3, 2] with (r,s)=(1,1)

maximal_vector()

Return the unique element of classical weight \( s \Lambda_r \) in self.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['C',2,1],1,2)
sage: K.maximal_vector()
[[1, 1]]
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.maximal_vector()
[(1,)]
sage: K = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: K.maximal_vector()
[[1], [2]]
```python
>>> K = crystals.KirillovReshetikhin(['E', Integer(6), Integer(1)],
    ~Integer(1), Integer(1))
>>> K.maximal_vector()
[(1,)]

>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],
    ~Integer(2), Integer(1))
>>> K.maximal_vector()
[[1], [2]]
```

### module_generator()

Return the unique module generator of classical weight \( s \Lambda_r \) of the Kirillov-Reshetikhin crystal \( B^{r,s} \).

**EXAMPLES:**

```python
sage: La = RootSystem(['G',2,1]).weight_space().fundamental_weights()
sage: K = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: K.module_generator()
(-Lambda[0] + Lambda[1],)
```

### q_dimension(q=None, prec=None, use_product=False)

Return the \( q \)-dimension of \( t \).

The \( q \)-dimension of a KR crystal is defined as the \( q \)-dimension of the underlying classical crystal.

**EXAMPLES:**

```python
sage: KRC = crystals.KirillovReshetikhin(['A',2,1], 2,2)
sage: KRC.q_dimension()
qu^4 + q^3 + 2*q^2 + q + 1
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: KRC.q_dimension()
qu^{10} + q^9 + 3*q^8 + 3*q^7 + 4*q^6 + 4*q^5 + 4*q^4 + 3*q^3 + 3*q^2 + q + 2
```

### \( \mathfrak{r} () \)

Return the value \( r \) in \( self \) written as \( B^{r,s} \).

**EXAMPLES:**
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,4)
sage: K.r()
2

>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A',Integer(3),Integer(1)],
   Integer(2),Integer(4))
>>> K.r()
2

Return the value s in self written as \( B^{r,s} \).

EXAMPLES:

class TensorProducts (category, *args)
Bases: TensorProductsCategory
The category of tensor products of Kirillov-Reshetikhin crystals.

class ElementMethods
Bases: object

affine_grading()
Return the affine grading of self.

The affine grading is calculated by finding a path from self to a ground state path (using the helper method \texttt{e_string_to_ground_state()}) and counting the number of affine Kashiwara operators \( e_0 \) applied on the way.

OUTPUT: an integer

EXAMPLES:
sage: K = crystals.KirillovReshetikhin(['C',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
    print("{} {}\n".format(b, b.affine_grading()))
[[[1]], [[2]], [[1]]] 1
[[[3]], [[2]], [[1]]] 0

sage: K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
                                         <-Integer(1),Integer(1))

>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
                                         <-Integer(1),Integer(1))

>>> T = crystals.TensorProduct(K,K)

>>> t = T.module_generators[Integer(0)]

>>> t.affine_grading()
1

>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
                                         <-Integer(1),Integer(1))

>>> T = crystals.TensorProduct(K,K)

>>> hw = T.classically_highest_weight_vectors()

>>> for b in hw:
    print("{} {}\n".format(b, b.affine_grading()))
[[[1]], [[1]], [[1]]] 3
[[[2]], [[1]], [[1]]] 2
[[[1]], [[2]], [[1]]] 1
[[[3]], [[2]], [[1]]] 0

>>> K = crystals.KirillovReshetikhin(['C',Integer(2),Integer(1)],
                                         <-Integer(1),Integer(1))

>>> T = crystals.TensorProduct(K,K,K)

>>> hw = T.classically_highest_weight_vectors()

>>> for b in hw:
    print("{} {}\n".format(b, b.affine_grading()))
[[[1]], [[1]], [[1]]] 2
[[[2]], [[1]], [[1]]] 1
[[[-1]], [[1]], [[1]]] 1
[[[-2]], [[2]], [[1]]] 0
[[[1]], [[-1]], [[1]]] 0

e_string_to_ground_state()
Return a string of integers in the index set \((i_1, \ldots, i_k)\) such that \(e_{i_k} \cdots e_{i_1}\) of self is the ground state.

This method calculates a path from self to a ground state path using Demazure arrows as defined in Lemma 7.3 in [ST2011].

OUTPUT: a tuple of integers \((i_1, \ldots, i_k)\)

EXAMPLES:
energy_function\( (algorithm=None)\)

Return the energy function of self.

**ALGORITHM:**
Let $T$ be a tensor product of Kirillov-Reshetikhin crystals. Let $R_i$ and $H_i$ be the combinatorial $R$-matrix and local energy functions, respectively, acting on the $i$ and $i+1$ factors. Let $D_B$ be the energy function of a single Kirillov-Reshetikhin crystal. The energy function is given by

$$D = \sum_{j>i} H_i R_{i+1} R_{i+2} \cdots R_{j-1} + \sum_j D_B R_1 R_2 \cdots R_{j-1},$$

where $D_B$ acts on the rightmost factor.

**grading**

If `self` is an element of $T$, a tensor product of perfect crystals of the same level, then use the affine grading to determine the energy. Specifically, let $g$ denote the affine grading of `self` and $d$ the affine grading of the maximal vector in $T$. Then the energy of `self` is given by $d - g$.

For more details, see Theorem 7.5 in [ST2011].

**INPUT:**

- `algorithm` – (default: None) use one of the following algorithms to determine the energy function:
  - 'definition' – use the definition of the energy function;
  - 'grading' – use the affine grading;
  - if not specified, then this uses 'grading' if all factors are perfect of the same level and otherwise this uses 'definition'

**OUTPUT:** an integer

**EXAMPLES:**

```sage
K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
    ....:    print("{} {}").format(b, b.energy_function())

[[[1]], [[1]], [[1]]] 0
[[[2]], [[1]], [[1]]] 1
[[[1]], [[2]], [[1]]] 2
[[[3]], [[2]], [[1]]] 3

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: T = crystals.TensorProduct(K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
    ....:    print("{} {}").format(b, b.energy_function())

[[[1, 1]], [[1, 1]]] 4
[[[1, 1]], [[1]]] 3
[[[1, 1]], [[2, 1]]] 1
[[[1, 1]], [[1, 1]]] 0
[[[1, 2]], [[1, 1]]] 1
[[[2, 2]], [[1, 1]]] 2
[[[-1, -1]], [[1, 1]]] 2
[[[1, -1]], [[1, 1]]] 2
[[[2, -1]], [[1, 1]]] 2

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: T = crystals.TensorProduct(K)
```

(continues on next page)
sage: t = T.module_generators[0]
sage: t.energy_function('grading')
Traceback (most recent call last):
... 
NotImplementedError: all crystals in the tensor product need to be perfect of the same level

>>> from sage.all import *
>>> K = crystals.KirillovReshetikhin(['A', Integer(2), Integer(1)],...
       Integer(1), Integer(1))
>>> T = crystals.TensorProduct(K,K)
>>> hw = T.classically_highest_weight_vectors()
>>> for b in hw:
...     print("{} {}".format(b, b.energy_function()))
[[[1]], [[1]], [[1]]] 0
[[[2]], [[1]], [[1]]] 1
[[[1]], [[2]], [[1]]] 2
[[[3]], [[2]], [[1]]] 3

>>> K = crystals.KirillovReshetikhin(['C', Integer(2), Integer(1)],...
       Integer(1), Integer(2))
>>> T = crystals.TensorProduct(K)
>>> hw = T.classically_highest_weight_vectors()
>>> for b in hw:
...     print("{} {}".format(b, b.energy_function()))
[[[1]], [[1]], [[1]]] 4
[[[1], [1]], [[1]]] 3
[[[1], [[1]], [[1]]]] 1
[[[1], [[1]], [[1]]]] 0
[[[1], [1], [[1]]] 1
[[[2], [1]], [[1], [1]]] 2
[[[-1], [-1]], [[1], [1]]] 2
[[[1], [-1]], [[1], [1]]] 2
[[[2], [-1]], [[1], [1]]] 2

>>> K = crystals.KirillovReshetikhin(['C', Integer(2), Integer(1)],...
       Integer(1), Integer(1))
>>> T = crystals.TensorProduct(K)
>>> t = T.module_generators[Integer(0)]
>>> t.energy_function('grading')
Traceback (most recent call last):
... 
NotImplementedError: all crystals in the tensor product need to be perfect of the same level

class ParentMethods

    Bases: object

    cardinality()

        Return the cardinality of self.

    EXAMPLES:

        sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.cardinality()
100

(continues on next page)
sage: len(RC.list())
100

sage: RC = RiggedConfigurations(['E', 7, 1], [[1,1]])
sage: RC.cardinality()
134
sage: len(RC.list())
134

sage: RC = RiggedConfigurations(['B', 3, 1], [[2,2],[1,2]])
sage: RC.cardinality()
5130

sage: from sage.all import *

sage: RC = RiggedConfigurations(['A', Integer(3), Integer(1)],
         [[Integer(3), Integer(2)], [Integer(1), Integer(2)]]
sage: RC.cardinality()
100
sage: len(RC.list())
100

sage: RC = RiggedConfigurations(['E', Integer(7), Integer(1)],
         [[Integer(1),Integer(1)]]
sage: RC.cardinality()
134
sage: len(RC.list())
134

sage: RC = RiggedConfigurations(['B', Integer(3), Integer(1)],
         [[Integer(2),Integer(2)],[Integer(1),Integer(2)]]
sage: RC.cardinality()
5130

classically_highest_weight_vectors()

Return the classically highest weight elements of self.
This works by using a backtracking algorithm since if $b_2 \otimes b_1$ is classically highest weight then $b_1$ is classically highest weight.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: T.classically_highest_weight_vectors()
([[[1]]], [[1]], [[1]]),
([[[2]]], [[1]], [[1]]),
([[[1]]], [[2]], [[1]]),
([[[3]]], [[2]], [[1]])

>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],
        Integer(1),Integer(1))

>>> T = crystals.TensorProduct(K,K,K)

>>> T.classically_highest_weight_vectors()
([[[1]], [[1]], [[1]]],
 [[[[2]], [[1]], [[1]]],
 [}[Continues on next page]
maximal_vector()

Return the maximal vector of self.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.maximal_vector()
[[[1]], [[2]], [[1]]]
```

one_dimensional_configuration_sum(q=None, group_components=True)

Compute the one-dimensional configuration sum of self.

INPUT:
- q = (default: None) a variable or None; if None, a variable \( q \) is set in the code
- group_components = (default: True) boolean; if True, then the terms are grouped by classical component

The one-dimensional configuration sum is the sum of the weights of all elements in the crystal weighted by the energy function.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
sage: R.<t> = ZZ[]
sage: T.one_dimensional_configuration_sum(t, False)
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.one_dimensional_configuration_sum()  # long time
False
...:
```

```
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
+ B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]

>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> T.one_dimensional_configuration_sum(t, False)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
+ B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]

>>> R = RootSystem(['A',Integer(2),Integer(1)])
>>> La = R.weight_space().basis()
>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])
>>> (LS.one_dimensional_configuration_sum()  # long time
... == T.one_dimensional_configuration_sum())
True

extra_super_categories()

EXAMPLES:

sage: from sage.categories.loop_crystals import...
        KirillovReshetikhinCrystals
sage: KirillovReshetikhinCrystals().TensorProducts().extra_super_...
        categories()
[Category of finite regular loop crystals]

super_categories()

EXAMPLES:

sage: from sage.categories.loop_crystals import KirillovReshetikhinCrystals
sage: KirillovReshetikhinCrystals().super_categories()
[Category of finite regular loop crystals]

class sage.categories.loop_crystals.LocalEnergyFunction (B, Bp, normalization=0)

Bases: Map

The local energy function.

Let $B$ and $B'$ be Kirillov-Reshetikhin crystals with maximal vectors $u_B$ and $u_{B'}$ respectively. The local energy function $H : B \otimes B' \to \mathbb{Z}$ is the function which satisfies

\[
H(c_0(b \otimes b')) = H(b \otimes b') + \begin{cases}
1 & \text{if } i = 0 \text{ and LL}, \\
-1 & \text{if } i = 0 \text{ and RR}, \\
0 & \text{otherwise},
\end{cases}
\]
where LL (resp. RR) denote $e_0$ acts on the left (resp. right) on both $b \otimes b'$ and $R(b \otimes b')$, and normalized by $H(u_B \otimes u_{B'}) = 0$.

**INPUT:**

- $b$ – a Kirillov-Reshetikhin crystal
- $B_p$ – a Kirillov-Reshetikhin crystal
- `normalization` – (default: 0) the normalization value

**EXAMPLES:**

```python
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K2 = crystals.KirillovReshetikhin(['C',2,1], 2,1)
sage: H = K.local_energy_function(K2)
sage: T = tensor([K, K2])
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
    print(b, H(b))
([[], [[1], [2]]], 1)
([[[1], [1]], [[1], [2]]], 0)
([[[2], [-2]], [[1], [2]]], 1)
([[[1], [-2]], [[1], [2]]], 1)
```

**REFERENCES:**

[KKMMNN1992]
sage: C.example()
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)

```python
>>> from sage.all import *
>>> from sage.categories.loop_crystals import LoopCrystals

>>> C = LoopCrystals()

>>> C
Category of loop crystals

>>> C.super_categories()
[Category of crystals]

>>> C.example()
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)
```

```python
class ParentMethods
    Bases: object
digraph (subset=None, index_set=None)
        Return the DiGraph associated to self.

        INPUT:
        • subset – (optional) a subset of vertices for which the digraph should be constructed
        • index_set – (optional) the index set to draw arrows

        See also:
        sage.categories.crystals.Crystals.ParentMethods.digraph()

    EXAMPLES:

    sage: C = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
    sage: G = C.digraph()
    sage: G.latex_options()  # optional - dot2tex
    LaTeX options for Digraph on 29 vertices:
    {...'edge_options': <function ... at ...>...}
    sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested...
    (opens external window)

    sage: C = crystals.KirillovReshetikhin(['D',Integer(4),Integer(1)],...
                   Integer(2), Integer(1))
    sage: G = C.digraph()
    sage: G.latex_options()  # optional - dot2tex
    LaTeX options for Digraph on 29 vertices:
    {...'edge_options': <function ... at ...>...}
    sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested...
    (opens external window)
```

```python
weight_lattice_realization()

Return the weight lattice realization used to express weights of elements in self.

The default is to use the non-extended affine weight lattice.

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
```
```
### sage: K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]

```python
>>> from sage.all import *

>>> C = crystals.Letters(['A', Integer(5)])
>>> C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]

>>> K = crystals.KirillovReshetikhin(['A',Integer(2),Integer(1)],...
    -Integer(1), Integer(1))
>>> K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

### example (n=3)

Return an example of Kirillov-Reshetikhin crystals, as per `Category.example()`.

**EXAMPLES:**

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: B = LoopCrystals().example(); B
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)

>>> from sage.all import *

>>> from sage.categories.loop_crystals import LoopCrystals

>>> B = LoopCrystals().example(); B
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)
```

### super_categories()

**EXAMPLES:**

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: LoopCrystals().super_categories()
[Category of crystals]

>>> from sage.all import *

>>> from sage.categories.loop_crystals import LoopCrystals

>>> LoopCrystals().super_categories()
[Category of crystals]
```

### class sage.categories.loop_crystals.RegularLoopCrystals

**Bases:** `Category_singleton`

The category of regular \( U'_\mathfrak{g} \)-crystals, where \( \mathfrak{g} \) is of affine type.

**class ElementMethods**

**Bases:** `object`

**classical_weight ()**

Return the classical weight of `self`.

**EXAMPLES:**

```python
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: hw = LS.classically_highest_weight_vectors()
```
sage: [(v.weight(), v.classical_weight()) for v in hw]
[(-2*Lambda[0] + 2*Lambda[1], (2, 0, 0)),
 (-Lambda[0] + Lambda[2], (1, 1, 0))]

>>> from sage.all import *

>>> R = RootSystem(['A', Integer(2), Integer(1)])

>>> La = R.weight_space().basis()

>>> LS = crystals.ProjectedLevelZeroLSPaths(Integer(2)*La[Integer(1)])

>>> hw = LS.classically_highest_weight_vectors()

>>> [(v.weight(), v.classical_weight()) for v in hw]
[(-2*Lambda[0] + 2*Lambda[1], (2, 0, 0)),
 (-Lambda[0] + Lambda[2], (1, 1, 0))]

4.110 L-trivial semigroups

class sage.categories.l_trivial_semigroups.LTrivialSemigroups(base_category)

Bases: CategoryWithAxiom

Commutative_extra_super_categories()

Implement the fact that a commutative $R$-trivial semigroup is $J$-trivial.

EXAMPLES:

sage: Semigroups().LTrivial().Commutative_extra_super_categories()
[Category of j trivial semigroups]

RTrivial_extra_super_categories()

Implement the fact that an $L$-trivial and $R$-trivial semigroup is $J$-trivial.

EXAMPLES:

sage: Semigroups().LTrivial().RTrivial_extra_super_categories()
[Category of j trivial magmas]
extra_super_categories()

Implement the fact that a $L$-trivial semigroup is $H$-trivial.

EXAMPLES:

```python
sage: Semigroups().LTrivial().RTrivial_extra_super_categories()
[Category of j trivial magmas]
```

4.111 Magmas

```python
>>> from sage.all import *
>>> Semigroups().LTrivial().RTrivial_extra_super_categories()
[Category of j trivial magmas]
```

```python
>>> from sage.all import *
>>> Semigroups().LTrivial().RTrivial_extra_super_categories()
[Category of h trivial semigroups]
```

```python
>>> from sage.all import *
>>> Semigroups().LTrivial().extra_super_categories()
[Category of h trivial semigroups]
```

```python
>>> from sage.all import *
>>> Semigroups().LTrivial().extra_super_categories()
[Category of h trivial semigroups]
```

The following axioms are defined by this category:

```
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Unital()
Category of unital magmas
sage: Magmas().Commutative()
Category of commutative magmas
sage: Magmas().Unital().Inverse()
Category of inverse unital magmas
```

(continues on next page)
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Associative().Unital()
Category of monoids
sage: Magmas().Associative().Unital().Inverse()
Category of groups

from sage.all import *

Magmas().Associative()
Category of semigroups
Magmas().Unital()
Category of monoids
Magmas().Commutative()
Category of commutative magmas
Magmas().Unital().Inverse()
Category of inverse unital magmas
Magmas().Associative()
Category of semigroups
Magmas().Associative().Unital()
Category of monoids
Magmas().Associative().Unital().Inverse()
Category of groups

class Algebras(category, *args)
    Bases: AlgebrasCategory
class ParentMethods
    Bases: object

    is_field(proof=True)
        Return True if self is a field.
        For a magma algebra $RS$ this is always false unless $S$ is trivial and the base ring $R$ is a field.

EXAMPLES:

sage: SymmetricGroup(1).algebra(QQ).is_field() # needs sage.combinat sage.groups
True
sage: SymmetricGroup(2).algebra(ZZ).is_field() # needs sage.combinat sage.groups
False
sage: SymmetricGroup(2).algebra(QQ).is_field() # needs sage.combinat sage.groups
False

from sage.all import *

SymmetricGroup(Integer(1)).algebra(QQ).is_field() # needs sage.combinat sage.groups
True
SymmetricGroup(Integer(1)).algebra(ZZ).is_field() # needs sage.combinat sage.groups
False
SymmetricGroup(Integer(2)).algebra(QQ).is_field() # needs sage.combinat sage.groups
False
extra_super_categories()

EXAMPLES:

```
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.extra_super_categories()
[Category of commutative magmas]
```

```
>>> from sage.all import *

>>> MCA = Magmas().Commutative().Algebras(QQ)

>>> MCA.extra_super_categories()
[Category of commutative magmas]
```

This implements the fact that the algebra of a commutative magma is commutative:

```
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

```
>>> from sage.all import *

>>> MCA = Magmas().Commutative().Algebras(QQ)

>>> MCA.super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

In particular, commutative monoid algebras are commutative algebras:

```
sage: MoCA = Monoids().Commutative().Algebras(QQ)
sage: MoCA.is_subcategory(Algebras(QQ).Commutative())
True
```

```
>>> from sage.all import *

>>> MoCA = Monoids().Commutative().Algebras(QQ)

>>> MoCA.is_subcategory(Algebras(QQ).Commutative())
True
```

Associative

alias of Semigroups

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

class ParentMethods
    Bases: object

    product(left, right)

    EXAMPLES:

```
sage: C = Magmas().CartesianProducts().example(); C
The Cartesian product of (Rational Field, Integer Ring, Integer Ring)
sage: x = C.an_element(); x
(1/2, 1, 1)
sage: x * x
(1/4, 1, 1)
```

```
sage: # needs sage.groups sage.modules
sage: A = SymmetricGroupAlgebra(QQ, 3)
sage: x = cartesian_product([A([1,3,2]), A([2,3,1])])
```

(continues on next page)
sage: y = cartesian_product([A([1,3,2]), A([2,3,1])])
sage: cartesian_product([A,A]).product(x,y)
B(0, [1, 2, 3]) + B(1, [3, 1, 2])
sage: x*y
B(0, [1, 2, 3]) + B(1, [3, 1, 2])

```python
>>> from sage.all import *
>>>
```

extra_super_categories()
This implements the fact that a subquotient (and therefore a quotient or subobject) of a finite set is finite.

EXAMPLES:

```python
sage: Semigroups().CartesianProducts().extra_super_categories()
[Category of semigroups]
sage: Semigroups().CartesianProducts().super_categories()
[Category of semigroups, Category of Cartesian products of magmas]
```

```python
>>> from sage.all import *
>>> Semigroups().CartesianProducts().extra_super_categories()
[Category of semigroups]
```

```python
>>> from sage.all import *
>>> Semigroups().CartesianProducts().super_categories()
[Category of semigroups, Category of Cartesian products of magmas]
```

class Commutative(base_category)
Bases: CategoryWithAxiom_singleton

class Algebras(category, *args)
Bases: AlgebrasCategory

extra_super_categories()

EXAMPLES:

```python
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.extra_super_categories()
[Category of commutative magmas]
```

```python
>>> from sage.all import *
>>> MCA = Magmas().Commutative().Algebras(QQ)
```

```python
>>> MCA.extra_super_categories()
[Category of commutative magmas]
```

This implements the fact that the algebra of a commutative magma is commutative:

```python
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

```python
>>> from sage.all import *
```

```python
>>> MCA = Magmas().Commutative().Algebras(QQ)
```

```python
>>> MCA.is_subcategory(Algebras(QQ).Commutative())
True
```

This implements the fact that the algebra of a commutative magma is commutative:

```python
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

```python
>>> from sage.all import *
```

```python
>>> MCA = Magmas().Commutative().Algebras(QQ)
```

```python
>>> MCA.is_subcategory(Algebras(QQ).Commutative())
True
```

class CartesianProducts(category, *args)
Bases: CartesianProductsCategory
extra_super_categories()

Implement the fact that a Cartesian product of commutative additive magmas is still a commutative additive magmas.

EXAMPLES:

```sage
sage: C = Magmas().Commutative().CartesianProducts()
sage: C.extra_super_categories()
[Category of commutative magmas]
sage: C.axioms()
frozenset({'Commutative'})
```

```python
>>> from sage.all import *
>>> C = Magmas().Commutative().CartesianProducts()
>>> C.extra_super_categories()
[Category of commutative magmas]
```

```python
>>> from sage.all import *
>>> C = Magmas().Commutative().CartesianProducts()
```

class ParentMethods

Bases: object

is_commutative()

Return True, since commutative magmas are commutative.

EXAMPLES:

```sage
sage: Parent(QQ, category=CommutativeRings()).is_commutative()
True
```

```python
>>> from sage.all import *
>>> Parent(QQ, category=CommutativeRings()).is_commutative()
True
```

class ElementMethods

Bases: object

is_idempotent()

Test whether self is idempotent.

EXAMPLES:

```sage
sage: S = Semigroups().example("free"); S
An example of a semigroup:
the free semigroup generated by ('a', 'b', 'c', 'd')
sage: a = S('a')
sage: a^2
'aa'
sage: a.is_idempotent()
False
```

```python
>>> from sage.all import *
>>> S = Semigroups().example("free"); S
An example of a semigroup:
The free semigroup generated by ('a', 'b', 'c', 'd')
>>> a = S('a')
>>> a**Integer(2)
```

(continues on next page)
>>> a.is_idempotent()
False

sage: L = Semigroups().example("leftzero"); L
An example of a semigroup: the left zero semigroup
sage: x = L('x')
sage: x^2
'x'
sage: x.is_idempotent()
True

>>> from sage.all import *
>>> L = Semigroups().example("leftzero"); L
An example of a semigroup: the left zero semigroup
>>> x = L('x')
>>> x**Integer(2)
'x'
>>> x.is_idempotent()
True

FinitelyGeneratedAsMagma

alias of FinitelyGeneratedMagmas

class JTrivial(base_category)

Bases: CategoryWithAxiom

class ParentMethods

Bases: object

multiplication_table(names='letters', elements=None)

Returns a table describing the multiplication operation.

Note: The order of the elements in the row and column headings is equal to the order given by the table's list() method. The association can also be retrieved with the dict() method.

INPUT:

• names – the type of names used
  – 'letters' – lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by column_keys(), padded to a common width with leading 'a's.
  – 'digits' – base 10 representation of the elements' positions in the list given by column_keys(), padded to a common width with leading zeros.
  – 'elements' – the string representations of the elements themselves.
  – a list - a list of strings, where the length of the list equals the number of elements.

• elements – default = None. A list of elements of the magma, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering the S.list method returns. Or the elements can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.

OUTPUT:

The multiplication table as an object of the class OperationTable which defines several methods for manipulating and displaying the table. See the documentation there for full details to supplement the documentation here.
EXAMPLES:
The default is to represent elements as lowercase ASCII letters.

```
sage: G = CyclicPermutationGroup(5)  # needs sage.groups
sage: G.multiplication_table()  # needs sage.groups
* a b c d e
+--------
a| a b c d e
b| b c d e a
c| c d e a b
d| d e a b c
e| e a b c d
```

All that is required is that an algebraic structure has a multiplication defined. A `LeftRegularBand` is an example of a finite semigroup. The `names` argument allows displaying the elements in different ways.

```
sage: from sage.categories.examples.finite_semigroups import LeftRegularBand
sage: L = LeftRegularBand(('a', 'b'))
sage: T = L.multiplication_table(names='digits')  # needs sage.modules
sage: T.column_keys()  # needs sage.modules
('a', 'ab', 'b', 'ba')
sage: T  # needs sage.modules
* 0 1 2 3
+--------
0| 0 1 1 1
1| 1 1 1 1
2| 3 3 2 3
3| 3 3 3 3
```

(continues on next page)
Specifying the elements in an alternative order can provide more insight into how the operation behaves.

```
sage: L = LeftRegularBand(('a', 'b', 'c'))
sage: elts = sorted(L.list())
sage: L.multiplication_table(elements=elts)  # needs sage.modules
```

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>g</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>k</td>
<td>l</td>
<td>m</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>k</td>
<td>i</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>l</td>
<td>l</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>
```

```python
>>> from sage.all import *
>>> L = LeftRegularBand(('a', 'b', 'c'))
>>> elts = sorted(L.list())
>>> L.multiplication_table(elements=elts)  # needs sage.modules
```

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>m</th>
<th>n</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>b</td>
<td>b</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>d</td>
<td>d</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>g</td>
<td>g</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>g</td>
<td>g</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
<td>j</td>
</tr>
<tr>
<td>k</td>
<td>l</td>
<td>m</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>k</td>
<td>i</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>l</td>
<td>l</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>n</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>
```
The `elements` argument can be used to provide a subset of the elements of the structure. The subset must be closed under the operation. Elements need only be in a form that can be coerced into the set. The `names` argument can also be used to request that the elements be represented with their usual string representation.

```python
sage: L = LeftRegularBand(('a', 'b', 'c'))
sage: elts = ['a', 'c', 'ac', 'ca']
sage: L.multiplication_table(names='elements', elements=elts)  # needs sage.modules
    *  |  a  |  c  |  ac  |  ca  
    +--------+-------+-------+-------+-------+
    a  |  a   |  ac  |  ac  |  ac  
    c  |  ca  |  c   |  ac  |  ac  
    ac |  ac  |  ac  |  ac  |  ac  
    ca |  ca  |  ca  |  ca  |  ca  
```

The table returned can be manipulated in various ways. See the documentation for `OperationTable` for more comprehensive documentation.

```python
sage: # needs sage.groups sage.modules
sage: G = AlternatingGroup(3)
sage: T = G.multiplication_table()
sage: T.column_keys()
((), (1,2,3), (1,3,2))
sage: T.translation()
{'a': (), 'b': (1,2,3), 'c': (1,3,2)}
sage: T.change_names(['x', 'y', 'z'])
sage: T.translation()
{'x': (), 'y': (1,2,3), 'z': (1,3,2)}
sage: T
    *  x  y  z
    +-------
    x  |  x  y  z
    y  |  y  z  x
    z  |  z  x  y
```

(continues on next page)
product \((x, y)\)

The binary multiplication of the magma.

INPUT:
- \(x, y\) – elements of this magma

OUTPUT:
- an element of the magma (the product of \(x\) and \(y\))

EXAMPLES:

```
sage: S = Semigroups().example("free")
sage: x = S('a'); y = S('b')
sage: S.product(x, y)
'a*b'
```

A parent in \texttt{Magmas()} must either implement \texttt{product()} in the parent class or \texttt{*_mul_*} in the element class. By default, the addition method on elements \texttt{x._mul_(y)} calls \texttt{S.product(x, y)}, and reciprocally.

As a bonus, \texttt{S.product} models the binary function from \texttt{S} to \texttt{S}:

```
sage: bin = S.product
sage: bin(x,y)
'a*b'
```

Currently, \texttt{S.product} is just a bound method:

```
sage: bin
<bound method FreeSemigroup.product of An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')>
```

```
>>> from sage.all import *

>>> S = Semigroups().example("free")

>>> x = S('a'); y = S('b')

>>> S.product(x, y)
'a*b'
```

A parent in \texttt{Magmas()} must either implement \texttt{product()} in the parent class or \texttt{*_mul_*} in the element class. By default, the addition method on elements \texttt{x._mul_(y)} calls \texttt{S.product(x, y)}, and reciprocally.

As a bonus, \texttt{S.product} models the binary function from \texttt{S} to \texttt{S}:

```
sage: bin = S.product
sage: bin(x,y)
'a*b'
```

Currently, \texttt{S.product} is just a bound method:

```
sage: bin
<bound method FreeSemigroup.product of An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')>
```

```
>>> from sage.all import *

>>> bin = S.product

>>> bin(x,y)
'a*b'
```

Currently, \texttt{S.product} is just a bound method:
When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich \texttt{S.product} with extra mathematical structure. This will typically be implemented using lazy attributes:

```
sage: bin # todo: not implemented
Generic binary morphism:
From: (S x S)
To: S
```

\texttt{product\_from\_element\_class\_mul}(x, y)

The binary multiplication of the magma.

**INPUT:**
- \(x, y\) – elements of this magma

**OUTPUT:**
- an element of the magma (the product of \(x\) and \(y\))

**EXAMPLES:**

```
sage: S = Semigroups().example("free")
sage: x = S('a'); y = S('b')
sage: S.product(x, y)
'ab'
```

A parent in \texttt{Magmas()} must either implement \texttt{product()} in the parent class or \texttt{\_mul\_} in the element class. By default, the addition method on elements \(x.\_mul\_(y)\) calls \texttt{S.product(x, y)}, and reciprocally.

As a bonus, \texttt{S.product} models the binary function from \(S\) to \(S\):

```
sage: bin = S.product
sage: bin(x,y)
'ab'
```

Currently, \texttt{S.product} is just a bound method:

```
sage: bin
<bound method FreeSemigroup.product of An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')>
```
When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich $S.product$ with extra mathematical structure. This will typically be implemented using lazy attributes:

```
from sage.all import *

bin
# todo: not implemented
```

Generic binary morphism:
From: (S x S)
To: S

```python
from sage.all import *

bin
# todo: not implemented
```

```
class Realizations(category, *args)
    Bases: RealizationsCategory

class ParentMethods
    Bases: object

    product_by_coercion(left, right)
    Default implementation of product for realizations.
    This method coerces to the realization specified by self.realization_of().
    a_realization(), computes the product in that realization, and then coerces back.

    EXAMPLES:
```

```
sage: # needs sage.combinat sage.modules
sage: Out = Sets().WithRealizations().example().Out(); Out
The subset algebra of {1, 2, 3} over Rational Field
in the Out basis
sage: Out.product
<bound method Magmas.Realizations.ParentMethods.product_by_coercion
of The subset algebra of {1, 2, 3} over Rational Field
in the Out basis>
sage: Out.product.__module__
'sage.categories.magmas'
sage: x = Out.an_element()
sage: y = Out.an_element()
sage: Out.product(x, y)
Out[{}] + 4*Out[{1}] + 9*Out[{2}] + Out[{1, 2}]
```

```python
from sage.all import *

# needs sage.combinat sage.modules

Out = Sets().WithRealizations().example().Out(); Out
The subset algebra of {1, 2, 3} over Rational Field
in the Out basis
```

(continues on next page)
class SubcategoryMethods
    Bases: object

    Associative()
    Return the full subcategory of the associative objects of self.
    A (multiplicative) magma \( M \) is associative if, for all \( x, y, z \in M \),
    \[ x \cdot (y \cdot z) = (x \cdot y) \cdot z \]

    See also:
    Wikipedia article Associative_property

    EXAMPLES:
    sage: Magmas().Associative()
    Category of semigroups
    >>> from sage.all import *
    >>> Magmas().Associative()
    Category of semigroups

    Commutative()
    Return the full subcategory of the commutative objects of self.
    A (multiplicative) magma \( M \) is commutative if, for all \( x, y \in M \),
    \[ x \cdot y = y \cdot x \]

    See also:
    Wikipedia article Commutative_property

    EXAMPLES:
    sage: Magmas().Commutative()
    Category of commutative magmas
    sage: Monoids().Commutative()
    Category of commutative monoids
    >>> from sage.all import *
    >>> Magmas().Commutative()
    Category of commutative magmas
    >>> Monoids().Commutative()
    Category of commutative monoids
Distributive()

Return the full subcategory of the objects of self where \(*\) is distributive on +.

**INPUT:**

- self – a subcategory of Magmas and AdditiveMagmas

Given that Sage does not yet know that the category MagmasAndAdditiveMagmas is the intersection of the categories Magmas and AdditiveMagmas, the method MagmasAndAdditiveMagmas.SubcategoryMethods.Distributive() is not available, as would be desirable, for this intersection.

This method is a workaround. It checks that self is a subcategory of both Magmas and AdditiveMagmas and upgrades it to a subcategory of MagmasAndAdditiveMagmas before applying the axiom. It complains otherwise, since the Distributive axiom does not make sense for a plain magma.

**EXAMPLES:**

```python
sage: (Magmas() & AdditiveMagmas()).Distributive()
Category of distributive magmas and additive magmas
sage: (Monoids() & CommutativeAdditiveGroups()).Distributive()
Category of rings
sage: Magmas().Distributive()
Traceback (most recent call last):
... ValueError: The distributive axiom only makes sense on a magma which is simultaneously an additive magma
sage: Semigroups().Distributive()
Traceback (most recent call last):
... ValueError: The distributive axiom only makes sense on a magma which is simultaneously an additive magma
```

```python
>>> from sage.all import *
>>> (Magmas() & AdditiveMagmas()).Distributive()
Category of distributive magmas and additive magmas
>>> (Monoids() & CommutativeAdditiveGroups()).Distributive()
Category of rings
>>> Magmas().Distributive()
Traceback (most recent call last):
... ValueError: The distributive axiom only makes sense on a magma which is simultaneously an additive magma
>>> Semigroups().Distributive()
Traceback (most recent call last):
... ValueError: The distributive axiom only makes sense on a magma which is simultaneously an additive magma
```

FinitelyGenerated()

Return the subcategory of the objects of self that are endowed with a distinguished finite set of (multiplicative) magma generators.

**EXAMPLES:**

This is a shorthand for FinitelyGeneratedAsMagma(), which see:
An error is raised if this is ambiguous:

```python
sage: (Magmas() & AdditiveMagmas()).FinitelyGenerated()
Traceback (most recent call last):
  ... ValueError: FinitelyGenerated is ambiguous for
Join of Category of magmas and Category of additive magmas.
Please use explicitly one of the FinitelyGeneratedAsXXX methods
```

**Note:** Checking that there is no ambiguity currently assumes that all the other “finitely generated” axioms involve an additive structure. As of Sage 6.4, this is correct.

The use of this shorthand should be reserved for casual interactive use or when there is no risk of ambiguity.

**FinitelyGeneratedAsMagma()**

Return the subcategory of the objects of `self` that are endowed with a distinguished finite set of (multiplicative) magma generators.

A set \( S \) of elements of a multiplicative magma form a *set of generators* if any element of the magma can be expressed recursively from elements of \( S \) and products thereof.

It is not imposed that morphisms shall preserve the distinguished set of generators; hence this is a full subcategory.

**See also:**

Wikipedia article Unital_magma#unital

**EXAMPLES:**

```python
sage: Magmas().FinitelyGeneratedAsMagma()
Category of finitely generated magmas
```
Being finitely generated does depend on the structure: for a ring, being finitely generated as a magma, as an additive magma, or as a ring are different concepts. Hence the name of this axiom is explicit:

```
sage: Rings().FinitelyGeneratedAsMagma()
Category of finitely generated as magma enumerated rings
```

On the other hand, it does not depend on the multiplicative structure: for example a group is finitely generated if and only if it is finitely generated as a magma. A short hand is provided when there is no ambiguity, and the output tries to reflect that:

```
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated enumerated groups
sage: Semigroups().FinitelyGenerated().axioms()
frozenset({'Associative', 'Enumerated', 'FinitelyGeneratedAsMagma'})
```

Note that the set of generators may depend on the actual category; for example, in a group, one can often use less generators since it is allowed to take inverses.

**JTrivial()**

Return the full subcategory of the \( J \)-trivial objects of \( \text{self} \).

This axiom is in fact only meaningful for semigroups. This stub definition is here as a workaround for Issue #20515, in order to define the \( J \)-trivial axiom as the intersection of the \( L \) and \( R \)-trivial axioms.

See also:

```
Semigroups.SubcategoryMethods.JTrivial()
```

**Unital()**

Return the subcategory of the unital objects of \( \text{self} \).

A (multiplicative) magma \( \text{Magma} \ M \) is unital if it admits an element \( 1 \), called unit, such that for all \( x \in M \),

\[
1 \ast x = x \ast 1 = x
\]

This element is necessarily unique, and should be provided as \( M\text{.one()} \).
See also:
Wikipedia article Unital_magma#unital

EXAMPLES:

```
sage: Magmas().Unital()
Category of unital magmas
sage: Semigroups().Unital()
Category of monoids
sage: Monoids().Unital()
Category of monoids
sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: AssociativeAlgebras(QQ).Unital()
Category of algebras over Rational Field
```

```python
>>> from sage.all import *
>>> Magmas().Unital()
Category of unital magmas
>>> Semigroups().Unital()
Category of monoids
>>> Monoids().Unital()
Category of monoids
>>> from sage.categories.associative_algebras import AssociativeAlgebras
>>> AssociativeAlgebras(QQ).Unital()
Category of algebras over Rational Field
```

class Subquotients (category, *args)

Bases: SubquotientsCategory

The category of subquotient magmas.

See Sets.SubcategoryMethods.Subquotients() for the general setup for subquotients. In the case of a subquotient magma $S$ of a magma $G$, the condition that $r$ be a morphism in $\mathbb{A}S$ can be rewritten as follows:

- for any two $a, b \in S$ the identity $a \times_S b = r(l(a) \times_G l(b))$ holds.

This is used by this category to implement the product $\times_S$ of $S$ from $l$ and $r$ and the product of $G$.

EXAMPLES:

```
sage: Semigroups().Subquotients().all_super_categories()
[Category of subquotients of semigroups, Category of semigroups,
 Category of subquotients of magmas, Category of magmas,
 Category of subquotients of sets, Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

```python
>>> from sage.all import *
>>> Semigroups().Subquotients().all_super_categories()
[Category of subquotients of semigroups, Category of semigroups,
 Category of subquotients of magmas, Category of magmas,
 Category of subquotients of sets, Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

class ParentMethods

Bases: object
product \((x, y)\)

Return the product of two elements of \(self\).

EXAMPLES:

```
sage: S = Semigroups().Subquotients().example()
sage: S
An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup
sage: S.product(S(19), S(3))
19
```

Here is a more elaborate example involving a sub algebra:

```
sage: Z = SymmetricGroup(5).algebra(QQ).center()  # needs sage.combinat sage.groups
sage: B = Z.basis()  # needs sage.combinat sage.groups
```

```python
>>> from sage.all import *

>>> Z = SymmetricGroup(Integer(5)).algebra(QQ).center()  # needs sage.combinat sage.groups

>>> B = Z.basis()  # needs sage.combinat sage.groups

>>> B[Integer(3)] * B[Integer(2)]
```

class Unital \((base\_category)\)

Bases: CategoryWithAxiom\_singleton

class Algebras \((category, *\text{args})\)

Bases: AlgebrasCategory

extra\_super\_categories()  

EXAMPLES:

```
sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.extra_super_categories()
[Category of commutative magmas]
```

```python
>>> from sage.all import *

>>> MCA = Magmas().Commutative().Algebras(QQ)

>>> MCA.extra_super_categories()
[Category of commutative magmas]```
This implements the fact that the algebra of a commutative magma is commutative:

```python
sage: MCA.super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

In particular, commutative monoidal algebras are commutative algebras:

```python
sage: MoCA = Monoids().Commutative().Algebras(QQ)
sage: MoCA.is_subcategory(Algebras(QQ).Commutative())
True
```

```python
>>> from sage.all import *

MoCA = Monoids().Commutative().Algebras(QQ)
MoCA.is_subcategory(Algebras(QQ).Commutative())
True
```

class **CartesianProducts** *(category, *args)*  
Bases: **CartesianProductsCategory**

class **ElementMethods**  
Bases: object

class **ParentMethods**  
Bases: object

```python
one()
```

Return the unit of this Cartesian product.

It is built from the units for the Cartesian factors of `self`.

**EXAMPLES:**

```python
sage: cartesian_product([QQ, ZZ, RR]).one()  
# needs sage.rings.real_mpfr
(1, 1, 1.00000000000000)
```

```python
>>> from sage.all import *

cartesian_product([QQ, ZZ, RR]).one()  
# needs sage.rings.real_mpfr
(1, 1, 1.00000000000000)
```

**extra_super_categories()**

Implement the fact that a Cartesian product of unital magmas is a unital magma

**EXAMPLES:**

```python
sage: C = Magmas().Unital().CartesianProducts()
sage: C.extra_super_categories()
[Category of unital magmas]
sage: C.axioms()
frozenset({'Unital'})
```

(continues on next page)
sage: Monoids().CartesianProducts().is_subcategory(Monoids())
True

```python
>>> from sage.all import *
``` 
```
>>> C = Magmas().Unital().CartesianProducts()
``` 
```
>>> C.extra_super_categories()
[Category of unital magmas]
``` 
```
>>> C.axioms()
frozenset({'Unital'})
``` 
```
>>> Monoids().CartesianProducts().is_subcategory(Monoids())
True
``` 

```python
class ElementMethods
    Bases: object
class Inverse(base_category)
    Bases: CategoryWithAxiom_singleton
class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory
    extra_super_categories()
        Implement the fact that a Cartesian product of magmas with inverses is a magma with inverse.
        EXAMPLES:
        ```
sage: C = Magmas().Unital().Inverse().CartesianProducts()
sage: C.extra_super_categories()
[Category of inverse unital magmas]
sage: sorted(C.axioms())
['Inverse', 'Unital']
``` 
```
```python
>>> from sage.all import *
``` 
```
>>> C = Magmas().Unital().Inverse().CartesianProducts()
``` 
```
>>> C.extra_super_categories()
[Category of inverse unital magmas]
``` 
```
>>> sorted(C.axioms())
['Inverse', 'Unital']
``` 

```python
class ParentMethods
    Bases: object
    is_empty()
        Return whether self is empty.
        Since this set is a unital magma it is not empty and this method always return False.
        EXAMPLES:
        ```
sage: S = SymmetricGroup(2)
# needs sage.groups
sage: S.is_empty()
# needs sage.groups
False
``` 
```
one()

Return the unit of the monoid, that is the unique neutral element for *.

**Note:** The default implementation is to coerce 1 into `self`. It is recommended to override this method because the coercion from the integers:

- is not always meaningful (except for 1);
- often uses `self.one()`.

**EXAMPLES:**

```python
sage: M = Monoids().example(); M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.one()
```

```python
sage: M = Monoids().example(); M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
```

```python
>>> from sage.all import *
>>> M = Monoids().example(); M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
```

```python
>>> M = Monoids().example()
>>> M.is_empty()
False
```

```python
>>> S = SymmetricGroup(Integer(2))
˓→ needs sage.groups
>>> S.is_empty()
# needs sage.groups
False
```

```python
>>> M = Monoids().example()
>>> M.is_empty()
False
```

```python
sage: M = Monoids().example()  # needs sage.combinat sage.groups
sage: from sage.combinat.root_system.extended_affine_weyl_group import ExtendedAffineWeylGroup
sage: PvW0 = ExtendedAffineWeylGroup(['A',2,1]).PvW0()
```
class SubcategoryMethods

    Bases: object

    Inverse()

        Return the full subcategory of the inverse objects of self.

        An inverse class: (multiplicative) magma \(<\text{Magmas}>\) is a \textit{unital magma} such that every element admits both an inverse on the left and on the right. Such a magma is also called a \textit{loop}.

        See also:

        Wikipedia article Inverse_element, Wikipedia article Quasigroup

    EXAMPLES:

        sage: Magmas().Unital().Inverse()
        Category of inverse unital magmas
        sage: Monoids().Inverse()
        Category of groups

additional_structure()

    Return self.

    Indeed, the category of unital magmas defines an additional structure, namely the unit of the magma which shall be preserved by morphisms.

    See also:

    Category.additional_structure()
>>> from sage.all import *
>>> Magmas().Unital().additional_structure()
Category of unital magmas

super_categories()

EXAMPLES:

```
sage: Magmas().super_categories()
[Category of sets]
```

### 4.112 Magmas and Additive Magmas

**class** `sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas`

**Bases:** `Category_singleton`

The category of sets $(S, +, \cdot)$ with an additive operation `$+$` and a multiplicative operation `\cdot`

**EXAMPLES:**

```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: C = MagmasAndAdditiveMagmas(); C
Category of magmas and additive magmas
```

This is the base category for the categories of rings and their variants:

```
sage: C.Distributive()
Category of distributive magmas and additive magmas
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().
    AdditiveUnital().AdditiveInverse()
Category of rngs
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().
    AdditiveUnital().Unital()
Category of semirings
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().
    AdditiveUnital().AdditiveInverse().Unital()
Category of rings
```

(continues on next page)
This category is really meant to represent the intersection of the categories of Magmas and AdditiveMagmas; however Sage's infrastructure does not allow yet to model this:

```
sage: Magmas() & AdditiveMagmas()
Join of Category of magmas and Category of additive magmas

sage: Magmas() & AdditiveMagmas()   # todo: not implemented
Category of magmas and additive magmas
```

```
>>> from sage.all import *
>>> Magmas() & AdditiveMagmas()
Join of Category of magmas and Category of additive magmas

>>> Magmas() & AdditiveMagmas()   # todo: not implemented
Category of magmas and additive magmas
```

class CartesianProducts(category, *args):
    Bases: CartesianProductsCategory

    extra_super_categories()

    Implement the fact that this structure is stable under Cartesian products.

Distributive
    alias of DistributiveMagmasAndAdditiveMagmas

class SubcategoryMethods
    Bases: object

    Distributive()

    Return the full subcategory of the objects of self where * is distributive on +.

    A magma and additive magma $M$ is distributive if, for all $x, y, z \in M$,

    $$x \ast (y + z) = x \ast y + x \ast z \text{ and } (x + y) \ast z = x \ast z + y \ast z$$

    EXAMPLES:

    ```
sage: from sage.categories.magmas_and_additive_magmas import _
    __MagmasAndAdditiveMagmas
sage: C = MagmasAndAdditiveMagmas().Distributive(); C
Category of distributive magmas and additive magmas
```

```
>>> from sage.all import *
>>> from sage.categories.magmas_and_additive_magmas import _
    __MagmasAndAdditiveMagmas
>>> C = MagmasAndAdditiveMagmas().Distributive(); C
Category of distributive magmas and additive magmas
```
Note: Given that Sage does not know that `MagmasAndAdditiveMagmas` is the intersection of `Magmas` and `AdditiveMagmas`, this method is not available for:

```sage
Magmas() & AdditiveMagmas()
```

Join of Category of magmas and Category of additive magmas

Still, the natural syntax works:

```sage
(Magmas() & AdditiveMagmas()).Distributive()
```

Category of distributive magmas and additive magmas

thanks to a workaround implemented in `Magmas.SubcategoryMethods.Distributive()`:

```sage
(Magmas() & AdditiveMagmas()).Distributive.__module__
```

'sage.categories.magmas'

### additional_structure()

Return None.

Indeed, this category is meant to represent the join of `AdditiveMagmas` and `Magmas`. As such, it defines no additional structure.

See also:

`Category.additional_structure()`

EXAMPLES:

```sage
from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
MagmasAndAdditiveMagmas().additional_structure()
```

```sage
from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
MagmasAndAdditiveMagmas().additional_structure()
```

### super_categories()

EXAMPLES:

```sage
from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
MagmasAndAdditiveMagmas().additional_structure()
```
4.113 Non-unital non-associative algebras

```python
sage: MagmasAndAdditiveMagmas().super_categories()
[Category of magmas, Category of additive magmas]
```

```python
>>> from sage.all import *
>>> from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
>>> MagmasAndAdditiveMagmas().super_categories()
[Category of magmas, Category of additive magmas]
```

class sage.categories.magmatic_algebras.MagmaticAlgebras(base, name=None)

Bases: Category_over_base_ring

The category of algebras over a given base ring.

An algebra over a ring $R$ is a module over $R$ endowed with a bilinear multiplication.

**Warning:** `MagmaticAlgebras` will eventually replace the current `Algebras` for consistency with e.g. Wikipedia article `Algebras` which assumes neither associativity nor the existence of a unit (see Issue #15043).

**EXAMPLES:**

```python
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: C = MagmaticAlgebras(ZZ); C
Category of magmatic algebras over Integer Ring
sage: C.super_categories()
[Category of additive commutative additive associative additive unital distributive magmas and additive magmas, Category of modules over Integer Ring]
```

```python
>>> from sage.all import *
>>> from sage.categories.magmatic_algebras import MagmaticAlgebras
>>> C = MagmaticAlgebras(ZZ); C
>>> C.super_categories()
[Category of additive commutative additive associative additive unital distributive magmas and additive magmas, Category of modules over Integer Ring]
```

**Associative**

alias of ` AssociativeAlgebras`

**class** `ParentMethods`

Bases: `object`

`algebra_generators()`

Return a family of generators of this algebra.

**EXAMPLES:**
Unital

alias of UnitalAlgebras

class WithBasis(base_category)

Bases: CategoryWithAxiom_over_base_ring

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom_over_base_ring

class ParentMethods

Bases: object

derivations_basis()

Return a basis for the Lie algebra of derivations of self as matrices.

A derivation $D$ of an algebra is an endomorphism of $A$ such that

$$D(ab) = D(a)b + aD(b)$$

for all $a, b \in A$. The set of all derivations form a Lie algebra.

EXAMPLES:

We construct the Heisenberg Lie algebra as a multiplicative algebra:

```python
sage: A = algebras.FiniteDimensional(QQ, [p_mult, q_mult, matrix(QQ, 3, 3)], 'p,q,z')
sage: A.inject_variables()
Defining p, q, z
sage: A.derivations_basis()
```

4.1.13. Non-unital non-associative algebras
We construct another example using the exterior algebra and verify we obtain a derivation:

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
... p_mult = matrix([[Integer(0),Integer(0),Integer(0)],
...                   [Integer(0), Integer(0), Integer(1)],
...                   [Integer(0), Integer(0), Integer(0)]]
... q_mult = matrix([[Integer(0),Integer(0),Integer(1)],
...                   [Integer(0), Integer(0), Integer(0)],
...                   [Integer(0), Integer(0), Integer(0)]]
... A = algebras.FiniteDimensional(QQ,
...                              [p_mult, q_mult, matrix(QQ, Integer(3), Integer(3))],
...                              'p,q,z')
>>> A.inject_variables()
Defining p, q, z
>>> p * q
z
>>> q * p
-z
>>> A.derivations_basis()
(...
[1 0 0] [0 1 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0]
[0 0 0] [0 0 0] [1 0 0] [0 1 0] [0 0 0] [0 0 0]
[0 0 1], [0 0 0], [0 0 0], [0 0 1], [1 0 0], [0 1 0])

We construct another example using the exterior algebra and verify we obtain a derivation:

```python
sage: # needs sage.combinat sage.modules
sage: A = algebras.Exterior(QQ, 1)
sage: A.derivations_basis()
...
[0 0]
[0 1]

We construct another example using the exterior algebra and verify we obtain a derivation:

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = algebras.Exterior(QQ, Integer(1))
>>> A.derivations_basis()
...
D = A.module_morphism(matrix=A.derivations_basis()[Integer(0)],
...   codomain=A)
>>> one, e = A.basis()
>>> all(D(a*b) == D(a) * b + a * D(b))
...   for a in A.basis() for b in A.basis())
True

REFERENCES:

Wikipedia article Derivation_(differential_algebra)
Bases: object

\textbf{algebra\_generators}()

Return generators for this algebra.

This default implementation returns the basis of this algebra.

\textbf{OUTPUT: a family}

\textbf{See also:}

• \texttt{basis()}
• \texttt{MagmaticAlgebras.ParentMethods.algebra\_generators()}

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: D4 = DescentAlgebra(QQ, 4).B()}
\texttt{\# needs sage.combinat sage.groups sage.modules}
\texttt{sage: D4.algebra\_generators()}
\texttt{\# needs sage.combinat sage.groups sage.modules}
\texttt{Lazy family (...)\_\{i in Compositions of 4\}}
\texttt{sage: R.<x> = ZZ[]}\texttt{[[]}
\texttt{sage: P = PartitionAlgebra(1, x, R)}\texttt{\# needs sage.combinat sage.modules}
\texttt{sage: P.algebra\_generators()}
\texttt{\# needs sage.combinat sage.modules}
\texttt{Lazy family (Term map from Partition diagrams of order 1}
\texttt{\to Partition Algebra of rank 1 with parameter x}
\texttt{\over Univariate Polynomial Ring in x}
\texttt{\over Integer Ring}(i)\_\{i in Partition diagrams of order 1\}}
\end{verbatim}

\begin{verbatim}
\texttt{\>>> from sage\_all import *}
\texttt{\>>> D4 = DescentAlgebra(QQ, Integer(4)).B()}\texttt{\# needs sage.combinat sage.groups sage.modules}
\texttt{\>>> D4.algebra\_generators()}\texttt{\# needs sage.combinat sage.groups sage.modules}
\texttt{Lazy family (...)\_\{i in Compositions of 4\}}
\texttt{\>>> R = ZZ[\texttt{[[]}}\texttt{[\texttt{[\texttt{\['\texttt{x}']}\texttt{]}\texttt{]}\texttt{]}\texttt{]}\texttt{]; (x,)}\texttt{=}\texttt{R._\texttt{first\_ngens}(1)}
\texttt{\>>> P = PartitionAlgebra(Integer(1), x, R)}\texttt{\# needs sage.combinat sage.modules}
\texttt{\>>> P.algebra\_generators()}\texttt{\# needs sage.combinat sage.modules}
\texttt{Lazy family (Term map from Partition diagrams of order 1}
\texttt{\to Partition Algebra of rank 1 with parameter x}
\texttt{\over Univariate Polynomial Ring in x}
\texttt{\over Integer Ring}(i)\_\{i in Partition diagrams of order 1\}}
\end{verbatim}

\textbf{product}()

The product of the algebra, as per \texttt{Magmas.ParentMethods.product()}

By default, this is implemented using one of the following methods, in the specified order:

• \texttt{product\_on\_basis()}
• \texttt{product\_by\_coercion()}

\textbf{EXAMPLES:}

4.113. Non-unital non-associative algebras 995
product_on_basis (i, j)

The product of the algebra on the basis (optional).

INPUT:
• i, j – the indices of two elements of the basis of self

Return the product of the two corresponding basis elements indexed by i and j.

If implemented, product() is defined from it by bilinearity.

EXAMPLES:

additional_structure()

Return None.

Indeed, the category of (magnetic) algebras defines no new structure: a morphism of modules and of magmas between two (magnetic) algebras is a (magnetic) algebra morphism.

See also:

Category.additional_structure()
EXAMPLES:

```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(ZZ).additional_structure()
```

```
>>> from sage.all import *
>>> from sage.categories.magmatic_algebras import MagmaticAlgebras
>>> MagmaticAlgebras(ZZ).additional_structure()
```

```
sage: super_categories()
```

EXAMPLES:

```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MA = MagmaticAlgebras(ZZ)
sage: MA.super_categories()
[Category of additive commutative additive associative additive
 unital distributive magmas and additive magmas,
 Category of modules over Integer Ring]
```

```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
```

```
sage: MA.is_subcategory((AdditiveSemigroups() & Magmas()).Distributive())
True
```

```
>>> from sage.all import *
>>> from sage.categories.magmatic_algebras import MagmaticAlgebras
>>>
```

```
sage: super_categories()
```

EXAMPLES:

```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MA = MagmaticAlgebras(ZZ)
sage: MA.super_categories()
[Category of additive commutative additive associative additive
 unital distributive magmas and additive magmas,
 Category of modules over Integer Ring]
```

```
>>> from sage.categories.additive_semigroups import AdditiveSemigroups
```

```
>>> MA.is_subcategory((AdditiveSemigroups() & Magmas()).Distributive())
True
```

4.114 Manifolds

```
class sage.categories.manifolds.ComplexManifolds(base, name=None)
```

```
Bases: Category_over_base_ring
```

The category of complex manifolds.

A $d$-dimensional complex manifold is a manifold whose underlying vector space is $\mathbb{C}^d$ and has a holomorphic atlas.

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

```
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> Manifolds(RR).super_categories()
```

(continues on next page)
class sage.categories.manifolds.Manifolds(base, name=None)

Bases: Category_over_base_ring

The category of manifolds over any topological field.

Let $k$ be a topological field. A $d$-dimensional $k$-manifold $M$ is a second countable Hausdorff space such that the neighborhood of any point $x \in M$ is homeomorphic to $k^d$.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR); C
Category of manifolds over Real Field with 53 bits of precision
sage: C.super_categories()
[Category of topological spaces]
```

class AlmostComplex(base_category)

Bases: CategoryWithAxiom_over_base_ring

The category of almost complex manifolds.

An almost complex manifold $M$ is a manifold with a smooth tensor field $J$ of rank $(1, 1)$ such that $J^2 = -1$ when regarded as a vector bundle isomorphism $J : TM \to TM$ on the tangent bundle. The tensor field $J$ is called the almost complex structure of $M$.

extra_super_categories()

Return the extra super categories of self.

An almost complex manifold is smooth.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).AlmostComplex().super_categories() # indirect doctest
[Category of smooth manifolds over Real Field with 53 bits of precision]
```
class **Analytic**(*base_category*)

Bases: `CategoryWithAxiom_over_base_ring`

The category of complex manifolds.

An analytic manifold is a manifold with an analytic atlas.

**extra_super_categories**()

Return the extra super categories of `self`.

An analytic manifold is smooth.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Analytic().super_categories()  # indirect doctest
[Category of smooth manifolds over Real Field with 53 bits of precision]
```

class **Connected**(*base_category*)

Bases: `CategoryWithAxiom_over_base_ring`

The category of connected manifolds.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).Connected()
sage: TestSuite(C).run(skip="_test_category_over_bases")
```

class **Differentiable**(*base_category*)

Bases: `CategoryWithAxiom_over_base_ring`

The category of differentiable manifolds.

A differentiable manifold is a manifold with a differentiable atlas.

class **FiniteDimensional**(*base_category*)

Bases: `CategoryWithAxiom_over_base_ring`

Category of finite dimensional manifolds.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).FiniteDimensional()
sage: TestSuite(C).run(skip="_test_category_over_bases")
```
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> C = Manifolds(RR).FiniteDimensional()
>>> TestSuite(C).run(skip="_test_category_over_bases")

class ParentMethods
    Bases: object

    dimension()

        Return the dimension of self.

        EXAMPLES:

        sage: from sage.categories.manifolds import Manifolds
        sage: M = Manifolds(RR).example()
        sage: M.dimension()
        3

    class Smooth(base_category)
        Bases: CategoryWithAxiom_over_base_ring

        The category of smooth manifolds.
        A smooth manifold is a manifold with a smooth atlas.

        extra_super_categories()

            Return the extra super categories of self.
            A smooth manifold is differentiable.

            EXAMPLES:

            sage: from sage.categories.manifolds import Manifolds
            sage: Manifolds(RR).Smooth().super_categories()  # indirect doctest  
            Category of differentiable manifolds over Real Field with 53 bits of precision

    class SubcategoryMethods
        Bases: object

        AlmostComplex()

            Return the subcategory of the almost complex objects of self.

            EXAMPLES:
```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).AlmostComplex()  # needs sage.rings.real_mpfr
Category of almost complex manifolds over Real Field with 53 bits of precision
```
Release 10.4

---

 needs sage.rings.real_mpfr

Category of connected manifolds
over Real Field with 53 bits of precision

>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> Manifolds(RR).Connected()
# needs sage.rings.real_mpfr

Category of connected manifolds
over Real Field with 53 bits of precision

**Differentiable()**

Return the subcategory of the differentiable objects of self.

**EXAMPLES:**

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Differentiable()
# needs sage.rings.real_mpfr
```

Category of differentiable manifolds
over Real Field with 53 bits of precision

```
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>>
Manifolds(RR).Differentiable()
# needs sage.rings.real_mpfr
```

**FiniteDimensional()**

Return the full subcategory of the finite dimensional objects of self.

**EXAMPLES:**

```
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).Connected().FiniteDimensional(); C
# needs sage.rings.real_mpfr
```

Category of finite dimensional connected manifolds
over Real Field with 53 bits of precision

```
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> C = Manifolds(RR).Connected().FiniteDimensional(); C
# needs sage.rings.real_mpfr
```

**Smooth()**

Return the subcategory of the smooth objects of self.

**EXAMPLES:**

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Smooth()
# needs sage.rings.real_mpfr
```

(continues on next page)
Category Framework, Release 10.4

Category of smooth manifolds
over Real Field with 53 bits of precision

```python
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> Manifolds(RR).Smooth()
# needs sage.rings.real_mpfr
Category of smooth manifolds
over Real Field with 53 bits of precision
```

**additional_structure()**

Return None.

Indeed, the category of manifolds defines no new structure: a morphism of topological spaces between manifolds is a manifold morphism.

See also:

*Category.additional_structure()*

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).additional_structure()
```

```python
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> Manifolds(RR).additional_structure()
```

**super_categories()**

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

```python
>>> from sage.all import *
>>> from sage.categories.manifolds import Manifolds
>>> Manifolds(RR).super_categories()
```

[Category of topological spaces]

---

### 4.115 Matrix algebras

**class** `sage.categories.matrix_algebras.MatrixAlgebras` *(base, name=None)*

**Bases:** *Category_over_base_ring*

The category of matrix algebras over a field.

**EXAMPLES:**

```python
sage: MatrixAlgebras(RationalField())
Category of matrix algebras over Rational Field
```

4.115. Matrix algebras 1003
>>> from sage.all import *
>>> MatrixAlgebras(RationalField())
Category of matrix algebras over Rational Field

\texttt{super\_categories()}

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: MatrixAlgebras(QQ).super_categories()
  [Category of algebras over Rational Field]
\end{verbatim}

\section{4.116 Metric Spaces}

\texttt{class sage.categories.metric_spaces.MetricSpaces\ (category, *args)}

Bases: \texttt{MetricSpacesCategory}

The category of metric spaces.

A \textit{metric} on a set $S$ is a function $d : S \times S \to \mathbb{R}$ such that:

\begin{itemize}
  \item $d(a, b) \geq 0$,
  \item $d(a, b) = 0$ if and only if $a = b$.
\end{itemize}

A metric space is a set $S$ with a distinguished metric.

\textbf{Implementation}

Objects in this category must implement either a \texttt{dist} on the parent or the elements or \texttt{metric} on the parent; otherwise this will cause an infinite recursion.

\textbf{Todo:}

\begin{itemize}
  \item Implement a general geodesics class.
  \item Implement a category for metric additive groups and move the generic distance $d(a, b) = |a - b|$ there.
  \item Incorporate the length of a geodesic as part of the default distance cycle.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: from sage.categories.metric_spaces import MetricSpaces
  sage: C = MetricSpaces()
  sage: C
  Category of metric spaces
  sage: TestSuite(C).run()
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> from sage.categories.metric_spaces import MetricSpaces
>>> C = MetricSpaces()

(continues on next page)
class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory

class ParentMethods
    Bases: object

dist(a, b)

    Return the distance between a and b in self.
    It is defined as the maximum of the distances within the Cartesian factors.

EXAMPLES:

    sage: from sage.categories.metric_spaces import MetricSpaces
    sage: Q2 = QQ.cartesian_product(QQ)
    sage: Q2.category()  # Join of Category of Cartesian products of commutative rings and Category of Cartesian products of metric spaces
    sage: Q2 in MetricSpaces()  # True
    sage: Q2.dist((Integer(0), Integer(0)), (Integer(2), Integer(3)))
    3

extra_super_categories()

    Implement the fact that a (finite) Cartesian product of metric spaces is a metric space.

EXAMPLES:

    sage: from sage.categories.metric_spaces import MetricSpaces
    sage: C = MetricSpaces().CartesianProducts()
    sage: C.extra_super_categories()  # [Category of metric spaces]
    sage: C.super_categories()  # [Category of Cartesian products of topological spaces, Category of metric spaces]
    sage: C.axioms()  # frozenset()
```python
>>> from sage.all import *
>>> from sage.categories.metric_spaces import MetricSpaces
>>> C = MetricSpaces().CartesianProducts()
>>> C.extra_super_categories()
[Category of metric spaces]
>>> C.super_categories()
[Category of Cartesian products of topological spaces,
 Category of metric spaces]
>>> C.axioms()
frozenset()
```

class Complete(base_category)

Bases: CategoryWithAxiom

The category of complete metric spaces.

class CartesianProducts(category, *args)

Bases: CartesianProductsCategory

extra_super_categories()

Implement the fact that a (finite) Cartesian product of complete metric spaces is a complete metric space.

EXAMPLES:

```python
sage: from sage.categories.metric_spaces import MetricSpaces
sage: C = MetricSpaces().Complete().CartesianProducts()
sage: C.extra_super_categories()
[Category of complete metric spaces]
sage: C.super_categories()
[Category of Cartesian products of metric spaces,
 Category of complete metric spaces]
sage: C.axioms()
frozenset({'Complete'})
```

```python
sage: R2 = RR.cartesian_product(RR)
sage: R2 in MetricSpaces()  # True
sage: R2 in MetricSpaces().Complete()  # True
sage: QR = QQ.cartesian_product(RR)
sage: QR in MetricSpaces()  # True
sage: QR in MetricSpaces().Complete()  # False
```

```python
(continues on next page)
```

1006 Chapter 4. Individual Categories
class ElementMethods

Bases: object

abs()

Return the absolute value of self.

EXAMPLES:

```python
sage: CC(I).abs()  # needs sage.rings.real_mpfr sage.symbolic
1.00000000000000
```

```python
>>> from sage.all import *
```

```python
sage: CC(I).abs()  # needs sage.rings.real_mpfr sage.symbolic
1.00000000000000
```

dist(b)

Return the distance between self and other.

EXAMPLES:

```python
sage: # needs sage.symbolic
sage: UHP = HyperbolicPlane().UHP()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1 + I)
sage: p1.dist(p2)
arccosh(33/7)
```

```python
>>> from sage.all import *
>>> # needs sage.symbolic
>>> UHP = HyperbolicPlane().UHP()
>>> p1 = UHP.get_point(Integer(5) + Integer(7)*I)
>>> p2 = UHP.get_point(Integer(1) + I)
>>> p1.dist(p2)
arccosh(33/7)
```

class Homsets (category, *args)

Bases: HomsetsCategory

The category of homsets of metric spaces

It consists of the metric maps, that is, the Lipschitz functions with Lipschitz constant 1.
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

dist(a, b)
    Return the distance between a and b in self.

EXAMPLES:

sage: # needs sage.symbolic
sage: UHP = HyperbolicPlane().UHP()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1.0 + I)
sage: UHP.dist(p1, p2)
2.23230104635820

sage: PD = HyperbolicPlane().PD()
# needs sage.symbolic
sage: PD.dist(PD.get_point(0), PD.get_point(I/2))
# needs sage.symbolic
arccosh(5/3)

_metric(*args, **kwds)
    Deprecated: Use metric_function() instead. See Issue #30062 for details.

metric_function()
    Return the metric function of self.

EXAMPLES:

sage: # needs sage.symbolic
sage: UHP = HyperbolicPlane().UHP()
sage: m = UHP.metric_function()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1.0 + I)
sage: m(p1, p2)
2.23230104635820

>>> from sage.all import *
>>> # needs sage.symbolic
>>> UHP = HyperbolicPlane().UHP()
>>> m = UHP.metric_function()
>>> p1 = UHP.get_point(5 + 7*I)
>>> p2 = UHP.get_point(1.0 + I)
>>> m(p1, p2)
2.23230104635820

>>> from sage.all import *
>>> # needs sage.symbolic
>>> UHP = HyperbolicPlane().UHP()
>>> m = UHP.metric_function()
>>> p1 = UHP.get_point(Integer(5) + Integer(7)*I)
>>> p2 = UHP.get_point(RealNumber('1.0') + I)
>>> m(p1, p2)
2.23230104635820

class SubcategoryMethods
Bases: object

Complete()

Return the full subcategory of the complete objects of self.

EXAMPLES:

sage: Sets().Metric().Complete()
Category of complete metric spaces

from sage.all import *

sage: Sets().Metric().Complete()
Category of complete metric spaces

class WithRealizations (category, *args)
Bases: WithRealizationsCategory

class ParentMethods
Bases: object

dist(a, b)

Return the distance between a and b by converting them to a realization of self and doing the computation.

EXAMPLES:

from sage.all import *

# needs sage.symbolic
H = HyperbolicPlane()
P = H.PD()
p1 = PD.get_point(0)
p2 = PD.get_point(I/2)
H.dist(p1, p2)
arccosh(5/3)

from sage.all import *

# needs sage.symbolic
H = HyperbolicPlane()
P = H.PD()
p1 = PD.get_point(Integer(0))
p2 = PD.get_point(I/Integer(2))
H.dist(p1, p2)
arccosh(5/3)

class sage.categories.metric_spaces.MetricSpacesCategory (category, *args)
Bases: RegressiveCovariantConstructionCategory

classmethod default_super_categories (category)

Return the default super categories of category.Metric().

Mathematical meaning: if A is a metric space in the category C, then A is also a topological space.
INPUT:

- **cls** – the class `MetricSpaces`
- **category** – a category `Cat`

OUTPUT:

A (join) category

In practice, this returns `category.Metric()`, joined together with the result of the method `RegressiveCovariantConstructionCategory.default_super_categories()` (that is the join of `category` and `cat.Metric()` for each `cat` in the super categories of `category`).

EXAMPLES:

Consider `category=Groups()`. Then, a group `G` with a metric is simultaneously a topological group by itself, and a metric space:

```python
sage: Groups().Metric().super_categories()
[Category of topological groups, Category of metric spaces]
```

This resulted from the following call:

```python
sage: sage.categories.metric_spaces.MetricSpacesCategory.default_super_categories(Groups())
```

4.117 Modular abelian varieties

```python
class sage.categories.modular_abelian_varieties.ModularAbelianVarieties(Y)
Bases: Category_over_base
```

The category of modular abelian varieties over a given field.

EXAMPLES:

```python
sage: ModularAbelianVarieties(QQ)
Category of modular abelian varieties over Rational Field
```

```python
>>> from sage.all import *
```
class Endset(base_category)
    Bases: CategoryWithAxiom
extra_super_categories()
    Implement the fact that an endset of modular abelian variety is a ring.
    EXAMPLES:

    sage: ModularAbelianVarieties(QQ).Endsets().extra_super_categories()
    [Category of rings]

    >>> from sage.all import *
    >>> ModularAbelianVarieties(QQ).Endsets().extra_super_categories()
    [Category of rings]

base_field()
    EXAMPLES:

    sage: ModularAbelianVarieties(QQ).base_field()
    Rational Field

    >>> from sage.all import *
    >>> ModularAbelianVarieties(QQ).base_field()
    Rational Field

super_categories()
    EXAMPLES:

    sage: ModularAbelianVarieties(QQ).super_categories()
    [Category of sets]

    >>> from sage.all import *
    >>> ModularAbelianVarieties(QQ).super_categories()
    [Category of sets]

4.118 Modules

class sage.categories.modules.Modules(base, name=None)
    Bases: Category_module

    The category of all modules over a base ring $R$.

    An $R$-module $M$ is a left and right $R$-module over a commutative ring $R$ such that:

    \[ r \cdot (x \cdot s) = (r \cdot x) \cdot s \quad \forall r, s \in R \text{ and } x \in M \]

    INPUT:

    - base_ring -- a ring $R$ or subcategory of Rings()
    - dispatch -- a boolean (for internal use; default: True)

    When the base ring is a field, the category of vector spaces is returned instead (unless dispatch == False).
Warning: Outside of the context of symmetric modules over a commutative ring, the specifications of this category are fuzzy and not yet set in stone (see below). The code in this category and its subcategories is therefore prone to bugs or arbitrary limitations in this case.

EXAMPLES:

```
sage: Modules(ZZ)
Category of modules over Integer Ring
sage: Modules(QQ)
Category of vector spaces over Rational Field
sage: Modules(Rings())
Category of modules over rings
sage: Modules(FiniteFields())
Category of vector spaces over finite enumerated fields
sage: Modules(Integers(9))
Category of modules over Ring of integers modulo 9
sage: Modules(Integers(9)).super_categories()
[Category of bimodules over Ring of integers modulo 9 on the left
 and Ring of integers modulo 9 on the right]
sage: Modules(ZZ).super_categories()
[Category of bimodules over Integer Ring on the left
 and Integer Ring on the right]
sage: Modules == RingModules
True
sage: Modules(ZZ['x']).is_abelian()  # see #6081
True
```

```python
>>> from sage.all import *

>>> Modules(ZZ)
Category of modules over Integer Ring
>>> Modules(QQ)
Category of vector spaces over Rational Field
>>> Modules(Rings())
Category of modules over rings
>>> Modules(FiniteFields())
Category of vector spaces over finite enumerated fields
>>> Modules(Integers(Integer(9)))
Category of modules over Ring of integers modulo 9
>>> Modules(Integers(Integer(9))).super_categories()
[Category of bimodules over Ring of integers modulo 9 on the left
 and Ring of integers modulo 9 on the right]
>>> Modules(ZZ).super_categories()
[Category of bimodules over Integer Ring on the left
 and Integer Ring on the right]
>>> Modules == RingModules
True
```
Todo:

- Clarify the distinction, if any, with \texttt{BiModules(R, R)}. In particular, if \( R \) is a commutative ring (e.g. a field), some pieces of the code possibly assume that \( M \) is a symmetric \( R \)-\( R \)-bimodule:

\[
r \cdot x = x \cdot r \quad \forall r \in R \text{ and } x \in M
\]

- Make sure that non-symmetric modules are properly supported by all the code, and advertise it.
- Make sure that non-commutative rings are properly supported by all the code, and advertise it.
- Add support for base semirings.
- Implement a \texttt{FreeModules(R)} category, when so prompted by a concrete use case: e.g. modeling a free module with several bases (using \texttt{Sets.SubcategoryMethods.Realizations()}) or with an atlas of local maps (see e.g. Issue \#15916).

```python
from sage.all import *
```

```python
>>> Modules(ZZ['x']).is_abelian()  # see \#6081
True
```

\texttt{Todo:}

```python
class CartesianProducts(category, *args):
    Bases: CartesianProductsCategory

    The category of modules constructed as Cartesian products of modules

    This construction gives the direct product of modules. The implementation is based on the following resources:

    - http://groups.google.fr/group/sage-devel/browse_thread/thread/35a72b1d0a2fc77a/348f42ae77a66d16
    - Wikipedia article Direct_product

    class ElementMethods
        Bases: object

    class ParentMethods
        Bases: object

    extra_super_categories()

    A Cartesian product of modules is endowed with a natural module structure.

    EXAMPLES:
```

```python
sage: Modules(ZZ).CartesianProducts().extra_super_categories()
[Category of modules over Integer Ring]
```

```python
sage: Modules(ZZ).CartesianProducts().super_categories()
[Category of Cartesian products of commutative additive groups, Category of modules over Integer Ring]
```

\```
class ElementMethods
    Bases: object

Filtered
    alias of FilteredModules

class FiniteDimensional(base_category)
    Bases: CategoryWithAxiom_over_base_ring

class TensorProducts(category, *args)
    Bases: TensorProductsCategory

    extra_super_categories()
    Implement the fact that a (finite) tensor product of finite dimensional modules is a finite dimensional module.

    EXAMPLES:

    sage: Modules(ZZ).FiniteDimensional().TensorProducts().extra_super_categories()
    [Category of finite dimensional modules over Integer Ring]
    sage: Modules(QQ).FiniteDimensional().TensorProducts().
    -FiniteDimensional()
    Category of tensor products of finite dimensional vector spaces over Rational Field

    >>> from sage.all import *
    >>> Modules(ZZ).FiniteDimensional().TensorProducts().extra_super_categories()
    [Category of finite dimensional modules over Integer Ring]
    >>> Modules(QQ).FiniteDimensional().TensorProducts().
    -FiniteDimensional()
    Category of tensor products of finite dimensional vector spaces over Rational Field

    extra_super_categories()
    Implement the fact that a finite dimensional module over a finite ring is finite.

    EXAMPLES:

    sage: Modules(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
    [Category of finite sets]
    sage: Modules(ZZ).FiniteDimensional().extra_super_categories()
    []
    sage: Modules(GF(5)).FiniteDimensional().is_subcategory(Sets().Finite())
    True
    sage: Modules(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
    False
    sage: Modules(Rings().Finite()).FiniteDimensional().is_subcategory(Sets().
    -Finite())
    True
class FinitelyPresented(base_category)

Bases: CategoryWithAxiom_over_base_ring

extra_super_categories()

Implement the fact that a finitely presented module over a finite ring is finite.

EXAMPLES:

sage: Modules(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
[Category of finite sets]
sage: Modules(ZZ).FiniteDimensional().extra_super_categories()
[]
sage: Modules(GF(5)).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: Modules(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
False
sage: Modules(Rings().Finite()).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: Modules(Rings()).FiniteDimensional().is_subcategory(Sets().Finite())
False

>>> from sage.all import *

>>> Modules(IntegerModRing(Integer(4))).FiniteDimensional().extra_super_categories()
[Category of finite sets]
>>> Modules(ZZ).FiniteDimensional().extra_super_categories()
[]
>>> Modules(GF(Integer(5))).FiniteDimensional().is_subcategory(Sets().Finite())
True
>>> Modules(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
False

(continues on next page)
Graded
alias of `GradedModules`

class `Homsets` *(category, *args)*

Bases: `HomsetsCategory`

The category of homomorphism sets hom\((X, Y)\) for \(X, Y\) modules.

class `Endset` *(base_category)*

Bases: `CategoryWithAxiom_over_base_ring`

The category of endomorphism sets End\((X)\) for \(X\) a module (this is not used yet)

extra_super_categories()

Implement the fact that the endomorphism set of a module is an algebra.

See also:
CategoryWithAxiom.extra_super_categories()

EXAMPLES:

```sage
sage: Modules(ZZ).Endsets().extra_super_categories()
[Category of magmatic algebras over Integer Ring]

sage: End(ZZ^3) in Algebras(ZZ)  # needs sage.modules
True

>>> from sage.all import *

sage: Modules(ZZ).Endsets().extra_super_categories()
[Category of magmatic algebras over Integer Ring]

sage: End(ZZ**Integer(3)) in Algebras(ZZ)  # needs sage.modules
True

class `ParentMethods`

Bases: object

base_ring()

Return the base ring of self.

EXAMPLES:

```sage
sage: # needs sage.modules
sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
sage: H = Hom(E, F)
sage: H.base_ring()
Integer Ring
```
```python
>>> from sage.all import *

# needs sage.modules

E = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2), Integer(3)])
F = CombinatorialFreeModule(ZZ, [Integer(2), Integer(3), Integer(4)])
H = Hom(E, F)
H.base_ring()
```

```
Integer Ring
```

This `base_ring` method is actually overridden by `sage.structure.category_object.CategoryObject.base_ring()`:

```
sage: H.base_ring.__module__  # needs sage.modules
˓→needs sage.modules
```

```
>>> from sage.all import *

H.base_ring.__module__  # needs sage.modules
˓→needs sage.modules
```

Here we call it directly:

```
sage: method = H.category().parent_class.base_ring  # needs sage.modules
˓→needs sage.modules
sage: method.__get__(H)()  # needs sage.modules
```

```
Integer Ring
```

`zero()`

**EXAMPLES:**

```
sage: # needs sage.modules
sage: E = CombinatorialFreeModule(ZZ, [1, 2, 3])
sage: F = CombinatorialFreeModule(ZZ, [2, 3, 4])
sage: H = Hom(E, F)
sage: f = H.zero()
sage: f
Generic morphism:
    From: Free module generated by {1, 2, 3} over Integer Ring
    To:   Free module generated by {2, 3, 4} over Integer Ring
sage: f(E.monomial(2))
0
sage: f(E.monomial(3)) == F.zero()
```

```
True
```

```
>>> from sage.all import *

# needs sage.modules
>>> E = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2), Integer(3)])
>>> F = CombinatorialFreeModule(ZZ, [Integer(2), Integer(3), Integer(4)])
>>> H = Hom(E, F)
>>> f = H.zero()
```

(continues on next page)
\section*{Category Framework, Release 10.4}

\begin{lstlisting}[language=Python]
>>> f
Generic morphism:  
    From: Free module generated by \{1, 2, 3\} over Integer Ring
    To:   Free module generated by \{2, 3, 4\} over Integer Ring
>>> f(E.monomial(Integer(2)))
0
>>> f(E.monomial(Integer(3))) == F.zero()
True
\end{lstlisting}

**base_ring()**

**EXAMPLES:**

```
sage: Modules(ZZ).Homsets().base_ring()
Integer Ring
>>> from sage.all import *
>>> Modules(ZZ).Homsets().base_ring()
Integer Ring
```

**Todo:** Generalize this so that any homset category of a full subcategory of modules over a base ring is a category over this base ring.

**extra_super_categories()**

**EXAMPLES:**

```
sage: Modules(ZZ).Homsets().extra_super_categories()
[Category of modules over Integer Ring]
>>> from sage.all import *
>>> Modules(ZZ).Homsets().extra_super_categories()
[Category of modules over Integer Ring]
```

**class ParentMethods**

**Bases:** object

**linear_combination(iter_of_elements_coeff, factor_on_left=True)**

Return the linear combination \(\lambda_1 v_1 + \cdots + \lambda_k v_k\) (resp. the linear combination \(v_1 \lambda_1 + \cdots + v_k \lambda_k\)) where \(\text{iter}_\text{of}_\text{elements}_\text{coeff}\) iterates through the sequence \([(\lambda_1, v_1), \ldots, (\lambda_k, v_k)]\).

**INPUT:**

- \text{iter}_\text{of}_\text{elements}_\text{coeff} – iterator of pairs \((\text{element}, \text{coeff})\) with \text{element} in \text{self} and \text{coeff} in \text{self}.base_ring()
- \text{factor}_\text{on}_\text{left} – (optional) if True, the coefficients are multiplied from the left; if False, the coefficients are multiplied from the right

**EXAMPLES:**

```
sage: m = matrix([[0,1], [1,1]])  # needs sage.modules
sage: J.<a,b,c> = JordanAlgebra(m)  # needs sage.combinat,sage.modules
sage: J.linear_combination(((a+b, 1), (-2*b + c, -1)))  # needs sage.combinat,sage.modules
1 + (3, -1)
```

Chapter 4. Individual Categories

1018
>>> from sage.all import *
>>> m = matrix([[Integer(0), Integer(1)], [Integer(1), Integer(1)]])._needs_sage.modules
>>> J = JordanAlgebra(m, names=('a', 'b', 'c')); (a, b, c) = J._first_ngens(3)._needs_sage.combinat.sage.modules
>>> J.linear_combination(((a+b, Integer(1)), (-Integer(2)*b + c, -Integer(1))))._needs_sage.combinat.sage.modules
1 + (3, -1)

module_morphism (function, category, codomain, **keywords)

Construct a module morphism from self to codomain.

Let self be a module $X$ over a ring $R$. This constructs a morphism $f : X \to Y$.

INPUT:

• self – a parent $X$ in Modules(R).
• function – a function $f$ from $X$ to $Y$
• codomain – the codomain $Y$ of the morphism (default: $f$.codomain() if it’s defined; otherwise it must be specified)
• category – a category or None (default: None)

EXAMPLES:

sage: # needs sage.modules
sage: V = FiniteRankFreeModule(QQ, 2)

sage: e = V.basis('e'); e
Basis (e_0,e_1) on the 2-dimensional vector space over the Rational Field

sage: neg = V.module_morphism(function=operator.neg, codomain=V); neg
Generic endomorphism of 2-dimensional vector space over the Rational Field

sage: neg(e[0])
Element -e_0 of the 2-dimensional vector space over the Rational Field

 >>> from sage.all import *
 >>> # needs sage.modules
 >>> V = FiniteRankFreeModule(QQ, Integer(2))
 >>> e = V.basis('e'); e
 Basis (e_0,e_1) on the 2-dimensional vector space over the Rational Field
 >>> neg = V.module_morphism(function=operator.neg, codomain=V); neg
 Generic endomorphism of 2-dimensional vector space over the Rational Field
 >>> neg(e[Integer(0)])
 Element -e_0 of the 2-dimensional vector space over the Rational Field

quotient (submodule, check=True, **kwds)

Construct the quotient module self / submodule.

INPUT:

• submodule – a submodule with basis of self, or something that can be turned into one via self.submodule(submodule)
• check, other keyword arguments: passed on to quotient_module().

This method just delegates to quotient_module(). Classes implementing modules should override that method.

Parents in categories with additional structure may override quotient(). For example, in algebras, quotient() will be the same as quotient_ring().
EXAMPLES:

```python
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
# needs sage.modules
sage: TA = TensorAlgebra(C)
# needs sage.combinat sage.modules
sage: TA.quotient
# needs sage.combinat sage.modules
<bound method Rings.ParentMethods.quotient of Tensor Algebra of Free module generated by {'a', 'b', 'c'} over Rational Field>
```

```python
>>> from sage.all import *

>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
# needs sage.modules

>>> TA = TensorAlgebra(C)
# needs sage.combinat sage.modules

>>> TA.quotient
# needs sage.combinat sage.modules
<bound method Rings.ParentMethods.quotient of Tensor Algebra of Free module generated by {'a', 'b', 'c'} over Rational Field>
```

tensor_square()

Returns the tensor square of self

EXAMPLES:

```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
# needs sage.groups sage.modules
sage: A.tensor_square()
# needs sage.groups sage.modules
```

An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6
as a permutation group over Rational Field
An example of Hopf algebra with basis: the group algebra of the Dihedral
group of order 6 as a permutation group over Rational Field

```python
>>> from sage.all import *

>>> A = HopfAlgebrasWithBasis(QQ).example()
# needs sage.groups sage.modules

>>> A.tensor_square()
# needs sage.groups sage.modules
```

An example of Hopf algebra with basis:
the group algebra of the Dihedral group of order 6
as a permutation group over Rational Field
An example of Hopf algebra with basis: the group algebra of the Dihedral
group of order 6 as a permutation group over Rational Field

class SubcategoryMethods

Bases: object

DualObjects()

Return the category of spaces constructed as duals of spaces of self.

The dual of a vector space \( V \) is the space consisting of all linear functionals on \( V \) (see Wikipedia article Dual space). Additional structure on \( V \) can endow its dual with additional structure; for example, if \( V \) is a finite dimensional algebra, then its dual is a coalgebra.
This returns the category of spaces constructed as dual of spaces in \textit{self}, endowed with the appropriate additional structure.

\textbf{Warning:}

\begin{itemize}
  \item This semantic of dual and DualObject is imposed on all subcategories, in particular to make dual a covariant functorial construction.
    
    A subcategory that defines a different notion of dual needs to use a different name.
    
  \item Typically, the category of graded modules should define a separate graded_dual construction (see Issue \#15647). For now the two constructions are not distinguished which is an oversimplified model.
\end{itemize}

See also:

\begin{itemize}
  \item dual.DualObjectsCategory
    
  \item CovariantFunctorialConstruction.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
  sage: VectorSpaces(QQ).DualObjects()
  Category of duals of vector spaces over Rational Field

  >>> from sage.all import *
  >>> VectorSpaces(QQ).DualObjects()
  Category of duals of vector spaces over Rational Field
\end{verbatim}

The dual of a vector space is a vector space:

\begin{verbatim}
  sage: VectorSpaces(QQ).DualObjects().super_categories()
  [Category of vector spaces over Rational Field]

  >>> from sage.all import *
  >>> VectorSpaces(QQ).DualObjects().super_categories()
  [Category of vector spaces over Rational Field]
\end{verbatim}

The dual of an algebra is a coalgebra:

\begin{verbatim}
  sage: sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
  [Category of coalgebras over Rational Field,
   Category of duals of vector spaces over Rational Field]

  >>> from sage.all import *
  >>> sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
  [Category of coalgebras over Rational Field,
   Category of duals of vector spaces over Rational Field]
\end{verbatim}

The dual of a coalgebra is an algebra:

\begin{verbatim}
  sage: sorted(Coalgebras(QQ).DualObjects().super_categories(), key=str)
  [Category of algebras over Rational Field,
   Category of duals of vector spaces over Rational Field]

  >>> from sage.all import *
  >>> sorted(Coalgebras(QQ).DualObjects().super_categories(), key=str)
\end{verbatim}

(continues on next page)
As a shorthand, this category can be accessed with the `dual()` method:

```sage
VectorSpaces(QQ).dual()
```

```
Category of duals of vector spaces over Rational Field
```

```
>>> from sage.all import *

>>> VectorSpaces(QQ).dual()

```

**Filtered** *(base_ring=None)*

Return the subcategory of the filtered objects of `self`.

**INPUT:**

- `base_ring` – this is ignored

**EXAMPLES:**

```sage
Modules(ZZ).Filtered()
```

```
Category of filtered modules over Integer Ring
```

```sage
Coalgebras(QQ).Filtered()
```

```
Category of filtered coalgebras over Rational Field
```

```sage
AlgebrasWithBasis(QQ).Filtered()
```

```
Category of filtered algebras with basis over Rational Field
```

```
>>> from sage.all import *


>>> Modules(ZZ).Filtered()

```

```
Category of filtered modules over Integer Ring
```

```
>>> Coalgebras(QQ).Filtered()

```

```
Category of filtered coalgebras over Rational Field
```

```
>>> AlgebrasWithBasis(QQ).Filtered()

```

```
Category of filtered algebras with basis over Rational Field
```

**Todo:**

- Explain why this does not commute with `WithBasis()`
- Improve the support for covariant functorial constructions categories over a base ring so as to get rid of the `base_ring` argument.

**FiniteDimensional**

Return the full subcategory of the finite dimensional objects of `self`.

**EXAMPLES:**

```sage
Modules(ZZ).FiniteDimensional()
```

```
Category of finite dimensional modules over Integer Ring
```

```sage
Coalgebras(QQ).FiniteDimensional()
```

```
Category of finite dimensional coalgebras over Rational Field
```

```sage
AlgebrasWithBasis(QQ).FiniteDimensional()
```

```
Category of finite dimensional algebras with basis over Rational Field
```
>>> from sage.all import *
>>> Modules(ZZ).FiniteDimensional()
Category of finite dimensional modules over Integer Ring
>>> Coalgebras(QQ).FiniteDimensional()
Category of finite dimensional coalgebras over Rational Field
>>> AlgebrasWithBasis(QQ).FiniteDimensional()
Category of finite dimensional algebras with basis over Rational Field

FinitelyPresented()

Return the full subcategory of the finitely presented objects of self.

EXAMPLES:

sage: Modules(ZZ).FinitelyPresented()
Category of finitely presented modules over Integer Ring
sage: A = SteenrodAlgebra(2)
# needs sage.combinat sage.modules
sage: from sage.modules.fp_graded.module import FPModule
# needs sage.combinat sage.modules
sage: FPModule(A, [0, 1], [[Sq(2), Sq(1)]]).category()
Category of finitely presented graded modules
over mod 2 Steenrod algebra, milnor basis

Graded(base_ring=None)

Return the subcategory of the graded objects of self.

INPUT:
• base_ring – this is ignored

EXAMPLES:

sage: Modules(ZZ).Graded()
Category of graded modules over Integer Ring
sage: Coalgebras(QQ).Graded()
Category of graded coalgebras over Rational Field
sage: AlgebrasWithBasis(QQ).Graded()
Category of graded algebras with basis over Rational Field

(continues on next page)
>>> Coalgebras(QQ).Graded()
Category of graded coalgebras over Rational Field

>>> AlgebrasWithBasis(QQ).Graded()
Category of graded algebras with basis over Rational Field

Todo:
- Explain why this does not commute with `WithBasis()`
- Improve the support for covariant functorial constructions categories over a base ring so as to get rid of the `base_ring` argument.

Super `(base_ring=None)`
Return the super-analogue category of `self`.

INPUT:
- `base_ring` – this is ignored

EXAMPLES:

```python
sage: Modules(ZZ).Super()
Category of super modules over Integer Ring

sage: Coalgebras(QQ).Super()
Category of super coalgebras over Rational Field

sage: AlgebrasWithBasis(QQ).Super()
Category of super algebras with basis over Rational Field
```

Todo:
- Explain why this does not commute with `WithBasis()`
- Improve the support for covariant functorial constructions categories over a base ring so as to get rid of the `base_ring` argument.

TensorProducts()
Return the full subcategory of objects of `self` constructed as tensor products.

See also:
- `tensor.TensorProductsCategory`
- `RegressiveCovariantFunctorialConstruction`.

EXAMPLES:
**WithBasis()**

Return the full subcategory of the objects of *self* with a distinguished basis.

**EXAMPLES:**

```python
sage: Modules(QQ).WithBasis()
Category of modules with basis over Integer Ring
sage: Coalgebras(QQ).WithBasis()
Category of coalgebras with basis over Rational Field
sage: AlgebrasWithBasis(QQ).WithBasis()
Category of algebras with basis over Rational Field
```

**base_ring()**

Return the base ring (category) for *self*.

This implements a `base_ring` method for all subcategories of `Modules(K)`.

**EXAMPLES:**

```python
sage: C = Modules(QQ) & Semigroups(); C
Join of Category of semigroups and Category of vector spaces over Rational Field
sage: C.base_ring()
Rational Field
sage: C.base_ring.__module__
sage.categories.modules
sage: C2 = Modules(Rings()) & Semigroups(); C2
Join of Category of semigroups and Category of modules over rings
sage: C2.base_ring()
Category of rings
sage: C2.base_ring.__module__
sage.categories.modules
sage: # needs sage.combinat sage.groups sage.modules
sage: C3 = DescentAlgebra(QQ,3).B().category()
sage: C3.base_ring.__module__
sage.categories.modules
sage: C3.base_ring()
Rational Field
```
sage: # needs sage.combinat sage.modules
sage: C4 = QuasiSymmetricFunctions(QQ).F().category()

sage: C4.base_ring.__module__
'sage.categories.modules'

sage: C4.base_ring()
Rational Field

>>> from sage.all import *

>>> C = Modules(QQ) & Semigroups(); C
Join of Category of semigroups
and Category of vector spaces over Rational Field

>>> C.base_ring()
Rational Field

>>> C.base_ring.__module__
'sage.categories.modules'

>>> C2 = Modules(Rings()) & Semigroups(); C2
Join of Category of semigroups and Category of modules over rings

>>> C2.base_ring()
Category of rings

>>> C2.base_ring.__module__
'sage.categories.modules'

>>> # needs sage.combinat sage.groups sage.modules

>>> C3 = DescentAlgebra(QQ,Integer(3)).B().category()

>>> C3.base_ring.__module__
'sage.categories.modules'

>>> C3.base_ring()
Rational Field

>>> # needs sage.combinat sage.modules

>>> C4 = QuasiSymmetricFunctions(QQ).F().category()

>>> C4.base_ring.__module__
'sage.categories.modules'

>>> C4.base_ring()
Rational Field

``dual ()``

Return the category of spaces constructed as duals of spaces of `self`.

The `dual` of a vector space $V$ is the space consisting of all linear functionals on $V$ (see Wikipedia article `Dual_space`). Additional structure on $V$ can endow its dual with additional structure; for example, if $V$ is a finite dimensional algebra, then its dual is a coalgebra.

This returns the category of spaces constructed as dual of spaces in `self`, endowed with the appropriate additional structure.

**Warning:**

- This semantic of `dual` and `DualObject` is imposed on all subcategories, in particular to make `dual` a covariant functorial construction.

  A subcategory that defines a different notion of dual needs to use a different name.

- Typically, the category of graded modules should define a separate `graded_dual construc-
tion (see Issue #15647). For now the two constructions are not distinguished which is an oversimplified model.

See also:

- `dual.DualObjectsCategory`
- `CovariantFunctorialConstruction`.

EXAMPLES:

```python
sage: VectorSpaces(QQ).DualObjects()
Category of duals of vector spaces over Rational Field
```

The dual of a vector space is a vector space:

```python
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```

```python
>>> from sage.all import *

>>> VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```

The dual of an algebra is a coalgebra:

```python
sage: sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
[Category of coalgebras over Rational Field,
Category of duals of vector spaces over Rational Field]
```

```python
>>> from sage.all import *

>>> sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
[Category of coalgebras over Rational Field,
Category of duals of vector spaces over Rational Field]
```

The dual of a coalgebra is an algebra:

```python
sage: sorted(Coalgebras(QQ).DualObjects().super_categories(), key=str)
[Category of algebras over Rational Field,
Category of duals of vector spaces over Rational Field]
```

```python
>>> from sage.all import *

>>> sorted(Coalgebras(QQ).DualObjects().super_categories(), key=str)
[Category of algebras over Rational Field,
Category of duals of vector spaces over Rational Field]
```

As a shorthand, this category can be accessed with the `dual()` method:

```python
sage: VectorSpaces(QQ).dual()
Category of duals of vector spaces over Rational Field
```
>>> from sage.all import *
>>> VectorSpaces(QQ).dual()
Category of duals of vector spaces over Rational Field

Super
alias of SuperModules
class TensorProducts(category, *args)
    Bases: TensorProductsCategory
    The category of modules constructed by tensor product of modules.
class ParentMethods
    Bases: object
    Implement operations on tensor products of modules.
    construction()
        Return the construction of self.
        EXAMPLES:
        sage: A = algebras.Free(QQ, 2)
        # needs sage.combinat sage.modules
        sage: T = A.tensor(A)
        # needs sage.combinat sage.modules
        sage: T.construction()
        (The tensor functorial construction,
         (Free Algebra on 2 generators (None0, None1) over Rational Field,
          Free Algebra on 2 generators (None0, None1) over Rational Field))

tensor_factors()
    Return the tensor factors of this tensor product.
    EXAMPLES:
    sage: # needs sage.modules
    sage: F = CombinatorialFreeModule(ZZ, [1,2])
    sage: F.rename("F")
    sage: G = CombinatorialFreeModule(ZZ, [3,4])
    sage: G.rename("G")
    sage: T = tensor([F, G]); T
    F # G
    sage: T.tensor_factors()
    (F, G)
>>> from sage.all import *
>>> # needs sage.modules

>>> F = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2)])
>>> F.rename("F")

>>> G = CombinatorialFreeModule(ZZ, [Integer(3), Integer(4)])
>>> G.rename("G")

>>> T = tensor([F, G]); T
F # G

T.tensor_factors()
(F, G)

**extra_super_categories()**

EXAMPLES:

```
sage: Modules(ZZ).TensorProducts().extra_super_categories()
[Category of modules over Integer Ring]
sage: Modules(ZZ).TensorProducts().super_categories()
[Category of modules over Integer Ring]
```

**WithBasis**

alias of *ModulesWithBasis*

**additional_structure()**

Return None.

Indeed, the category of modules defines no additional structure: a bimodule morphism between two modules is a module morphism.

See also:

`Category.additional_structure()`

**Todo:** Should this category be a *CategoryWithAxiom*?

EXAMPLES:

```
sage: Modules(ZZ).additional_structure()
```

**super_categories()**

EXAMPLES:

```
sage: Modules(ZZ).super_categories()
[Category of bimodules over Integer Ring on the left
 and Integer Ring on the right]
```

4.118. Modules
Not a bene:

```python
>>> from sage.all import *
>>> Modules(ZZ).super_categories()
[Category of bimodules over Integer Ring on the left
    and Integer Ring on the right]
```

4.119 Modules With Basis

AUTHORS:
- Jason Bandlow and Florent Hivert (2010): Triangular Morphisms
- Christian Stump (2010): Issue #9648 module_morphism’s to a wider class of codomains

```python
class sage.categories.modules_with_basis.ModulesWithBasis(base_category):
    Bases: CategoryWithAxiom_over_base_ring
    The category of modules with a distinguished basis.
    The elements are represented by expanding them in the distinguished basis. The morphisms are not required to respect the distinguished basis.
```

EXAMPLES:

```python
sage: ModulesWithBasis(ZZ)
Category of modules with basis over Integer Ring
sage: ModulesWithBasis(ZZ).super_categories()
[Category of modules over Integer Ring]
```

If the base ring is actually a field, this constructs instead the category of vector spaces with basis:

```python
sage: ModulesWithBasis(QQ)
Category of vector spaces with basis over Rational Field
sage: ModulesWithBasis(QQ).super_categories()
```

(continues on next page)
Let $X$ and $Y$ be two modules with basis. We can build $\text{Hom}(X,Y)$:

```sage
sage: X = CombinatorialFreeModule(QQ, [1,2]); X.rename("X")  # needs sage.modules
sage: Y = CombinatorialFreeModule(QQ, [3,4]); Y.rename("Y")  # needs sage.modules
sage: H = Hom(X, Y); H  # needs sage.modules
Set of Morphisms from X to Y
in Category of finite dimensional vector spaces with basis over Rational Field
```

The simplest morphism is the zero map:

```sage
sage: H.zero()  # todo: move this test into module once we have an example  # needs sage.modules
Generic morphism:
  From: X
  To:  Y
```

which we can apply to elements of $X$:

```sage
sage: x = X.monomial(1) + 3 * X.monomial(2)  # needs sage.modules
sage: H.zero()(x)  # needs sage.modules
0
```
EXAMPLES:

We now construct a more interesting morphism by extending a function by linearity:

```python
>>> phi = H(on_basis=lambda i: Y.monomial(i + 2)); phi
Generic morphism:
  From: X
  To:  Y
sage: phi(x)
```

We can retrieve the function acting on indices of the basis:

```python
>>> f = phi.on_basis()
sage: f(1), f(2)
(B[3], B[4])
```

\( \mathcal{H}om(X, Y) \) has a natural module structure (except for the zero, the operations are not yet implemented though). However since the dimension is not necessarily finite, it is not a module with basis; but see \texttt{FiniteDimensionalModulesWithBasis} and \texttt{GradedModulesWithBasis}:

```python
>>> H in ModulesWithBasis(QQ), H in Modules(QQ)
(False, True)
```
Some more playing around with categories and higher order homsets:

```python
sage: H.category() #...
Category of homsets of finite dimensional modules with basis over Rational Field
sage: Hom(H, H).category() #...
Category of endsets of homsets of finite dimensional modules with basis over Rational Field

Todo: End(X) is an algebra.

Note: This category currently requires an implementation of an element method support. Once Issue #18066 is merged, an implementation of an items method will be required.

class CartesianProducts (category, *args)
    Bases: CartesianProductsCategory
    The category of modules with basis constructed by Cartesian products of modules with basis.
class ParentMethods
    Bases: object
    extra_super_categories()
    EXAMPLES:

sage: ModulesWithBasis(QQ).CartesianProducts().extra_super_categories()
[Category of vector spaces with basis over Rational Field]
sage: ModulesWithBasis(QQ).CartesianProducts().super_categories()
[Category of Cartesian products of modules with basis over Rational Field, Category of vector spaces with basis over Rational Field, Category of Cartesian products of vector spaces over Rational Field]

>>> from sage.all import *

```
**extra_super_categories()**

**EXAMPLES:**

```
sage: ModulesWithBasis(ZZ).DualObjects().extra_super_categories()
[Category of modules over Integer Ring]
sage: ModulesWithBasis(QQ).DualObjects().super_categories()
[Category of duals of vector spaces over Rational Field,
 Category of duals of modules with basis over Rational Field]
```

```python
>>> from sage.all import *
```
Categories Framework, Release 10.4

(continued from previous page)

```
-2
>>> z.coefficient(Partition([Integer(2), Integer(1)]))
-2
>>> z.coefficient([Integer(1), Integer(2)])
Traceback (most recent call last):
...
AssertionError: [1, 2] should be an element of Partitions
>>> z.coefficient(Composition([Integer(2), Integer(1)]))
Traceback (most recent call last):
...
AssertionError: [2, 1] should be an element of Partitions
```

Test that coefficient also works for those parents that do not have an element_class:

```
sage: # needs sage.modules sage.rings.padics
sage: H = pAdicWeightSpace(3)
sage: F = CombinatorialFreeModule(QQ, H)
sage: hasattr(H, "element_class")
False
sage: h = H.an_element()
sage: (2*F.monomial(h)).coefficient(h)
2
```

```
>>> from sage.all import *
>>> # needs sage.modules sage.rings.padics
>>> H = pAdicWeightSpace(Integer(3))
>>> F = CombinatorialFreeModule(QQ, H)
>>> hasattr(H, "element_class")
False
>>> h = H.an_element()
>>> (Integer(2)*F.monomial(h)).coefficient(h)
2
```

coefficients (sort=True)

Return a list of the (non-zero) coefficients appearing on the basis elements in self (in an arbitrary order).

INPUT:
• sort – (default: True) to sort the coefficients based upon the default ordering of the indexing set

See also:
dense_coefficient_list()
```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> B = F.basis()
>>> f = B['a'] - Integer(3)*B['c']
>>> f.coefficients()
[1, -3]
>>> f = B['c'] - Integer(3)*B['a']
>>> f.coefficients()
[-3, 1]

sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
>>> z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])  # needs sage.combinat sage.modules
>>> z.coefficients()  # needs sage.combinat sage.modules
[1, 1, 1, 1]

is_zero()

Return True if and only if self == 0.

EXAMPLES:

sage: # needs sage.modules
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.is_zero()
False
sage: F.zero().is_zero()
True
```

(continues on next page)
False
sage: s(0).is_zero()
True
sage: (s([2,1]) - s([2,1])).is_zero()
True

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> s = SymmetricFunctions(QQ).schur()
>>> s([Integer(2),Integer(1)]).is_zero()
False
>>> s(Integer(0)).is_zero()
True
>>> (s([Integer(2),Integer(1)]) - s([Integer(2),Integer(1)])).is_zero()
True

**leading_coefficient** (*args, **kwds)

Return the leading coefficient of `self`.

This is the coefficient of the term whose corresponding basis element is maximal. Note that this may not be the term which actually appears first when `self` is printed.

If the default term ordering is not what is desired, a comparison key, `key(x,y)`, can be provided.

**EXAMPLES:**

```
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_coefficient()
1
sage: def key(x):
    return -x
sage: x.leading_coefficient(key=key)
3
```

```sage
sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
sage: f.leading_coefficient()  # needs sage.combinat sage.modules
-5
```

```sage
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.monomial(Integer(1)) +
 X.monomial(Integer(2)) + X.monomial(Integer(3))
>>> x.leading_coefficient()
1
>>> def key(x):
    return -x
>>> x.leading_coefficient(key=key)
3

>>> s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
```
leading_item(*args, **kwds)

Return the pair \((k, c)\) where

\[ c \cdot \text{(the basis element indexed by } k) \]

is the leading term of \(self\).

Here 'leading term' means that the corresponding basis element is maximal. Note that this may not be the term which actually appears first when \(self\) is printed.

If the default term ordering is not what is desired, a comparison function, \(key(x)\), can be provided.

EXAMPLES:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)
sage: x.leading_item()
(3, 4)
sage: def key(x): return -x
sage: x.leading_item(key=key)
(1, 3)
sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
sage: f.leading_item()
([3], -5)
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.
>>> monomial(Integer(2)) + Integer(4)*X.monomial(Integer(3))
>>> x.leading_item()
(3, 4)
>>> def key(x): return -x
>>> x.leading_item(key=key)
(1, 3)
>>> s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
>>> f = Integer(2)*s[Integer(1)] + Integer(3)*s[Integer(2),Integer(1)]  # needs sage.combinat sage.modules
>>> f.leading_item()
([3], -5)
```
**leading_monomial**(*args, **kwds*)

Return the leading monomial of `self`.

This is the monomial whose corresponding basis element is maximal. Note that this may not be the term which actually appears first when `self` is printed.

If the default term ordering is not what is desired, a comparison key, `key(x)`, can be provided.

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_monomial()
B[3]
sage: def key(x): return -x
sage: x.leading_monomial(key=key)
B[1]
sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
sage: f.leading_monomial()

s[3]
```

>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.monomial(Integer(2)) + X.monomial(Integer(3))
>>> x.leading_monomial()
B[3]
>>> def key(x): return -x
>>> x.leading_monomial(key=key)
B[1]

```python
>>> s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
>>> f = Integer(2)*s[Integer(1)] + Integer(3)*s[Integer(2),Integer(1)]
>>> f.leading_monomial()  # needs sage.combinat sage.modules
s[3]
```

**leading_support**(*args, **kwds*)

Return the maximal element of the support of `self`.

Note that this may not be the term which actually appears first when `self` is printed.

If the default ordering of the basis elements is not what is desired, a comparison key, `key(x)`, can be provided.

**EXAMPLES:**

```python
```
leading_term(*args, **kwds)

Return the leading term of self.

This is the term whose corresponding basis element is maximal. Note that this may not be the term which actually appears first when self is printed.

If the default term ordering is not what is desired, a comparison key, key(x), can be provided.

EXAMPLES:

sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_term()
B[3]
sage: def key(x): return -x
sage: x.leading_term(key=key)
3*B[1]
length()  
Return the number of basis elements whose coefficients in `self` are nonzero.

EXAMPLES:

```python
sage: # needs sage.modules
sage: s = SymmetricFunctions(QQ).schur()  
       # needs sage.combinat sage.modules
       # needs sage.combinat sage.modules
sage: f.leading_term()  
       # needs sage.combinat sage.modules
-5*s[3]

>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);  
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.  
       monomial(Integer(2)) + X.monomial(Integer(3))
>>> x.leading_term()
B[3]
>>> def key(x): return -x
>>> x.leading_term(key=key)
3*B[1]
```

```python
sage: # needs sage.modules
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])  
       # needs sage.combinat sage.modules
sage: z.length()  
       # needs sage.combinat sage.modules
4
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a','b','c'])
>>> B = F.basis()
>>> f = B['a'] - 3*B['c']
>>> f.length()
2
```

```python
sage: # needs sage.modules
sage: s = SymmetricFunctions(QQ).schur()  
       # needs sage.combinat sage.modules
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])  
       # needs sage.combinat sage.modules
sage: z.length()  
       # needs sage.combinat sage.modules
4
```
map_coefficients \((f, \text{new\_base\_ring}=\text{None})\)

Return the element obtained by applying \(f\) to the non-zero coefficients of \(self\).

If \(f\) is a \textit{sage.categories.map.Map}, then the resulting polynomial will be defined over the codomain of \(f\). Otherwise, the resulting polynomial will be over the same ring as \(self\). Set \text{new\_base\_ring} to override this behaviour.

An error is raised if the coefficients cannot be converted to the new base ring.

INPUT:

- \(f\) – a callable that will be applied to the coefficients of \(self\)
- \text{new\_base\_ring} – (optional) if given, the resulting element will be defined over this ring

EXAMPLES:

```python
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.map_coefficients(lambda x: x + 5)
6*B['a'] + 2*B['c']
```

Killed coefficients are handled properly:

```python
sage: f.map_coefficients(lambda x: 0) # needs sage.modules
0
sage: list(f.map_coefficients(lambda x: 0)) # needs sage.modules
[]
```

```python
>>> from sage.all import *  
>>> f.map_coefficients(lambda x: Integer(0)) # needs sage.modules
0
>>> list(f.map_coefficients(lambda x: Integer(0))) # needs sage.modules
[]
```
We can map into a different base ring:

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: a = 1/2*B['a'] + 3*B['c']; a
1/2*B['a'] + 3/2*B['c']
sage: b = a.map_coefficients(lambda c: 2*c, ZZ); b
B['a'] + 3*B['c']
sage: b.parent()
Free module generated by {'a', 'b', 'c'} over Integer Ring
sage: b.map_coefficients(lambda c: 1/2*c, ZZ)
Traceback (most recent call last):
  ...
TypeError: no conversion of this rational to integer
```

Coefficients are converted to the new base ring after applying the map:

```python
sage: B['a'].map_coefficients(lambda c: 2*c, GF(2))
B['a']
sage: B['a'].map_coefficients(lambda c: GF(2)(c), QQ)
B['a']
```

(continues on next page)
map_item\(f\)

Mapping a function on items.

INPUT:

• \(f\) – a function mapping pairs \((\text{index}, \text{coeff})\) to other such pairs

Return a new element of \(\text{self.parent()}\) obtained by applying the function \(f\) to all items \((\text{index}, \text{coeff})\) of \(\text{self}\).

EXAMPLES:

\[
\begin{align*}
\text{sage: } & B = \text{CombinatorialFreeModule}(\mathbb{Z}, [-1, 0, 1]) \quad \# \text{ needs sage.modules} \\
\text{sage: } & x = B.\text{an_element}(); x \\
\text{sage: } & 2*B[-1] + 2*B[0] + 3*B[1] \\
\text{sage: } & x.\text{map_item}(\lambda i, c: (-i, 2*c)) \\
\text{sage: } & 6*B[-1] + 4*B[0] + 4*B[1]
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} \\
\text{>>> B = CombinatorialFreeModule(ZZ, [-Integer(1), Integer(0), Integer(1)]) \quad \# \text{ needs sage.modules} } \\
\text{>>> x = B.\text{an_element}(); x} \\
\text{>>> 2*B[-1] + 2*B[0] + 3*B[1]} \\
\text{>>> x.\text{map_item}(\lambda i, c: (-i, \text{Integer}(2)*c))} \\
\text{>>> 6*B[-1] + 4*B[0] + 4*B[1]}
\end{align*}
\]

\(f\) needs not be injective:

\[
\begin{align*}
\text{sage: } & x.\text{map_item}(\lambda i, c: (1, \text{Integer}(2)*c)) \quad \# \text{ needs sage.modules} \\
\text{sage: } & 14*B[1] \\
\text{sage: } & s = \text{SymmetricFunctions}(\mathbb{Q}).\text{schur()} \quad \# \text{ needs sage.combinat sage.modules} \\
\text{sage: } & f = \lambda m, c: (m.\text{conjugate}(), \text{2 * c}) \\
\text{sage: } & a = s([2,1]) + s([1,1,1]) \quad \# \text{ needs sage.combinat sage.modules} \\
\text{sage: } & a.\text{map_item}(f) \quad \# \text{ needs sage.combinat sage.modules} \\
\text{sage: } & 2*s[2, 1] + 2*s[3]
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} \\
\text{>>> x.\text{map_item}(\lambda i, c: (\text{Integer}(1), \text{Integer}(2)*c))} \quad \# \text{ needs sage.modules} \\
\text{>>> 14*B[1]} \\
\text{>>> s = SymmetricFunctions(\mathbb{Q}).schur()} \quad \# \text{ needs sage.combinat sage.modules} \\
\text{>>> f = \lambda m, c: (m.\text{conjugate}(), \text{Integer}(2) * c)}
\end{align*}
\]
a = s([Integer(2), Integer(1)]) + s([Integer(1), Integer(1),
  → Integer(1)])  # needs sage.
  → combinat sage.modules
>>> a.map_item(f)  # needs sage.combinat sage.modules
2*s[2, 1] + 2*s[3]

map_support(f)

Mapping a function on the support.

INPUT:

• f – an endofunction on the indices of the free module

Return a new element of self.parent() obtained by applying the function f to all of the objects indexing the basis elements.

EXAMPLES:

sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])  # needs sage.modules
sage: x = B.an_element(); x  # needs sage.modules
2*B[-1] + 2*B[0] + 3*B[1]
sage: x.map_support(lambda i: -i)  # needs sage.modules
3*B[-1] + 2*B[0] + 2*B[1]

f needs not be injective:

sage: x.map_support(lambda i: 1)  # needs sage.modules
7*B[1]
sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
sage: a = s([2, 1]) + 2*s([3, 2])  # needs sage.combinat sage.modules
sage: a.map_support(lambda x: x.conjugate())  # needs sage.combinat sage.modules
s[2, 1] + 2*s[2, 2, 1]

sage: from sage.all import *

... from sage.all import *
... B = CombinatorialFreeModule(ZZ, [-Integer(1), Integer(0),
  → Integer(1)])  # needs sage.modules
... x = B.an_element(); x  # needs sage.modules
2*B[-1] + 2*B[0] + 3*B[1]
... x.map_support(lambda i: -i)  # needs sage.modules
3*B[-1] + 2*B[0] + 2*B[1]

7*B[1]

sage: s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
sage: a = s([2, 1]) + 2*s([3, 2])  # needs sage.combinat sage.modules
sage: a.map_support(lambda x: x.conjugate())  # needs sage.combinat sage.modules
s[2, 1] + 2*s[2, 2, 1]

... from sage.all import *

... x.map_support(lambda i: Integer(1))  # needs sage.modules
7*B[1]  # needs sage.modules

... s = SymmetricFunctions(QQ).schur()  # needs sage.combinat sage.modules
... a = s([2, 1]) + 2*s([3, 2])  # needs sage.combinat sage.modules
... a.map_support(lambda x: x.conjugate())  # needs sage.combinat sage.modules
s[2, 1] + 2*s[2, 2, 1]
map_support_skip_none(f)

Mapping a function on the support.

INPUT:

• f – an endofunction on the indices of the free module

Returns a new element of self.parent() obtained by applying the function f to all of the objects indexing the basis elements.

EXAMPLES:

```python
sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])
# needs sage.modules
sage: x = B.an_element(); x
# needs sage.modules
2*B[-1] + 2*B[0] + 3*B[1]
sage: x.map_support_skip_none(lambda i: -i if i else None)
# needs sage.modules
3*B[-1] + 2*B[1]
```

f needs not be injective:

```python
sage: x.map_support_skip_none(lambda i: 1 if i else None)
# needs sage.modules
5*B[1]
```

monomial_coefficients(copy=True)

Return a dictionary whose keys are indices of basis elements in the support of self and whose values are the corresponding coefficients.

INPUT:

• copy – (default: True) if self is internally represented by a dictionary d, then make a copy of d; if False, then this can cause undesired behavior by mutating d

EXAMPLES:

```python
```

1046  Chapter 4. Individual Categories
```python
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']
```
```python
sage: d = f.monomial_coefficients()
```
```python
sage: d['a']
1
sage: d['c']
3
```
```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> B = F.basis()
>>> f = B['a'] + Integer(3)*B['c']
>>> d = f.monomial_coefficients()
>>> d['a']
1
>>> d['c']
3
```

### support()

Return an iterable of the objects indexing the basis of `self.parent()` whose corresponding coefficients of `self` are non-zero.

This method returns these objects in an arbitrary order.

```python
sage: (F.zero()).support()  # needs sage.modules
[]
```
```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> B = F.basis()
>>> f = B['a'] + Integer(2)*B['c']
>>> f.monomials()
```
```python
B['a'], B['c']
```
```python
>>> (F.zero()).monomials()  # needs sage.modules
[]
```
```python
```
EXAMPLES:

```python
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: sorted(f.support())
['a', 'c']
```

```python
from sage.all import *
from sage.all import # needs sage.modules
F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
B = F.basis()
f = B['a'] - Integer(3)*B['c']
sorted(f.support())
['a', 'c']
```

```python
s = SymmetricFunctions(QQ).schur() # needs sage.combinat sage.modules
z = s([4]) + s([2,1]) + s([1,1,1]) + s([1]) # needs sage.combinat sage.modules
sorted(z.support()) # needs sage.combinat sage.modules
[[1], [1, 1, 1], [2, 1], [4]]
```

```python
from sage.all import *
from sage.all import # needs sage.modules
s = SymmetricFunctions(QQ).schur() # needs sage.combinat sage.modules
z = s([Integer(4)]) + s([Integer(2),Integer(1)]) + s([Integer(1),
Integer(1),Integer(1)]) + s([Integer(1)]) # needs sage.combinat sage.modules
sorted(z.support()) # needs sage.combinat sage.modules
[[1], [1, 1, 1], [2, 1], [4]]
```

**support_of_term()**

Return the support of `self`, where `self` is a monomial (possibly with coefficient).

EXAMPLES:

```python
sage: X = CombinatorialFreeModule(QQ, [1,2,3,4]); X.rename("X") # needs sage.modules
sage: X.monomial(2).support_of_term() # needs sage.modules
2
sage: X.term(3, 2).support_of_term() # needs sage.modules
3
```

```python
from sage.all import *
from sage.all import # needs sage.modules
X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
Integer(4)]); X.rename("X") # needs sage.modules
X.monomial(Integer(2)).support_of_term() # needs sage.modules
2
X.term(Integer(3), Integer(2)).support_of_term() # needs sage.modules
2
```
An exception is raised if \texttt{self} has more than one term:

\begin{verbatim}
>>> from sage.all import *
>>> (X.monomial(Integer(2)) + X.monomial(Integer(3))).support_of_term()  
...
Traceback (most recent call last):
...  
\end{verbatim}

\begin{verbatim}
 from sage.all import *
>>> (X.monomial(Integer(2)) + X.monomial(Integer(3))).support_of_term()  
...
Traceback (most recent call last):
...  
\end{verbatim}

\texttt{tensor} (*\texttt{elements})

Return the tensor product of its arguments, as an element of the tensor product of the parents of those elements.

\textbf{EX^3\P manually:} is this a policy that we want to enforce on all parents?

\texttt{terms}()

Return a list of the (non-zero) terms of \texttt{self} (in an arbitrary order).

\textbf{See also:}

\texttt{monomials}()

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()  
...
\end{verbatim}

\begin{verbatim}
 >>> from sage.all import *
 >>> C = AlgebrasWithBasis(QQ)
 >>> A = C.example()  
 ...
\end{verbatim}
trailing_coefficient(*args, **kwds)

Return the trailing coefficient of self.

This is the coefficient of the monomial whose corresponding basis element is minimal. Note that this
may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key key(x), can be provided.

EXAMPLES:

>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> B = F.basis()
>>> f = B['a'] + Integer(2)*B['c']
>>> f.terms()
[B['a'], 2*B['c']]

>>> s = SymmetricFunctions(QQ).schur()
# needs sage.combinat sage.modules
>>> f = Integer(2)*s[1] + Integer(3)*s[2,1] - Integer(5)*s[3]
# needs sage.combinat sage.modules
>>> f.trailing_coefficient()
2
trailing_item(*args, **kwds)

Return the pair $(c, k)$ where $c\cdot\text{self}.\text{parent()}.\text{monomial}(k)$ is the trailing term of self.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key `key(x)`, can be provided.

EXAMPLES:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = X.monomial(1) + X.monomial(2) + X.monomial(3)
sage: x.trailing_item()
(1, 3)
sage: def key(x):
    return -x
sage: x.trailing_item(key=key)
(3, 1)
```

```python
sage: # needs sage.combinat sage.modules
sage: s = SymmetricFunctions(QQ).schur()
sage: f.trailing_item()
([1], 2)
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.monomial(Integer(2)) + X.monomial(Integer(3))
>>> x.trailing_item()
(1, 3)
>>> def key(x):
    return -x
>>> x.trailing_item(key=key)
(3, 1)
```

```python
sage: # needs sage.combinat sage.modules
sage: f = Integer(2)*s[Integer(1)] + Integer(3)*s[Integer(2),Integer(1)] -
    Integer(5)*s[Integer(3)]
```

trailing_monomial(*args, **kwds)

Return the trailing monomial of self.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key `key(x)`, can be provided.
trailing_support(*args, **kwds)

Return the minimal element of the support of self. Note that this may not be the term which actually appears last when self is printed.

If the default ordering of the basis elements is not what is desired, a comparison key, key(x), can be provided.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.trailing_support() # needs sage.modules
1
```
trailing_term(*args, **kwds)

Return the trailing term of self.

This is the term whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key key(x), can be provided.

EXAMPLES:

```python
sage: x.trailing_support(key=key)  #--
needs sage.modules
3

sage: s = SymmetricFunctions(QQ).schur()  #--
needs sage.combinat sage.modules
needs sage.combinat sage.modules
sage: f.trailing_support()  #--
needs sage.combinat sage.modules
[1]
```
Continued from previous page:

```python
sage: f.trailing_term()  # needs sage.combinat sage.modules
2*s[1]
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]);
>>> X.rename("X")
>>> x = Integer(3)*X.monomial(Integer(1)) + Integer(2)*X.monomial(Integer(2)) + X.monomial(Integer(3))
>>> x.trailing_term()
3*B[1]
```
Category Framework, Release 10.4

(continued from previous page)

```python
sage: x = X.basis()
sage: f = H(lambda x: Y.zero()).on_basis()
sage: f(2)
0
sage: f = lambda i: Y.monomial(i) + 2*Y.monomial(i+1)
sage: g = H(on_basis=f).on_basis()
sage: g(2)
sage: g == f
True
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]);
→ X.rename("X")
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
→ Integer(4)]); Y.rename("Y")
>>> H = Hom(X, Y)
>>> x = X.basis()
>>> f = H(lambda x: Y.zero()).on_basis()
>>> f(Integer(2))
0
>>> f = lambda i: Y.monomial(i) + Integer(2)*Y.monomial(i+Integer(1))
>>> g = H(on_basis=f).on_basis()
>>> g(Integer(2))
>>> g == f
True
```

```python
class ParentMethods
Bases: object

basis()

Return the basis of self.

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
# needs sage.modules
sage: F.basis()
# needs sage.modules
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> # needs sage.modules
>>> F.basis()
>>> # needs sage.modules
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```python
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
# needs sage.combinat sage.groups sage.modules
sage: list(QS3.basis())
# needs sage.combinat sage.groups sage.modules
[[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

4.119. Modules With Basis 1055
>>> from sage.all import *
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
˓→ # needs sage.combinat sage.groups sage.modules
>>> list(QS3.basis())
˓→ # needs sage.combinat sage.groups sage.modules
[[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

cardinality()
Return the cardinality of self.

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.cardinality()
+Infinity
sage: S = SymmetricGroupAlgebra(GF(2), 4)
sage: S.cardinality()
16777216
sage: S.cardinality().factor()
2^24
sage: # needs sage.modules
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.cardinality()
+Infinity
sage: E.<x,y> = ExteriorAlgebra(GF(3))
sage: E.cardinality()
81
sage: s = SymmetricFunctions(GF(2)).s()  # needs sage.combinat sage.modules
˓→ s.cardinality()  # needs sage.combinat sage.modules
+Infinity
sage: M = CombinatorialFreeModule(QQ, [])
sage: M.dimension()
0
sage: M.cardinality()
1

(continues on next page)
%Infinity

>>> E = ExteriorAlgebra(GF(Integer(3)), names=('x', 'y',)); (x, y) = E._first_ngens(2)
>>> E.cardinality()
81

>>> s = SymmetricFunctions(GF(Integer(2))).s()  # needs sage.combinat sage.modules
>>> s.cardinality()  # needs sage.combinat sage.modules
+Infinity

>>> M = CombinatorialFreeModule(QQ, [])
>>> M.dimension()
0
>>> M.cardinality()
1

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: A.<x,y> = algebras.DifferentialWeyl(QQ)  # needs sage.modules
```

```
sage: A.dimension()
+Infinity
```

```
>>> from sage.all import *
```

```
>>> A = algebras.DifferentialWeyl(QQ, names=('x', 'y',)); (x, y) = A._first_ngens(2)  # needs sage.modules
```

```
>>> A.dimension()  # needs sage.modules
+Infinity
```

ehelon_form(elements, row_reduced=False, order=None)

Return a basis in echelon form of the subspace spanned by a finite set of elements.

INPUT:
- elements - a list or finite iterable of elements of self
- row_reduced - (default: False) whether to compute the basis for the row reduced echelon form
- order - (optional) either something that can be converted into a tuple or a key function

OUTPUT:
A list of elements of self whose expressions as vectors form a matrix in echelon form. If base_ring is specified, then the calculation is achieved in this base ring.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: C = CombinatorialFreeModule(R, ZZ, prefix='z')  # needs sage.modules
```

```
sage: z = C.basis()  # needs sage.modules
```
sage: C.echelon_form([z[0] - z[1], 2*z[1] - 2*z[2], z[0] - z[2]])  # needs sage.libs.singular sage.modules
[z[0] - z[2], z[1] - z[2]]

>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> C = CombinatorialFreeModule(R, ZZ, prefix='z')  # needs sage.modules
>>> z = C.basis()  # needs sage.modules
>>> C.echelon_form([z[Integer(0)] - z[Integer(1)],
˓→Integer(2)*z[Integer(1)] - Integer(2)*z[Integer(2)], z[Integer(0)] -
˓→z[Integer(2)]])  # needs sage.libs.singular sage.modules
[z[0] - z[2], z[1] - z[2]]

intersection (other)

Return the intersection of self with other.

EXAMPLES:

sage: X = CombinatorialFreeModule(QQ, range(4)); x = X.basis()
sage: U = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: F = CombinatorialFreeModule(QQ, ['a','b','c','d'])
sage: G = F.submodule([F.basis()['a']])
sage: X.intersection(X) is X
True
sage: X.intersection(U) is U
True
sage: X.intersection(F)
Traceback (most recent call last):
...
TypeError: other must be a submodule
sage: X.intersection(G)
Traceback (most recent call last):
...
ArithmeticError: this module must be the ambient

is_finite ()
Return whether \texttt{self} is finite.
This is true if and only if \texttt{self.basis().keys()} and \texttt{self.base_ring()} are both finite.

**EXAMPLES:**

```python
sage: GroupAlgebra(SymmetricGroup(2), IntegerModRing(10)).is_finite() # needs sage.combinat sage.groups sage.modules
True
sage: GroupAlgebra(SymmetricGroup(2)).is_finite() # needs sage.combinat sage.groups sage.modules
False
sage: GroupAlgebra(AbelianGroup(1), IntegerModRing(10)).is_finite() # needs sage.groups sage.modules
False
```

```python
>>> from sage.all import *
>>> GroupAlgebra(SymmetricGroup(Integer(2)), IntegerModRing(Integer(10))).is_finite() # needs sage.combinat sage.groups sage.modules
True
>>> GroupAlgebra(SymmetricGroup(Integer(2))).is_finite() # needs sage.combinat sage.groups sage.modules
False
>>> GroupAlgebra(AbelianGroup(Integer(1)), IntegerModRing(Integer(10))).is_finite() # needs sage.groups sage.modules
False
```

\texttt{module_morphism}(on\_basis=None, matrix=None, function=None, diagonal=None, triangular=None, unitriangular=False, **keywords)

Construct a module morphism from \texttt{self} to \texttt{codomain}.

Let \texttt{self} be a module \(X\) with a basis indexed by \(I\). This constructs a morphism \(f: X \to Y\) by linearity from a map \(I \to Y\) which is to be its restriction to the basis \((x_i)_{i \in I}\) of \(X\). Some variants are possible too.

**INPUT:**

- \texttt{self} – a parent \(X\) in \texttt{ModulesWithBasis(R)} with basis \(x = (x_i)_{i \in I}\).
- \(I\) – a function \(f\) from \(I\) to \(Y\) as a function \(d\) from \(I\) to \(R\)
- \(f\) – a function \(f\) from \(X\) to \(Y\)
- \(X\) – a matrix of size \(\dim Y \times \dim X\)
- \(Y\) – the zero of the codomain (default: \texttt{codomain.zero()}); can be used (with care) to define affine maps. Only meaningful with \texttt{on\_basis}.
- \(I\) – a non-negative integer specifying which positional argument is used as the input of the function \(f\) (default: 0); this is currently only used with \texttt{on\_basis}.
- \(I\) – a boolean. Only meaningful for a triangular morphism. As a shorthand, one may use \texttt{unitriangular}="lower" for \texttt{triangular}="lower", uni-
triangular=True.

• side = “left” or “right” (default: “left”) Only meaningful for a morphism built from a matrix.

EXAMPLS:

With the on_basis option, this returns a function $g$ obtained by extending $f$ by linearity on the position-th positional argument. For example, for position == 1 and a ternary function $f$, one has:

$$g\left(a, \sum_i \lambda_i x_i, c\right) = \sum_i \lambda_i f(a, i, c).$$

```
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); X.rename("X")
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4]); Y.rename("Y")
sage: def f(i):
    ....:     return Y.monomial(i) + 2*Y.monomial(i+1)
sage: phi = X.module_morphism(f, codomain=Y)
sage: x = X.basis(); y = Y.basis()
sage: phi(x[1] + x[3])
sage: phi
Generic morphism:
From: X
To: Y
```

By default, the category is the first of Modules(R).WithBasis().FiniteDimensional(), Modules(R).WithBasis(), Modules(R), and CommutativeAdditiveMonoids() that contains both the domain and the codomain:

```
sage: phi.category_for()  # needs sage.modules
Category of finite dimensional vector spaces with basis
over Rational Field
```

```
>>> from sage.all import *

>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)])

>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
\,...
Integer(4)]); Y.rename("Y")

>>> def f(i):
...     return Y.monomial(i) + Integer(2)*Y.monomial(i+Integer(1))

>>> phi = X.module_morphism(f, codomain=Y)

>>> x = X.basis(); y = Y.basis()

>>> phi(x[Integer(1)] + x[Integer(3)])

>>> phi
Generic morphism:
From: X
To: Y
```

```
>>> from sage.all import *

>>> phi.category_for()  # needs sage.modules
Category of finite dimensional vector spaces with basis
over Rational Field
```

Chapter 4. Individual Categories
With the `zero` argument, one can define affine morphisms:

```python
sage: def f(i):
    ....:     return Y.monomial(i) + 2*Y.monomial(i+1)
sage: phi = X.module_morphism(f, codomain=Y, zero=10*y[1])
```

```
# needs sage.modules
sage: phi(x[1] + x[3])
```

```
# needs sage.modules
```

In this special case, the default category is `Sets()`:

```python
sage: phi.category_for()
```

```
# needs sage.modules
Category of sets
```

One can construct morphisms with the base ring as codomain:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(ZZ, [1, -1])
sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=ZZ)
sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))
```

```
-1
```

Or more generally any ring admitting a coercion map from the base ring:
On can also define module morphisms between free modules over different base rings; here we implement the natural map from $X = \mathbb{R}^2$ to $Y = \mathbb{C}$:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(RR, ['x', 'y'])
sage: Y = CombinatorialFreeModule(CC, ['z'])
sage: x = X.monomial('x')
sage: y = X.monomial('y')
sage: z = Y.monomial('z')
sage: def on_basis(a):
....:     if a == 'x':
```

(continues on next page)
....:     return CC(1) * z
....:     elif a == 'y':
....:         return CC(I) * z
sage: phi = X.module_morphism(on_basis=on_basis, codomain=Y)
sage: v = 3 * x + 2 * y; v
3.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
With the \texttt{diagonal=d} argument, this constructs the module morphism \( g \) such that

\[
\hat{g}(x_i) = d(i)y_i.
\]

This assumes that the respective bases \( x \) and \( y \) of \( X \) and \( Y \) have the same index set \( I \):

\[
sage: \# \text{ needs sage.modules}
\]
\[
sage: X = CombinatorialFreeModule(ZZ, [1, 2, 3]); X.rename("X")
\]
\[
sage: from sage.arith.misc import factorial
\]
\[
sage: phi = X.module_morphism(diagonal=factorial, codomain=X)
\]
\[
sage: x = X.basis()
\]
\[
sage: phi(x[1]), phi(x[2]), phi(x[3])
\]
\[
(B[1], 2*B[2], 6*B[3])
\]
See also: `sage.modules.with_basis.morphism.DiagonalModuleMorphism`.

With the matrix=m argument, this constructs the module morphism whose matrix in the distinguished basis of X and Y is m:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X")
sage: x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); Y.rename("Y")
sage: y = Y.basis()
sage: m = matrix([[0,1,2], [3,5,0]])
sage: phi = X.module_morphism(matrix=m, codomain=Y)
sage: phi(x[1])
3*B[4]
sage: phi(x[2])
```

See also: `sage.modules.with_basis.morphism.ModuleMorphismFromMatrix`.

With triangular="upper", the constructed module morphism is assumed to be upper triangular; that is its matrix in the distinguished basis of X and Y would be upper triangular with invertible elements on its diagonal. This is used to compute preimages and to invert the morphism:

```
sage: I = list(range(1, 200))
sage: X = CombinatorialFreeModule(QQ, I); X.rename("X"); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, I); Y.rename("Y"); y = Y.basis()
sage: f = Y.sum_of_monomials * divisors
sage: phi = X.module_morphism(f, triangular="upper", codomain=Y)
sage: phi(x[2])
```

(continues on next page)
Since Issue #8678, one can also define a triangular morphism from a function:

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> I = list(range(Integer(1), Integer(200)))
>>> X = CombinatorialFreeModule(QQ, I); x = X.basis()
>>> Y = CombinatorialFreeModule(QQ, I); y = Y.basis()
>>> f = Y.sum_of_monomials * divisors
>>> phi = X.module_morphism(f, triangular="upper", codomain=Y)
>>> phi(x[Integer(2)])
>>> phi(x[Integer(6)])
>>> phi(x[Integer(30)])
>>> phi.preimage(y[Integer(2)])
>>> phi.preimage(y[Integer(6)])
>>> phi.preimage(y[Integer(30)])
>>> (phi**-Integer(1))(y[Integer(30)])
```
\begin{verbatim}
>>> phi = X.module_morphism(function=f, codomain=X,
...                      triangular="upper")
>>> phi(x[Integer(2)] + Integer(3)*x[Integer(4)])
>>> phi.preimage(_)
\end{verbatim}

For details and further optional arguments, see `sage.modules.with_basis.morphism.TriangularModuleMorphism`.

**Warning:** As a temporary measure, until multivariate morphisms are implemented, the constructed morphism is in \texttt{Hom(codomain, domain, category)}. This is only correct for unary functions.

**Todo:**
- Should codomain be \texttt{self} by default in the diagonal, triangular, and matrix cases?
- Support for diagonal morphisms between modules not sharing the same index set

\textbf{monomial}(i)

Return the basis element indexed by \(i\).

**INPUT:**
- \(i\) – an element of the index set

**EXAMPLES:**

\begin{verbatim}
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])

sage: F.monomial('a')
B['a']
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])

>>> F.monomial('a')
B['a']
\end{verbatim}

\(F\text{.monomial}\) is in fact (almost) a map:

\begin{verbatim}
sage: F.monomial

Term map from \{a', 'b', 'c}\) to Free module generated by \{a', 'b', 'c\} over Rational Field
\end{verbatim}

\begin{verbatim}

\textbf{monomial\_or\_zero\_if\_none}(i)
\end{verbatim}
EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # needs sage.modules
sage: F.monomial_or_zero_if_none('a')  # needs sage.modules
B['a']
sage: F.monomial_or_zero_if_none(None)  # needs sage.modules
0
```

```python
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # needs sage.modules
>>> F.monomial_or_zero_if_none('a')  # needs sage.modules
B['a']
```

**quotient_module** *(submodule, check=True, already_echelonized=False, category=None)*

Construct the quotient module self / submodule.

**INPUT:**

- **submodule** – a submodule with basis of self, or something that can be turned into one via self.submodule(submodule)
- **check**, **already_echelonized** – passed down to ModulesWithBasis.

**Warning:** At this point, this only supports quotients by free submodules admitting a basis in unitriangular echelon form. In this case, the quotient is also a free module, with a basis consisting of the retract of a subset of the basis of self.

EXAMPLES:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: Y = X.quotient_module([x[0] - x[1], x[1] - x[2]], ....:
    already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by {2} over Rational Field
sage: y = Y.basis()
sage: y[2]
y[2]
sage: y[2].lift()
x[2]
sage: Y.retract(x[0] + 2*x[1])
3*y[2]
```

```python
sage: # needs sage.modules
sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
sage: x = C.basis()
```

(continues on next page)
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
sage: Y = C.quotient_module(gens)

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix="x")
>>> x = X.basis()
>>> Y = X.quotient_module([x[Integer(0)] - x[Integer(1)], x[Integer(1)] -
˓→x[Integer(2)]], already_echelonized=True)
>>> y = Y.basis()
>>> Y.print_options(prefix=y); Y
Free module generated by {2} over Rational Field
>>> y = Y.basis()
>>> y[Integer(2)]
y[2]
>>> y[Integer(2)].lift()
x[2]
>>> Y.retract(x[Integer(0)] + Integer(2)*x[Integer(1)])
3*y[2]
>>> # needs sage.modules
>>> R = QQ['a, b']; (a, b,) = R._first_ngens(2)
>>> C = CombinatorialFreeModule(R, range(Integer(3)), prefix=x)
>>> x = C.basis()
>>> gens = [x[Integer(0)] - x[Integer(1)], Integer(2)*x[Integer(1)] -
˓→Integer(2)*x[Integer(2)], x[Integer(0)] - x[Integer(2)]]
>>> Y = C.quotient_module(gens)
```

See also:
- Modules.WithBasis.ParentMethods.submodule()
- Rings.ParentMethods.quotient()
- sage.modules.with_basis.subquotient.QuotientModuleWithBasis

`random_element` (n=2)

Return a ‘random’ element of self.

INPUT:
- n – integer (default: 2); number of summands

ALGORITHM:

Return a sum of n terms, each of which is formed by multiplying a random element of the base ring by a random element of the group.

EXAMPLES:

```python
sage: x = DihedralGroup(6).algebra(QQ).random_element()  # needs sage.groups sage.modules
sage: x.parent() is DihedralGroup(6).algebra(QQ)  # needs sage.groups sage.modules
True
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> x = DihedralGroup(Integer(6)).algebra(QQ).random_element()  # needs sage.groups sage.modules
>>> x.parent() is DihedralGroup(Integer(6)).algebra(QQ)  # needs sage.groups sage.modules
True
```
Note, this result can depend on the PRNG state in libgap in a way that depends on which packages are loaded, so we must re-seed GAP to ensure a consistent result for this example:

```python
sage: libgap.set_seed(0)  # needs sage.libs.gap
0
sage: m = SU(2, 13).algebra(QQ).random_element()  # needs sage.groups sage.libs.pari sage.modules
sage: m.parent() is SU(2, 13).algebra(QQ)  # needs sage.groups sage.libs.pari sage.modules
True
sage: p = CombinatorialFreeModule(ZZ, Partitions(4)).random_element()  # needs sage.combinat sage.libs.pari sage.modules
sage: p.parent() is CombinatorialFreeModule(ZZ, Partitions(4))  # needs sage.combinat sage.libs.pari sage.modules
True
```

```python
>>> from sage.all import *
>>> libgap.set_seed(Integer(0))  # needs sage.libs.gap
0
>>> m = SU(Integer(2), Integer(13)).algebra(QQ).random_element()  # needs sage.groups sage.libs.pari sage.modules
>>> m.parent() is SU(Integer(2), Integer(13)).algebra(QQ)  # needs sage.groups sage.libs.pari sage.modules
True
>>> p = CombinatorialFreeModule(ZZ, Partitions(Integer(4))).random_element()  # needs sage.combinat sage.libs.pari sage.modules
>>> p.parent() is CombinatorialFreeModule(ZZ, Partitions(Integer(4)))  # needs sage.combinat sage.libs.pari sage.modules
True
```

submodule (gens, check=True, already_echelonized=False, unitriangular=False, support_order=None, category=None, submodule_class=None, *args, **opts)

The submodule spanned by a finite set of elements.

**INPUT:**

- **gens** – a list or family of elements of self
- **check** – (default: True) whether to verify that the elements of gens are in self
- **already_echelonized** – (default: False) whether the elements of gens are already in (not necessarily reduced) echelon form
- **unitriangular** – (default: False) whether the lift morphism is unitriangular
- **support_order** – (optional) either something that can be converted into a tuple or a key function
- **category** – (optional) the category of the submodule
- **submodule_class** – (optional) the class of the submodule to return

If already_echelonized is False, then the generators are put in reduced echelon form using echelonize(), and reindexed by 0, 1,....
Warning: At this point, this method only works for finite dimensional submodules and if matrices can be echelonized over the base ring.

If in addition `unitriangular` is `True`, then the generators are made such that the coefficients of the pivots are 1, so that lifting map is unitriangular.

The basis of the submodule uses the same index set as the generators, and the lifting map sends \( y_i \) to \( \text{gens[i]} \).

See also:

- ModulesWithBasis.FiniteDimensional.ParentMethods.quotient_module()
- sage.modules.with_basis.subquotient.SubmoduleWithBasis

EXAMPLES:

We construct a submodule of the free \( \mathbb{Q} \)-module generated by \( x_0, x_1, x_2 \). The submodule is spanned by \( y_0 = x_0 - x_1 \) and \( y_1 = x_1 - x_2 \), and its basis elements are indexed by 0 and 1:

```python
sage: # needs sage.modules
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: gens = [x[0] - x[1], x[1] - x[2]]; gens
[\( x_0 - x_1 \), \( x_1 - x_2 \)]
sage: Y = X.submodule(gens, already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by \{0, 1\} over Rational Field
sage: y = Y.basis()
sage: y[1]
y[1]
sage: y[1].lift()  
x[1] - x[2]
sage: Y.retract(x[0] - x[2])
y[0] + y[1]
```

ValueError: \( x[0] \) is not in the image

```
>>> from sage.all import *
>>> # needs sage.modules
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix="x")
>>> x = X.basis()
>>> gens = [x[Integer(0)] - x[Integer(1)], x[Integer(1)] - x[Integer(2)]]; gens
[\( x_0 - x_1 \), \( x_1 - x_2 \)]
>>> Y = X.submodule(gens, already_echelonized=True)
>>> Y.print_options(prefix='y'); Y
Free module generated by \{0, 1\} over Rational Field
>>> y = Y.basis()
>>> y[Integer(1)]
y[1]
```
By using a family to specify a basis of the submodule, we obtain a submodule whose index set coincides with the index set of the family:

```
sage: # needs sage.modules/nsage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: gens = Family({1: x[0] - x[1], 3: x[1] - x[2]}); gens
Finite family {1: x[0] - x[1], 3: x[1] - x[2]}
sage: Y = X.submodule(gens, already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by {1, 3} over Rational Field
sage: y = Y.basis()
sage: y[1]
y[1]
sage: y[1].lift()
x[0] - x[1]
sage: y[3].lift()
x[1] - x[2]
sage: Y.retract(x[0] - x[2])
y[1] + y[3]
sage: Y.retract(x[0])
Traceback (most recent call last):
...   ValueError: x[0] is not in the image
```

It is not necessary that the generators of the submodule form a basis (an explicit basis will be computed):
We now implement by hand the center of the algebra of the symmetric group $S_3$:

```python
sage: # needs sage.combinat sage.groups sage.modules
sage: S3 = SymmetricGroup(3)
sage: S3A = S3.algebra(QQ)
sage: basis = S3A.annihilator_basis(S3A.algebra_generators(),
    ....:     S3A.bracket)
sage: basis
    (((), (1,2,3) + (1,3,2), (2,3) + (1,2) + (1,3))
sage: center = S3A.submodule(basis,
    ....:     category=AlgebrasWithBasis(QQ).Subobjects(),
    ....:     already_echelonized=True)
sage: center
Free module generated by {0, 1, 2} over Rational Field
sage: center in Algebras
True
sage: center.print_options(prefix='c')
sage: c = center.basis()
sage: c[1].lift()
(1,2,3) + (1,3,2)
sage: c[0]^2
(1,2,3) + (1,3,2)
sage: e = 1/6 * (c[0]+c[1]+c[2])
sage: e.is_idempotent()
True
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups sage.modules
>>> S3 = SymmetricGroup(Integer(3))
>>> S3A = S3.algebra(QQ)
>>> basis = S3A.annihilator_basis(S3A.algebra_generators(),
    ....:     S3A.bracket)
```
... S3A.bracket)
>>> basis
(((), (1,2,3) + (1,3,2), (2,3) + (1,2) + (1,3))
>>> center = S3A.submodule(basis,
...   category=AlgebrasWithBasis(QQ).Subobjects(),
...   already_echelonized=True)
>>> center
Free module generated by {0, 1, 2} over Rational Field
>>> center in Algebras
True
>>> center.print_options(prefix='c')
>>> c = center.basis()
>>> c[Integer(1)].lift()
(1,2,3) + (1,3,2)
>>> c[Integer(0)]**Integer(2)
c[0]
>>> e = Integer(1)/Integer(6) * c[Integer(0)] + c[Integer(1)] + c[Integer(2)]
>>> e.is_idempotent()
True

Of course, this center is best constructed using:

```python
sage: center = S3A.center()  # needs sage.combinat sage.groups sage.modules
```

We can also automatically construct a basis such that the lift morphism is (lower) unitriangular:

```python
sage: # needs sage.modules
sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
sage: x = C.basis()
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
sage: Y = C.submodule(gens, unitriangular=True)
sage: Y.lift.matrix()
[ 1 0]
[ 0 1]
[-1 -1]
```

We now construct a (finite-dimensional) submodule of an infinite dimensional free module:
sage: # needs sage.modules
sage: C = CombinatorialFreeModule(QQ, ZZ, prefix='z')
sage: z = C.basis()
sage: gens = [z[0] - z[1], 2*z[1] - 2*z[2], z[0] - z[2]]
sage: Y = C.submodule(gens)
sage: [Y.lift(b) for b in Y.basis()]
[z[0] - z[2], z[1] - z[2]]

>>> from sage.all import *
>>> # needs sage.modules
>>> C = CombinatorialFreeModule(QQ, ZZ, prefix='z')
>>> z = C.basis()
>>> gens = [z[Integer(0)] - z[Integer(1)], Integer(2)*z[Integer(1)] -
Integer(2)*z[Integer(2)], z[Integer(0)] - z[Integer(2)]]
>>> Y = C.submodule(gens)
>>> [Y.lift(b) for b in Y.basis()]
[z[0] - z[2], z[1] - z[2]]

**sum_of_monomials()**

Return the sum of the basis elements with indices in *indices*.

**INPUT:**

• *indices* – a list (or iterable) of indices of basis elements

**EXAMPLES:**

sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # needs sage.modules
sage: F.sum_of_monomials(['a', 'b'])  # needs sage.modules
B['a'] + B['b']
sage: F.sum_of_monomials(['a', 'b', 'a'])  # needs sage.modules
2*B['a'] + B['b']

>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # needs sage.modules
>>> F.sum_of_monomials(['a', 'b'])  # needs sage.modules
B['a'] + B['b']
>>> F.sum_of_monomials(['a', 'b', 'a'])  # needs sage.modules
2*B['a'] + B['b']

F.sum_of_monomials is in fact (almost) a map:

sage: F.sum_of_monomials  # needs sage.modules
A map to Free module generated by {'a', 'b', 'c'} over Rational Field

>>> from sage.all import *
>>> F.sum_of_monomials  # needs sage.modules
A map to Free module generated by {'a', 'b', 'c'} over Rational Field
**sum_of_terms** *(terms)*
Construct a sum of terms of self.

**INPUT:**
- terms — a list (or iterable) of pairs (index, coeff)

**OUTPUT:**
Sum of coeff * B[index] over all (index, coeff) in terms, where B is the basis of self.

**EXAMPLES:**
```
sage: m = matrix([[0,1], [1,1]])  # needs sage.modules
sage: J.<a,b,c> = JordanAlgebra(m) # needs sage.combinat sage.modules
sage: J.sum_of_terms([[0, 2], [2, -3]]) # needs sage.combinat sage.modules
2 + (0, -3)
```

**tensor** *(parents, **kwargs)*
Return the tensor product of the parents.

**EXAMPLES:**
```
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A") # needs sage.combinat sage.modules
sage: A.tensor(A, A) # A # A # A
sage: A.rename(None) # needs sage.combinat sage.modules
```

**term** *(index, coeff=None)*
Construct a term in self.

**INPUT:**
- index — the index of a basis element
- coeff — an element of the coefficient ring (default: one)
OUTPUT:

\[ \text{coeff} \times B[\text{index}], \text{where} \ B \text{ is the basis of self.} \]

EXAMPLES:

```python
sage: m = matrix([[0,1], [1,1]])  # needs sage.modules
sage: J.<a,b,c> = JordanAlgebra(m)  # needs sage.combinat sage.modules
sage: J.term(1, -2)  # needs sage.combinat sage.modules
0 + (-2, 0)
```

```python
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)], [Integer(1),Integer(1)]])  # needs sage.modules
>>> J = JordanAlgebra(m, names=('a', 'b', 'c',)); (a, b, c,) = J._first_
>>> J.term(Integer(1), -Integer(2))  # needs sage.combinat sage.modules
0 + (-2, 0)
```

Design: should this do coercion on the coefficient ring?

**Super**

alias of `SuperModulesWithBasis`

**class TensorProducts** (category, *args)

Bases: `TensorProductsCategory`

The category of modules with basis constructed by tensor product of modules with basis.

**class ElementMethods**

Bases: `object`

Implements operations on elements of tensor products of modules with basis.

**apply_multilinear_morphism** *(f, codomain=None)*

Return the result of applying the morphism induced by f to `self`

**INPUT:**

- `f` – a multilinear morphism from the component modules of the parent tensor product to any module
- `codomain` – the codomain of f (optional)

By the universal property of the tensor product, f induces a linear morphism from `self.parent()` to the target module. Returns the result of applying that morphism to `self`

The codomain is used for optimizations purposes only. If it’s not provided, it’s recovered by calling `f` on the zero input.

**EXAMPLES:**

We start with simple (admittedly not so interesting) examples, with two modules `A` and `B`:

```python
sage: # needs sage.modules
sage: A = CombinatorialFreeModule(ZZ, [1,2], prefix="A")
sage: A.rename("A")
sage: B = CombinatorialFreeModule(ZZ, [3,4], prefix="B")
```

4.119. Modules With Basis
and $f$ the bilinear morphism $(a, b) \mapsto b \otimes a$ from $A \times B$ to $B \otimes A$:

```python
sage: def f(a, b):
    ... return tensor([b, a])
```

Now, calling applying $f$ on $a \otimes b$ returns the same as $f(a, b)$:

```python
sage: f(a, b)  # needs sage.modules
```

$f$ may be a bilinear morphism to any module over the base ring of $A$ and $B$. Here the codomain is $Z$:

```python
sage: def f(a, b):
    .... return sum(a.coefficients(), 0) * sum(b.coefficients(), 0)
sage: f(a, b)  # needs sage.modules
-3
```

(continues on next page)
Mind the 0 in the sums above; otherwise $f$ would not return 0 in $\mathbb{Z}$:

```python
sage: def f(a, b):
    ....:    return sum(a.coefficients()) * sum(b.coefficients())
```

Which would be wrong and break this method:

```python
sage: tensor([a, b]).apply_multilinear_morphism(f)
```

Here we consider an example where the codomain is a module with basis with a different base ring:

```python
sage: C = CombinatorialFreeModule(QQ, [(1,3),(2,4)], prefix="C")
sage: C.rename("C")
sage: def f(a, b):
    ....:    return C.sum_of_terms([(1,3), QQ(a[1]*b[3])],
    ....:               [(2,4), QQ(a[2]*b[4])]))
sage: f(a,b)
C[(1, 3)] - 4*C[(2, 4)]
sage: tensor([a, b]).apply_multilinear_morphism(f)
```

We conclude with a real life application, where we check that the antipode of the Hopf algebra of Symmetric functions on the Schur basis satisfies its defining formula:
We recover the constant term of $x$, as desired.

Todo: Extract a method to linearize a multilinear morphism, and delegate the work there.

class ParentMethods

Bases: object

Implements operations on tensor products of modules with basis.

extra_super_categories()

EXAMPLES:

```sage
e: ModulesWithBasis(QQ).TensorProducts().extra_super_categories()
[Category of vector spaces with basis over Rational Field]
e: ModulesWithBasis(QQ).TensorProducts().super_categories()
[Category of tensor products of modules with basis over Rational Field,
Category of vector spaces with basis over Rational Field,
Category of tensor products of vector spaces over Rational Field]
```

```>>> from sage.all import *
>>> ModulesWithBasis(QQ).extra_super_categories()
[Category of vector spaces with basis over Rational Field]
>>> ModulesWithBasis(QQ).super_categories()
[Category of tensor products of modules with basis over Rational Field,
Category of vector spaces with basis over Rational Field,
Category of tensor products of vector spaces over Rational Field]
```

is_abelian()

Return whether this category is abelian.

This is the case if and only if the base ring is a field.

EXAMPLES:

```sage: ModulesWithBasis(QQ).is_abelian()
True
sage: ModulesWithBasis(ZZ).is_abelian()
False
```

```>>> from sage.all import *
>>> ModulesWithBasis(QQ).is_abelian()
True
>>> ModulesWithBasis(ZZ).is_abelian()
False
```
4.120 Monoid algebras

`sage.categories.monoid_algebras.MonoidAlgebras(base_ring)`

The category of monoid algebras over base_ring.

```python
sage: C = MonoidAlgebras(QQ); C
Category of monoid algebras over Rational Field
```

```python
sage: sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
 Category of semigroup algebras over Rational Field,
 Category of unital magma algebras over Rational Field]
```

This is just an alias for:

```python
sage: C is Monoids().Algebras(QQ)
True
```

4.121 Monoids

```python
class sage.categories.monoids.Monoids(base_category)

Bases: CategoryWithAxiom_singleton

The category of (multiplicative) monoids.

A monoid is a unital semigroup, that is a set endowed with a multiplicative binary operation * which is associative and admits a unit (see Wikipedia article Monoid).

EXAMPLES:
```
```
sage: Monoids().axioms()
frozenset({'Associative', 'Unital'})
sage: Semigroups().Unital()
Category of monoids

sage: Monoids().example()
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> Monoids()
Category of monoids
>>> Monoids().super_categories()
[Category of semigroups, Category of unital magmas]
>>> Monoids().all_super_categories()
[Category of monoids,
 Category of semigroups,
 Category of unital magmas, Category of magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

>>> Monoids().axioms()
frozenset({'Associative', 'Unital'})

class Algebras(category, *args)
    Bases: AlgebrasCategory

class ElementMethods
    Bases: object

    is_central()
    Return whether the element self is central.

    EXAMPLES:

sage: SG4 = SymmetricGroupAlgebra(ZZ,4)  # needs sage.groups sage.modules
sage: SG4(1).is_central()  # needs sage.groups sage.modules
True
sage: SG4(Permutation([1,3,2,4])).is_central()  # needs sage.groups sage.modules
False

sage: A = GroupAlgebras(QQ).example(); A  # needs sage.groups sage.modules
Algebra of Dihedral group of order 8 as a permutation group over Rational Field
sage: sum(A.basis()).is_central()  # needs sage.groups sage.modules
True
```python
>>> from sage.all import *
>>> SG4 = SymmetricGroupAlgebra(ZZ, Integer(4))
˓→ # needs sage.groups sage.modules
>>>
SG4(Integer(1)).is_central()
˓→ # needs sage.groups sage.modules
True
>>>
SG4(Permutation([Integer(1), Integer(3), Integer(2), Integer(4)])).is_central()
˓→ # needs sage.groups sage.modules
False
>>>
A = GroupAlgebras(QQ).example(); A
˓→ # needs sage.groups sage.modules
Algebra of Dihedral group of order 8
as a permutation group over Rational Field
>>>
sum(A.basis()).is_central()
˓→ # needs sage.groups sage.modules
True
```

class ParentMethods

Bases: object

algebra_generators()

Return generators for this algebra.

For a monoid algebra, the algebra generators are built from the monoid generators if available and from the semigroup generators otherwise.

See also:

• Semigroups.Algebras.ParentMethods.algebra_generators()
• MagmaticAlgebras.ParentMethods.algebra_generators().

EXAMPLES:

sage: M = Monoids().example(); M
An example of a monoid:
the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.monoid_generators()
Finite family {'a': a, 'b': b, 'c': c, 'd': d}
sage: M.algebra(ZZ).algebra_generators()
˓→ # needs sage.modules
Finite family {'a': B[a], 'b': B[b], 'c': B[c], 'd': B[d]}

sage: Z12 = Monoids().Finite().example(); Z12
An example of a finite multiplicative monoid:
the integers modulo 12
sage: Z12.monoid_generators()
Traceback (most recent call last):
...
AttributeError: 'IntegerModMonoid_with_category' object has no attribute 'monoid_generators'...
sage: Z12.semgroup_generators()
Family (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
sage: Z12.algebra(QQ).algebra_generators()
˓→ # needs sage.modules
Family (B[0], B[1], B[2], B[3], B[4], B[5], B[6], B[7], B[8], B[9], ...
˓→ B[10], B[11])
```
one_basis()

Return the unit of the monoid, which indexes the unit of this algebra, as per
AlgebrasWithBasis.ParentMethods.one_basis().

EXAMPLES:

```python
sage: # needs sage.modules
sage: A = Monoids().example().algebra(ZZ)
  ...
  sage: A.one()
  B['']
  sage: A(3)
  3*B['']

>>> from sage.all import *
  >>> # needs sage.modules
  >>> A = Monoids().example().algebra(ZZ)
  >>> A.one()
  B['']
  >>> A(Integer(3))
  3*B['']
```

`extra_super_categories()`

The algebra of a monoid is a bialgebra and a monoid.

EXAMPLES:

```python
sage: C = Monoids().Algebras(QQ)
  sage: C.extra_super_categories()
  [Category of bialgebras over Rational Field,
   Category of monoids]
  sage: Monoids().Algebras(QQ).super_categories()
  [Category of bialgebras with basis over Rational Field,
   Category of semigroup algebras over Rational Field,
   Category of unital magma algebras over Rational Field]
  >>> from sage.all import *
  >>> C = Monoids().Algebras(QQ)
  >>> C.extra_super_categories()
  [Category of bialgebras over Rational Field,
   Category of monoids]
  >>> Monoids().Algebras(QQ).super_categories()
  [Category of bialgebras with basis over Rational Field,
   Category of semigroup algebras over Rational Field,
   Category of unital magma algebras over Rational Field]
```

class `CartesianProducts` *(category, *args)*

```
Bases: CartesianProductsCategory
```

The category of monoids constructed as Cartesian products of monoids.

This construction gives the direct product of monoids. See Wikipedia article Direct_product for more information.

class `ElementMethods`

```
Bases: object
```
**multiplicative_order()**

Return the multiplicative order of this element.

**EXAMPLES:**

```python
sage: # needs sage.groups sage.modules
sage: G1 = SymmetricGroup(3)
sage: G2 = SL(2, 3)
sage: G = cartesian_product([G1, G2])
sage: G((G1.gen(0), G2.gen(1))).multiplicative_order()
12
```

```python
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> G1 = SymmetricGroup(Integer(3))
>>> G2 = SL(Integer(2), Integer(3))
>>> G = cartesian_product([G1, G2])
>>> G((G1.gen(Integer(0)), G2.gen(Integer(1)))).multiplicative_order()
12
```

class **ParentMethods**

**Bases:** object

**monoid_generators()**

Return the generators of self.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.groups
sage: M = Monoids.free([1, 2, 3])
sage: N = Monoids.free(['a', 'b'])
sage: C = cartesian_product([M, N])
sage: C.monoid_generators()
Family ((F[1], 1), (F[2], 1), (F[3], 1),
        (1, F['a']), (1, F['b']))
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> M = Monoids.free([Integer(1), Integer(2), Integer(3)])
>>> N = Monoids.free(['a', 'b'])
>>> C = cartesian_product([M, N])
>>> C.monoid_generators()
Family ((F[1], 1), (F[2], 1), (F[3], 1),
        (1, F['a']), (1, F['b']))
```

An example with an infinitely generated group (a better output is needed):

```python
sage: N = Monoids.free(ZZ)  # needs sage.combinat
sage: C = cartesian_product([M, N])  # needs sage.combinat sage.groups
sage: C.monoid_generators()  # needs sage.combinat sage.groups
Lazy family (gen(i))_{i in The Cartesian product of {...}}
```

```python
>>> from sage.all import *
>>> N = Monoids.free(ZZ)  # needs sage.combinat
```

(continues on next page)
A Cartesian product of monoids is endowed with a natural group structure.

**EXAMPLES:**

```python
sage: C = Monoids().CartesianProducts()
sage: C.extra_super_categories()
[Category of monoids]
sage: sorted(C.super_categories(), key=str)
[Category of Cartesian products of semigroups,
 Category of Cartesian products of unital magmas,
 Category of monoids]
```

### class Commutative(base_category)

**Bases:** CategoryWithAxiom_singleton

A monoid $M$ is commutative if $xy = yx$ for all $x, y \in M$.

**static free(index_set=None, names=None, **kwds)**

Return a free abelian monoid on $n$ generators or with the generators indexed by a set $I$.

A free monoid is constructed by specifying either:

- the number of generators and/or the names of the generators, or
- the indexing set for the generators.

**INPUT:**

- `index_set` – (optional) an index set for the generators; if an integer, then this represents $\{0, 1, \ldots, n - 1\}$
- `names` – a string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix

**EXAMPLES:**

```python
sage: Monoids().Commutative().free(index_set=ZZ) #...
Free abelian monoid indexed by Integer Ring
sage: Monoids().Commutative().free(ZZ) #...
Free abelian monoid indexed by Integer Ring
sage: F.<x,y,z> = Monoids().Commutative().free(); F #...
Free abelian monoid indexed by {'x', 'y', 'z'}
```
```python
>>> from sage.all import *
    Monoids().Commutative().free(index_set=ZZ)  # needs sage.combinat
Free abelian monoid indexed by Integer Ring
>>> Monoids().Commutative().free(ZZ)  # needs sage.combinat
Free abelian monoid indexed by Integer Ring
>>> F = Monoids().Commutative().free(names=('x', 'y', 'z',)); (x, y, z,)
    # needs sage.combinat
Free abelian monoid indexed by ('x', 'y', 'z')
```

```python
class ElementMethods
    Bases: object

    inverse()
        Return the multiplicative inverse of self.
        This is an alias for inversion, which can also be invoked by \(-x\) for an element \(x\).
        Nota Bene: Element classes should implement \_\_invert\_\_ only.
        EXAMPLES:

        sage: AA(sqrt(-2)).inverse()  # needs sage.rings.number_field sage.symbolic
        1.414213562373095?

        >>> from sage.all import *
        >>> AA(sqrt(-Integer(2))).inverse()  # needs sage.rings.number_field sage.symbolic
        1.414213562373095?

        is_one()
            Return whether self is the one of the monoid.
            The default implementation is to compare with self.one().

        powers(n)
            Return the list \([x^0, x^1, \ldots, x^{n-1}]\).
            EXAMPLES:

        sage: A = Matrix([[1, 1], [-1, 0]])  # needs sage.modules
        sage: A.powers(6)  # needs sage.modules
        [ [1 0] [ 1 1] [ 0 1] [-1 0] [-1 -1] [ 0 -1]
          [0 1], [-1 0], [-1 -1], [ 0 -1], [ 1 0], [ 1 1] ]
```

```python
>>> from sage.all import *
>>> A = Matrix([[Integer(1), Integer(1)], [-Integer(1), Integer(0)]]); A.powers(Integer(6))  # needs sage.modules
    # needs sage.modules
[
[1 0] [ 1 1] [ 0 1] [-1 0] [-1 -1] [ 0 -1]
[0 1], [-1 0], [-1 -1], [ 0 -1], [ 1 0], [ 1 1]
]  
(continues on next page)```

1088 Chapter 4. Individual Categories
Finite
alias of FiniteMonoids

Inverse
alias of Groups

class ParentMethods
Bases: object

\textbf{prod}(\textit{args})
n-ary product of elements of self.

\textbf{INPUT}:
\begin{itemize}
\item \textit{args} – a list (or iterable) of elements of self
\end{itemize}

Returns the product of the elements in \textit{args}, as an element of self.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: S = Monoids().example()
sage: S.prod([S('a'), S('b')]) \texttt{'ab'}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> S = Monoids().example()
>>> S.prod([S('a'), S('b')]) \texttt{'ab'}
\end{verbatim}

\textbf{semigroup_generators}()

Return the generators of \textit{self} as a semigroup.

The generators of a monoid \textit{M} as a semigroup are the generators of \textit{M} as a monoid and the unit.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: M = Monoids().free([1,2,3]) # needs sage.combinat
tsage: M.semigroup_generators() # needs sage.combinat
Family (1, F[1], F[2], F[3])
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> M = Monoids().free([Integer(1),Integer(2),Integer(3)]) # needs sage.combinat
>>> M.semigroup_generators() # needs sage.combinat
Family (1, F[1], F[2], F[3])
\end{verbatim}

\textbf{submonoid}(\textit{generators, category=\texttt{None}})

Return the multiplicative submonoid generated by \textit{generators}.

\textbf{INPUT}:
\begin{itemize}
\item \textit{generators} – a finite family of elements of \textit{self}, or a list, iterable, ... that can be converted into one (see Family).
\item \textit{category} – a category
\end{itemize}
This is a shorthand for `Semigroups.ParentMethods.subsemigroup()` that specifies that this is a submonoid, and in particular that the unit is `self.one()`.

**EXAMPLES:**

```python
sage: R = IntegerModRing(15)
sage: M = R.submonoid([R(3), R(5)]); M
A submonoid of (Ring of integers modulo 15) with 2 generators
sage: M.list()
[1, 3, 5, 9, 0, 10, 12, 6]
```

Not the presence of the unit, unlike in:

```python
sage: S = R.subsemigroup([R(3), R(5)]); S
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: S.list()
[3, 5, 9, 0, 10, 12, 6]
```

This method is really a shorthand for subsemigroup:

```python
sage: M2 = R.subsemigroup([R(3), R(5)], one=R.one())
sage: M2 is M
True
```

```python
>>> from sage.all import *
>>> R = IntegerModRing(Integer(15))
>>> M = R.submonoid([R(Integer(3)), R(Integer(5))]); M
A submonoid of (Ring of integers modulo 15) with 2 generators
>>> M.list()
[1, 3, 5, 9, 0, 10, 12, 6]
```

```python
>>> from sage.all import *
>>> S = R.subsemigroup([R(Integer(3)), R(Integer(5))]); S
A subsemigroup of (Ring of integers modulo 15) with 2 generators
>>> S.list()
[3, 5, 9, 0, 10, 12, 6]
```

```python
class Subquotients(category, *args)
    Bases: SubquotientsCategory
```
class ParentMethods
    Bases: object
    
    one()

    Returns the multiplicative unit of this monoid, obtained by retracting that of the ambient monoid.

    EXAMPLES:
    >>> from sage.all import *
    >>> S = Monoids().Subquotients().example() # todo: not implemented
    >>> S.one()
    # todo: not implemented

class WithRealizations (category, *args)
    Bases: WithRealizationsCategory
    
    class ParentMethods
        Bases: object
        
        one()

        Return the unit of this monoid.

        This default implementation returns the unit of the realization of self given by a_realization().

        EXAMPLES:
        sage: A = Sets().WithRealizations().example(); A # needs sage.modules
        The subset algebra of {1, 2, 3} over Rational Field
        sage: A.one._module_ # needs sage.modules
        'sage.categories.monoids'
        sage: A.one() # needs sage.modules
        F[{}
        
        >>> from sage.all import *
        >>> A = Sets().WithRealizations().example(); A # needs sage.modules
        The subset algebra of {1, 2, 3} over Rational Field
        >>> A.one._module_ # needs sage.modules
        'sage.categories.monoids'
        >>> A.one() # needs sage.modules
        F[{}

    static free (index_set=None, names=None, **kwds)

    Return a free monoid on \( n \) generators or with the generators indexed by a set \( I \).

    A free monoid is constructed by specifying either:
    • the number of generators and/or the names of the generators
    • the indexing set for the generators
INPUT:

- index_set – (optional) an index set for the generators; if an integer, then this represents \{0, 1, \ldots, n-1\}
- names – a string or list/tuple/iterable of strings (default: ‘x’); the generator names or name prefix

EXAMPLES:

```python
sage: Monoids().free(index_set=ZZ)  # needs sage.combinat
Free monoid indexed by Integer Ring
sage: F.<x,y,z> = Monoids().free(); F  # needs sage.combinat
Free monoid indexed by {'x', 'y', 'z'}
```

```
>>> from sage.all import *
>>> Monoids().free(index_set=ZZ)  # needs sage.combinat
Free monoid indexed by Integer Ring
>>> F = Monoids().free(names=('x', 'y', 'z')); (x, y, z) = F._first_
# needs sage.combinat
Free monoid indexed by {'x', 'y', 'z'}
```

### 4.122 Number fields

```python
class sage.categories.number_fields.NumberFields
    Bases: Category_singleton

The category of number fields.

EXAMPLES:

We create the category of number fields:

```python
sage: C = NumberFields()
sage: C
Category of number fields
```

By definition, it is infinite:

```python
sage: NumberFields().Infinite() is NumberFields()
True
```
Notice that the rational numbers $\mathbb{Q}$ are considered as an object in this category:

```python
sage: RationalField() in C
True
```

However, we can define a degree 1 extension of $\mathbb{Q}$, which is of course also in this category:

```python
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: K = NumberField(x - 1, 'a'); K
Number Field in a with defining polynomial x - 1
sage: K in C
True
```

Number fields all lie in this category, regardless of the name of the variable:

```python
sage: K = NumberField(x^2 + 1, 'a')
```

```
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object
        zeta_function(prec=53, max_imaginary_part=0, max_asymptotic_coeffs=40, algorithm='pari')
        Return the Dedekind zeta function of this number field.
```
Actually, this returns an interface for computing with the Dedekind zeta function $\zeta_F(s)$ of the number field $F$.

**INPUT:**
- prec – optional integer (default 53) bits precision
- max_imaginary_part – optional real number (default 0)
- max_asymptotic_coeffs – optional integer (default 40)
- algorithm – optional (default “pari”) either “gp” or “pari”

**OUTPUT:** The zeta function of this number field.

If algorithm is “gp”, this returns an interface to Tim Dokchitser’s gp script for computing with L-functions.

If algorithm is “pari”, this returns instead an interface to Pari’s own general implementation of L-functions.

**EXAMPLES:**

```python
sage: K.<a> = NumberField(ZZ['x'].0^2 + ZZ['x'].0 - 1) # needs sage.rings.number_field
sage: Z = K.zeta_function(); Z
PARI zeta function associated to Number Field in a with defining polynomial x^2 + x - 1
sage: Z(-1) # needs sage.rings.number_field sage.symbolic
0.033333333333333

sage: x = polygen(QQ, 'x')
sage: L.<a, b, c> = NumberField([x^2 - 5, x^2 + 3, x^2 + 1]) # needs sage.rings.number_field
sage: Z = L.zeta_function() # needs sage.rings.number_field sage.symbolic
sage: Z(5) # needs sage.rings.number_field sage.symbolic
1.00199015670185
```

```python
>>> from sage.all import *

>>> K = NumberField(ZZ['x'].gen(0)**Integer(2) + ZZ['x'].gen(0) - Integer(1), names=(a,)) # needs sage.rings.number_field
>>> Z = K.zeta_function(); Z
PARI zeta function associated to Number Field in a
with defining polynomial x^2 + x - 1
>>> Z(-Integer(1)) # needs sage.rings.number_field sage.symbolic
0.033333333333333

>>> x = polygen(QQ, 'x')

>>> L = NumberField([x**Integer(2) - Integer(5), x**Integer(2) + Integer(3), x**Integer(2) + Integer(1)], names=(a, b, c,)); (a, b, c) = L._first_ngens(3) # needs sage.rings.number_field
>>> Z = L.zeta_function() # needs sage.rings.number_field sage.symbolic
>>> Z(Integer(5)) # needs sage.rings.number_field sage.symbolic
1.00199015670185
```
Using the algorithm “pari”:

```python
sage: K.<a> = NumberField(ZZ['x'].0^2 + ZZ['x'].0 - 1)  # needs sage.rings.number_field
sage: Z = K.zeta_function(algorithm="pari")  # needs sage.rings.number_field sage.symbolic
sage: Z(-1)  # needs sage.rings.number_field sage.symbolic
0.0333333333333333
sage: x = polygen(QQ, 'x')
sage: L.<a, b, c> = NumberField([x^2 - 5, x^2 + 3, x^2 + 1])  # needs sage.rings.number_field
sage: Z = L.zeta_function(algorithm="pari")  # needs sage.rings.number_field sage.symbolic
sage: Z(5)  # needs sage.rings.number_field sage.symbolic
1.00199015670185
```

```
>>> from sage.all import *
>>>
>> from sage.all import *
```
4.123 Objects

class sage.categories.objects.Objects
    Bases: Category_singleton
    The category of all objects the basic category

    EXAMPLES:

    sage: Objects()
    Category of objects
    sage: Objects().super_categories()
    []

    >>> from sage.all import *
    >>> Objects()
    Category of objects
    >>> Objects().super_categories()
    []

class ParentMethods
    Bases: object
    Methods for all category objects

class SubcategoryMethods
    Bases: object

    Endsets()
    Return the category of endsets between objects of this category.

    EXAMPLES:

    sage: Sets().Endsets()
    Category of endsets of sets
    sage: Rings().Endsets()
    Category of endsets of unital magmas and additive unital additive magmas

    >>> from sage.all import *
    >>> Sets().Endsets()
    Category of endsets of sets
    >>> Rings().Endsets()
    Category of endsets of unital magmas and additive unital additive magmas

    See also:
    • Homsets()

    Homsets()
    Return the category of homsets between objects of this category.

    EXAMPLES:

    sage: Sets().Homsets()
    Category of homsets of sets
```python
sage: Rings().Homsets()
Category of homsets of unital magmas and additive unital additive magmas
```

```python
>>> from sage.all import *

```
```python
>>> Sets().Homsets()
Category of homsets of sets
```
```python
>>> Rings().Homsets()
Category of homsets of unital magmas and additive unital additive magmas
```

**Note:** Background

Information, code, documentation, and tests about the category of homsets of a category \(Cs\) should go in the nested class \(Cs.Homsets\). They will then be made available to homsets of any subcategory of \(Cs\).

Assume, for example, that homsets of \(Cs\) are \(Cs\) themselves. This information can be implemented in the method \(Cs.Homsets.extra_super_categories\) to make \(Cs.Homsets()\) a subcategory of \(Cs()\).

Methods about the homsets themselves should go in the nested class \(Cs.Homsets\). ParentMethods.

Methods about the morphisms can go in the nested class \(Cs.Homsets.ElementMethods\). However it’s generally preferable to put them in the nested class \(Cs.MorphismMethods\); indeed they will then apply to morphisms of all subcategories of \(Cs\), and not only full subcategories.

**See also:**

FunctorialConstruction

**Todo:**

- Design a mechanism to specify that an axiom is compatible with taking subsets. Examples: Finite, Associative, Commutative (when meaningful), but not Infinite nor Unital.
- Design a mechanism to specify that, when \(B\) is a subcategory of \(A\), a \(B\)-homset is a subset of the corresponding \(A\) homset. And use it to recover all the relevant axioms from homsets in super categories.
- For instances of redundant code due to this missing feature, see:
  - AdditiveMonoids.Homsets.extra_super_categories()
  - HomsetsCategory.extra_super_categories() (slightly different nature)
  - plus plenty of spots where this is not implemented.

**additional_structure()**

Return None

Indeed, by convention, the category of objects defines no additional structure.

**See also:**

Category.additional_structure()

**EXAMPLES:**

```python
sage: Objects().additional_structure()
```
super_categories()

EXAMPLES:

sage: Objects().super_categories()
[]

4.124 Partially ordered monoids

class sage.categories.partially_ordered_monoids.PartiallyOrderedMonoids
    Bases: Category_singleton

The category of partially ordered monoids, that is partially ordered sets which are also monoids, and such that multiplication preserves the ordering: \( x \leq y \) implies \( x \ast z < y \ast z \) and \( z \ast x < z \ast y \).

See Wikipedia article Ordered_monoid

EXAMPLES:

sage: PartiallyOrderedMonoids()
Category of partially ordered monoids
sage: PartiallyOrderedMonoids().super_categories()
[Category of posets, Category of monoids]

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

super_categories()

EXAMPLES:

sage: PartiallyOrderedMonoids().super_categories()
[Category of posets, Category of monoids]
4.125 Permutation groups

class sage.categories.permutation_groups.PermutationGroups
    Bases: Category

The category of permutation groups.

A permutation group is a group whose elements are concretely represented by permutations of some set. In other words, the group comes endowed with a distinguished action on some set.

This distinguished action should be preserved by permutation group morphisms. For details, see Wikipedia article Permutation_group#Permutation_isomorphic_groups.

Todo: shall we accept only permutations with finite support or not?

EXAMPLES:

sage: PermutationGroups()
Category of permutation groups
sage: PermutationGroups().super_categories()
[Category of groups]

>>> from sage.all import *

>>> PermutationGroups()
Category of permutation groups

>>> PermutationGroups().super_categories()
[Category of groups]

The category of permutation groups defines additional structure that should be preserved by morphisms, namely the distinguished action:

sage: PermutationGroups().additional_structure()
Category of permutation groups

>>> from sage.all import *

>>> PermutationGroups().additional_structure()
Category of permutation groups

Finite

alias of FinitePermutationGroups

super_categories()

Return a list of the immediate super categories of self.

EXAMPLES:

sage: PermutationGroups().super_categories()
[Category of groups]

>>> from sage.all import *

>>> PermutationGroups().super_categories()
[Category of groups]
4.126 Pointed sets

class sage.categories.pointed_sets.PointedSets

Bases: Category_singleton

The category of pointed sets.

EXAMPLES:

sage: PointedSets()
Category of pointed sets

>>> from sage.all import *
>>> PointedSets()
Category of pointed sets

super_categories()

EXAMPLES:

sage: PointedSets().super_categories()
[Category of sets]

>>> from sage.all import *
>>> PointedSets().super_categories()
[Category of sets]

4.127 Polyhedral subsets of free ZZ, QQ or RR-modules.

class sage.categories.polyhedra.PolyhedralSets(R)

Bases: Category_over_base_ring

The category of polyhedra over a ring.

EXAMPLES:

We create the category of polyhedra over QQ:

sage: PolyhedralSets(QQ)
Category of polyhedral sets over Rational Field

>>> from sage.all import *
>>> PolyhedralSets(QQ)
Category of polyhedral sets over Rational Field

super_categories()

EXAMPLES:

sage: PolyhedralSets(QQ).super_categories()
[Category of commutative magmas, Category of additive monoids]

>>> from sage.all import *
>>> PolyhedralSets(QQ).super_categories()
[Category of commutative magmas, Category of additive monoids]
4.128 Posets

```python
class sage.categories.posets.Posets
    Bases: Category

    The category of posets i.e. sets with a partial order structure.

    EXAMPLES:

    sage: Posets()
    Category of posets
    sage: Posets().super_categories()
    [Category of sets]
    sage: P = Posets().example(); P
    An example of a poset: sets ordered by inclusion

    The partial order is implemented by the mandatory method le():

    sage: x = P(Set([1,3])); y = P(Set([1,2,3]))
    sage: x, y
    ({1, 3}, {1, 2, 3})
    sage: P.le(x, y)
    True
    sage: P.le(x, x)
    True
    sage: P.le(y, x)
    False

    >>> from sage.all import *
    >>> x = P(Set([Integer(1),Integer(3)])); y = P(Set([Integer(1),Integer(2),
    ... Integer(3)]))
    >>> x, y
    ({1, 3}, {1, 2, 3})
    >>> P.le(x, y)
    True
    >>> P.le(x, x)
    True
    >>> P.le(y, x)
    False

    The other comparison methods are called lt(), ge(), gt(), following Python's naming convention in operator. Default implementations are provided:

    sage: P.lt(x, x)
    False
    sage: P.ge(y, x)
    True
```
Unless the poset is a facade (see \texttt{Sets.Facade}), one can compare directly its elements using the usual Python operators:

```python
sage: D = Poset((divisors(30), attrcall("divides")), facade = False)
sage: D(3) <= D(6)  # True
sage: D(3) <= D(3)  # True
sage: D(3) <= D(5)  # False
sage: D(3) < D(3)   # False
sage: D(10) >= D(5)  # True
```

At this point, this has to be implemented by hand. Once Issue \#10130 will be resolved, this will be automatically provided by this category:

```python
sage: # not implemented
sage: x < y  # True
sage: x < x  # False
sage: x <= x  # True
sage: y >= x  # True
```

See also:

`Poset()`, `FinitePosets`, `LatticePosets`

class ElementMethods

Bases: object

Finite

alias of `FinitePosets`

class ParentMethods

Bases: object

CartesianProduct

alias of `CartesianProductPoset`

directed_subset (elements, direction)

Return the order filter or the order ideal generated by a list of elements.

If `direction` is ‘up’, the order filter (upper set) is being returned.

If `direction` is ‘down’, the order ideal (lower set) is being returned.

INPUT:

• `elements` – a list of elements.
• `direction` – ‘up’ or ‘down’.

EXAMPLES:

```python
sage: B = posets.BooleanLattice(4)
sage: B.directed_subset([3, 8], 'up')
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: B.directed_subset([7, 10], 'down')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

```python
>>> from sage.all import *

>>> B = posets.BooleanLattice(Integer(4))

>>> B.directed_subset([Integer(3), Integer(8)], 'up')
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]

>>> B.directed_subset([Integer(7), Integer(10)], 'down')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

ge (x, y)

Return whether $x \geq y$ in the poset `self`.

INPUT:

• `x, y` – elements of `self`.

This default implementation delegates the work to `le()`.

EXAMPLES:

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.ge(6, 3)
True
sage: D.ge(3, 3)
True
sage: D.ge(3, 5)
False
```

4.128. Posets
gt \,(x, y) 
Return whether $x > y$ in the poset self.

INPUT:
• $x, y$ – elements of self.
This default implementation delegates the work to $it()$.

EXAMPLES:

```sage
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.gt( 3, 6 )
False
sage: D.gt( 3, 3 )
False
sage: D.gt( 3, 5 )
False
```

is_antichain_of_poset \,(o) 
Return whether an iterable $o$ is an antichain of self.

INPUT:
• $o$ – an iterable (e. g., list, set, or tuple) containing some elements of self

OUTPUT:
True if the subset of self consisting of the entries of $o$ is an antichain of self, and False otherwise.

EXAMPLES:

```sage
sage: P = Poset((divisors(12), attrcall("divides")),
....: facade=True, linear_extension=True)
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_antichain_of_poset([1, 3])
False
sage: P.is_antichain_of_poset([3, 1])
False
sage: P.is_antichain_of_poset([1, 1, 3])
False
sage: P.is_antichain_of_poset([])
True
```
sage: P.is_antichain_of_poset([1])
True
sage: P.is_antichain_of_poset([1, 1])
True
sage: P.is_antichain_of_poset([3, 4])
True
sage: P.is_antichain_of_poset([3, 4, 12])
False
sage: P.is_antichain_of_poset([6, 4])
True
sage: P.is_antichain_of_poset(i for i in divisors(12)
....: if (2 < i and i < 6))
True
sage: P.is_antichain_of_poset(i for i in divisors(12)
....: if (2 <= i and i < 6))
False
sage: Q = Poset({2: [3, 1], 3: [4], 1: [4]})
sage: Q.is_antichain_of_poset((1, 2))
False
sage: Q.is_antichain_of_poset((2, 4))
False
sage: Q.is_antichain_of_poset((4, 2))
False
sage: Q.is_antichain_of_poset((2, 2))
True
sage: Q.is_antichain_of_poset((3, 4))
True
sage: Q.is_antichain_of_poset((3, 1))
True
sage: Q.is_antichain_of_poset((1, ))
True
sage: Q.is_antichain_of_poset(()
True

>>> from sage.all import *

>>> P = Poset((divisors(Integer(12)), attrcall("divides")),
... facade=True, linear_extension=True)

>>> sorted(P.list())
[1, 2, 3, 4, 6, 12]
An infinite poset:

```python
sage: from sage.categories.examples.posets import FiniteSetsOrderedByInclusion
sage: R = FiniteSetsOrderedByInclusion()
sage: R.is_antichain_of_poset([R(set([Integer(3), Integer(1), Integer(2)])), ...
R(set([Integer(1), Integer(4)]), R(set([Integer(4), Integer(5)]))])
True
sage: R.is_antichain_of_poset([R(set([Integer(3), Integer(1), Integer(2), ...
Integer(4)])), ...
R(set([Integer(1), Integer(4)]), R(set([Integer(4), Integer(5)]))])
False
```

`is_chain_of_poset(o, ordered=False)`

Return whether an iterable `o` is a chain of `self`, including a check for `o` being ordered from smallest to largest element if the keyword `ordered` is set to `True`. 

```python
>>> from sage.all import *
>>> from sage.categories.examples.posets import FiniteSetsOrderedByInclusion
>>> R = FiniteSetsOrderedByInclusion()
>>> R.is_antichain_of_poset([R(set([Integer(3), Integer(1), Integer(2)])), ...
R(set([Integer(1), Integer(4)]), R(set([Integer(4), Integer(5)]))])
True
>>> R.is_antichain_of_poset([R(set([Integer(3), Integer(1), Integer(2), Integer(4)])), ...
R(set([Integer(1), Integer(4)]), R(set([Integer(4), Integer(5)]))])
False
```
INPUT:
• \textit{o} – an iterable (e. g., list, set, or tuple) containing some elements of \texttt{self}
• \texttt{ordered} – a Boolean (default: \texttt{False}) which decides whether the notion of a chain includes being ordered

OUTPUT:

If \texttt{ordered} is set to \texttt{False}, the truth value of the following assertion is returned: The subset of \texttt{self} formed by the elements of \texttt{o} is a chain in \texttt{self}.

If \texttt{ordered} is set to \texttt{True}, the truth value of the following assertion is returned: Every element of the list \texttt{o} is (strictly!) smaller than its successor in \texttt{self}. (This makes no sense if \texttt{ordered} is a set.)

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), ...
               facade=\texttt{True}, linear_extension=\texttt{True})
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_chain_of_poset([1, 3])
True
sage: P.is_chain_of_poset([3, 1])
True
sage: P.is_chain_of_poset([1, 3], ordered=\texttt{True})
False
sage: P.is_chain_of_poset([3, 1, ordered=\texttt{True})
False
sage: P.is_chain_of_poset([])
True
sage: P.is_chain_of_poset([], ordered=\texttt{True})
True
sage: P.is_chain_of_poset((2, 12, 6))
True
sage: P.is_chain_of_poset((2, 6), ordered=\texttt{True})
True
sage: P.is_chain_of_poset((2, 12, 6), ordered=\texttt{True})
False
sage: P.is_chain_of_poset((2, 12, 6, 3))
False
sage: P.is_chain_of_poset((2, 3))
False
sage: Q = Poset({2: [3, 1], 3: [4], 1: [4]})
sage: Q.is_chain_of_poset([1, 2], ordered=\texttt{True})
False
sage: Q.is_chain_of_poset([1, 2])
True
sage: Q.is_chain_of_poset([2, 1], ordered=\texttt{True})
True
sage: Q.is_chain_of_poset([2, 1, 1], ordered=\texttt{True})
False
sage: Q.is_chain_of_poset([3])
True
sage: Q.is_chain_of_poset([4, 2, 3])
True
sage: Q.is_chain_of_poset([4, 2, 3], ordered=\texttt{True})
False
sage: Q.is_chain_of_poset([2, 3, 4], ordered=\texttt{True})
True
```
>>> from sage.all import *  
>>> P = Poset((divisors(Integer(12)), attrcall("divides")),  
... facetd=True, linear_extension=True)  
>>> sorted(P.list())  
[1, 2, 3, 4, 6, 12]  
>>> P.is_chain_of_poset([Integer(1), Integer(3)])  
True  
>>> P.is_chain_of_poset([Integer(3), Integer(1)])  
True  
>>> P.is_chain_of_poset([Integer(1), Integer(3)], ordered=True)  
True  
>>> P.is_chain_of_poset([Integer(3), Integer(1)], ordered=True)  
False  
>>> P.is_chain_of_poset([])  
True  
>>> P.is_chain_of_poset([], ordered=True)  
True  
>>> P.is_chain_of_poset((Integer(2), Integer(12), Integer(6)))  
True  
>>> P.is_chain_of_poset((Integer(2), Integer(6), Integer(12)),  
... Ordered=True)  
True  
>>> P.is_chain_of_poset((Integer(2), Integer(12), Integer(6)),  
... Ordered=True)  
False  
>>> P.is_chain_of_poset((Integer(2), Integer(12), Integer(6), Integer(3)))  
False  
>>> P.is_chain_of_poset((Integer(2), Integer(3)))  
False  
>>> Q = Poset({Integer(2): [Integer(3), Integer(1)], Integer(3):  
... [Integer(4)], Integer(1): Integer(4)[]})  
>>> Q.is_chain_of_poset([Integer(1), Integer(2)], ordered=True)  
False  
>>> Q.is_chain_of_poset([Integer(1), Integer(2)])  
True  
>>> Q.is_chain_of_poset([Integer(2), Integer(1)], ordered=True)  
True  
>>> Q.is_chain_of_poset([Integer(2), Integer(1), Integer(1)],  
... ordered=True)  
False  
>>> Q.is_chain_of_poset([Integer(3)])  
True  
>>> Q.is_chain_of_poset([Integer(4), Integer(2), Integer(3)])  
True  
>>> Q.is_chain_of_poset([Integer(4), Integer(2), Integer(3)],  
... ordered=True)  
False  
>>> Q.is_chain_of_poset([Integer(2), Integer(3), Integer(4)],  
... ordered=True)  
True

Examples with infinite posets:

```python
sage: from sage.categories.examples.posets import ...  
... FiniteSetsOrderedByInclusion
sage: R = FiniteSetsOrderedByInclusion()
```
(continues on next page)


```python
sage: R.is_chain_of_poset([R(set([3, 1, 2]))],
....:     R(set([1, 4]))),
....:     R(set([4, 5]))])
False
sage: R.is_chain_of_poset([R(set([3, 1, 2]))],
....:     R(set([1, 2])),
....:     R(set([1])), ordered=True)
False
sage: R.is_chain_of_poset([R(set([3, 1, 2]))],
....:     R(set([1, 2])), R(set([1])))
True
```

```python
sage: from sage.categories.examples.posets import...
˓→PositiveIntegersOrderedByDivisibilityFacade
sage: T = PositiveIntegersOrderedByDivisibilityFacade()
sage: T.is_chain_of_poset((T(3), T(6), T(3)))
False
sage: T.is_chain_of_poset((T(3), T(6), T(3)), ordered=True)
False
sage: T.is_chain_of_poset((T(3), T(3), T(6)))
True
sage: T.is_chain_of_poset((T(3), T(3), T(6)), ordered=True)
False
sage: T.is_chain_of_poset((T(3), T(6)), ordered=True)
True
sage: T.is_chain_of_poset((), ordered=True)
True
sage: T.is_chain_of_poset((T(q) for q in divisors(27)))
True
sage: T.is_chain_of_poset((T(q) for q in divisors(18)))
False
```

```python
>>> from sage.all import *
>>> from sage.categories.examples.posets import...
˓→FiniteSetsOrderedByInclusion
>>> R = FiniteSetsOrderedByInclusion()
>>> R.is_chain_of_poset([R(set([Integer(3), Integer(1), Integer(2)]))],
˓→R(set([Integer(1), Integer(4)])),
˓→R(set([Integer(4), Integer(5)])))
False
>>> R.is_chain_of_poset([R(set([Integer(3), Integer(1), Integer(2)]))],
˓→R(set([Integer(1), Integer(2)])),
˓→R(set([Integer(1)])), ordered=True)
False
>>> R.is_chain_of_poset([R(set([Integer(3), Integer(1), Integer(2)]))],
˓→R(set([Integer(1), Integer(2)])), ...
˓→R(set([Integer(1)])))
True
```

```python
>>> from sage.categories.examples.posets import...
˓→PositiveIntegersOrderedByDivisibilityFacade
>>> T = PositiveIntegersOrderedByDivisibilityFacade()
```

(continues on next page)
is_order_filter(o)

Return whether o is an order filter of self, assuming self has no infinite ascending path.

INPUT:
• o—a list (or set, or tuple) containing some elements of self

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")),
....:     facade=True, linear_extension=True)
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_order_filter([4, 12])
True
sage: P.is_order_filter([])
True
sage: P.is_order_filter({3, 4, 12})
False
sage: P.is_order_filter({3, 6, 12})
True
```
**is_order_ideal** 

Return whether $o$ is an order ideal of `self`, assuming `self` has no infinite descending path.

**INPUT:**
- $o$ - a list (or set, or tuple) containing some elements of `self`

**EXAMPLES:**
```
sage: P = Poset((divisors(12), attrcall("divides")),
             facade=True, linear_extension=True)
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_order_ideal([1, 3])
True
sage: P.is_order_ideal([])
True
sage: P.is_order_ideal({1, 3})
True
sage: P.is_order_ideal([1, 3, 4])
False
```

**le**

Return whether $x \leq y$ in the poset `self`.

**INPUT:**
- $x, y$ - elements of `self`

**EXAMPLES:**
```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.le(3, 6)
True
sage: D.le(3, 3)
True
sage: D.le(3, 5)
False
```

```
lower_covers (x)

Return the lower covers of x, that is, the elements y such that y < x and there exists no z such that y < z < x.

EXAMPLES:

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lower_covers(15)
[3, 5]
```

lt (x, y)

Return whether x < y in the poset self.

INPUT:

• x, y – elements of self.

This default implementation delegates the work to le().

EXAMPLES:

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lt(3, 6)
True
sage: D.lt(3, 3)
False
sage: D.lt(3, 5)
False
```

order_filter (elements)

Return the order filter generated by a list of elements.

A subset I of a poset is said to be an order filter if, for any x in I and y such that y ≥ x, then y is in I. This is also called the upper set generated by these elements.

EXAMPLES:

```python
sage: B = posets.BooleanLattice(4)
sage: B.order_filter([3, 8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```
order_ideal (elements)

Return the order ideal in self generated by the elements of an iterable elements.

A subset I of a poset is said to be an order ideal if, for any x in I and y such that y ≤ x, then y is in I.

This is also called the lower set generated by these elements.

EXAMPLES:

```python
sage: B = posets.BooleanLattice(4)
sage: B.order_ideal([7,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

order_ideal_toggle (I, v)

Return the result of toggling the element v in the order ideal I.

If v is an element of a poset P, then toggling the element v is an automorphism of the set J(P) of all order ideals of P. It is defined as follows: If I is an order ideal of P, then the image of I under toggling the element v is

• the set I ∪ {v}, if v ∉ I but every element of P smaller than v is in I;
• the set I \ {v}, if v ∈ I but no element of P greater than v is in I;
• I otherwise.

This image always is an order ideal of P.

EXAMPLES:

```python
sage: P = Poset({1: [2,3], 2: [4], 3: []})
sage: I = Set({1, 2})
sage: I in P.order_ideals_lattice()  # needs sage.modules
True
sage: P.order_ideal_toggle(I, 1)
{1, 2}
sage: P.order_ideal_toggle(I, 2)
{1}
sage: P.order_ideal_toggle(I, 3)
{1, 2, 3}
sage: P.order_ideal_toggle(I, 4)
{1, 2, 4}
sage: P4 = Posets(4)
sage: all(all(all(P.order_ideal_toggle(P.order_ideal_toggle(I, i), i) == I)) for i in range(4)) for I in P.order_ideals_lattice(facade=True)) for P in P4)
True
```
order_ideal_toggles (I, vs)

Return the result of toggling the elements of the list (or iterable) vs (one by one, from left to right) in the order ideal I.

See order_ideal_toggle() for a definition of toggling.

EXAMPLES:

sage: P = Poset({1: [2, 3], 2: [4], 3: []})
sage: I = Set({1, 2})
sage: P.order_ideal_toggles(I, [1, 2, 3, 4])
{1, 3}
sage: P.order_ideal_toggles(I, (1, 2, 3, 4))
{1, 3}

principal_lower_set (x)

Return the order ideal generated by an element x.

This is also called the lower set generated by this element.

EXAMPLES:

sage: B = posets.BooleanLattice(4)
sage: B.principal_order_ideal(6)
[0, 2, 4, 6]
principal_order_filter \( (x) \)
Return the order filter generated by an element \( x \).
This is also called the upper set generated by this element.

EXAMPLES:
```
sage: B = posets.BooleanLattice(4)
sage: B.principal_order_filter(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

principal_order_ideal \( (x) \)
Return the order ideal generated by an element \( x \).
This is also called the lower set generated by this element.

EXAMPLES:
```
sage: B = posets.BooleanLattice(4)
sage: B.principal_order_ideal(6)
[0, 2, 4, 6]
```

principal_upper_set \( (x) \)
Return the order filter generated by an element \( x \).
This is also called the upper set generated by this element.

EXAMPLES:
```
sage: B = posets.BooleanLattice(4)
sage: B.principal_upper_set(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

upper_covers \( (x) \)
Return the upper covers of \( x \), that is, the elements \( y \) such that \( x < y \) and there exists no \( z \) such that \( x < z < y \).

EXAMPLES:
example (choice=None)

Return examples of objects of Posets(), as per Category.example().

EXAMPLES:

```python
sage: Posets().example()
An example of a poset: sets ordered by inclusion
sage: Posets().example("facade")
An example of a facade poset:
the positive integers ordered by divisibility
```

super_categories()

Return a list of the (immediate) super categories of self, as per Category.super_categories().

EXAMPLES:

```python
sage: Posets().super_categories()
[Category of sets]
```

4.129 Principal ideal domains

class sage.categories.principal_ideal_domains.PrincipalIdealDomains

Bases: Category_singleton

The category of (constructive) principal ideal domains

By constructive, we mean that a single generator can be constructively found for any ideal given by a finite set of generators. Note that this constructive definition only implies that finitely generated ideals are principal. It is not clear what we would mean by an infinitely generated ideal.

EXAMPLES:
### sage: PrincipalIdealDomains()
Category of principal ideal domains

### sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]

```python
>>> from sage.all import *
```

```python
>>>
```

```python
PrincipalIdealDomains()
Category of principal ideal domains
```

```python
PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]
```

See also [Wikipedia article Principal ideal domain](https://en.wikipedia.org/wiki/Principal_ideal_domain)

#### class ElementMethods

**Bases:** object

#### class ParentMethods

**Bases:** object

```python
class_group()
```

Return the trivial group, since the class group of a PID is trivial.

**EXAMPLES:**

```python
sage: QQ.class_group()  # needs sage.groups
Trivial Abelian group
```

```python
>>> from sage.all import *
```

```python
>>>
```

```python
QQ.class_group()  # needs sage.groups
Trivial Abelian group
```

```python
content (x, y, coerce=True)
```

Return the content of $x$ and $y$.

This is the unique element $c$ of self such that $x/c$ and $y/c$ are coprime and integral.

**EXAMPLES:**

```python
sage: QQ.content(ZZ(42), ZZ(48)); type(QQ.content(ZZ(42), ZZ(48)))
6
<class 'sage.rings.rational.Rational'>
```

```python
sage: QQ.content(1/2, 1/3)
1/6
```

```python
sage: factor(1/2); factor(1/3); factor(1/6)
2^-1
3^-1
2^-1 * 3^-1
```

```python
sage: a = (2*3)/(7*11); b = (13*17)/(19*23)
```

```python
sage: factor(a); factor(b); factor(QQ.content(a,b))
2 * 3 * 7^-1 * 11^-1
13 * 17 * 19^-1 * 23^-1
7^-1 * 11^-1 * 19^-1 * 23^-1
```

```python
>>> from sage.all import *
```

```python
>>>
```

```python
QQ.content(ZZ(Integer(42)), ZZ(Integer(48))); type(QQ.
```

(continues on next page)
sage: c = (Integer(2)*Integer(3))/(Integer(7)*Integer(11)); d = (Integer(13)*Integer(17))/(Integer(7)*Integer(19)*Integer(23))
sage: factor(c); factor(d); factor(QQ.content(c,d))
2 * 3 * 7^-1 * 11^-1
7^-1 * 13 * 17 * 19^-1 * 23^-1
7^-1 * 11^-1 * 19^-1 * 23^-1
sage: e = (2*3)/(7*11); f = (13*17)/(7^3*19*23)
sage: factor(e); factor(f); factor(QQ.content(e,f))
2 * 3 * 7^-1 * 11^-1
7^-3 * 13 * 17 * 19^-1 * 23^-1
7^-3 * 11^-1 * 19^-1 * 23^-1

Note the change to the second entry:

sage: c = (2*3)/(7*11); d = (13*17)/(7*19*23)
sage: factor(c); factor(d); factor(QQ.content(c,d))
2 * 3 * 7^-1 * 11^-1
7^-1 * 13 * 17 * 19^-1 * 23^-1
7^-1 * 11^-1 * 19^-1 * 23^-1
sage: e = (2*3)/(7*11); f = (13*17)/(7^3*19*23)
sage: factor(e); factor(f); factor(QQ.content(e,f))
2 * 3 * 7^-1 * 11^-1
7^-3 * 13 * 17 * 19^-1 * 23^-1
7^-3 * 11^-1 * 19^-1 * 23^-1

gcd(x, y, coerce=True)

Return the greatest common divisor of x and y, as elements of self.

EXAMPLES:

The integers are a principal ideal domain and hence a GCD domain:

sage: ZZ.gcd(42, 48)
6
sage: 42.factor(); 48.factor()
2 * 3 * 7
2^4 * 3
In a field, any nonzero element is a GCD of any nonempty set of nonzero elements. In previous versions, Sage used to return 1 in the case of the rational field. However, since Issue #10771, the rational field is considered as the fraction field of the integer ring. For the fraction field of an integral domain that provides both GCD and LCM, it is possible to pick a GCD that is compatible with the GCD of the base ring:

```
sage: ZZ.gcd(Integer(42), Integer(48))
6
sage: Integer(42).factor(); Integer(48).factor()
2 * 3 * 7
2 * 3 * 3
sage: ZZ.gcd(Integer(2)**Integer(4)*Integer(7)**Integer(2)*Integer(11), Integer(2)**Integer(3)*Integer(11)*Integer(13))
88
sage: Integer(88).factor()
2^3 * 11
```

Polynomial rings over fields are GCD domains as well. Here is a simple example over the ring of polynomials over the rationals as well as over an extension ring. Note that \( \text{gcd} \) requires \( x \) and \( y \) to be coercible:

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = NumberField(x^2 - 2, 'a')
sage: f = (x - a)*(x + a); g = (x - a)*(x^2 - 2)
sage: print(f); print(g)
x^2 - 2
x^3 - a*x^2 - 2*x + 2*a
sage: f in R
True
sage: g in R
False
sage: R.gcd(f, g)
Traceback (most recent call last):
...       TypeError: Unable to coerce 2*a to a rational
```
sage: R.base_extend(S).gcd(f, g)
x^2 - 2
sage: R.base_extend(S).gcd(f, (x - a)*(x^2 - 3))
x - a

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> S = NumberField(x**Integer(2) - Integer(2), 'a', names=('a',)); (a,)
... = S._first_ngens(1)
>>> f = (x - a)*(x + a); g = (x - a)*(x**Integer(2) - Integer(2))
>>> print(f); print(g)
x^2 - 2
x^3 - a*x^2 - 2*x + 2*a
>>> f in R
True
>>> g in R
False
>>> R.gcd(f, g)
Traceback (most recent call last):
... TypeError: Unable to coerce 2*a to a rational
>>> R.base_extend(S).gcd(f, g)
x^2 - 2
>>> R.base_extend(S).gcd(f, (x - a)*(x**Integer(2) - Integer(3)))
x - a

is_noetherian()

Every principal ideal domain is Noetherian, so we return True.

EXAMPLES:

sage: Zp(5).is_noetherian()  # needs sage.rings.padics
True

additional_structure()

Return None.

Indeed, the category of principal ideal domains defines no additional structure: a ring morphism between two principal ideal domains is a principal ideal domain morphism.

EXAMPLES:

sage: PrincipalIdealDomains().additional_structure()
4.130 Quotient fields

class sage.categories.quotient_fields.QuotientFields

Bases: Category_singleton

The category of quotient fields over an integral domain

EXAMPLES:

```python
sage: QuotientFields()
Category of quotient fields

sage: QuotientFields().super_categories()
[Category of fields]
```

```python
>>> from sage.all import *

sage: QuotientFields()
Category of quotient fields

sage: QuotientFields().super_categories()
[Category of fields]
```

class ElementMethods

Bases: object

denominator()

Constructor for abstract methods

EXAMPLES:

```python
sage: def f(x):
    ....:    "doc of f"
    ....:    return 1
sage: x = abstract_method(f); x
<abstract method f at ...>

sage: x.__doc__
'doc of f'

sage: x.__name__
'f'

sage: x.__module__
'__main__'
```

```python
>>> from sage.all import *

>>> def f(x):
...    "doc of f"
...    return Integer(1)

>>> x = abstract_method(f); x
<abstract method f at ...>

>>> x.__doc__
```

(continues on next page)
derivative(*args)

The derivative of this rational function, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:

_derivative()

EXAMPLES:

```python
sage: F.<x> = Frac(QQ['x'])
sage: (1/x).derivative()
-1/x^2
```

```python
>>> from sage.all import *
```
Here is an example to show that Issue #7868 has been resolved:

```python
sage: R.<x,y> = GF(2)[]
sage: f = x*y/(x+y)
sage: f.factor()  # needs sage.rings.finite_rings
(x + y)^-1 * y * x
```

**gcd**(other)

Greatest common divisor

**Note:** In a field, the greatest common divisor is not very informative, as it is only determined up to a unit. But in the fraction field of an integral domain that provides both gcd and lcm, it is possible to be a bit more specific and define the gcd uniquely up to a unit of the base ring (rather than in the fraction field).

**AUTHOR:**
- Simon King (2011-02): See Issue #10771

**EXAMPLES:**

```python
sage: # needs sage.libs.pari
sage: p = (1+x)^3*(1+2*x^2)/(1-x^5)
sage: q = (1+x)^2*(1+3*x^2)/(1-x^4)
sage: gcd(p, q)
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1
```

4.130. Quotient fields
```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> p = (Integer(1)+x)**Integer(3)*(Integer(1)+Integer(2)*x**Integer(2))/
    (-Integer(1)-x**Integer(5))
>>> q = (Integer(1)+x)**Integer(2)*(Integer(1)+Integer(3)*x**Integer(2))/
    (-Integer(1)-x**Integer(4))
>>> factor(p)
(-2) * (x - 1)^-1 * (x + 1)^3 * (x^2 + 1/2) * (x^4 + x^3 + x^2 + x + 1)^-1
>>> factor(q)
(-3) * (x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^2 + 1/3)
>>> gcd(p, q)
(x + 1)/(x^7 + x^5 - x^2 - 1)
>>> factor(gcd(p, q))
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^4 + x^3 + x^2 + x + 1)^-1
>>> factor(gcd(p, Integer(1) + x))
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^4 + x^3 + x^2 + x + 1)^-1
>>> factor(gcd(Integer(1) + x, q))
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1
```

1cm (other)

Least common multiple

In a field, the least common multiple is not very informative, as it is only determined up to a unit. But in the fraction field of an integral domain that provides both gcd and lcm, it is reasonable to be a bit more specific and to define the least common multiple so that it restricts to the usual least common multiple in the base ring and is unique up to a unit of the base ring (rather than up to a unit of the fraction field).

The least common multiple is easily described in terms of the prime decomposition. A rational number can be written as a product of primes with integer (positive or negative) powers in a unique way. The least common multiple of two rational numbers \(x\) and \(y\) can then be defined by specifying that the exponent of every prime \(p\) in lcm\((x, y)\) is the supremum of the exponents of \(p\) in \(x\), and the exponent of \(p\) in \(y\) (where the primes that do not appear in the decomposition of \(x\) or \(y\) are considered to have exponent zero).

AUTHOR:

• Simon King (2011-02): See Issue #10771

EXAMPLES:

```
sage: lcm(2/3, 1/5)
2
```

```
>>> from sage.all import *
>>> lcm(Integer(2)/Integer(3), Integer(1)/Integer(5))
2
```

Indeed \(2/3 = 2^1 3^{-1} 5^0\) and \(1/5 = 2^0 3^0 5^{-1}\), so \(lcm(2/3, 1/5) = 2^1 3^0 5^0 = 2\).

```
sage: lcm(1/3, 1/5)
sage: lcm(1/3, 1/6)
```

Some more involved examples:

```
sage: # needs sage.libs.pari
sage: R.<x> = QQ[]
sage: p = (1+x)^3*(1+2*x^2)/(1-x^5)
sage: q = (1+x)^2*(1+3*x^2)/(1-x^4)
sage: factor(p)
(-2) * (x - 1)^-1 * (x + 1)^3 * (x^2 + 1/2) * (x^4 + x^3 + x^2 + x + 1)^-1
sage: factor(q)
```

(continues on next page)
\[ (-3) \cdot (x - 1)^{-1} \cdot (x + 1) \cdot (x^2 + 1)^{-1} \cdot (x^2 + 1/3) \]

\[ \text{sage: factor(lcm(p, q))} \]
\[ (x - 1)^{-1} \cdot (x + 1)^3 \cdot (x^2 + 1/3) \cdot (x^2 + 1/2) \]

\[ \text{sage: factor(lcm(p, 1 + x))} \]
\[ (x + 1)^3 \cdot (x^2 + 1/2) \]

\[ \text{sage: factor(lcm(1 + x, q))} \]
\[ (x + 1) \cdot (x^2 + 1/3) \]

\[ \text{from sage.all import *} \]
\[ \text{# needs sage.libs.pari} \]
\[ R = \text{QQ['x']; (x,) = R._first_ngens(1)} \]
\[ p = (\text{Integer(1)+x})^{\text{Integer(3)}} \cdot (\text{Integer(1)+Integer(2)} \cdot x^{\text{Integer(2)}}) / (\text{Integer(1)-x^{\text{Integer(5)}}}) \]
\[ q = (\text{Integer(1)+x})^{\text{Integer(2)}} \cdot (\text{Integer(1)+Integer(3)} \cdot x^{\text{Integer(2)}}) / (\text{Integer(1)-x^{\text{Integer(4)}}}) \]

\[ \text{factor(p)} \]
\[ (-2) \cdot (x - 1)^{-1} \cdot (x + 1)^3 \cdot (x^2 + 1/2) \cdot (x^4 + x^3 + x^2 + x + 1)^{-1} \]

\[ \text{factor(q)} \]
\[ (-3) \cdot (x - 1)^{-1} \cdot (x + 1) \cdot (x^2 + 1)^{-1} \cdot (x^2 + 1/3) \]

\[ \text{factor(lcm(p, q))} \]
\[ (x - 1)^{-1} \cdot (x + 1)^3 \cdot (x^2 + 1/3) \cdot (x^2 + 1/2) \]

\[ \text{factor(lcm(p, 1 + x))} \]
\[ (x + 1)^3 \cdot (x^2 + 1/2) \]

\[ \text{factor(lcm(1 + x, q))} \]
\[ (x + 1) \cdot (x^2 + 1/3) \]

\[ \text{numerator()} \]

Constructor for abstract methods

**EXAMPLES:**

\[ \text{sage: def f(x):} \]
\[ \text{...: } \text{"doc of f"} \]
\[ \text{....: } \text{return 1} \]
\[ \text{sage: x = abstract_method(f); x} \]
\[ \text{<abstract method f at ...>} \]
\[ \text{sage: x.__doc__} \]
\[ \text{'doc of f'} \]
\[ \text{sage: x.__name__} \]
\[ \text{'f'} \]
\[ \text{sage: x.__module__} \]
\[ \text{'__main__'} \]

\[ \text{from sage.all import *} \]
\[ \text{def f(x):} \]
\[ \text{... } \text{"doc of f"} \]
\[ \text{... } \text{return Integer(1)} \]
\[ \text{x = abstract_method(f); x} \]
\[ \text{<abstract method f at ...>} \]
\[ \text{x.__doc__} \]
\[ \text{'doc of f'} \]
\[ \text{x.__name__} \]
\[ \text{'f'} \]
\[ \text{x.__module__} \]
\[ \text{'__main__'} \]

\[ \text{partial_fraction_decomposition (decompose_powers=True)} \]
Decompose fraction field element into a whole part and a list of fraction field elements over prime power denominators.

The sum will be equal to the original fraction.

**INPUT:**
- *decompose_powers* – boolean (default: True); whether to decompose prime power denominators as opposed to having a single term for each irreducible factor of the denominator

**OUTPUT:**
Partial fraction decomposition of self over the base ring.

**AUTHORS:**

**EXAMPLES:**

```
sage: # needs sage.libs.pari
sage: S.<t> = QQ[]
sage: q = 1/(t+1) + 2/(t+2) + 3/(t-3); q
(6*t^2 + 4*t - 6)/(t^3 - 7*t - 6)
sage: whole, parts = q.partial_fraction_decomposition(); parts
[3/(t - 3), 1/(t + 1), 2/(t + 2)]
sage: sum(parts) == q
True
sage: q = 1/(t^3+1) + 2/(t^2+2) + 3/(t-3)^5
sage: whole, parts = q.partial_fraction_decomposition(); parts
[1/3/(t + 1), 3/(t^5 - 15*t^4 + 90*t^3 - 270*t^2 + 405*t - 243),
(-1/3*t + 2/3)/(t^2 - t + 1), 2/(t^2 + 2)]
sage: sum(parts) == q
True
sage: q = 2*t / (t + 3)^2
sage: q.partial_fraction_decomposition()
(0, [2/(t + 3), -6/(t^2 + 6*t + 9)])
sage: for p in q.partial_fraction_decomposition()[1]:
    print(p.factor())
(2) * (t + 3)^-1
(-6) * (t + 3)^-2
sage: q.partial_fraction_decomposition(decompose_powers=False)
(0, [2*t/(t^2 + 6*t + 9)])
```

(continues on next page)
\[
q = \frac{2t}{(t + 3)^2}
\]
\[
q.\text{partial_fraction_decomposition}()
\]
\[
(0, \frac{2}{t + 3}, -\frac{6}{(t + 3)^2})
\]
\[
\text{for } p \text{ in } q.\text{partial_fraction_decomposition}[1]:
\]
\[
\text{print}(p.\text{factor}())
\]
\[
2(t + 3)^{-1}
\]
\[
-6(t + 3)^{-2}
\]
\[
q.\text{partial_fraction_decomposition}() [\text{decompose_powers} = \text{False}]
\]
\[
(0, \frac{2t}{(t^2 + 6t + 9)})
\]

We can decompose over a given algebraic extension:

\[
sage: R.<x> = QQ[sqrt(2)][]
\]
\[
sage: r = 1/(x^4 + 1)
\]
\[
sage: r.\text{partial_fraction_decomposition}()
\]
\[
(0, \frac{-1/4*sqrt2*x + 1/2}{(x^2 - sqrt2*x + 1)}, \frac{1/4*sqrt2*x + 1/2}{(x^2 + sqrt2*x + 1)})
\]

\[
sage: R.<x> = QQ[I][] # of QQ[sqrt(-1)]
\]
\[
sage: r = 1/(x^4 + 1)
\]
\[
sage: r.\text{partial_fraction_decomposition}()
\]
\[
(0, \frac{-1/2*I}{(x^2 - I)}, \frac{1/2*I}{(x^2 + I)})
\]

We can also ask Sage to find the least extension where the denominator factors in linear terms:

\[
sage: # \text{needs sage.rings.number_field sage.symbolic}
\]
\[
sage: R.<x> = QQ[] \text{ needs sage.rings.number_field}
\]
\[
sage: r = Integer(1)/(x**Integer(4)+Integer(1)) \text{ needs sage.rings.number_field}
\]
\[
sage: r.\text{partial_fraction_decomposition}()
\]
\[
(0, \frac{-1/4*sqrt2*x + 1/2}{(x^2 - sqrt2*x + 1)}, \frac{1/4*sqrt2*x + 1/2}{(x^2 + sqrt2*x + 1)})
\]

\[
sage: # \text{needs sage.rings.number_field sage.symbolic}
\]
\[
sage: R.<x> = QQ[I][] \text{ needs sage.rings.number_field}
\]
\[
sage: r = Integer(1)/(x**Integer(4)+Integer(1)) \text{ needs sage.rings.number_field}
\]
\[
sage: r.\text{partial_fraction_decomposition}()
\]
\[
(0, \frac{-1/2*I}{(x^2 - I)}, \frac{1/2*I}{(x^2 + I)})
\]
sage: R1.<x1> = N[]
sage: r1 = 1/(x1^4+2)  
sage: r1.partial_fraction_decomposition()
(0, 
[-1/224*a^6 + 13/448*a^4 - 5/56*a^2 - 25/224)/(x1 - 1/28*a^6 + 13/56*a^4 - 5/7*a^2 - 25/28),
(1/224*a^6 - 13/448*a^4 + 5/56*a^2 + 25/224)/(x1 + 1/28*a^6 - 13/56*a^4 + 5/7*a^2 + 25/28),
(-5/1344*a^7 + 43/1344*a^5 - 85/672*a^3 - 31/672*a)/(x1 - 5/168*a^7 + 43/168*a^5 - 85/84*a^3 - 31/84*a),
(5/1344*a^7 - 43/1344*a^5 + 85/672*a^3 + 31/672*a)/(x1 + 5/168*a^7 - 43/168*a^5 + 85/84*a^3 + 31/84*a])

Or we may work directly over an algebraically closed field:

sage: R.<x> = QQbar[]
# needs sage.rings.number_field

sage: r = 1/(x^4+1)  
# needs sage.rings.number_field

sage: r.partial_fraction_decomposition()
(0, 
[-0.1767766952966369? - 0.1767766952966369?*I)/(x - 0.7071067811865475? - 0.7071067811865475?*I),
(-0.1767766952966369? + 0.1767766952966369?*I)/(x - 0.7071067811865475? + 0.7071067811865475?*I),
(0.1767766952966369? - 0.1767766952966369?*I)/(x + 0.7071067811865475? - 0.7071067811865475?*I),
(0.1767766952966369? + 0.1767766952966369?*I)/(x + 0.7071067811865475? + 0.7071067811865475?*I))
We do the best we can over inexact fields:

```python
sage: # needs sage.rings.number_field sage.rings.real_mpfr
sage: R.<x> = RealField(20)[]
```

```python
sage: q = 1/(x^2 + x + 2)^2 + 1/(x-1); q
(x^4 + 2.0000*x^3 + 5.0000*x^2 + 5.0000*x + 3.0000)/(x^5 + x^4 + 3.0000*x^3 - x^2 - 4.0000)
```

```python
sage: whole, parts = q.partial_fraction_decomposition(); parts
[1.0000/(x - 1.0000), 1.0000/(x^4 + 2.0000*x^3 + 5.0000*x^2 + 4.0000*x + 4.0000)]
```

```python
sage: sum(parts)
(x^4 + 2.0000*x^3 + 5.0000*x^2 + 5.0000*x + 3.0000)/(x^5 + x^4 + 3.0000*x^3 - x^2 - 4.0000)
```

```
```

**xgcd** *(other)*

Return a triple \((g, s, t)\) of elements of that field such that \(g\) is the greatest common divisor of \(self\) and \(other\) and \(g = s*\text{self} + t*\text{other}\).

**Note:** In a field, the greatest common divisor is not very informative, as it is only determined up to a unit. But in the fraction field of an integral domain that provides both xgcd and lcm, it is possible to be a bit more specific and define the gcd uniquely up to a unit of the base ring (rather than in the fraction field).

**EXAMPLES:**

---

4.130. Quotient fields 1129
An example without a well defined gcd or xgcd on its base ring:

```python
sage: # needs sage.rings.number_field
sage: K = QuadraticField(5)
sage: O = K.maximal_order()
sage: R = PolynomialRing(O, 'x')
```

(continues on next page)
sage: F = R.fraction_field()
sage: x = F.gen(0)
sage: x.gcd(x+1)
1
sage: x.xgcd(x+1)
(1, 1/x, 0)
sage: zero = F.zero()
sage: zero.gcd(x)
1
sage: zero.xgcd(x)
(1, 0, 1/x)
sage: zero.xgcd(zero)
(0, 0, 0)

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(5))
>>> O = K.maximal_order()
>>> R = PolynomialRing(O, 'x')
>>> F = R.fraction_field()
>>> x = F.gen(Integer(0))
>>> x.gcd(x+Integer(1))
1
>>> x.xgcd(x+Integer(1))
(1, 1/x, 0)
>>> zero = F.zero()
>>> zero.gcd(x)
1
>>> zero.xgcd(x)
(1, 0, 1/x)
>>> zero.xgcd(zero)
(0, 0, 0)

class ParentMethods
    Bases: object

    super_categories()

    EXAMPLES:

    sage: QuotientFields().super_categories()
    [Category of fields]

    >>> from sage.all import *
    >>> QuotientFields().super_categories()
    [Category of fields]
4.131 Quantum Group Representations

AUTHORS:

- Travis Scrimshaw (2018): initial version

```python
class sage.categories.quantum_group_representations.QuantumGroupRepresentations(base, name=None):
    Bases: Category_module

    The category of quantum group representations.

class ParentMethods
    Bases: object

    cartan_type()
    Return the Cartan type of self.

    EXAMPLES:

    sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
    sage: C = crystals.Tableaux(['C',4], shape=[1])
    sage: R = ZZ['q'].fraction_field()
    sage: V = MinusculeRepresentation(R, C)
    sage: V.cartan_type()
    ['C', 4]

    index_set()
    Return the index set of self.

    EXAMPLES:

    sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
    sage: C = crystals.Tableaux(['C',4], shape=[1])
    sage: R = ZZ['q'].fraction_field()
    sage: V = MinusculeRepresentation(R, C)
    sage: V.index_set()
    (1, 2, 3, 4)
```

(continues on next page)
C = crystals.Tableaux(['C',4], shape=[Integer(1)])
R = ZZ['q'].fraction_field()
V = MinusculeRepresentation(R, C)
V.index_set()
(1, 2, 3, 4)

q()
Return the quantum parameter $q$ of self.

EXAMPLES:

```python
sage: from sage.algebras.quantum_groups.representations import...
    MinusculeRepresentation
sage: C = crystals.Tableaux(['C',4], shape=[1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculeRepresentation(R, C)
sage: V.q()
q
```

```python
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import...
    MinusculeRepresentation
>>> C = crystals.Tableaux(['C',4], shape=[1])
>>> R = ZZ['q'].fraction_field()
>>> V = MinusculeRepresentation(R, C)
>>> V.q()
```

class TensorProducts (category, *args)

Bases: TensorProductsCategory

The category of quantum group representations constructed by tensor product of quantum group representations.

Warning: We use the reversed coproduct in order to match the tensor product rule on crystals.

class ParentMethods

Bases: object

cartan_type()
Return the Cartan type of self.

EXAMPLES:

```python
sage: from sage.algebras.quantum_groups.representations import...
    MinusculeRepresentation
sage: C = crystals.Tableaux(['C',2], shape=[1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculeRepresentation(R, C)
sage: T = tensor([V,V])
sage: T.cartan_type()
['C', 2]
```
```python
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import ...
   ->MinusculeRepresentation
>>> C = crystals.Tableaux(['C',Integer(2)], shape=[Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = MinusculeRepresentation(R, C)
>>> T = tensor([V,V])
>>> T.cartan_type()
['C', 2]
```

```python
element_methods()

EXAMPLES:
```
```python
sage: from sage.categories.quantum_group_representations import ...
   ->QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
```
```python
class WithBasis (base_category)

Bases: CategoryWithAxiom_over_base_ring

The category of quantum group representations with a distinguished basis.

class ElementMethods

Bases: object

\( K(i, \text{power}=1) \)

Return the action of \( K_i \) on \text{self} to the power \text{power}.

INPUT:

\* i -- an element of the index set
\* power -- (default: 1) the power of \( K_i \)

EXAMPLES:
```
```python
sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import ...
   ->AdjointRepresentation
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
```
sage: v.K(1, 2)
2*B[[1]] + 2*q^4*B[[[1]]] + 3/q^4*B[[[2]]]
sage: v.K(1, -1)
2*B[[1]] + 2/q^2*B[[[1]]] + 3*q^2*B[[[2]]]

>>> from sage.all import *
>>> # needs sage.combinat sage.graphs sage.modules
>>> from sage.algebras.quantum_groups.representations import AdjointRepresentation
>>> K = crystals.KirillovReshetikhin(['D', Integer(4)], Integer(1), Integer(1))
>>> R = ZZ['q'].fraction_field()
>>> V = AdjointRepresentation(R, K)
>>> v = V.an_element(); v
2*B[[1]] + 2*B[[[1]]] + 3*B[[[2]]]
>>> v.K(Integer(0))
2*B[[1]] + 2*B[[[1]]] + 3*B[[[2]]]
>>> v.K(Integer(1))
2*B[[1]] + 2*q^2*B[[[1]]] + 3/q^2*B[[[2]]]
>>> v.K(Integer(1), Integer(2))
2*B[[1]] + 2*q^4*B[[[1]]] + 3/q^4*B[[[2]]]
>>> v.K(Integer(1), -Integer(1))
2*B[[1]] + 2/q^2*B[[[1]]] + 3*q^2*B[[[2]]]

\textbf{e}(i)

Return the action of $e_i$ on self.

INPUT:

• $i$ – an element of the index set

EXAMPLES:

sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import AdjointRepresentation
sage: C = crystals.Tableaux(['G',2], shape=[1,1])
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, C)
sage: v = V.an_element(); v
2*B[[[1], [2]]] + 2*B[[[1], [3]]] + 3*B[[[2], [3]]]
sage: v.e(1)
((3*q^4+3*q^2+3)/q^2)*B[[[1], [3]]]
sage: v.e(2)
2*B[[[1], [2]]]

(continues on next page)
\begin{verbatim}
>>> v.e(Integer(2))
2*B[[[1], [2]]]

f(i)

Return the action of \( f_i \) on self.

INPUT:

\cdot i – an element of the index set

EXAMPLES:

sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import...
    AdjointRepresentation
sage: K = crystals.KirillovReshetikhin(['D', 4, 1], 2, 1)
sage: R = ZZ['q'].fraction_field()

sage: V = AdjointRepresentation(R, K)

sage: v = V.an_element(); v
2*B[[1]] + 2*B[[[1], [2]]] + 3*B[[[1], [3]]]

sage: v.f(0)
((2*q^2+2)/q)*B[[[1], [2]]]

sage: v.f(1)
3*B[[2], [3]]

sage: v.f(2)
2*B[[[1], [3]]]

sage: v.f(3)
3*B[[[1], [4]]]

sage: v.f(4)
3*B[[[1], [-4]]]

sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import...
    AdjointRepresentation
sage: K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(1)],
    Integer(2), Integer(1))

sage: R = ZZ['q'].fraction_field()

sage: V = AdjointRepresentation(R, K)

sage: v = V.an_element(); v
2*B[[1]] + 2*B[[[1], [2]]] + 3*B[[[1], [3]]]

sage: v.f(Integer(0))
((2*q^2+2)/q)*B[[[1], [2]]]

sage: v.f(Integer(1))
3*B[[2], [3]]

sage: v.f(Integer(2))
2*B[[[1], [3]]]

sage: v.f(Integer(3))
3*B[[[1], [4]]]

sage: v.f(Integer(4))
3*B[[[1], [-4]]]

class ParentMethods

    Bases: object

    tensor(*factors)

    Return the tensor product of self with the representations factors.

    EXAMPLES:
\end{verbatim}
sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import (MinusculeRepresentation, AdjointRepresentation)

R = ZZ['q'].fraction_field()
CM = crystals.Tableaux(['D',4], shape=[1])
CA = crystals.Tableaux(['D',4], shape=[1,1])
V = MinusculeRepresentation(R, CM)
V tensor(V, V)

Traceback (most recent call last):
... ValueError: all factors must be of the same Cartan type

A = MinusculeRepresentation(R, CA)
V tensor(A)

Traceback (most recent call last):
... ValueError: all factors must be of the same Cartan type

B = crystals.Tableaux(['A',2], shape=[1])
W = MinusculeRepresentation(R, B)
tensor([W, V])

TensorProducts

The category of quantum group representations with a distinguished basis constructed by tensor product of quantum group representations with a distinguished basis.

class ParentMethods

Bases: object

K_on_basis(i, b, power=1)
Return the action of $K_i$ on the basis element indexed by $b$ to the power $\text{power}$.

**INPUT:**
- $i$ – an element of the index set
- $b$ – an element of basis keys
- $\text{power}$ – (default: 1) the power of $K_i$

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import (...
   MinusculeRepresentation, AdjointRepresentation)
   R = ZZ['q'].fraction_field()
   CM = crystals.Tableaux(['A',2], shape=[1])
   VM = MinusculeRepresentation(R, CM)
   VA = AdjointRepresentation(R, CA)
   v = tensor([sum(VM.basis()), VA.module_generator()]); v
   B[[[1]]] # B[[[1, 1], [2]]]
   + B[[[2]]] # B[[[1, 1], [2]]]
   + B[[[3]]] # B[[[1, 1], [2]]]

sage: v.K(1) # indirect doctest
q^2*B[[[1]]] # B[[[1, 1], [2]]]
   + B[[[2]]] # B[[[1, 1], [2]]]
   + q*B[[[3]]] # B[[[1, 1], [2]]]

sage: v.K(2, -1) # indirect doctest
1/q*B[[[1]]] # B[[[1, 1], [2]]]
   + 1/q^2*B[[[2]]] # B[[[1, 1], [2]]]
   + B[[[3]]] # B[[[1, 1], [2]]]
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs sage.modules
>>> from sage.algebras.quantum_groups.representations import (...
   MinusculeRepresentation, AdjointRepresentation)
>>> R = ZZ['q'].fraction_field()
>>> CM = crystals.Tableaux(['A',Integer(2)], shape=[Integer(1)])
>>> VM = MinusculeRepresentation(R, CM)
>>> CA = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2), Integer(1)])
>>> VA = AdjointRepresentation(R, CA)
>>> v = tensor([sum(VM.basis()), VA.module_generator()]); v
   B[[[1]]] # B[[[1, 1], [2]]]
   + B[[[2]]] # B[[[1, 1], [2]]]
   + B[[[3]]] # B[[[1, 1], [2]]]

>>> v.K(Integer(1)) # indirect doctest
q^2*B[[[1]]] # B[[[1, 1], [2]]]
   + B[[[2]]] # B[[[1, 1], [2]]]
   + q*B[[[3]]] # B[[[1, 1], [2]]]

>>> v.K(Integer(2), -Integer(1)) # indirect doctest
1/q*B[[[1]]] # B[[[1, 1], [2]]]
   + 1/q^2*B[[[2]]] # B[[[1, 1], [2]]]
   + B[[[3]]] # B[[[1, 1], [2]]]
```

\[ e_on_basis(i, b) \]

Return the action of $e_i$ on the basis element indexed by $b$.

**INPUT:**
- $i$ – an element of the index set
• b – an element of basis keys

EXAMPLES:

```python
sage: # needs sage.combinat sage.graphs sage.modules
sage: from sage.algebras.quantum_groups.representations import (  
    MinusculeRepresentation, AdjointRepresentation)
sage: R = ZZ['q'].fraction_field()
sage: CM = crystals.Tableaux(['D',4], shape=[1])
sage: VM = MinusculeRepresentation(R, CM)
sage: CA = crystals.Tableaux(['D',4], shape=[1,1])
sage: VA = AdjointRepresentation(R, CA)

sage: v = tensor([VM.an_element(), VA.an_element()]); v
4*B[[1]] # B[[1], [2]] + 4*B[[1]] # B[[1], [3]]
+ 6*B[[1]] # B[[2], [3]] + 4*B[[2]] # B[[1], [2]]
+ 4*B[[2]] # B[[1], [3]] + 6*B[[2]] # B[[2], [3]]
+ 6*B[[3]] # B[[1], [2]] + 6*B[[3]] # B[[1], [3]]
+ 9*B[[3]] # B[[2], [3]]

sage: v.e(1)  # indirect doctest
4*B[[1]] # B[[1], [2]]
+ ((4*q+6)/q)*B[[1]] # B[[1], [3]]
+ 6*B[[1]] # B[[2], [3]]
+ 6*q*B[[2]] # B[[1], [3]]
+ 9*B[[3]] # B[[2], [3]]

sage: v.e(2)  # indirect doctest
4*B[[1]] # B[[1], [2]]
+ ((6*q+4)/q)*B[[2]] # B[[1], [2]]
+ 6*B[[2]] # B[[1], [3]]
+ 9*B[[2]] # B[[2], [3]]
+ 6*q*B[[3]] # B[[1], [2]]

sage: v.e(3)  # indirect doctest
0

sage: v.e(4)  # indirect doctest
0

>>> from sage.all import *
>>> # needs sage.combinat sage.graphs sage.modules
>>> from sage.algebras.quantum_groups.representations import (  
    MinusculeRepresentation, AdjointRepresentation)
>>> R = ZZ['q'].fraction_field()
>>> CM = crystals.Tableaux(['D',4], shape=[Integer(1)])
>>> VM = MinusculeRepresentation(R, CM)
>>> CA = crystals.Tableaux(['D',4], shape=[Integer(1),  
    Integer(1)])

>>> VA = AdjointRepresentation(R, CA)
>>> v = tensor([VM.an_element(), VA.an_element()]); v
4*B[[1]] # B[[1], [2]]
+ ((4*q+6)/q)*B[[1]] # B[[1], [3]]
+ 6*B[[1]] # B[[2], [3]]
+ 4*B[[2]] # B[[1], [2]]
+ 4*B[[2]] # B[[1], [3]]
+ 6*B[[2]] # B[[2], [3]]
+ 6*B[[3]] # B[[1], [2]]
+ 6*B[[3]] # B[[1], [3]]
+ 9*B[[3]] # B[[2], [3]]

>>> v.e(Integer(1))  # indirect doctest
4*B[[1]] # B[[1], [2]]
+ ((4*q+6)/q)*B[[1]] # B[[1], [3]]
+ 6*B[[1]] # B[[2], [3]]
+ 6*q*B[[2]] # B[[1], [3]]
+ 9*B[[3]] # B[[2], [3]]

>>> v.e(Integer(2))  # indirect doctest
4*B[[1]] # B[[1], [2]]
```

(continues on next page)
\[ + \frac{(6q+4)}{q} B[[[2]]] B[[[1], [2]]] \]
\[ + 6 B[[[2]]] B[[[1], [3]]] \]
\[ + 9 B[[[2]]] B[[[2], [3]]] \]
\[ + 6 q^2 B[[[3]]] B[[[1], [2]]] \]

```
>>> v.e(Integer(3))  # indirect doctest
0
```
```
>>> v.e(Integer(4))  # indirect doctest
0
```

### f_on_basis \((i, b)\)

Return the action of \(f_i\) on the basis element indexed by \(b\).

**INPUT:**
- \(i\) – an element of the index set
- \(b\) – an element of basis keys

**EXAMPLES:**

```python
sage: from sage.algebras.quantum_groups.representations import (...
....: MinusculeRepresentation, AdjointRepresentation)

sage: R = ZZ['q'].fraction_field()

sage: KM = crystals.KirillovReshetikhin(['B',3,1], 3,1)

sage: VM = MinusculeRepresentation(R, KM)

sage: KA = crystals.KirillovReshetikhin(['B',3,1], 2,1)

sage: VA = AdjointRepresentation(R, KA)

sage: v = tensor([VM.an_element(), VA.an_element()]); v

\[ 4 B[[++, [1]]] + 2 B[[++, [2]]] + B[[++, [3]]] \]
\[ + 4 B[[++-, [1]]] B[[[1], [3]]] + 4 B[[++-, [2]]] B[[[]]] \]
\[ + 4 B[[++-, [3]]] B[[[1], [2]]] \]
\[ + 4 B[[+-+, [1]]] B[[[1], [3]]] + 4 B[[+-+, [2]]] B[[[1]]] \]
\[ + 9 B[[++-, [1]]] B[[[1], [2]]] \]
```
```
>>> v.f(0)  # indirect doctest
((4q^4+4)/q^2) B[[++, [1]]] B[[[1], [2]]] + ((4q^4+4)/q^2) B[[++, [2]]] B[[[1], [2]]] + ((6q^4+6)/q^2) B[[++, [3]]] B[[[]]]
```
```
>>> v.f(1)  # indirect doctest
6 B[[++, [2]]] + 6 B[[++, [3]]] B[[[2], [3]]] + 9 B[[++, [1]]] B[[[1], [2]]] + 6 B[[++-, [1]]] B[[[1]]] + 6 B[[++-, [2]]] B[[[1], [3]]] + 9 q^2 B[[-++, [1]]] B[[[1], [2]]]
```
```
>>> v.f(2)  # indirect doctest
4 B[[++, [1]]] B[[[1], [3]]] + 4 B[[++, [2]]] B[[[1], [3]]] + 4 B[[++, [3]]] B[[[1], [3]]] + 4 B[[+-+, [1]]] B[[[1]]] + 4 B[[+-+, [2]]] B[[[1], [2]]] + (6q^2/4) B[[+-+, [1]]] B[[[1], [3]]]
```
```
>>> v.f(3)  # indirect doctest
6 B[[++, [1]]] B[[[1], [0]]] + 4 B[[++, [2]]] B[[[1]]] + 4 B[[++, [3]]] B[[[1], [2]]] + 6 q^2 B[[+-+, [1]]] B[[[1], [3]]] + 6 B[[++, [1]]] B[[[1], [0]]] + 9 B[[++-, [1]]] B[[[1], [0]]]
```
```
>>> from sage.all import *

>>> # needs sage.combinat sage.graphs sage.modules

>>> from sage.algebras.quantum_groups.representations import *

>>> R = ZZ['q'].fraction_field()

>>> KM = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],...
   –> Integer(3), Integer(1))

>>> VM = MinusculeRepresentation(R, KM)

>>> KA = crystals.KirillovReshetikhin(['B', Integer(3), Integer(1)],...
   –> Integer(2), Integer(1))

>>> VA = AdjointRepresentation(R, KA)

>>> v = tensor([VM.an_element(), VA.an_element()]); v

    4*B[+++, []] # B[[1], [2]]
    + 4*B[+++, []] # B[[1], [2]]
    + 6*B[+++, []] # B[[1], [3]]
    + 4*B[+++, []] # B[[1], [2]]
    + 6*B[+++, []] # B[[1], [3]]
    + 6*B[+++, []] # B[[1], [3]]
    + 9*B[+++, []] # B[[1], [3]]

>>> v.f(Integer(0))  # indirect doctest

    ((4*q^4+4)/q^2)*B[+++, []] # B[[1], [2]]
    + ((4*q^4+4)/q^2)*B[+++, []] # B[[1], [1]]
    + ((6*q^4+6)/q^2)*B[+++, []] # B[[1], [2]]

>>> v.f(Integer(1))  # indirect doctest

    6*B[+++, []] # B[[2], [3]]
    + 6*B[+++, []] # B[[2], [3]]
    + 9*B[+++, []] # B[[2], [3]]
    + 6*B[+++, []] # B[[1], [1]]
    + 6*B[+++, []] # B[[1], [1]]
    + 9*B[+++, []] # B[[1], [1]]

>>> v.f(Integer(2))  # indirect doctest

    4*B[+++, []] # B[[1], [3]]
    + 4*B[+++, []] # B[[1], [3]]
    + 4*B[+++, []] # B[[1], [1]]
    + 4*q^2*B[+++, []] # B[[1], [2]]
    + ((6*q^2+6)/q^2)*B[+++, []] # B[[1], [3]]

>>> v.f(Integer(3))  # indirect doctest

    6*B[+++, []] # B[[1], [0]]
    + 4*B[+++, []] # B[[1], [0]]
    + 4*B[+++, []] # B[[1], [1]]
    + 6*q^2*B[+++, []] # B[[1], [3]]
    + 6*B[+++, []] # B[[1], [1]]
    + 6*B[+++, []] # B[[1], [0]]
    + 6*B[+++, []] # B[[1], [0]]
    + 6*B[+++, []] # B[[1], [2]]
    + 6*B[+++, []] # B[[1], [2]]
    + 9*q^2*B[+++, []] # B[[1], [3]]
example()
Return an example of a quantum group representation as per Category.example.

EXAMPLES:

sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.example()
V((2, 1, 0))

super_categories()
Return the super categories of self.

EXAMPLES:

sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: QuantumGroupRepresentations(ZZ['q'].fraction_field()).super_categories()
[Category of vector spaces over Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
4.132 Regular Crystals

class sage.categories.regular_crystals.RegularCrystals

    Bases: Category_singleton

    The category of regular crystals.

    A crystal is called regular if every vertex \( b \) satisfies

    \[
    \varepsilon_i(b) = \max\{k \mid e_i^k(b) \neq 0\} \quad \text{and} \quad \varphi_i(b) = \max\{k \mid f_i^k(b) \neq 0\}.
    \]

Note: Regular crystals are sometimes referred to as normal. When only one of the conditions (on either \( \varphi_i \) or \( \varepsilon_i \)) holds, these crystals are sometimes called seminormal or semiregular.

EXAMPLES:

```
sage: C = RegularCrystals()
sage: C
Category of regular crystals
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

```bash
>>> from sage.all import *
>>> C = RegularCrystals()
>>> C
Category of regular crystals
>>> C.super_categories()
[Category of crystals]
>>> C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

class ElementMethods

    Bases: object

    demazure_operator_simple \( i, \text{ring=}None \)

    Return the Demazure operator \( D_i \) applied to \text{self}.

    INPUT:
    • \( i \) – an element of the index set of the underlying crystal
    • \text{ring} – (default: \( \mathbb{Q} \)) a ring

    OUTPUT:

    An element of the \text{ring}-free module indexed by the underlying crystal.

    Let \( r = (\text{wt}(b), \alpha_i^\vee) \), then \( D_i(b) \) is defined as follows:
    • If \( r \geq 0 \), this returns the sum of the elements obtained from \text{self} by application of \( f_i^k \) for \( 0 \leq k \leq r \).
    • If \( r < 0 \), this returns the opposite of the sum of the elements obtained by application of \( e_i^k \) for \( 0 < k < -r \).

    REFERENCES:
    • [Li1995]
    • [Ka1993]

    EXAMPLES:
```python
sage: t = T(rows=[[1,2],[2]])
sage: t.demazure_operator_simple(2)
B([[1, 2], [2]]) + B([[1, 3], [2]]) + B([[1, 3], [3]])
sage: t.demazure_operator_simple(2).parent()
Algebra of The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]
   over Integer Ring
sage: t.demazure_operator_simple(1)
0
sage: K = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: t = K(rows=[[3],[2]])
sage: t.demazure_operator_simple(0)
B([[1, 2]]) + B([[2, 3]])
```

dual_equivalence_class(index_set=None)

Return the dual equivalence class indexed by index_set of self.

The dual equivalence class of an element \( b \in B \) is the set of all elements of \( B \) reachable from \( b \) via sequences of \( i \)-elementary dual equivalence relations (i.e., \( i \)-elementary dual equivalence transformations and their inverses) for \( i \) in the index set of \( B \).

For this to be well-defined, the element \( b \) has to be of weight 0 with respect to \( I \); that is, we need to have \( \varepsilon_j(b) = \varphi_j(b) \) for all \( j \in I \).

See [As2008]. See also dual_equivalence_graph() for a definition of \( i \)-elementary dual equivalence transformations.

INPUT:
- index_set – (optional) the index set \( I \) (default: the whole index set of the crystal); this has to be a subset of the index set of the crystal (as a list or tuple)

OUTPUT:

The dual equivalence class of self indexed by the subset index_set. This class is returned as an undirected edge-colored multigraph. The color of an edge is the index \( i \) of the dual equivalence relation it encodes.

See also:
- dual_equivalence_graph()
• sage.combinat.partition.Partition.dual_equivalence_graph()

EXAMPES:

```python
crystals.Tableaux(['A', 3], shape=[2, 2])
crystals.Tableaux(['A', 4], shape=[3, 2])
crystals.Tableaux(['A', Integer(3)], shape=[Integer(2), Integer(2)])
crystals.Tableaux(['A', Integer(4)], shape=[Integer(3), Integer(2)])
crystals.Letters(['A', 5])
crystals.Letters(['A', Integer(5)])
```

\( \epsilon(i) \)

Return \( \epsilon_i \) of self.

EXAMPLES:

```python
crystals.Letters(['A', 5])
crystals.Letters(['A', Integer(5)])
```

\( \phi(i) \)

Return \( \phi_i \) of self.

EXAMPLES:

```python
crystals.Letters(['A', 5])
crystals.Letters(['A', Integer(5)])
```
stembridgeDel_depth\( (i, j) \)

Return the difference in the \( j \)-depth of \( \text{self} \) and \( f_i \) of \( \text{self} \), where \( i \) and \( j \) are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The \( i \)-depth of a crystal node \( x \) is \( \epsilon_i(x) \).

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = T(rows=[[1,1],[2]])
sage: t.stembridgeDel_depth(1,2)
0
sage: s = T(rows=[[1,3],[3]])
sage: s.stembridgeDel_depth(1,2)
-1
```

stembridgeDel_rise\( (i, j) \)

Return the difference in the \( j \)-rise of \( \text{self} \) and \( f_i \) of \( \text{self} \), where \( i \) and \( j \) are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The \( i \)-rise of a crystal node \( x \) is \( \varphi_i(x) \).

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = T(rows=[[1,1],[2]])
sage: t.stembridgeDel_rise(1,2)
-1
sage: s = T(rows=[[1,3],[3]])
sage: s.stembridgeDel_rise(1,2)
0
```

(continues on next page)
stembridgeDelta_depth(i, j)

Return the difference in the $j$-depth of $\text{self}$ and $e_i$ of $\text{self}$, where $i$ and $j$ are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The $i$-depth of a crystal node $x$ is $-\varepsilon_i(x)$.

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: t = T(rows=[[1, 2], [2]])
sage: t.stembridgeDelta_depth(1, 2)
0
sage: s = T(rows=[[2, 3], [3]])
sage: s.stembridgeDelta_depth(1, 2)
-1
```

stembridgeDelta_rise(i, j)

Return the difference in the $j$-rise of $\text{self}$ and $e_i$ of $\text{self}$, where $i$ and $j$ are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The $i$-rise of a crystal node $x$ is $\varphi_i(x)$.

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: t = T(rows=[[1, 2], [2]])
sage: t.stembridgeDelta_rise(1, 2)
-1
sage: s = T(rows=[[2, 3], [3]])
sage: s.stembridgeDelta_rise(1, 2)
0
```
stembridgeTriple \((i,j)\)

Let \(A\) be the Cartan matrix of the crystal, \(x\) a crystal element, and let \(i\) and \(j\) be in the index set of the crystal. Further, set \(b = \text{stembridgeDelta}_\text{depth}(x, i, j)\), and \(c = \text{stembridgeDelta}_\text{rise}(x, i, j)\). If \(x.e(i)\) is non-empty, this function returns the triple \((A_{ij}, b, c)\); otherwise it returns None. By the Stembridge local characterization of crystal bases, one should have \(A_{ij} = b + c\).

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = T(rows=[[1,1],[2]])
sage: t.stembridgeTriple(1,2)
(-1, 0, -1)

sage: T = crystals.Tableaux(['B',2], shape=[2,1])
sage: t = T(rows=[[1,2],[2]])
sage: t.stembridgeTriple(1,2)
(-2, 0, -2)
```

weight()

Return the weight of this crystal element.

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).weight()
(1, 0, 0, 0, 0)
```

```python
>>> from sage.all import *

>>> C = crystals.Letters(['A',Integer(5)])
```
class MorphismMethods
Bases: object

is_isomorphism()
Check if self is a crystal isomorphism, which is true if and only if this is a strict embedding with the same number of connected components.

EXAMPLES:

```python
sage: A21 = RootSystem(['A',2,1])

sage: La = A21.weight_space(extended = True).fundamental_weights()

sage: B = crystals.LSPaths(La[0])

sage: C = crystals.GeneralizedYoungWalls(2, La[0])

sage: H = Hom(B, C)

sage: from sage.categories.highest_weight_crystals import HighestWeightCrystalMorphism

sage: class Psi(HighestWeightCrystalMorphism):
...     def is_strict(self):
...         return True

sage: psi = Psi(H, C.module_generators); psi
[A, 2, 1] Crystal morphism:
    From: The crystal of LS paths of type [A, 2, 1] and weight Lambda[0]
    To: Highest weight crystal of generalized Young walls of Cartan type [A, 2, 1] and highest weight Lambda[0]
    Defn: (Lambda[0],) |--> []

sage: psi.is_isomorphism()
True
```

```python
>>> from sage.all import *
```

```python
A21 = RootSystem(['A',Integer(2),Integer(1)])

La = A21.weight_space(extended = True).fundamental_weights()

B = crystals.LSPaths(La[0])

C = crystals.GeneralizedYoungWalls(2, La[0])

H = Hom(B, C)

>>> from sage.categories.highest_weight_crystals import HighestWeightCrystalMorphism

>>> class Psi(HighestWeightCrystalMorphism):
...     def is_strict(self):
...         return True

>>> psi = Psi(H, C.module_generators); psi
['A', 2, 1] Crystal morphism:
    From: The crystal of LS paths of type ['A', 2, 1] and weight Lambda[0]
    To: Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1] and highest weight Lambda[0]
    Defn: (Lambda[0],) |--> []

>>> psi.is_isomorphism()
True
```

class ParentMethods
Bases: object
demazure_operator \((element, \text{reduced\_word})\)

Returns the application of Demazure operators \(D_i\) for \(i\) from \text{reduced\_word} on \text{element}.

INPUT:
- \text{element} — an element of a free module indexed by the underlying crystal
- \text{reduced\_word} — a reduced word of the Weyl group of the same type as the underlying crystal

OUTPUT:
- an element of the free module indexed by the underlying crystal

EXAMPLES:

\begin{verbatim}
sage: T = crystals.Tableaux('A2', shape=[3])
sage: C = CombinatorialFreeModule(QQ, T)
sage: t = T.highest_weight_vector()
sage: b = 2*C(t)
sage: T.demazure_operator(b,[1,2,1])
2*B[[[1, 1], [2]]] + 2*B[[[1, 2], [2]]] + 2*B[[[1, 3], [2]]] + 2*B[[[2, 1], [2]]] + 2*B[[[2, 2], [2]]] + 2*B[[[2, 3], [2]]] + 2*B[[[2, 3], [3]]] + 2*B[[[2, 3], [3]]]
\end{verbatim}

The Demazure operator is idempotent:

\begin{verbatim}
sage: b = C(T.module_generators[0]); b
B[[[1, 1, 1, 1]]]
sage: e = T.demazure_operator(b,[1]); e
B[[[1, 1, 1, 1]]] + B[[[1, 2, 2, 2]]] + B[[[2, 2, 2, 2]]]
sage: e == T.demazure_operator(e,[1])
True
sage: all(T.demazure_operator(T.demazure_operator(C(t),[1]),[1]) == T.demazure_operator(C(t),[1]) for t in T)
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> T = crystals.Tableaux('A1', shape=[4])
>>> C = CombinatorialFreeModule(QQ, T)
>>> b = C(T.module_generators[0]); b
B[[[1, 1, 1, 1]]]
\end{verbatim}
\begin{verbatim}
>>> e = T.demazure_operator(b,[Integer(1)]); e
B[[[1, 1, 1, 1]]] + B[[[1, 1, 1, 2]]] + B[[[1, 1, 2, 2]]] + B[[[2, 2, 2, 2]]] + B[[[2, 2, 2, 2]]]
>>> e == T.demazure_operator(e,[Integer(1)])
True
>>> all(T.demazure_operator(T.demazure_operator(C(t),[Integer(1)]),
  ...[Integer(1)])
== T.demazure_operator(C(t),[Integer(1)]) for t in T)
True

demazure_subcrystal (element, reduced_word, only_support=True)

Return the subcrystal corresponding to the application of Demazure operators \( D_i \) for \( i \) from reduced_word on element.

INPUT:

• element – an element of a free module indexed by the underlying crystal
• reduced_word – a reduced word of the Weyl group of the same type as the underlying crystal
• only_support – (default: True) only include arrows corresponding to the support of reduced_word

OUTPUT:

• the Demazure subcrystal

EXAMPLES:

\begin{verbatim}
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = T.highest_weight_vector()
sage: S = T.demazure_subcrystal(t, [1,2])
sage: list(S)
[[[1, 1], [2]], [[1, 2], [2]], [[1, 1], [3]], [[1, 2], [3]], [[2, 2], [3]]]
sage: S = T.demazure_subcrystal(t, [2,1])
sage: list(S)
[[[1, 1], [2]], [[1, 2], [2]], [[1, 1], [3]], [[1, 2], [3]], [[1, 3], [2]], [[1, 3], [3]]]

>>> from sage.all import *

>>> T = crystals.Tableaux(['A',Integer(2)], shape=[Integer(2),Integer(1)])

>>> t = T.highest_weight_vector()

>>> S = T.demazure_subcrystal(t, [Integer(1),Integer(2)])

>>> list(S)
[[[1, 1], [2]], [[1, 2], [2]], [[1, 1], [3]], [[1, 2], [3]], [[2, 2], [3]]]

>>> S = T.demazure_subcrystal(t, [Integer(2),Integer(1)])

>>> list(S)
[[[1, 1], [2]], [[1, 2], [2]], [[1, 1], [3]], [[1, 3], [2]], [[1, 3], [3]]]
\end{verbatim}

We construct an example where we don't only want the arrows indicated by the support of the reduced word:

\begin{verbatim}
sage: K = crystals.KirillovReshetikhin(['A',1,1], 1, 2)
sage: mg = K.module_generator()
sage: S = K.demazure_subcrystal(mg, [1])
sage: S.digraph().edges(sort=True)
((([[1, 1]], [[1, 2]], 1), ((((1, 2), [1, 1], 1))))
sage: S = K.demazure_subcrystal(mg, [1], only_support=False)
\end{verbatim}
\end{verbatim}
sage: S.digraph().edges(sort=True)

\[
\left(([[1, 1]], [[1, 2]], 1),
( [[1, 2]], [[1, 1]], 0),
( [[1, 2]], [[2, 2]], 1),
( [[2, 2]], [[1, 2]], 0)
\right)
\]

```python
>>> from sage.all import *

>>> K = crystals.KirillovReshetikhin([A,Integer(1),Integer(1)],
˓→Integer(1), Integer(2))

>>> mg = K.module_generator()

>>> S = K.demazure_subcrystal(mg, [Integer(1)])

>>> S.digraph().edges(sort=True)

\[
\left(([[1, 1]], [[1, 2]], 1),
( [[1, 2]], [[2, 2]], 1)
\right)
\]

```

```python
>>> S = K.demazure_subcrystal(mg, [Integer(1)], only_support=False)

>>> S.digraph().edges(sort=True)

\[
\left(([[1, 1]], [[1, 2]], 1),
( [[1, 2]], [[1, 1]], 0),
( [[1, 2]], [[2, 2]], 1),
( [[2, 2]], [[1, 2]], 0)
\right)
\]

dual_equivalence_graph (X=None, index_set=None, directed=True)

Return the dual equivalence graph indexed by index_set on the subset X of self.

Let \( b \in B \) be an element of weight 0, so \( \varepsilon_j(b) = \varphi_j(b) \) for all \( j \in I \), where \( I \) is the indexing set. We say \( b' \) is an \( i \)-elementary dual equivalence transformation of \( b \) (where \( i \in I \)) if

- \( \varepsilon_i(b) = 1 \) and \( \varepsilon_{i-1}(b) = 0 \), and
- \( b' = f_{i-1}f_if_{i}e_{i-1}e_i b \).

We can do the inverse procedure by interchanging \( i \) and \( i-1 \) above.

**Note:** If the index set is not an ordered interval, we let \( i-1 \) mean the index appearing before \( i \) in \( I \).

This definition comes from [As2008] Section 4 (where our \( \varphi_j(b) \) and \( \varepsilon_j(b) \) are denoted by \( e(b, j) \) and \( -\delta(b, j) \), respectively).

The dual equivalence graph of \( B \) is defined to be the colored graph whose vertices are the elements of \( B \) of weight 0, and whose edges of color \( i \) (for \( i \in I \)) connect pairs \( \{b, b'\} \) such that \( b' \) is an \( i \)-elementary dual equivalence transformation of \( b \).

**Note:** This dual equivalence graph is a generalization of \( \mathcal{G}(\mathcal{X}) \) in [As2008] Section 4 except we do not require \( \varepsilon_i(b) = 0,1 \) for all \( i \).

This definition can be generalized by choosing a subset \( X \) of the set of all vertices of \( B \) of weight 0, and restricting the dual equivalence graph to the vertex set \( X \).

**INPUT:**
- \( X \) – (optional) the vertex set \( X \) (default: the whole set of vertices of self of weight 0)
- \( \text{index\_set} \) – (optional) the index set \( I \) (default: the whole index set of self); this has to be a subset of the index set of self (as a list or tuple)
- \( \text{directed} \) – (default: True) whether to have the dual equivalence graph be directed, where the head of an edge \( b - b' \) is \( b \) and the tail is \( b' = f_{i-1}f_i e_{i-1}e_i b \)

**See also:**
sage.combinat.partition.Partition.dual_equivalence_graph()
EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',3], shape=[2, 2])
sage: G = T.dual_equivalence_graph()
sage: G.edges(sort=True)
[(([1, 3], [2, 4]), ([1, 2], [3, 4]), 2),
 ([1, 2], [3, 4]), ([1, 3], [2, 4]), 3)]
```

```python
sage: T = crystals.Tableaux(['A',4], shape=[3, 2])
sage: G = T.dual_equivalence_graph()
sage: G.edges(sort=True)
[(([1, 3, 5], [2, 4]), ([1, 3, 4], [2, 5]), 4),
 ([1, 3, 5], [2, 4]), ([1, 2, 5], [3, 4]), 2),
 ([1, 3, 4], [2, 5]), ([1, 2, 4], [3, 5]), 2),
 ([1, 2, 5], [3, 4]), ([1, 3, 5], [2, 4]), 3),
 ([1, 2, 4], [3, 5]), ([1, 2, 3], [4, 5]), 3),
 ([1, 2, 3], [4, 5]), ([1, 2, 4], [3, 5]), 4)]
```

```python
sage: T = crystals.Tableaux(['A',4], shape=[3, 1])
sage: G = T.dual_equivalence_graph(index_set=[1,2,3])
sage: G.vertices(sort=True)
[['[1, 3, 4], [2]'], ['[1, 2, 4], [3]'], ['[1, 2, 3], [4]']]
```

```python
>>> from sage.all import *
>>> T = crystals.Tableaux(['A',Integer(3)], shape=[Integer(2),Integer(2)])
>>> G = T.dual_equivalence_graph()
>>> G.edges(sort=True)
[(([1, 3], [2, 4]), ([1, 2], [3, 4]), 2),
 ([1, 2], [3, 4]), ([1, 3], [2, 4]), 3)]
```

```python
class TensorProducts (category, *args)
Bases: TensorProductsCategory
The category of regular crystals constructed by tensor product of regular crystals.
```

```python
extra_super_categories ()
EXAMPLES:
```
sage: RegularCrystals().TensorProducts().extra_super_categories()
[Category of regular crystals]

>>> from sage.all import *
>>> RegularCrystals().TensorProducts().extra_super_categories()
[Category of regular crystals]

additional_structure()

Return None.

Indeed, the category of regular crystals defines no new structure: it only relates $\varepsilon_a$ and $\varphi_a$ to $e_a$ and $f_a$ respectively.

See also:
Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

sage: RegularCrystals().additional_structure()

>>> from sage.all import *
>>> RegularCrystals().additional_structure()

example ($n=3$)

Returns an example of highest weight crystals, as per Category.example().

EXAMPLES:

sage: B = RegularCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1

>>> from sage.all import *
>>> B = RegularCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1

super_categories()

EXAMPLES:

sage: RegularCrystals().super_categories()
[Category of crystals]

>>> from sage.all import *
>>> RegularCrystals().super_categories()
[Category of crystals]
4.133 Regular Supercrystals

class sage.categories.regular_supercrystals.RegularSuperCrystals
    Bases: Category_singleton

    The category of crystals for super Lie algebras.

    EXAMPLES:

    sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
    sage: C = RegularSuperCrystals()
    sage: C
    Category of regular super crystals
    sage: C.super_categories()
    [Category of finite super crystals]

    Parents in this category should implement the following methods:
    • either an attribute _cartan_type or a method cartan_type
    • module_generators: a list (or container) of distinct elements that generate the crystal using $f_i$ and $e_i$

    Furthermore, their elements $x$ should implement the following methods:
    • $x.e(i)$ (returning $e_i(x)$)
    • $x.f(i)$ (returning $f_i(x)$)
    • $x.weight()$ (returning $wt(x)$)

    EXAMPLES:

    sage: from sage.misc.abstract_method import abstract_methods_of_class
    sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
    sage: abstract_methods_of_class(RegularSuperCrystals().element_class)
    {'optional': [], 'required': ['e', 'f', 'weight']}

    class ElementMethods
        Bases: object
        epsilon(i)
            Return $\varepsilon_i$ of self.

        EXAMPLES:
phi (i)

Return $\phi_i$ of self.

EXAMPLES:

```sage
sage: C = crystals.Tableaux(['A',[1,2]], shape=[2,1])
sage: c = C.an_element(); c
[-2, -2], [-1]
sage: c.epsilon(2)
0
sage: c.epsilon(0)
0
sage: c.epsilon(-1)
0
```

```python
>>> from sage.all import *

>>> C = crystals.Tableaux(['A', [Integer(1), Integer(2)]],
  \shape=[Integer(2), Integer(1)])

>>> c = C.an_element(); c
[-2, -2], [-1]

>>> c.epsilon(Integer(2))
0

>>> c.epsilon(Integer(0))
0

>>> c.epsilon(-Integer(1))
0
```

class TensorProducts (category, *args)

Bases: TensorProductsCategory

The category of regular crystals constructed by tensor product of regular crystals.

extra_super_categories ()

EXAMPLES:
sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: RegularSuperCrystals().TensorProducts().extra_super_categories()
[Category of regular super crystals]

>>> from sage.all import *
>>> from sage.categories.regular_supercrystals import RegularSuperCrystals
>>> RegularSuperCrystals().TensorProducts().extra_super_categories()
[Category of regular super crystals]

super_categories()
EXAMPLES:

sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: C = RegularSuperCrystals()

EXAMPLES:

4.134 Right modules

class sage.categories.right_modules.RightModules (base, name=None)
Bases: Category_over_base_ring

The category of right modules right modules over an rng (ring not necessarily with unit), i.e. an abelian group with right multiplication by elements of the rng

EXAMPLES:

sage: RightModules(QQ)
Category of right modules over Rational Field
sage: RightModules(QQ).super_categories()
[Category of commutative additive groups]

>>> from sage.all import *
>>> RightModules(QQ)
Category of right modules over Rational Field

>>> RightModules(QQ).super_categories()
[Category of commutative additive groups]

class ElementMethods
Bases: object

class ParentMethods
Bases: object

super_categories()
EXAMPLES:
4.135 Ring ideals

class sage.categories.ring_ideals.RingIdeals(R)

Bases: Category_ideal

The category of two-sided ideals in a fixed ring.

EXAMPLES:

sage: Ideals(Integers(200))
Category of ring ideals in Ring of integers modulo 200
sage: C = Ideals(IntegerRing()); C
Category of ring ideals in Integer Ring
sage: I = C([8,12,18])
Principal ideal (2) of Integer Ring

See also: CommutativeRingIdeals.

Todo:

- If useful, implement RingLeftIdeals and RingRightIdeals of which RingIdeals would be a subcategory.
- Make RingIdeals(R), return CommutativeRingIdeals(R) when R is commutative.

super_categories()

EXAMPLES:

sage: RingIdeals(ZZ).super_categories()
[Category of modules over Integer Ring]
sage: RingIdeals(QQ).super_categories()
[Category of vector spaces over Rational Field]

(continues on next page)
4.136 Rings

class sage.categories.rings.Rings(base_category)

Bases: CategoryWithAxiom_singleton

The category of rings

Associative rings with unit, not necessarily commutative

EXAMPLES:

sage: Rings()
Category of rings

sage: sorted(Rings().super_categories(), key=str)
[Category of rngs, Category of semirings]

sage: sorted(Rings().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive', 'Unital']

sage: Rings() is (CommutativeAdditiveGroups() & Monoids()).Distributive()
True

sage: Rings() is Rngs().Unital()
True

sage: Rings() is Semirings().AdditiveInverse()
True

Todo: (see Issue #sage_trac/wiki/CategoriesRoadMap)

- Make Rings() into a subcategory or alias of Algebras(ZZ);
- A parent P in the category Rings() should automatically be in the category Algebras(P).
Commutative
alias of CommutativeRings

Division
alias of DivisionRings

class ElementMethods
    Bases: object

    inverse_of_unit()
    Return the inverse of this element if it is a unit.

    OUTPUT:
    An element in the same ring as this element.

    EXAMPLES:

    sage: R.<x> = ZZ[]
    sage: S = R.quo(x^2 + x + 1)
    # needs sage.libs.pari
    sage: S(1).inverse_of_unit()
    # needs sage.libs.pari
    1

    >>> from sage.all import *
    >>> R = ZZ['x']; (x,) = R._first_ngens(1)
    >>> S = R.quo(x**Integer(2) + x + Integer(1))
    # needs sage.libs.pari
    >>> S(Integer(1)).inverse_of_unit()
    # needs sage.libs.pari
    1

    This method fails when the element is not a unit:

    sage: 2.inverse_of_unit()
    Traceback (most recent call last):
      ...
    ArithmeticError: inverse does not exist

    >>> from sage.all import *
    >>> Integer(2).inverse_of_unit()
    Traceback (most recent call last):
      ...
    ArithmeticError: inverse does not exist

    The inverse returned is in the same ring as this element:

    sage: a = -1
    sage: a.parent()
    Integer Ring
    sage: a.inverse_of_unit().parent()
    Integer Ring

    >>> from sage.all import *
    >>> a = -Integer(1)
    >>> a.parent()
    (continues on next page)
Note that this is often not the case when computing inverses in other ways:

```
sage: (~a).parent()
Rational Field
sage: (1/a).parent()
Rational Field
```

```from sage.all import *
>>> (~a).parent()
Rational Field
>>> (Integer(1)/a).parent()
Rational Field```

**is_unit()**

Return whether this element is a unit in the ring.

**Note:** This is a generic implementation for (non-commutative) rings which only works for the one element, its additive inverse, and the zero element. Most rings should provide a more specialized implementation.

**EXAMPLES:**

```
sage: # needs sage.modules
sage: MS = MatrixSpace(ZZ, 2)
sage: MS.one().is_unit()
True
sage: MS.zero().is_unit()
False
sage: MS([[1,2,3,4]]).is_unit()
False
```

```from sage.all import *
>>> # needs sage.modules
>>> MS = MatrixSpace(ZZ, Integer(2))
>>> MS.one().is_unit()
True
>>> MS.zero().is_unit()
False
>>> MS([[Integer(1),Integer(2),Integer(3),Integer(4)]]).is_unit()
False```

**class MorphismMethods**

**Bases:** object

**extend_to_fraction_field()**

Return the extension of this morphism to fraction fields of the domain and the codomain.

**EXAMPLES:**
\begin{verbatim}
sage: S.<x> = QQ[]
sage: f = S.hom([x + 1]); f
Ring endomorphism of Univariate Polynomial Ring in x over Rational Field
    Defn: x |--> x + 1
sage: g = f.extend_to_fraction_field(); g
# needs sage.libs.singular
Ring endomorphism of Fraction Field of Univariate Polynomial Ring in x
over Rational Field
    Defn: x |--> x + 1
sage: g(x)
# needs sage.libs.singular
x + 1
sage: g(1/x)
# needs sage.libs.singular
1/(x + 1)
\end{verbatim}

If this morphism is not injective, it does not extend to the fraction field and an error is raised:

\begin{verbatim}
sage: f = GF(5).coerce_map_from(ZZ)
sage: f.extend_to_fraction_field()
Traceback (most recent call last):
... ValueError: the morphism is not injective
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> f = S.hom([x + Integer(1)]); f
Ring endomorphism of Univariate Polynomial Ring in x over Rational Field
    Defn: x |--> x + 1
>>> g = f.extend_to_fraction_field(); g
# needs sage.libs.singular
Ring endomorphism of Fraction Field of Univariate Polynomial Ring in x
over Rational Field
    Defn: x |--> x + 1
>>> g(x)
# needs sage.libs.singular
x + 1
>>> g(Integer(1)/x)
# needs sage.libs.singular
1/(x + 1)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> f = GF(Integer(5)).coerce_map_from(ZZ)
>>> f.extend_to_fraction_field()
Traceback (most recent call last):
... ValueError: the morphism is not injective
\end{verbatim}

**is_injective()**

Return whether or not this morphism is injective.

**EXAMPLES:**

\begin{verbatim}
sage: # needs sage.libs.singular
sage: R.<x,y> = QQ[]
\end{verbatim}
sage: R.hom([x, y^2], R).is_injective()
True
sage: R.hom([x, x^2], R).is_injective()
False
sage: S.<u,v> = R.quotient(x^3*y)
sage: R.hom([v, u], S).is_injective()
False
sage: S.hom([-u, v], S).is_injective()
True
sage: S.cover().is_injective()
False

>>> from sage.all import *

## needs sage.libs.singular

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> R.hom([x, y**Integer(2)], R).is_injective()
True
>>> R.hom([x, x**Integer(2)], R).is_injective()
False
>>> S = R.quotient(x**Integer(3)*y, names=(u, v,)); (u, v,) = S._
˓→first_ngens(2)
>>> R.hom([v, u], S).is_injective()
False
>>> S.hom([-u, v], S).is_injective()
True
>>> S.cover().is_injective()
False

If the domain is a field, the homomorphism is injective:

sage: K.<x> = FunctionField(QQ)
sage: L.<y> = FunctionField(QQ)
sage: f = K.hom([y]); f
Function Field morphism:
  From: Rational function field in x over Rational Field
  To:   Rational function field in y over Rational Field
  Defn: x |--> y
sage: f.is_injective()
True

>>> from sage.all import *

>>> K = FunctionField(QQ, names=('x',)); (x,) = K._first_ngens(1)
>>> L = FunctionField(QQ, names=('y',)); (y,) = L._first_ngens(1)
>>> f = K.hom([y]); f
Function Field morphism:
  From: Rational function field in x over Rational Field
  To:   Rational function field in y over Rational Field
  Defn: x |--> y
>>> f.is_injective()
True

Unless the codomain is the zero ring:

sage: codomain = Integers(1)
sage: f = QQ.hom([Zmod(1)(0)], check=False)
sage: f.is_injective()

(continues on next page)
Homomorphism from rings of characteristic zero to rings of positive characteristic can not be injective:

```
sage: R.<x> = ZZ[]
sage: f = R.hom([GF(3)(1)]); f
Ring morphism:
   From: Univariate Polynomial Ring in x over Integer Ring
   To:   Finite Field of size 3
   Defn: x |--> 1
sage: f.is_injective()
False
```

A morphism whose domain is an order in a number field is injective if the codomain has characteristic zero:

```
sage: K.<x> = FunctionField(QQ)
sage: f = ZZ.hom(K); f
Composite map:
   From: Integer Ring
   To:   Rational function field in x over Rational Field
   Defn: Conversion via FractionFieldElement_1poly_field map:
      From: Integer Ring
      To:   Fraction Field of Univariate Polynomial Ring in x
            over Rational Field
      then
      Isomorphism:
         From: Fraction Field of Univariate Polynomial Ring in x
               over Rational Field
         To:   Rational function field in x over Rational Field
sage: f.is_injective()
True
```
Defn: Conversion via FractionFieldElement_1poly_field map:
From: Integer Ring
To: Fraction Field of Univariate Polynomial Ring in x
over Rational Field
then
Isomorphism:
From: Fraction Field of Univariate Polynomial Ring in x
over Rational Field
To: Rational function field in x over Rational Field

```python
>>> f.is_injective()
True
```

A coercion to the fraction field is injective:

```python
sage: R = ZpFM(3)  # needs sage.rings.padics
sage: R.fraction_field().coerce_map_from(R).is_injective()
True
```

```python
>>> from sage.all import *
```

```python
>>> R = ZpFM(Integer(3))  # needs sage.rings.padics
>>> R.fraction_field().coerce_map_from(R).is_injective()
True
```

NoZeroDivisors
alias of Domains

class ParentMethods
Bases: object

`bracket(x, y)`

Return the Lie bracket \([x, y] = xy - yx\) of \(x\) and \(y\).

**INPUT:**
- \(x, y\) – elements of \(\text{self}\)

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: F = AlgebrasWithBasis(QQ).example()
sage: F
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field
sage: a, b, c = F.algebra_generators()
sage: F.bracket(a, b)
B[word: ab] - B[word: ba]
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.combinat sage.modules
>>> F = AlgebrasWithBasis(QQ).example()
>>> F
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field
>>> a, b, c = F.algebra_generators()
>>> F.bracket(a, b)
B[word: ab] - B[word: ba]
```
This measures the default of commutation between $x$ and $y$. $F$ endowed with the bracket operation is a Lie algebra; in particular, it satisfies Jacobi's identity:

```
sage: (F.bracket(F.bracket(a,b), c) + F.bracket(F.bracket(b,c), a))  
    + F.bracket(F.bracket(c,a), b))
```

```
>>> from sage.all import *

>>> (F.bracket(F.bracket(a,b), c) + F.bracket(F.bracket(b,c), a))  
    + F.bracket(F.bracket(c,a), b))
```

**characteristic()**

Return the characteristic of this ring.

**EXAMPLES:**

```
sage: QQ.characteristic()
0
sage: GF(19).characteristic()
19
sage: Integers(8).characteristic()
8
sage: Zp(5).characteristic()  
    + F.bracket(F.bracket(c,a), b))
```

```
>>> from sage.all import *

>>> QQ.characteristic()
0

>>> GF(Integer(19)).characteristic()
19
>>> Integers(Integer(8)).characteristic()
8
>>> Zp(Integer(5)).characteristic()  
    + F.bracket(F.bracket(c,a), b))
```

**free_module** *(base=None, basis=None, map=True)*

Return a free module $V$ over the specified subring together with maps to and from $V$.

The default implementation only supports the case that the base ring is the ring itself.

**INPUT:**

- `base` – a subring $R$ so that this ring is isomorphic to a finite-rank free $R$-module $V$
- `basis` – (optional) a basis for this ring over the base
- `map` – boolean (default True), whether to return $R$-linear maps to and from $V$

**OUTPUT:**

- A finite-rank free $R$-module $V$
- An $R$-module isomorphism from $V$ to this ring (only included if map is True)
- An $R$-module isomorphism from this ring to $V$ (only included if map is True)

**EXAMPLES:**

```
sage: # needs sage.modules

sage: R.<x> = QQ[]
```

(continues on next page)
ideal(*args, **kws)

Create an ideal of this ring.

**Note:** The code is copied from the base class `Ring`. This is because there are rings that do not inherit from that class, such as matrix algebras. See Issue #7797.

**INPUT:**

- An element or a list/tuple/sequence of elements.
- `coerce` (optional bool, default True): First coerce the elements into this ring.
- `side`, optional string, one of "twosided" (default), "left", "right": determines whether the resulting ideal is twosided, a left ideal or a right ideal.

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(QQ, 2, 2)
sage: isinstance(MS, Ring)
False
sage: MS in Rings()
True
sage: MS.ideal(2)
Twosided Ideal
   (([2 0],
     [0 2]))
```
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

\begin{sagecell}
MS.ideal([MS.0, MS.1], side='right')
\end{sagecell}

Right Ideal

\begin{verbatim}
( [1 0]
 [0 0],

 [0 1]
 [0 0]
)
\end{verbatim}

of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

\begin{sagecell}
from sage.all import *
# needs sage.modules
MS = MatrixSpace(QQ, Integer(2), Integer(2))
isinstance(MS, Ring)
False
MS in Rings()
True
MS.ideal(Integer(2))
Twosided Ideal

( [2 0]
 [0 2]
)

of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

MS.ideal([MS.gen(0), MS.gen(1)], side='right')
Right Ideal

( [1 0]
 [0 0],

 <BLANKLINE>
 [0 1]
 [0 0]
)

of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

\end{sagecell}

\textbf{ideal_monoid()}

The monoid of the ideals of this ring.

\textbf{Note: } The code is copied from the base class of rings. This is since there are rings that do not inherit from that class, such as matrix algebras. See \textbf{Issue #7797}.

\textbf{EXAMPLES:}

\begin{sagecell}
sage: # needs sage.modules
sage: MS = MatrixSpace(QQ, 2, 2)
isinstance(MS, Ring)
False
sage: MS in Rings()
True
sage: MS.ideal_monoid()
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
\end{sagecell}
>>> from sage.all import *
>>> # needs sage.modules
>>> MS = MatrixSpace(QQ, Integer(2), Integer(2))
>>> isinstance(MS, Ring)
False
>>> MS in Rings()
True
>>> MS.ideal_monoid()
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices
over Rational Field

Note that the monoid is cached:

sage: MS.ideal_monoid() is MS.ideal_monoid()  #...
← needs sage.modules
True

is_commutative()
Return whether the ring is commutative.

The answer is True only if the category is a sub-category of CommutativeRings.

It is recommended to use instead R in Rings().Commutative().

EXAMPLES:

sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -1, -1) ˓→ # needs sage.combinat sage.modules
sage: Q.is_commutative() ˓→ # needs sage.combinat sage.modules
False

is_integral_domain(proof=True)
Return True if this ring is an integral domain.

INPUT:
• proof – (default: True) Determines what to do in unknown cases

ALGORITHM:
If the parameter proof is set to True, the returned value is correct but the method might throw an error. Otherwise, if it is set to False, the method returns True if it can establish that self is an integral domain and False otherwise.

EXAMPLES:
This illustrates the use of the `proof` parameter:
sage: S.is_integral_domain(proof=False)                   # needs sage.libs.singular
False

```python
>>> from sage.all import *
>>> R = ZZ['a, b']; (a, b,) = R._first_ngens(2)
>>> S = R.quo((b**Integer(3)), names=('x', 'y',)); (x, y,) = S._first_ngens(2)  # needs sage.libs.singular
>>> S.is_integral_domain(proof=True)  # needs sage.libs.singular
Traceback (most recent call last):
... Not ImplementedError
```  

```
```python
>>> S.is_integral_domain(proof=False)  # needs sage.libs.singular
False
```

**is_noetherian()**

Return True if this ring is Noetherian.

**EXAMPLES:**

```python
sage: QQ.is_noetherian()  
True
sage: ZZ.is_noetherian()  
True
```

```python
>>> from sage.all import *
>>> QQ.is_noetherian()  
True
```

```python
>>> ZZ.is_noetherian()  
True
```

**is_ring()**

Return True, since this in an object of the category of rings.

**EXAMPLES:**

```python
sage: Parent(QQ, category=Rings()).is_ring()  
True
```

```python
>>> from sage.all import *
>>> Parent(QQ, category=Rings()).is_ring()  
True
```

**is_zero()**

Return True if this is the zero ring.

**EXAMPLES:**

```python
sage: Integers(1).is_zero()  
True
sage: Integers(2).is_zero()  
False
sage: QQ.is_zero()  
(continues on next page)
\texttt{sage}: \texttt{R.<x> = ZZ[]}
\texttt{sage}: \texttt{R.quo(1).is_zero()}
\texttt{True}
\texttt{sage}: \texttt{R.<x> = GF(101)[]}
\texttt{sage}: \texttt{R.quo(77).is_zero()}
\texttt{True}
\texttt{sage}: \texttt{R.quo(x^2 + 1).is_zero() \# needs sage.libs.pari}
\texttt{False}

\begin{verbatim}
>>> from sage.all import *
>>> Integers(Integer(1)).is_zero()
\texttt{True}
>>> Integers(Integer(2)).is_zero()
\texttt{False}
>>> QQ.is_zero()
\texttt{False}
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> R.quo(Integer(1)).is_zero()
\texttt{True}
>>> R = GF(Integer(101))['x']; (x,) = R._first_ngens(1)
>>> R.quo(Integer(77)).is_zero()
\texttt{True}
>>> R.quo(x**Integer(2) + Integer(1)).is_zero()  \# needs sage.libs.pari
\texttt{False}
\end{verbatim}

\texttt{quo}(I, \texttt{names}=None, **\texttt{kwds})

Quotient of a ring by a two-sided ideal.

\textbf{Note:} This is a synonym for \texttt{quotient()}.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: MS = MatrixSpace(QQ, 2) \# needs sage.modules
\texttt{---}
\texttt{sage}: I = MS * MS.gens() * MS \# needs sage.modules
\texttt{---}
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2)) \# needs sage.modules
>>> I = MS * MS.gens() * MS \# needs sage.modules
\end{verbatim}

\texttt{MS} is not an instance of \texttt{Ring}.

However it is an instance of the parent class of the category of rings. The quotient method is inherited from there:

\begin{verbatim}
\texttt{sage}: isinstance(MS, sage.rings.ring.Ring) \# needs sage.modules
\texttt{False}
\end{verbatim}
sage: isinstance(MS, Rings().parent_class)   # needs sage.modules
True

sage: MS.quo(I, names=['a', 'b', 'c', 'd'])   # needs sage.modules
Quotient of Full MatrixSpace of 2 by 2 dense matrices over Rational Field by the ideal
([1 0]
[0 0],
[0 1]
[0 0],
[0 0]
[1 0],
[0 0]
[0 1])

>>> from sage.all import *
>>> isinstance(MS, sage.rings.ring.Ring)   # needs sage.modules
False

>>> isinstance(MS, Rings().parent_class)   # needs sage.modules
True

sage: MS.quo(I, names=['a', 'b', 'c', 'd'])   # needs sage.modules
Quotient of Full MatrixSpace of 2 by 2 dense matrices over Rational Field by the ideal
([1 0]
[0 0],
[0 1]
[0 0],
[0 0]
[1 0],
[0 0]
[0 1])

A test with a subclass of Ring:

```
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quo((x^2, y))
sage: S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2, y)
sage: S.gens()
(a, 0)
```
\begin{quote}
\texttt{sage: a == b}
\texttt{False}
\end{quote}

\begin{quote}
\begin{verbatim}
from sage.all import *
# needs sage.libs.singular
R = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = R._first_ngens(2)
S = R.quo((x**Integer(2), y), names=('a', 'b',)); (a, b,) = S._first_ngens(2)
S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
S.gens()
(a, 0)
\end{verbatim}
\texttt{a == b}
\texttt{False}
\end{quote}

\textbf{quotient} \texttt{(I, names=None, **kwds)}

Quotient of a ring by a two-sided ideal.

INPUT:

- \texttt{I} – A two-sided ideal of this ring.
- \texttt{names} – (optional) names of the generators of the quotient (if there are multiple generators, you can specify a single character string and the generators are named in sequence starting with 0).
- further named arguments that may be passed to the quotient ring constructor.

EXAMPLES:

Usually, a ring inherits a method \texttt{sage.rings.ring.Ring.quotient()}. So, we need a bit of effort to make the following example work with the category framework:
zbar
sage: Q.0*Q.1
xbar*ybar
sage: Q.0*Q.1*Q.0
0

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = F._first_ ˓→gens(3)
>>> from sage.rings.noncommutative_ideals import Ideal_nc
>>> from itertools import product
>>> class PowerIdeal(Ideal_nc):
...     def __init__(self, R, n):
...         self._power = n
...         Ideal_nc.__init__(self, R, [R.prod(m) ˓→
...             for m in product(R.gens(), repeat=n)])
...     def reduce(self, x):
...         R = self.ring()
...         return add([c*R(m) for m, c in x ˓→
...             if len(m) < self._power], R(Integer(0)))

>>> I = PowerIdeal(F, Integer(3))
>>> Q = Rings().parent_class.quotient(F, I); Q
Quotient of Free Algebra on 3 generators (x, y, z) over Rational Field
by the ideal (x^3, x^2*y, x^2*z, x*y*x, x*y^2, x*y*z, x*z*x, ˓→
x*z*y, x*z^2, y*x^2, y*x*y, y*x*z, y^2*x, y^3,
y^2*z, y*z*x, y*z*y, y*z^2, z*x^2, z*x*y, z*x*z, ˓→
z*y*x, z*y^2, z*y*z, z^2*x, z^2*y, z^3)

>>> Q.gen(0)
xbar
>>> Q.gen(1)
ybar
>>> Q.gen(2)
zbar
>>> Q.gen(0)*Q.gen(1)
xbar*ybar
>>> Q.gen(0)*Q.gen(1)*Q.gen(0)
0

An example with polynomial rings:

sage: R.<x> = PolynomialRing(ZZ)
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient(I, 'a')
sage: S.gens()
(a,)
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quotient((x^2, y))
sage: S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
sage: S.gens()
(a, 0)
sage: a == b
False

```python
>>> from sage.all import *
>>> R = PolynomialRing(ZZ, names=('x',)); (x,) = R._first_ngens(1)
>>> I = R.ideal([Integer(4) + Integer(3)*x + x**Integer(2), Integer(1) +
                x**Integer(2)])
>>> S = R.quotient(I, 'a')
>>> S.gens()
(a,)

>>> # needs sage.libs.singular
>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = R._
                first_ngens(2)
>>> S = R.quotient((x**Integer(2), y), names=('a', 'b',)); (a, b,) = S._
                first_ngens(2)
>>> S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
>>> S.gens()
(a, 0)
>>> a == b
False
```

**quotient_ring**(I, names=None, **kwds)

Quotient of a ring by a two-sided ideal.

**Note:** This is a synonym for **quotient()**.

**INPUT:**
- I – an ideal of R
- names – (optional) names of the generators of the quotient. (If there are multiple generators, you can specify a single character string and the generators are named in sequence starting with 0.)
- further named arguments that may be passed to the quotient ring constructor.

**OUTPUT:**
- R/I – the quotient ring of R by the ideal I

**EXAMPLES:**

```python
sage: MS = MatrixSpace(QQ, 2)  # needs sage.modules
sage: I = MS * MS.gens() * MS  # needs sage.modules

>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2))  # needs sage.modules
>>> I = MS * MS.gens() * MS  # needs sage.modules
```

MS is not an instance of **Ring**, but it is an instance of the parent class of the category of rings. The quotient method is inherited from there:
sage: isinstance(MS, sage.rings.ring.Ring)  #-- needs sage.modules
False
sage: isinstance(MS, Rings().parent_class)  #-- needs sage.modules
True
sage: MS.quotient_ring(I, names=['a','b','c','d'])  #-- needs sage.modules
Quotient of Full MatrixSpace of 2 by 2 dense matrices
over Rational Field by the ideal
( [1 0]
[0 0],

[0 1]
[0 0],

[0 0]
[1 0],

[0 0]
[0 1]
)

>>> from sage.all import *

>>> isinstance(MS, sage.rings.ring.Ring)  #-- needs sage.modules
False
>>> isinstance(MS, Rings().parent_class)  #-- needs sage.modules
True
>>> MS.quotient_ring(I, names=['a','b','c','d'])  #-- needs sage.modules
Quotient of Full MatrixSpace of 2 by 2 dense matrices
over Rational Field by the ideal
( [1 0]
[0 0],

[0 1]
[0 0],

[0 0]
[1 0],

[0 0]
[0 1]
)

A test with a subclass of Ring:

sage: R.<x> = PolynomialRing(ZZ)
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I, 'a')
sage: S.gens()
(a,)

(continues on next page)
```python
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = R.quotient_ring((x^2, y))
sage: S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
sage: S.gens()
(a, 0)
sage: a == b
False
```

```python
>>> from sage.all import *
>>> R = PolynomialRing(ZZ, names=('x',)); (x,) = R._first_ngens(1)
>>> I = R.ideal([Integer(4) + Integer(3)*x + x**Integer(2), Integer(1) +
   →x**Integer(2)]
>>> S = R.quotient_ring(I, 'a')
>>> S.gens()
(a,

>>> # needs sage.libs.singular
>>> R = PolynomialRing(QQ,Integer(2), names=('x', y,)); (x, y,) = R._
   →first_ngens(2)
>>> S = R.quotient_ring((x**Integer(2), y), names=('a', 'b',)); (a, b,) =␣
   →S._first_ngens(2)
>>> S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
>>> S.gens()
(a, 0)
>>> a == b
False
```

```python
class SubcategoryMethods
Bases: object

Division()

Return the full subcategory of the division objects of self.

A ring satisfies the division axiom if all non-zero elements have multiplicative inverses.

EXAMPLES:

```python
sage: Rings().Division()
Category of division rings
sage: Rings().Commutative().Division()
Category of fields
```

```python
>>> from sage.all import *
>>> Rings().Division()
Category of division rings
>>> Rings().Commutative().Division()
Category of fields
```

NoZeroDivisors()

Return the full subcategory of the objects of self having no nonzero zero divisors.
A zero divisor in a ring $R$ is an element $x \in R$ such that there exists a nonzero element $y \in R$ such that $x \cdot y = 0$ or $y \cdot x = 0$ (see Wikipedia article Zero_divisor).

**EXAMPLES:**

```python
sage: Rings().NoZeroDivisors()
Category of domains
```

---

### 4.137 Rngs

**class** `sage.categories.rngs.Rngs(base_category)`

**Bases:** `CategoryWithAxiom_singleton`

The category of rngs.

An rng $(S, +, *)$ is similar to a ring but not necessarily unital. In other words, it is a combination of a commutative additive group $(S, +)$ and a multiplicative semigroup $(S, *)$, where $*$ distributes over $+$.

**EXAMPLES:**

```python
sage: C = Rngs(); C
Category of rngs
sage: sorted(C.super_categories(), key=str)
[Category of associative additive commutative additive associative additive...
unital distributive magmas and additive magmas,
Category of commutative additive groups]
```

```python
sage: C.is(Subsets() & Rings()).Unital()
Category of rings
```

---

4.137. Rngs 1179
Unital

alias of Rings

4.138 R-trivial semigroups

```python
class sage.categories.r_trivial_semigroups.RTrivialSemigroups(base_category):
    Bases: CategoryWithAxiom

    Commutative_extra_super_categories()
    Implement the fact that a commutative $R$-trivial semigroup is $J$-trivial.

    EXAMPLES:
    sage: Semigroups().RTrivial().Commutative_extra_super_categories()
    [Category of j trivial semigroups]
    >>> from sage.all import *
    >>> Semigroups().RTrivial().Commutative_extra_super_categories()
    [Category of j trivial semigroups]
```

```python
elemental_super_categories()
    Implement the fact that a $R$-trivial semigroup is $H$-trivial.

    EXAMPLES:
    sage: Semigroups().RTrivial().elemental_super_categories()
    [Category of h trivial semigroups]
    >>> from sage.all import *
    >>> Semigroups().RTrivial().elemental_super_categories()
    [Category of h trivial semigroups]
```

4.139 Schemes

```python
class sage.categories.schemes.AbelianVarieties(base):
    Bases: Schemes_over_base

    The category of abelian varieties over a given field.

    EXAMPLES:
    sage: AbelianVarieties(QQ)
    Category of abelian varieties over Rational Field
    sage: AbelianVarieties(ZZ)
    Traceback (most recent call last):
    ... ValueError: category of abelian varieties is only defined over fields
    >>> from sage.all import *
    >>> AbelianVarieties(QQ)
    Category of abelian varieties over Rational Field
    ```
>>> AbelianVarieties(ZZ)
Traceback (most recent call last):
...
ValueError: category of abelian varieties is only defined over fields

**class Homsets (category, *args)**

- Bases: :class:`HomsetsCategory`

  Overloaded :class:`Homsets` class to register the homset as an additive abelian group.

  .. EXAMPLES::

    sage: AbelianVarieties(QQ).Homsets().is_subcategory(CommutativeAdditiveGroups())
    True

    >>> from sage.all import *
    >>> AbelianVarieties(QQ).Homsets().is_subcategory(CommutativeAdditiveGroups())
    True

**class Endset (base_category)**

- Bases: :class:`CategoryWithAxiom`

  Overloaded :class:`Endset` class to register the endset as a ring.

  sage: AbelianVarieties(QQ).Endsets().is_subcategory(Rings())
  True

  .. extra_super_categories ()

    Register the endset as a ring.

    .. EXAMPLES::

      sage: End(EllipticCurve(j=1)) in Rings()
      True

      >>> from sage.all import *
      >>> End(EllipticCurve(j=Integer(1))) in Rings()
      True

  .. extra_super_categories ()

    Register the homset as an additive abelian group.

    .. EXAMPLES::

      sage: Hom(EllipticCurve(j=1), EllipticCurve(j=2)) in CommutativeAdditiveGroups()
      True

      >>> from sage.all import *
      >>> Hom(EllipticCurve(j=Integer(1)), EllipticCurve(j=Integer(2))) in CommutativeAdditiveGroups()
      True

**base_scheme ()

  .. EXAMPLES::

4.139. Schemes
sage: Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring

>>> from sage.all import *
>>> Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring

super_categories()
EXCEPTIONS:
sage: AbelianVarieties(QQ).super_categories()
[Category of schemes over Rational Field,
Category of commutative additive groups]

>>> from sage.all import *
>>> AbelianVarieties(QQ).super_categories()
[Category of schemes over Rational Field,
Category of commutative additive groups]

class sage.categories.schemes.Jacobians(base)

Bases: Schemes_over_Base
The category of Jacobians attached to curves or function fields.

EXAMPLES:
sage: Jacobians(QQ)
Category of Jacobians over Rational Field

>>> from sage.all import *
>>> Jacobians(QQ)
Category of Jacobians over Rational Field

class ParentMethods

Bases: object

base_curve()
Return the curve to which this Jacobian is attached.

EXAMPLES:
sage: # needs sage.rings.function_field
sage: K.<x> = FunctionField(GF(2))
sage: J = K.jacobian()
sage: J.base_curve()
Rational function field in x over Finite Field of size 2

>>> from sage.all import *
>>> # needs sage.rings.function_field
>>> K = FunctionField(GF(Integer(2)), names=('x',)); (x,) = K._first_...
>>> J = K.jacobian()
>>> J.base_curve()
Rational function field in x over Finite Field of size 2
**base_scheme()**

Return the base scheme of this Jacobians category.

EXAMPLES:

```
sage: Jacobians(QQ).base_scheme()
Spectrum of Rational Field
```

```
>>> from sage.all import *
>>> Jacobians(QQ).base_scheme()
Spectrum of Rational Field
```

**super_categories()**

Return the super categories of this Jacobians category.

EXAMPLES:

```
sage: Jacobians(QQ).super_categories()
[Category of abelian varieties over Rational Field]
```

```
>>> from sage.all import *
>>> Jacobians(QQ).super_categories()
[Category of abelian varieties over Rational Field]
```

class sage.categories.schemes.Schemes

Bases: Category

The category of all schemes.

EXAMPLES:

```
sage: Schemes()
Category of schemes
```

```
>>> from sage.all import *
>>> Schemes()
Category of schemes
```

Schemes can also be used to construct the category of schemes over a given base:

```
sage: Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

```
sage: Schemes(ZZ)
Category of schemes over Integer Ring
```

```
>>> from sage.all import *
>>> Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

```
>>> Schemes(ZZ)
Category of schemes over Integer Ring
```

Todo: Make Schemes() a singleton category (and remove Schemes from the workaround in category_types.Category_over_base._test_category_over_bases()).
This is currently incompatible with the dispatching below.

**super_categories()**

EXAMPLES:

```
sage: Schemes().super_categories()
[Category of sets]
```

```
>>> from sage.all import *
>>> Schemes().super_categories()
[Category of sets]
```

### class sage.categories.schemes.Schemes_over_base

**Bases:** *Category_over_base*

The category of schemes over a given base scheme.

**EXAMPLES:**

```
sage: Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

```
>>> from sage.all import *
>>> Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

### base_scheme()

**EXAMPLES:**

```
sage: Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring
```

```
>>> from sage.all import *
>>> Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring
```

### super_categories()

**EXAMPLES:**

```
sage: Schemes(Spec(ZZ)).super_categories()
[Category of schemes]
```

```
>>> from sage.all import *
>>> Schemes(Spec(ZZ)).super_categories()
[Category of schemes]
```
4.140 Semigroups

class sage.categories.semigroups.Semigroups(base_category)
    Bases: CategoryWithAxiom_singleton

The category of (multiplicative) semigroups.

A semigroup is an associative magma, that is a set endowed with a multiplicative binary operation \( \ast \) which is associative (see Wikipedia article Semigroup).

The operation \( \ast \) is not required to have a neutral element. A semigroup for which such an element exists is a monoid.

EXAMPLES:

```
sage: C = Semigroups(); C
Category of semigroups

sage: C.super_categories()
[Category of magmas]

sage: C.all_super_categories()
[Category of semigroups, Category of magmas,
 Category of sets, Category of sets with partial maps, Category of objects]

sage: C.axioms()
frozenset({'Associative'})

sage: C.example()
An example of a semigroup: the left zero semigroup
```

>>> from sage.all import *
>>> C = Semigroups(); C
Category of semigroups

```
Family ('a', 'b', 'c', 'd')

```
sage: M.algebra(ZZ).algebra_generators() #...
˓→needs sage.modules
Family (B['a'], B['b'], B['c'], B['d'])
```

```python
>>> from sage.all import *
" type="quote"
>>>
M = FiniteSemigroups().example(); M
An example of a finite semigroup:
the left regular band generated by ('a', 'b', 'c', 'd')
```

```python
>>> M.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

```python
>>> M.algebra(ZZ).algebra_generators() #...
˓→needs sage.modules
Family (B['a'], B['b'], B['c'], B['d'])
```

gen (i=0)

Return the i-th generator of self.

EXAMPLES:

```
sage: A = GL(3, GF(7)).algebra(ZZ) #...
˓→needs sage.modules
sage: A.gen(0) #...
˓→needs sage.groups sage.libs.pari sage.modules
[3 0 0]
[0 1 0]
[0 0 1]
```

```python
>>> from sage.all import *
" type="quote"
>>>
A = GL(Integer(3), GF(Integer(7))).algebra(ZZ)  # needs sage.modules
>>> A.gen(Integer(0))  # needs sage.groups sage.libs.pari sage.modules
[3 0 0]
[0 1 0]
[0 0 1]
```

gens ()

Return the generators of self.

EXAMPLES:

```
sage: a, b = SL2Z.algebra(ZZ).gens(); a, b #...
˓→needs sage.groups sage.modular sage.modules
(\[
[ 0 -1] \\
[ 1 0],
[1 1] \\
[0 1])
sage: 2*a + b #...
˓→needs sage.groups sage.modular sage.modules
2*[ 0 -1] + 
[1 1] 
[0 1]
```
ngens()  
Return the number of generators of self.

EXAMPLES:

```python
sage: SL2Z.algebra(ZZ).ngens()  # needs sage.groups sage.modular sage.modules
2
sage: DihedralGroup(4).algebra(RR).ngens()  # needs sage.groups sage.modules
2
```

product_on_basis(g1, g2)

Product, on basis elements, as per MagmaticAlgebras.WithBasis.ParentMethods.product_on_basis().

The product of two basis elements is induced by the product of the corresponding elements of the group.

EXAMPLES:

```python
sage: S = FiniteSemigroups().example(); S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(QQ)  # needs sage.modules
sage: a, b, c, d = A.algebra_generators()  # needs sage.modules
sage: a * b + b * d * c * d  # needs sage.modules
B['ab'] + B['bdc']
```

(continues on next page)
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')

```python
>>> A = S.algebra(QQ)  # needs sage.modules
>>> a, b, c, d = A.algebra_generators()  # needs sage.modules
>>> a * b + b * d * c * d  # needs sage.modules
B['ab'] + B['bdc']
```

**regular_representation** *(side='left')*

Return the regular representation of self.

**INPUT:**
- **side** – (default: "left") whether this is the "left" or "right" regular representation

**EXAMPLES:**

```python
sage: # needs sage.groups
tsage: G = groups.permutation.Dihedral(4)
tsage: A = G.algebra(QQ)  # needs sage.modules
tsage: V = A.regular_representation()  # needs sage.modules
tsage: V == G.regular_representation(QQ)  # needs sage.modules
True
```

```python
>>> from sage.all import *
>>> # needs sage.groups
>>> G = groups.permutation.Dihedral(Integer(4))
>>> A = G.algebra(QQ)  # needs sage.modules
>>> V = A.regular_representation()  # needs sage.modules
>>> V == G.regular_representation(QQ)  # needs sage.modules
True
```

**representation** *(module, on_basis, side='left', *args, **kwargs)*

Return a representation of self on module with the action of the semigroup given by on_basis.

**INPUT:**
- **module** – a module with a basis
- **on_basis** – function which takes as input \( g, m \) where \( g \) is an element of the semigroup and \( m \) is an element of the indexing set for the basis, and returns the result of \( g \) acting on \( m \)
- **side** – (default: "left") whether this is a "left" or "right" representation

**EXAMPLES:**

```python
sage: G = groups.permutation.Dihedral(5)
sage: CFM = CombinatorialFreeModule(GF(2), [1, 2, 3, 4, 5])
sage: A = G.algebra(GF(2))
sage: R = A.representation(CFM, lambda g, i: CFM.basis()[g(i)], side='right')
sage: R
Representation of Dihedral group of order 10 as a permutation group indexed by {1, 2, 3, 4, 5} over Finite Field of size 2
```
from sage.all import *

G = groups.permutation.Dihedral(Integer(5))

CFM = CombinatorialFreeModule(GF(Integer(2)), [Integer(1), Integer(2), Integer(3), Integer(4), Integer(5)])

A = G.algebra(GF(Integer(2)))

R = A.representation(CFM, lambda g, i: CFM.basis()[g(i)], side='right')

R

trivial_representation(side='twosided')

Return the trivial representation of self.

INPUT:
• side – ignored

EXAMPLES:

sage: # needs sage.groups
g = groups.permutation.Dihedral(4)
sage: A = G.algebra(QQ)  # needs sage.modules
sage: V = A.trivial_representation()  # needs sage.modules
sage: V == G.trivial_representation(QQ)  # needs sage.modules
True

extra_super_categories()

Implement the fact that the algebra of a semigroup is indeed a (not necessarily unital) algebra.

EXAMPLES:

sage: Semigroups().Algebras(QQ).extra_super_categories() [Category of semigroups]
sage: Semigroups().Algebras(QQ).super_categories() [Category of associative algebras over Rational Field, Category of magma algebras over Rational Field]
Aperiodic
    alias of AperiodicSemigroups

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory
    extra_super_categories()

    Implement the fact that a Cartesian product of semigroups is a semigroup.

    EXAMPLES:

>>> from sage.all import *
>>> Semigroups().CartesianProducts().extra_super_categories()
[Category of semigroups]
>>> Semigroups().CartesianProducts().super_categories()
[Category of semigroups, Category of Cartesian products of magmas]

    class ElementMethods
        Bases: object
        Finite
            alias of FiniteSemigroups
        FinitelyGeneratedAsMagma
            alias of FinitelyGeneratedSemigroups
        HTrivial
            alias of HTrivialSemigroups
        JTrivial
            alias of JTrivialSemigroups
        LTrivial
            alias of LTrivialSemigroups

    class ParentMethods
        Bases: object
        cayley_graph(side='right', simple=False, elements=None, generators=None, connecting_set=None)

        Return the Cayley graph for this finite semigroup.

        INPUT:
        • side – “left”, “right”, or “twosided”: the side on which the generators act (default:”right”)  
        • simple – boolean (default: False): if True, returns a simple graph (no loops, no labels, no  
          multiple edges)
        • generators – a list, tuple, or family of elements of self (default:  
          self.semigroup_generators())
        • connecting_set – alias for generators; deprecated
        • elements – a list (or iterable) of elements of self

        OUTPUT:
        • DiGraph

1190 Chapter 4. Individual Categories
EXAMPLES:

We start with the (right) Cayley graphs of some classical groups:

```python
sage: # needs sage.graphs sage.groups
sage: D4 = DihedralGroup(4); D4
Dihedral group of order 8 as a permutation group
sage: G = D4.cayley_graph()
sage: show(G, color_by_label=True, edge_labels=True)  # needs sage.plot
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.show3d(color_by_label=True, edge_size=0.01,  # needs sage.plot
           edge_size2=0.02, vertex_size=0.03)

sage: w = WeylGroup(['A', 3])
sage: d = w.cayley_graph(); d
Digraph on 24 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01,  # needs sage.plot
           edge_size2=0.02, vertex_size=0.03)
```

(continues on next page)
Alternative generators may be specified:

```
sage: # needs sage.graphs sage.groups
sage: G = A5.cayley_graph(generators=[A5.gens()[0]])
sage: G.num_edges()
60
sage: g = PermutationGroup([(i + 1, j + 1)
....:     for i in range(5)
....:     for j in range(5) if j != i])
sage: g.cayley_graph(generators=[(1,2), (2,3)])
Digraph on 120 vertices
```

If elements is specified, then only the subgraph induced and those elements is returned. Here we use it to display the Cayley graph of the free monoid truncated on the elements of length at most 3:

```
sage: # needs sage.combinat sage.graphs
sage: M = Monoids().example(); M
An example of a monoid:
the free monoid generated by ('a', 'b', 'c', 'd')
sage: elements = [M.prod(w)
....:     for w in sum((list(Words(M.semigroup_generators(), k))
....:     for k in range(Integer(4))), [])]
sage: G = M.cayley_graph(elements=elements)
sage: G.num_verts(), G.num_edges()
(85, 84)
sage: G.show3d(color_by_label=True, edge_size=0.001, vertex_size=0.01)  # needs sage.plot
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs
>>> G = M.cayley_graph(elements=elements)
```

(continues on next page)
We now illustrate the `side` and `simple` options on a semigroup:

```python
sage: S = FiniteSemigroups().example(alphabet=('a', 'b'))
sage: g = S.cayley_graph(simple=True)  # needs sage.graphs
sage: g.vertices(sort=True)  # needs sage.graphs
['a', 'ab', 'b', 'ba']
sage: g.edges(sort=True)  # needs sage.graphs
[('a', 'ab', None), ('b', 'ba', None)]
```
needs sage.graphs
[('a', 'ab', None), ('a', 'ba', None), ('ab', 'ba', None), ('b', 'ab', None), ('b', 'ba', None), ('ba', 'ab', None)]

>>> from sage.all import *
>>> g = S.cayley_graph(side="twosided", simple=True)

needs sage.graphs

>>> g.vertices(sort=True)

needs sage.graphs

['a', 'ab', 'b', 'ba']

>>> g.edges(sort=True)

needs sage.graphs

[('a', 'ab', None), ('a', 'ba', None), ('ab', 'ba', None), ('b', 'ab', None), ('b', 'ba', None), ('ba', 'ab', None)]

sage: s1 = SymmetricGroup(1); s = s1.cayley_graph()

needs sage.graphs sage.groups

sage: s.vertices(sort=False)

needs sage.graphs sage.groups

[]

>>> from sage.all import *
>>> s1 = SymmetricGroup(Integer(1)); s = s1.cayley_graph()

# needs sage.graphs sage.groups

>>> s.vertices(sort=False)

needs sage.graphs sage.groups
Todo:

- Add more options for constructing subgraphs of the Cayley graph, handling the standard use cases when exploring large/infinite semigroups (a predicate, generators of an ideal, a maximal length in term of the generators)
- Specify good default layout/plot/latex options in the graph
- Generalize to combinatorial modules with module generators / operators

AUTHORS:

- Bobby Moretti (2007-08-10)
- Nicolas M. Thiery (2008-12): extension to semigroups, side, simple, and elements options,
...

magma_generators ()

An alias for semigroup_generators ()

EXAMPLES:

```python
sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
sage: S.magma_generators()
Family ('a', 'b', 'c', 'd')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

```python
>>> from sage.all import *
>>> S = Semigroups().example("free")
>>> S.magma_generators()
Family ('a', 'b', 'c', 'd')
>>> S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

prod (args)

Return the product of the list of elements args inside self.

EXAMPLES:

```python
sage: S = Semigroups().example("free")
sage: S.prod([S('a'), S('b'), S('c')])
'abc'
sage: S.prod([])
Traceback (most recent call last):
  ...
AssertionError: Cannot compute an empty product in a semigroup
```

```python
>>> from sage.all import *
>>> S = Semigroups().example("free")
>>> S.prod([S('a'), S('b'), S('c')])
'abc'
```

(continues on next page)
regular_representation(base_ring=\text{None}, \text{side}=\text{\textquoteleft left\textquoteright})

Return the regular representation of self over base_ring.

- side – (default: \textquoteleft left\textquoteright) whether this is the \textquoteleft left\textquoteright or \textquoteleft right\textquoteright regular representation

**EXAMPLES:**

```sage```
S = groups.permutation.Dihedral(4)
# needs sage.groups
S.regular_representation()
# needs sage.groups
```

representation(module, on\_basis, side=\textquoteleft left\textquoteright, \textastargs, **\textasteriskcenteredkwargs)

Return a representation of self on module with the action given by on\_basis.

**INPUT:**

- module – a module with a basis
- on\_basis – function which takes as input \( g, m \), where \( g \) is an element of the semigroup and \( m \) is an element of the indexing set for the basis, and returns the result of \( g \) acting on \( m \)
- side – (default: \textquoteleft left\textquoteright) whether this is a \textquoteleft left\textquoteright or \textquoteleft right\textquoteright representation

**EXAMPLES:**

```sage```
sage: G = CyclicPermutationGroup(3)
sage: M = algebras.Exterior(QQ, 'x', 3)
sage: def on\_basis(g, m):
\hfill # cyclically permute generators
\hfill return M.prod([M.monomial(FrozenBitset([g(j+1)-1]))
\hfill for j in m])
sage: from sage.categories.algebras import Algebras
sage: R = G.representation(M, on\_basis, category=Algebras(QQ).WithBasis().FiniteDimensional())
sage: R
```

(continues on next page)
semigroup_generators()

Return distinguished semigroup generators for self.

OUTPUT: a family

This method is optional.

EXAMPLES:

```
sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

subsemigroup (generators, one=None, category=None)

Return the multiplicative subsemigroup generated by generators.

INPUT:
- generators – a finite family of elements of self, or a list, iterable, … that can be converted into one (see Family).
- one – a unit for the subsemigroup, or None.
- category – a category

This implementation lazily constructs all the elements of the semigroup, and the right Cayley graph relations between them, and uses the latter as an automaton.

See AutomaticSemigroup for details.

EXAMPLES:

```
sage: R = IntegerModRing(15)
sage: M = R.subsemigroup([R(3), R(5)]); M
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: M.list()
[3, 5, 9, 0, 10, 12, 6]
```

```
>>> from sage.all import *
>>> R = IntegerModRing(Integer(15))
>>> M = R.subsemigroup([R(Integer(3)), R(Integer(5))]); M
A subsemigroup of (Ring of integers modulo 15) with 2 generators
>>> M.list()
[3, 5, 9, 0, 10, 12, 6]
```
By default, \( M \) is just in the category of subsemigroups:

```
sage: M in Semigroups().Subobjects()  # needs sage.combinat
True
```

```
>>> from sage.all import *
>>> M in Semigroups().Subobjects()  # needs sage.combinat
True
```

In the following example, we specify that \( M \) is a submonoid of the finite monoid \( R \) (it shares the same unit), and a group by itself:

```
sage: M = R.subsemigroup([R(-1)],  # needs sage.combinat
category=Monoids().Finite().Subobjects() & Groups()); M
A submonoid of (Ring of integers modulo 15) with 1 generators
```

```
M.list()  # needs sage.combinat
[1, 14]
M.one()  # needs sage.combinat
1
```

```
>>> from sage.all import *
>>> M = R.subsemigroup([R(-Integer(1))],  # needs sage.combinat...
category=Monoids().Finite().Subobjects() & Groups()); M
A submonoid of (Ring of integers modulo 15) with 1 generators
```

```
>>> M.list()  # needs sage.combinat
[1, 14]
>>> M.one()  # needs sage.combinat
1
```

In the following example, \( M \) is a group; however, its unit does not coincide with that of \( R \), so \( M \) is only a subsemigroup, and we need to specify its unit explicitly:

```
sage: M = R.subsemigroup([R(5)],  # needs sage.combinat
category=Semigroups().Finite().Subobjects() & Groups()); M
Traceback (most recent call last):
... ValueError: For a monoid which is just a subsemigroup, the unit should be specified
```

```
sage: # needs sage.combinat sage.groups
sage: M = R.subsemigroup([R(5)], one=R(10),  # needs sage.combinat
category=Semigroups().Finite().Subobjects() & Groups()); M
A subsemigroup of (Ring of integers modulo 15) with 1 generators
```

```
sage: M in Groups()  # needs sage.combinat
True
sage: M.list()  # needs sage.combinat
[10, 5]
sage: M.one()  # needs sage.combinat
```

(continues on next page)
>>> from sage.all import *
>>> M = R.subsemigroup([R(Integer(5))], ␣ needs sage.combinat ...
... category=Semigroups().Finite().Subobjects() & Groups()); M
Traceback (most recent call last):
... ValueError: For a monoid which is just a subsemigroup, the unit should be specified

>>> # needs sage.combinat sage.groups
>>> M = R.subsemigroup([R(Integer(5))], one=R(Integer(10)), ...
... category=Semigroups().Finite().Subobjects() & Groups()); M
A subsemigroup of (Ring of integers modulo 15) with 1 generators

Todo:
• Fix the failure in TESTS by providing a default implementation of __invert__ for finite groups (or even finite monoids).
• Provide a default implementation of one for a finite monoid, so that we would not need to specify it explicitly?

trivial_representation(base_ring=None, side='twosided')

Return the trivial representation of self over base_ring.

INPUT:
• base_ring – (optional) the base ring; the default is Z
• side – ignored

EXAMPLES:

sage: G = groups.permutation.Dihedral(4) ␣ needs sage.groups
sage: G.trivial_representation() ␣ needs sage.groups
Trivial representation of Dihedral group of order 8
as a permutation group over Integer Ring

class Quotients (category, *args)

Bases: QuotientsCategory
class ParentMethods
    Bases: object

    semigroup_generators()
        Return semigroup generators for self by retracting the semigroup generators of the ambient semigroup.

        EXAMPLES:
        sage: S = FiniteSemigroups().Quotients().example().semigroup_generators()  # todo: not implemented
        >>> from sage.all import *
        >>> S = FiniteSemigroups().Quotients().example().semigroup_generators()  # todo: not implemented

    example()
        Return an example of quotient of a semigroup, as per Category.example().

        EXAMPLES:
        sage: Semigroups().Quotients().example()
        An example of a (sub)quotient semigroup: a quotient of the left zero semigroup
        >>> from sage.all import *
        >>> Semigroups().Quotients().example()
        An example of a (sub)quotient semigroup: a quotient of the left zero semigroup

RTrivial
    alias of RTrivialSemigroups

class SubcategoryMethods
    Bases: object

    Aperiodic()
        Return the full subcategory of the aperiodic objects of self.

        A (multiplicative) semigroup $S$ is aperiodic if for any element $s \in S$, the sequence $s, s^2, s^3, \ldots$ eventually stabilizes.

        In terms of variety, this can be described by the equation $s^2 s = s$.

        EXAMPLES:
        sage: Semigroups().Aperiodic()
        Category of aperiodic semigroups
        >>> from sage.all import *
        >>> Semigroups().Aperiodic()
        Category of aperiodic semigroups

        An aperiodic semigroup is $H$-trivial:
        sage: Semigroups().Aperiodic().axioms()
        frozenset({'Aperiodic', 'Associative', 'HTrivial'})
In the finite case, the two notions coincide:

```python
sage: Semigroups().Aperiodic().Finite() is Semigroups().HTrivial().Finite()
True
```

See also:
- [Wikipedia article Aperiodic semigroup](http://example.com/aperiodic_semigroup)
- [Semigroups.SubcategoryMethods.RTrivial](http://example.com/supplementary_materials/r_trivial)
- [Semigroups.SubcategoryMethods.LTrivial](http://example.com/supplementary_materials/l_trivial)
- [Semigroups.SubcategoryMethods.JTrivial](http://example.com/supplementary_materials/j_trivial)
- [Semigroups.SubcategoryMethods.Aperiodic](http://example.com/supplementary_materials/aperiodic)

**HTrivial()**

Return the full subcategory of the $H$-trivial objects of `self`.

Let $S$ be (multiplicative) semigroup. Two elements of $S$ are in the same $H$-class if they are in the same $L$-class and in the same $R$-class.

The semigroup $S$ is $H$-trivial if all its $H$-classes are trivial (that is of cardinality 1).

**EXAMPLES:**

```python
sage: C = Semigroups().HTrivial(); C
Category of h trivial semigroups
sage: Semigroups().HTrivial().Finite().example()
NotImplemented
```

See also:
- [Wikipedia article Green%27s_relations](http://example.com/green_relations)
- [Semigroups.SubcategoryMethods.RTrivial](http://example.com/supplementary_materials/r_trivial)
- [Semigroups.SubcategoryMethods.LTrivial](http://example.com/supplementary_materials/l_trivial)
- [Semigroups.SubcategoryMethods.JTrivial](http://example.com/supplementary_materials/j_trivial)
- [Semigroups.SubcategoryMethods.Aperiodic](http://example.com/supplementary_materials/aperiodic)

**JTrivial()**

Return the full subcategory of the $J$-trivial objects of `self`.

Let $S$ be (multiplicative) semigroup. The $J$-preorder $\leq_J$ on $S$ is defined by:

$$x \leq_J y \iff x \in SyS$$
The \( J \)-classes are the equivalence classes for the associated equivalence relation. The semigroup \( S \) is \( J \)-trivial if all its \( J \)-classes are trivial (that is of cardinality 1), or equivalently if the \( J \)-preorder is in fact a partial order.

EXAMPLES:

```python
sage: C = Semigroups().JTrivial(); C
Category of j trivial semigroups

>>> from sage.all import *
>>> C = Semigroups().JTrivial(); C
Category of j trivial semigroups
```

A semigroup is \( J \)-trivial if and only if it is \( L \)-trivial and \( R \)-trivial:

```python
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'JTrivial', 'LTrivial', 'RTrivial']
sage: Semigroups().LTrivial().RTrivial()
Category of j trivial semigroups

>>> from sage.all import *
>>> sorted(C.axioms())
['Associative', 'HTrivial', 'JTrivial', 'LTrivial', 'RTrivial']
```

For a commutative semigroup, all three axioms are equivalent:

```python
sage: Semigroups().Commutative().LTrivial()
Category of commutative j trivial semigroups
sage: Semigroups().Commutative().RTrivial()
Category of commutative j trivial semigroups

>>> from sage.all import *
>>> Semigroups().Commutative().LTrivial()
>>> Semigroups().Commutative().RTrivial()
Category of commutative j trivial semigroups
```

See also:
- Wikipedia article Green\%'s_relations
- Semigroups.SubcategoryMethods.LTrivial
- Semigroups.SubcategoryMethods.RTrivial
- Semigroups.SubcategoryMethods.HTrivial

\texttt{LTrivial()}

Return the full subcategory of the \( L \)-trivial objects of \texttt{self}.

Let \( S \) be (multiplicative) \texttt{semigroup}. The \( L \)-preorder \( \leq_L \) on \( S \) is defined by:

\[
x \leq_L y \iff x \in S y
\]

The \( L \)-classes are the equivalence classes for the associated equivalence relation. The semigroup \( S \) is \( L \)-trivial if all its \( L \)-classes are trivial (that is of cardinality 1), or equivalently if the \( L \)-preorder is in fact a partial order.

EXAMPLES:
A $L$-trivial semigroup is $H$-trivial:

```python
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'LTrivial']
```

See also:
- Wikipedia article Green%27s_relations
- Semigroups.SubcategoryMethods.RTrivial
- Semigroups.SubcategoryMethods.JTrivial
- Semigroups.SubcategoryMethods.HTrivial

**RTrivial()**

Return the full subcategory of the $R$-trivial objects of `self`.

Let $S$ be (multiplicative) semigroup. The $R$-preorder $\leq_R$ on $S$ is defined by:

$$x \leq_R y \iff x \in yS$$

The $R$-classes are the equivalence classes for the associated equivalence relation. The semigroup $S$ is $R$-trivial if all its $R$-classes are trivial (that is of cardinality 1), or equivalently if the $R$-preorder is in fact a partial order.

**EXAMPLES:**

```python
sage: C = Semigroups().RTrivial(); C
Category of r trivial semigroups

>>> from sage.all import *

C = Semigroups().RTrivial(); C
Category of r trivial semigroups
```

An $R$-trivial semigroup is $H$-trivial:

```python
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'RTrivial']
```

See also:
- Wikipedia article Green%27s_relations
- Semigroups.SubcategoryMethods.LTrivial
- Semigroups.SubcategoryMethods.JTrivial
class Subquotients(category, *args)
    Bases: SubquotientsCategory

The category of subquotient semi-groups.

EXAMPLES:

sage: Semigroups().Subquotients().all_super_categories()
[Category of subquotients of semigroups,
 Category of semigroups,
 Category of subquotients of magmas,
 Category of magmas,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

An example of a (sub)quotient semigroup: a quotient of the left zero...
Unital

alias of Monoids

example (choice='leftzero', **kwds)

Returns an example of a semigroup, as per Category.example().

INPUT:

• choice – str (default: 'leftzero'). Can be either 'leftzero' for the left zero semigroup, or 'free' for the free semigroup.

• **kwds – keyword arguments passed onto the constructor for the chosen semigroup.

EXAMPLES:

```python
sage: Semigroups().example(choice='leftzero')
An example of a semigroup: the left zero semigroup
sage: Semigroups().example(choice='free')
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd →')
sage: Semigroups().example(choice='free', alphabet=('a', 'b'))
An example of a semigroup: the free semigroup generated by ('a', 'b')
```

4.141 Semirings

class sage.categories.semirings.Semirings (base_category)

Bases: CategoryWithAxiom_singleton

The category of semirings.

A semiring \((S, +, *)\) is similar to a ring, but without the requirement that each element must have an additive inverse. In other words, it is a combination of a commutative additive monoid \((S, +)\) and a multiplicative monoid \((S, *)\), where \(*\) distributes over \(+\).

See also:

Wikipedia article Semiring

EXAMPLES:

```python
sage: Semirings()
Category of semirings
```

(continues on next page)
sage: Semirings().super_categories()
[Category of associative additive commutative additive associative additive...
→ unital distributive magmas and additive magmas,
Category of monoids]
sage: sorted(Semirings().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveUnital', 'Associative',
→ 'Distributive', 'Unital']
sage: Semirings() is (CommutativeAdditiveMonoids() & Monoids()).Distributive()
True
sage: Semirings().AdditiveInverse()
Category of rings

4.142 Semisimple Algebras

class sage.categories.semisimple_algebras.SemisimpleAlgebras(base, name=None)

    Bases: Category_over_base_ring

The category of semisimple algebras over a given base ring.

EXAMPLES:

sage: from sage.categories.semisimple_algebras import SemisimpleAlgebras
sage: C = SemisimpleAlgebras(QQ); C
Category of semisimple algebras over Rational Field

This category is best constructed as:
sage: D = Algebras(QQ).Semisimple(); D
Category of semisimple algebras over Rational Field
sage: D is C
True
sage: C.super_categories()
[Category of algebras over Rational Field]

>>> from sage.all import *

>>> D = Algebras(QQ).Semisimple(); D
Category of semisimple algebras over Rational Field
>>> D is C
True
>>> C.super_categories()
[Category of algebras over Rational Field]

Typically, finite group algebras are semisimple:
sage: DihedralGroup(5).algebra(QQ) in SemisimpleAlgebras
˓→ needs sage.groups
True

>>> from sage.all import *

>>> DihedralGroup(Integer(5)).algebra(QQ) in SemisimpleAlgebras
˓→ # needs sage.groups
True

Unless the characteristic of the field divides the order of the group:
sage: DihedralGroup(5).algebra(IntegerModRing(5)) in SemisimpleAlgebras
˓→ needs sage.groups
False
sage: DihedralGroup(5).algebra(IntegerModRing(7)) in SemisimpleAlgebras
˓→ needs sage.groups
True

>>> from sage.all import *

>>> DihedralGroup(Integer(5)).algebra(IntegerModRing(Integer(5))) in SemisimpleAlgebras
˓→ # needs sage.groups
False

See also:
Wikipedia article Semisimple_algebra

class FiniteDimensional(base_category)

    Bases: CategoryWithAxiom_over_base_ring

    WithBasis
        alias of FiniteDimensionalSemisimpleAlgebrasWithBasis

4.142. Semisimple Algebras
class ParentMethods
    Bases: object

    radical_basis(**keywords)
    Return a basis of the Jacobson radical of this algebra.
    • keywords – for compatibility; ignored.
    OUTPUT: the empty list since this algebra is semisimple.

    EXAMPLES:
    sage: A = SymmetricGroup(4).algebra(QQ)  # needs sage.combinat sage.groups
    sage: A.radical_basis()  # needs sage.combinat sage.groups
    ()

    super_categories()
    EXAMPLES:
    sage: Algebras(QQ).Semisimple().super_categories()
    [Category of algebras over Rational Field]

    >>> from sage.all import *
    >>> A = SymmetricGroup(Integer(4)).algebra(QQ)  # needs sage.combinat sage.groups
    >>> A.radical_basis()  # needs sage.combinat sage.groups
    ()

4.143 Sets

exception sage.categories.sets_cat.EmptySetError
    Bases: ValueError
    Exception raised when some operation can’t be performed on the empty set.

    EXAMPLES:
    sage: def first_element(st):
    ....:     if not st: raise EmptySetError("no elements")
    ....:     else: return st[0]
    sage: first_element(Set((1,2,3)))
    1
    sage: first_element(Set([]))
    Traceback (most recent call last):
    ... EmptySetError: no elements
```python
>>> from sage.all import *
>>> def first_element(st):
...    if not st: raise EmptySetError("no elements")
...    else: return st[Integer(0)]
>>> first_element(Set((Integer(1), Integer(2), Integer(3))))
1
>>> first_element(Set([]))
Traceback (most recent call last):
... EmptySetError: no elements
```

class sage.categories.sets_cat.Sets

Bases: Category_singleton

The category of sets.

The base category for collections of elements with = (equality).

This is also the category whose objects are all parents.

EXAMPLES:

```python
sage: Sets()
Category of sets
sage: Sets().super_categories()
[Category of sets with partial maps]
```

Let us consider an example of set:

```python
sage: P = Sets().example("inherits")
sage: P
Set of prime numbers
```

See `P??` for the code.

P is in the category of sets:

```python
sage: P.category()
Category of sets
```
and therefore gets its methods from the following classes:

```python
sage: for cl in P.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract'>
<class 'sage.structure.unique_representation.UniqueRepresentation'>
<class 'sage.structure.unique_representation.CachedRepresentation'>
<class 'sage.structure.unique_representation.WithPicklingByInitArgs'>
<class 'sage.misc.fast_methods.WithEqualityById'>
<class 'sage.structure.parent.Parent'>
<class 'sage.structure.category_object.CategoryObject'>
<class 'sage.structure.sage_object.SageObject'>
<class 'sage.categories.sets_cat.Sets.parent_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.parent_class'>
<class 'sage.categories.objects.Objects.parent_class'>
<class 'object'>
```

We run some generic checks on P:

```python
>>> from sage.all import *

>>> for cl in P.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract'>
<class 'sage.structure.unique_representation.UniqueRepresentation'>
<class 'sage.structure.unique_representation.CachedRepresentation'>
<class 'sage.structure.unique_representation.WithPicklingByInitArgs'>
<class 'sage.misc.fast_methods.WithEqualityById'>
<class 'sage.structure.parent.Parent'>
<class 'sage.structure.category_object.CategoryObject'>
<class 'sage.structure.sage_object.SageObject'>
<class 'sage.categories.sets_cat.Sets.parent_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.parent_class'>
<class 'sage.categories.objects.Objects.parent_class'>
<class 'object'>
```

```
sage: TestSuite(P).run(verbosel=True) #needs sage.libs.pari
running _test_an_element() ... pass
running _test_cardinality() ... pass
running _test_category() ... pass
running _test_construction() ... pass
running _test_elements() ... Running the test suite of self.an_element()
running _test_category() ... pass
running _test_eq() ... pass
running _test_new() ... pass
running _test_not_implemented_methods() ... pass
running _test_pickling() ... pass
pass
running _test_elements_eq_reflexive() ... pass
running _test_elements_eq_symmetric() ... pass
running _test_elements_eq_transitive() ... pass
running _test_elements_neq() ... pass
running _test_eq() ... pass
running _test_new() ... pass
running _test_not_implemented_methods() ... pass
```

(continues on next page)
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass

Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass

They get their methods from the following classes:

sage: for cl in x.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'>

(continues on next page)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits.Element'>
<class 'sage.rings.integer.IntegerWrapper'>
<class 'sage.rings.integer.Integer'>
<class 'sage.structure.element.EuclideanDomainElement'>
<class 'sage.structure.element.PrincipalIdealDomainElement'>
<class 'sage.structure.element.DedekindDomainElement'>
<class 'sage.structure.element.IntegralDomainElement'>
<class 'sage.structure.element.CommutativeRingElement'>
<class 'sage.structure.element.RingElement'>
<class 'sage.structure.element.ModuleElement'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element'>
<class 'sage.structure.element.Element'>
<class 'sage.structure.sage_object.SageObject'>
<class 'sage.categories.sets_cat.Sets.element_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.element_class'>
<class 'sage.categories.objects.Objects.element_class'>
<... object>

>>> from sage.all import *
>>> for cl in x.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.
˓→element_class'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits.Element'>
<class 'sage.rings.integer.IntegerWrapper'>
<class 'sage.rings.integer.Integer'>
<class 'sage.structure.element.EuclideanDomainElement'>
<class 'sage.structure.element.PrincipalIdealDomainElement'>
<class 'sage.structure.element.DedekindDomainElement'>
<class 'sage.structure.element.IntegralDomainElement'>
<class 'sage.structure.element.CommutativeRingElement'>
<class 'sage.structure.element.RingElement'>
<class 'sage.structure.element.ModuleElement'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element'>
<class 'sage.structure.element.Element'>
<class 'sage.structure.sage_object.SageObject'>
<class 'sage.categories.sets_cat.Sets.element_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.element_class'>
<class 'sage.categories.objects.Objects.element_class'>
<... 'object'>

FIXME: Objects.element_class is not very meaningful …

class Algebras(category, *args)
    Bases: AlgebrasCategory

class ParentMethods
    Bases: object

    construction()
    Return the functorial construction of self.

    EXAMPLES:

    sage: A = GroupAlgebra(KleinFourGroup(), QQ)  # needs sage.groups sage.modules
    sage: F, arg = A.construction(); F, arg  # needs sage.groups sage.modules

    (continues on next page)
This also works for structures such as monoid algebras (see Issue #27937):

```
sage: A = FreeAbelianMonoid('x,y').algebra(QQ)
vneds sage.combinat sage.modules
sage: F, arg = A.construction(); F, arg
vneds sage.groups sage.modules
(The algebra functorial construction,
Free abelian monoid on 2 generators (x, y))
sage: F(arg) is A
vneds sage.groups sage.modules
True
```

**extra_super_categories()**

**EXAMPLES:**

```
sage: Sets().Algebras(ZZ).super_categories()
[Category of modules with basis over Integer Ring]
sage: Sets().Algebras(QQ).extra_super_categories()
[Category of vector spaces with basis over Rational Field]
sage: Sets().example().algebra(ZZ).categories()
vneds sage.modules
[Category of set algebras over Integer Ring,
Category of modules with basis over Integer Ring,
... 
Category of objects]
```
from sage.all import *

Sets().Algebras(ZZ).super_categories()
[Category of modules with basis over Integer Ring]

Sets().Algebras(QQ).extra_super_categories()
[Category of vector spaces with basis over Rational Field]

Sets().example().algebra(ZZ).categories()  # needs sage.modules
[Category of set algebras over Integer Ring, 
Category of modules with basis over Integer Ring, ...
Category of objects]

class CartesianProducts(category, *args)

Bases: CartesianProductsCategory

EXAMPLES:

sage: C = Sets().CartesianProducts().example()
sage: C
The Cartesian product of (Set of prime numbers (basic implementation), 
An example of an infinite enumerated set: the non negative integers, 
An example of a finite enumerated set: {1,2,3})
sage: C.category()
Category of Cartesian products of sets
sage: C.categories()
[Category of Cartesian products of sets, Category of sets, 
Category of sets with partial maps, 
Category of objects]
sage: TestSuite(C).run()

from sage.all import *

C = Sets().CartesianProducts().example()
C
The Cartesian product of (Set of prime numbers (basic implementation), 
An example of an infinite enumerated set: the non negative integers, 
An example of a finite enumerated set: {1,2,3})
C.category()
Category of Cartesian products of sets
C.categories()
[Category of Cartesian products of sets, Category of sets, 
Category of sets with partial maps, 
Category of objects]
TestSuite(C).run()

class ElementMethods

Bases: object
cartesian_factors()

Return the Cartesian factors of self.

EXAMPLES:

# needs sage.modules
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename("F")
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename("G")
sage: H = CombinatorialFreeModule(ZZ, [4,7]); H.rename("H")
sage: S = cartesian_product([F, G, H])
sage: x = (S.monomial((0,4)) + 2 * S.monomial((0,5))
    ....: + 3 * S.monomial((1,6)) + 4 * S.monomial((2,4))
    ....: + 5 * S.monomial((2,7)))
sage: x.cartesian_factors()
sage: [s.parent() for s in x.cartesian_factors()]
[F, G, H]
sage: S.zero().cartesian_factors()
(0, 0, 0)
sage: [s.parent() for s in S.zero().cartesian_factors()]
[F, G, H]

**cartesian_projection**(i)

Return the projection of **self** onto the i-th factor of the Cartesian product.

**INPUT:**

- **i** – the index of a factor of the Cartesian product

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename("F")
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename("G")
sage: S = cartesian_product([F, G])
sage: x = (S.monomial((0,4)) + 2 * S.monomial((0,5))
    ....: + 3 * S.monomial((1,6))
    ....: + 4 * S.monomial((1,6))
    ....: + 5 * S.monomial((2,7)))
sage: x.cartesian_projection(0)
sage: x.cartesian_projection(1)
3*B[6]
```
```python
>>> from sage.all import *
>>> # needs sage.modules

>>> F = CombinatorialFreeModule(ZZ, [Integer(4), Integer(5)]); F.rename("F")
>>> G = CombinatorialFreeModule(ZZ, [Integer(4), Integer(6)]); G.rename("G")
>>> S = cartesian_product([F, G])
>>> x = (S.monomial((Integer(0), Integer(4))) + Integer(2) * S.monomial((Integer(0), Integer(5)))
    + Integer(3) * S.monomial((Integer(1), Integer(6))))

>>> x.cartesian_projection(Integer(0))
>>> x.cartesian_projection(Integer(1))
3*B[6]
```

```python
class ParentMethods
    Bases: object

    an_element()

    EXAMPLES:

    sage: C = Sets().CartesianProducts().example(); C
    The Cartesian product of (Set of prime numbers (basic implementation),
    An example of an infinite enumerated set: the non negative integers,
    An example of a finite enumerated set: \{1,2,3\})
    sage: C.an_element()
    (47, 42, 1)

    >>> from sage.all import *
    >>> C = Sets().CartesianProducts().example(); C
    The Cartesian product of (Set of prime numbers (basic implementation),
    An example of an infinite enumerated set: the non negative integers,
    An example of a finite enumerated set: \{1,2,3\})
    >>> C.an_element()
    (47, 42, 1)

cardinality()

    Return the cardinality of self.

    EXAMPLES:

    sage: E = FiniteEnumeratedSet([1,2,3])
    sage: C = cartesian_product([E, SymmetricGroup(4)])
    # needs sage.groups
    sage: C.cardinality()
    72

    sage: E = FiniteEnumeratedSet([[]])
    sage: C = cartesian_product([E, ZZ, QQ])
    sage: C.cardinality()
    0

    sage: C = cartesian_product([ZZ, QQ])
    sage: C.cardinality()
    +Infinity
```

(continues on next page)
cartesian_factors()

Return the Cartesian factors of self.

EXAMPLES:

```python
sage: cartesian_product([QQ, ZZ, ZZ]).cartesian_factors()
(Rational Field, Integer Ring, Integer Ring)
```

```python
>>> from sage.all import *
```

```python
>>> cartesian_product([QQ, ZZ, ZZ]).cartesian_factors()
(Rational Field, Integer Ring, Integer Ring)
```

cartesian_projection(i)

Return the natural projection onto the i-th Cartesian factor of self.

INPUT:

- i – the index of a Cartesian factor of self

EXAMPLES:

```python
sage: C = Sets().CartesianProducts().example(); C
The Cartesian product of (Set of prime numbers (basic implementation),
An example of an infinite enumerated set: the non negative integers,
An example of a finite enumerated set: {1,2,3})
sage: x = C.an_element(); x
(47, 42, 1)
sage: pi = C.cartesian_projection(1)
```

(continues on next page)
>>> from sage.all import *
>>> C = Sets().CartesianProducts().example(); C
The Cartesian product of (Set of prime numbers (basic implementation),
An example of an infinite enumerated set: the non negative integers,
An example of a finite enumerated set: {1,2,3})
>>> x = C.an_element(); x
(47, 42, 1)
>>> pi = C.cartesian_projection(Integer(1))
>>> pi(x)
42

construction()

The construction functor and the list of Cartesian factors.

EXAMPLES:

```python
sage: C = cartesian_product([QQ, ZZ, ZZ])
```

```python
sage: C.construction()
(The cartesian_product functorial construction,
(Rational Field, Integer Ring, Integer Ring))
```

```python
>>> from sage.all import *
```

```python
>>> C = cartesian_product([QQ, ZZ, ZZ])
```

```python
>>> C.construction()
(The cartesian_product functorial construction,
(Rational Field, Integer Ring, Integer Ring))
```

is_empty()

Return whether this set is empty.

EXAMPLES:

```python
sage: S1 = FiniteEnumeratedSet([1,2,3])
sage: S2 = Set([])
sage: cartesian_product([S1,ZZ]).is_empty()
False
```

```python
sage: cartesian_product([S1,S2,S1]).is_empty()
True
```

```python
>>> from sage.all import *
```

```python
>>> S1 = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])
```

```python
>>> S2 = Set([])
```

```python
>>> cartesian_product([S1,ZZ]).is_empty()
False
```

```python
>>> cartesian_product([S1,S2,S1]).is_empty()
True
```

is_finite()

Return whether this set is finite.

EXAMPLES:
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E, SymmetricGroup(4)]) # needs sage.groups
sage: C.is_finite() # needs sage.groups
True

sage: cartesian_product([ZZ,ZZ]).is_finite()
False
sage: cartesian_product([ZZ, Set(), ZZ]).is_finite()
True

random_element(*args)

Return a random element of this Cartesian product.

The extra arguments are passed down to each of the factors of the Cartesian product.

EXAMPLES:

sage: C = cartesian_product([Permutations(10)]*5)
sage: C.random_element() # random
([2, 9, 4, 7, 1, 8, 6, 10, 5, 3],
 [8, 6, 5, 7, 1, 4, 9, 3, 10, 2],
 [5, 10, 3, 8, 2, 9, 1, 4, 7, 6],
 [9, 6, 10, 3, 2, 1, 5, 8, 7, 4],
 [8, 5, 2, 9, 10, 3, 7, 1, 4, 6])

sage: C = cartesian_product([ZZ]*10)
sage: c1 = C.random_element()
sage: c1 # random
(3, 1, 4, 1, -3, 0, -4, -17, 2)
sage: c2 = C.random_element(4,7)
sage: c2 # random
(6, 5, 6, 4, 5, 6, 4, 5, 5)
sage: all(4 <= i < 7 for i in c2)
True

>>> from sage.all import *

>>> E = FiniteEnumeratedSet([Integer(1),Integer(2),Integer(3)])

>>> C = cartesian_product([E, SymmetricGroup(Integer(4))])

>>> C.is_finite() # needs sage.groups
True

>>> cartesian_product([ZZ,ZZ]).is_finite()
False

>>> cartesian_product([ZZ, Set(), ZZ]).is_finite()
True

(continues on next page)
\begin{verbatim}
>>> C = cartesian_product([ZZ]*Integer(10))
>>> c1 = C.random_element()
>>> c1
# random
(3, 1, 4, 1, 1, -3, 0, -4, -17, 2)
>>> c2 = C.random_element(Integer(4),Integer(7))
>>> c2
# random
(6, 5, 6, 4, 5, 6, 6, 4, 5, 5)
>>> all(Integer(4) <= i < Integer(7) for i in c2)
True
\end{verbatim}

\textbf{example()}

\textbf{EXAMPLES:}

\texttt{sage: Sets().CartesianProducts().example()}
The Cartesian product of (Set of prime numbers (basic implementation),
An example of an infinite enumerated set: the non negative integers,
An example of a finite enumerated set: \{1,2,3\})

\texttt{>>> from sage.all import *}
\texttt{>>> Sets().CartesianProducts().example()}
The Cartesian product of (Set of prime numbers (basic implementation),
An example of an infinite enumerated set: the non negative integers,
An example of a finite enumerated set: \{1,2,3\})

\textbf{extra_super_categories()}

A Cartesian product of sets is a set.

\textbf{EXAMPLES:}

\texttt{sage: Sets().CartesianProducts().extra_super_categories()}
[Category of sets]
\texttt{sage: Sets().CartesianProducts().super_categories()}
[Category of sets]

\texttt{>>> from sage.all import *}
\texttt{>>> Sets().CartesianProducts().extra_super_categories()}
[Category of sets]
\texttt{>>> Sets().CartesianProducts().super_categories()}
[Category of sets]

\textbf{class ElementMethods}

\textbf{Bases: object}

\textbf{cartesian_product(*elements)}

Return the Cartesian product of its arguments, as an element of the Cartesian product of the parents of those elements.

\textbf{EXAMPLES:}

\texttt{sage: C = AlgebrasWithBasis(QQ)}
\texttt{sage: A = C.example()}
# ...
\texttt{needs sage.combinat sage.modules}
\texttt{sage: a, b, c = A.algebra_generators()}
\end{verbatim}
needs sage.combinat sage.modules

sage: a.cartesian_product(b, c)  # ...

needs sage.combinat sage.modules

B[(0, word: a)] + B[(1, word: b)] + B[(2, word: c)]

>>> from sage.all import *

>>> C = AlgebrasWithBasis(QQ)

>>> A = C.example()  # ...

>>> a, b, c = A.algebra_generators()  # ...

>>> a.cartesian_product(b, c)  # ...

B[(0, word: a)] + B[(1, word: b)] + B[(2, word: c)]

FIXME: is this a policy that we want to enforce on all parents?

Enumerated

alias of EnumeratedSets

Facade

alias of FacadeSets

Finite

alias of FiniteSets

class Infinite(base_category)

Bases: CategoryWithAxiom_singleton

class ParentMethods

Bases: object

cardinality()

Count the elements of the enumerated set.

EXAMPLES:

sage: NN = InfiniteEnumeratedSets().example()
sage: NN.cardinality()
+Infinity

>>> from sage.all import *

>>> NN = InfiniteEnumeratedSets().example()

>>> NN.cardinality()
+Infinity

is_empty()

Return whether this set is empty.

Since this set is infinite this always returns False.

EXAMPLES:

sage: C = InfiniteEnumeratedSets().example()
sage: C.is_empty()
False
>>> from sage.all import *
>>> C = InfiniteEnumeratedSets().example()
>>> C.is_empty()
False

is_finite()
Return whether this set is finite.
Since this set is infinite this always returns False.

EXAMPLES:
sage: C = InfiniteEnumeratedSets().example()
sage: C.is_finite()
False

class IsomorphicObjects(category, *args)
Bases: IsomorphicObjectsCategory
A category for isomorphic objects of sets.

EXAMPLES:
sage: Sets().IsomorphicObjects()
Category of isomorphic objects of sets
sage: Sets().IsomorphicObjects().all_super_categories()
[Category of isomorphic objects of sets,
 Category of subobjects of sets, Category of quotients of sets,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

>>> from sage.all import *
>>> Sets().IsomorphicObjects()
Category of isomorphic objects of sets
>>> Sets().IsomorphicObjects().all_super_categories()
[Category of isomorphic objects of sets,
 Category of subobjects of sets, Category of quotients of sets,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

class ParentMethods
Bases: object

Metric
alias of MetricSpaces

class MorphismMethods
Bases: object
image (domain_subset=None)

Return the image of the domain or of domain_subset.

EXAMPLES:

```python
sage: # needs sage.combinat
sage: P = Partitions(6)
sage: H = Hom(P, ZZ)
sage: f = H(ZZ.sum)
sage: X = f.image() # needs sage.libs.flint
sage: list(X) # needs sage.libs.flint
[6]
```

is_injective()

Return whether this map is injective.

EXAMPLES:

```python
sage: f = ZZ.hom(GF(3)); f
Natural morphism:
    From: Integer Ring
    To:   Finite Field of size 3
sage: f.is_injective()
False
```

class ParentMethods

Bases: object

CartesianProduct

alias of CartesianProduct

algebra (base_ring, category=None, **kwds)

Return the algebra of self over base_ring.

INPUT:

- `self`: a parent $S$
- `base_ring`: a ring $K$
category – a super category of the category of \( S \), or None
This returns the space of formal linear combinations of elements of \( S \) with coefficients in \( K \),
edowed with whatever structure can be induced from that of \( S \). See the documentation of \texttt{sage.categories.algebra_functor} for details.

**EXAMPLES:**

If \( S \) is a group, the result is its group algebra \( KS \):

```python
sage: # needs sage.groups sage.modules
sage: S = DihedralGroup(4); S
Dihedral group of order 8 as a permutation group
sage: A = S.algebra(QQ); A
Algebra of Dihedral group of order 8 as a permutation group
over Rational Field
sage: A.category()
Category of finite group algebras over Rational Field
sage: a = A.an_element(); a
() + (1,3) + 2*(1,3)(2,4) + 3*(1,4,3,2)
```

This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication
of \( G \) to a multiplication on \( RG \):

```python
sage: a * a
# needs sage.groups sage.modules
6*() + 4*(2,4) + 3*(1,2)(3,4) + 12*(1,2,3,4) + 2*(1,3) + 13*(1,3)(2,4) + 6*(1,4,3,2) + 3*(1,4)(2,3)
```

If \( S \) is a monoid, the result is its monoid algebra \( KS \):

```python
sage: S = Monoids().example(); S
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(QQ); A
# needs sage.modules
Algebra of
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
over Rational Field
sage: A.category()
```

(continues on next page)
Similarly, we can construct algebras for additive magmas, monoids, and groups.

One may specify for which category one takes the algebra; here we build the algebra of the additive group $\mathbb{G}F_3$:

```
sage: # needs sage.modules
sage: from sage.categories.additive_groups import AdditiveGroups
sage: S = GF(7)
```

```
Algebra of Finite Field of size 7 over Rational Field
sage: A.category()
Category of finite dimensional additive group algebras over Rational Field
sage: a = A(S(1))
sage: a
1
sage: 1 + a * a * a
0 + 3
```

Note that the `category` keyword needs to be fed with the structure on $S$ to be used, not the induced structure on the result.

**an_element()**

Return a (preferably typical) element of this parent.

This is used both for illustration and testing purposes. If the set `self` is empty, `an_element()`
should raise the exception \texttt{EmptySetError}.

This default implementation calls \texttt{\_an\_element\_()} and caches the result. Any parent should implement either \texttt{an\_element()} or \texttt{\_an\_element\_()}.  

\textbf{EXAMPLES:}

```
sage: CDF.an_element()
    # needs sage.rings.complex_double
    1.0*I
sage: ZZ[['t']].an_element()
t
```

```
>>> from sage.all import *
>>> CDF.an_element()  #--
    needs sage.rings.complex_double
    1.0*I
>>> ZZ[['t']].an_element()
t
```

cartesian\_product (*parents, **kwargs)

Return the Cartesian product of the parents.

\textbf{INPUT:}

\begin{itemize}
    \item parents -- a list (or other iterable) of parents.
    \item category -- (default: None) the category the Cartesian product belongs to. If None is passed, then \texttt{category\_from\_parents()} is used to determine the category.
    \item extra\_category -- (default: None) a category that is added to the Cartesian product in addition to the categories obtained from the parents.
    \item other keyword arguments will passed on to the class used for this Cartesian product (see also \texttt{CartesianProduct}).
\end{itemize}

\textbf{OUTPUT:}

The Cartesian product.

\textbf{EXAMPLES:}

```
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A")  #--
    needs sage.combinat sage.modules
sage: A.cartesian_product(A, A)  #--
    needs sage.combinat sage.modules
A (+) A (+) A
sage: ZZ.cartesian_product(GF(2), FiniteEnumeratedSet([1,2,3]))
The Cartesian product of (Integer Ring,
    Finite Field of size 2, \{1, 2, 3\})
sage: C = ZZ.cartesian_product(A); C  #--
    needs sage.combinat sage.modules
The Cartesian product of (Integer Ring, A)
```

```
>>> from sage.all import *
>>> C = AlgebrasWithBasis(QQ)
>>> A = C.example(); A.rename("A")  #--
    needs sage.combinat sage.modules
>>> A.cartesian_product(A, A)  #--
    needs sage.combinat sage.modules
A (+) A (+) A
```

(continues on next page)
The Cartesian product of (Integer Ring, Finite Field of size 2, \{1, 2, 3\})

C = ZZ.cartesian_product(A); C

The Cartesian product of (Integer Ring, A)

---

**construction()**

Return a pair \((\text{functor}, \text{parent})\) such that \(\text{functor}(\text{parent})\) returns \(\text{self}\). If \(\text{self}\) does not have a functorial construction, return \(\text{None}\).

**EXAMPLES:**

```
sage: QQ.construction()
(FractionField, Integer Ring)
sage: f, R = QQ['x'].construction()
sage: f
Poly[x]
sage: R
Rational Field
sage: f(R)
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> QQ.construction()
(FractionField, Integer Ring)
```

```
>>> f, R = QQ['x'].construction()
```

```
>>> f
Poly[x]
```

```
>>> R
Rational Field
```

```
>>> f(R)
Univariate Polynomial Ring in x over Rational Field
```

**is_parent_of(element)**

Return whether \(\text{self}\) is the parent of \(\text{element}\).

**INPUT:**

- \(\text{element}\) – any object

**EXAMPLES:**

```
sage: S = ZZ
sage: S.is_parent_of(1)
True
sage: S.is_parent_of(2/1)
False
```

```
>>> from sage.all import *
>>> S = ZZ
```

```
>>> S.is_parent_of(Integer(1))
True
```

```
>>> S.is_parent_of(Integer(2)/Integer(1))
False
```

This method differs from \(\text{__contains__()}\) because it does not attempt any coercion:
some_elements()

Return a list (or iterable) of elements of self.

This is typically used for running generic tests (see TestSuite).

This default implementation calls an_element().

EXAMPLES:

sage: S = Sets().example(); S
Set of prime numbers (basic implementation)
sage: S.an_element()
47
sage: S.some_elements()
[47]
sage: S = Set([])
sage: list(S.some_elements())
[]

This method should return an iterable, not an iterator.

class Quotients (category, *args)

Bases: QuotientsCategory

A category for quotients of sets.

See also:

Sets().Quotients()

EXAMPLES:

sage: Sets().Quotients()
Category of quotients of sets
sage: Sets().Quotients().all_super_categories()
[Category of quotients of sets, Category of subquotients of sets,
>>> from sage.all import *
>>> Sets().Quotients()
Category of quotients of sets

>>> Sets().Quotients().all_super_categories()
[Category of quotients of sets,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

class ParentMethods
    Bases: object

class Realizations(category, *args)
    Bases: RealizationsCategory
class ParentMethods
    Bases: object

    realization_of()

    Return the parent this is a realization of.

    EXAMPLES:

    sage: A = Sets().WithRealizations().example(); A
        # needs sage.modules
    The subset algebra of {1, 2, 3} over Rational Field
    sage: In = A.In(); In
        # needs sage.modules
    The subset algebra of {1, 2, 3} over Rational Field in the In basis
    sage: In.realization_of()
        # needs sage.modules
    The subset algebra of {1, 2, 3} over Rational Field

>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
    # needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
>>> In = A.In(); In
    # needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the In basis
>>> In.realization_of()
    # needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field

class SubcategoryMethods
    Bases: object

    Algebras(base_ring)

    Return the category of objects constructed as algebras of objects of self over base_ring.

    INPUT:
    • base_ring—a ring
See \texttt{Sets.ParentMethods.algebra()} for the precise meaning in Sage of the \textit{algebra of an object}.

EXAMPLES:

\begin{verbatim}
sage: Monoids().Algebras(QQ)
Category of monoid algebras over Rational Field

sage: Groups().Algebras(QQ)
Category of group algebras over Rational Field

sage: AdditiveMagmas().AdditiveAssociative().Algebras(QQ)
Category of additive semigroup algebras over Rational Field

sage: Monoids().Algebras(Rings())
Category of monoid algebras over Category of rings
\end{verbatim}

See also:

\begin{itemize}
\item \texttt{algebra_functor.AlgebrasCategory}
\item \texttt{CovariantFunctorialConstruction}
\end{itemize}

\textbf{CartesianProducts()}

Return the full subcategory of the objects of \texttt{self} constructed as Cartesian products.

See also:

\begin{itemize}
\item \texttt{cartesian_product.CartesianProductFunctor}
\item \texttt{RegressiveCovariantFunctorialConstruction}
\end{itemize}

EXAMPLES:

\begin{verbatim}
>>> from sage.all import *

>>> Monoids().Algebras(QQ)
Category of monoid algebras over Rational Field

>>> Groups().Algebras(QQ)
Category of group algebras over Rational Field

>>> AdditiveMagmas().AdditiveAssociative().Algebras(QQ)
Category of additive semigroup algebras over Rational Field

>>> Monoids().Algebras(Rings())
Category of monoid algebras over Category of rings
\end{verbatim}
Enumerated()  
Return the full subcategory of the enumerated objects of self.

An enumerated object can be iterated to get its elements.

EXAMPLES:

```
sage: Sets().Enumerated()
Category of enumerated sets
sage: Rings().Finite().Enumerated()
Category of finite enumerated rings
sage: Rings().Infinite().Enumerated()
Category of infinite enumerated rings
```

Facade()  
Return the full subcategory of the facade objects of self.

What is a facade set?

Recall that, in Sage, *sets are modelled by “parents”*, and their elements know which distinguished set they belong to. For example, the ring of integers \( \mathbb{Z} \) is modelled by the parent \( \mathbb{Z} \), and integers know that they belong to this set:

```
sage: ZZ
Integer Ring
sage: 42.parent()
Integer Ring
```

Sometimes, it is convenient to represent the elements of a parent \( P \) by elements of some other parent. For example, the elements of the set of prime numbers are represented by plain integers:

```
sage: Primes()
Set of all prime numbers: 2, 3, 5, 7, ...
sage: p = Primes().an_element(); p
43
sage: p.parent()
Integer Ring
```

(continues on next page)
In this case, \( P \) is called a **facade set**.

This feature is advertised through the category of \( P \):

```
sage: Primes().category()
Category of facade infinite enumerated sets

sage: Sets().Facade()
Category of facade sets
```

Typical use cases include modeling a subset of an existing parent:

```
sage: Set([4,6,9])  # random
{4, 6, 9}

sage: Sets().Facade().example()
An example of facade set: the monoid of positive integers
```

or the union of several parents:

```
>>> from sage.all import *

>>> Set([Integer(4),Integer(6),Integer(9)])  # random
{4, 6, 9}

>>> Sets().Facade().example()
An example of facade set: the monoid of positive integers
```

or endowing an existing parent with more (or less!) structure:

```
sage: Posets().example("facade")
An example of a facade poset: the positive integers ordered by \( \_\_\_\_\_\_→ \) divisibility

>>> from sage.all import *

>>> Posets().example("facade")
An example of a facade poset: the positive integers ordered by \( \_\_\_\_\_\_→ \) divisibility
```

Let us investigate in detail a close variant of this last example: let \( P \) be set of divisors of 12 partially ordered by divisibility. There are two options for representing its elements:

1. as plain integers:
2. as integers, modified to be aware that their parent is $P$:

```python
sage: Q = Poset((divisors(12), attrcall("divides")), facade=False)  # needs sage.graphs

>>> from sage.all import *
>>> P = Poset((divisors(Integer(12)), attrcall("divides")),
            facade=True)  # needs sage.graphs

The advantage of option 1. is that one needs not do conversions back and forth between $P$ and $\mathbb{Z}$. The disadvantage is that this introduces an ambiguity when writing $2 < 3$: does this compare 2 and 3 w.r.t. the natural order on integers or w.r.t. divisibility?

```python
sage: 2 < 3
True

>>> from sage.all import *
>>> Integer(2) < Integer(3)
True

To raise this ambiguity, one needs to explicitly specify the underlying poset as in $2 < _P 3$:

```python
sage: P = Posets().example("facade")
sage: P.lt(2,3)
False

>>> from sage.all import *
>>> P = Posets().example("facade")
>>> P.lt(Integer(2),Integer(3))
False

On the other hand, with option 2. and once constructed, the elements know unambiguously how to compare themselves:

```python
sage: Q(2) < Q(3)  # needs sage.graphs
False
sage: Q(2) < Q(6)  # needs sage.graphs
True

```python

```python
>>> from sage.all import *
>>> Q(Integer(2)) < Q(Integer(3))  # needs sage.graphs
False
>>> Q(Integer(2)) < Q(Integer(6))  # needs sage.graphs
True

Beware that $P(2)$ is still the integer 2. Therefore $P(2) < P(3)$ still compares 2 and 3 as integers!:

```
In short, \( P \) being a facade parent is one of the programmatic counterparts (with e.g. coercions) of the usual mathematical idiom: “for ease of notation, we identify an element of \( P \) with the corresponding integer”. Too many identifications lead to confusion; the lack thereof leads to heavy, if not obfuscated, notations. Finding the right balance is an art, and even though there are common guidelines, it is ultimately up to the writer to choose which identifications to do. This is no different in code.

**See also:**

The following examples illustrate various ways to implement subsets like the set of prime numbers; look at their code for details:

```python
sage: Sets().example("facade")
Set of prime numbers (facade implementation)
sage: Sets().example("inherits")
Set of prime numbers
sage: Sets().example("wrapper")
Set of prime numbers (wrapper implementation)
```

**Specifications**

A parent which is a facade must either:
- call `Parent.__init__()` using the `facade` parameter to specify a parent, or tuple thereof.
- overload the method `facade_for()`.

**Note:** The concept of facade parents was originally introduced in the computer algebra system MuPAD.

**Finite()**

Return the full subcategory of the finite objects of `self`.

**Examples:**

```python
sage: Sets().Finite()
Category of finite sets
sage: Rings().Finite()
Category of finite rings
```

```python
>>> from sage.all import *

>>> Sets().Finite()
Category of finite sets
>>> Sets().Finite()
```

(continues on next page)
Rings().Finite()
Category of finite rings

Infinite()
Return the full subcategory of the infinite objects of self.

EXAMPLES:

```python
sage: Sets().Infinite()
Category of infinite sets
sage: Rings().Infinite()
Category of infinite rings
```

IsomorphicObjects()
Return the full subcategory of the objects of self constructed by isomorphism.

Given a concrete category $A()$ (i.e. a subcategory of Sets()), $A().IsomorphicObjects()$ returns the category of objects of $A()$ endowed with a distinguished description as the image of some other object of $A()$ by an isomorphism in this category.

See Subquotients() for background.

EXAMPLES:

In the following example, $A$ is defined as the image by $x \mapsto x^2$ of the finite set $B = \{1, 2, 3\}$:

```python
sage: A = FiniteEnumeratedSets().IsomorphicObjects().example(); A
The image by some isomorphism of An example of a finite enumerated set: "\{1, 2, 3\}
```

Since $B$ is a finite enumerated set, so is $A$:

```python
sage: A in FiniteEnumeratedSets()
True
sage: A.cardinality()
3
sage: A.list()
[1, 4, 9]
```
The isomorphism from $B$ to $A$ is available as:

```python
sage: A.retract(3)
9
```

```python
>>> from sage.all import *

>>> A.retract(Integer(3))
9
```

and its inverse as:

```python
sage: A.lift(9)
3
```

```python
>>> from sage.all import *

>>> A.lift(Integer(9))
3
```

It often is natural to declare those morphisms as coercions so that one can do $A(b)$ and $B(a)$ to go back and forth between $A$ and $B$ (TODO: refer to a category example where the maps are declared as a coercion). This is not done by default. Indeed, in many cases one only wants to transport part of the structure of $B$ to $A$. Assume for example, that one wants to construct the set of integers $B = \mathbb{Z}$, endowed with $\max$ as addition, and $+$ as multiplication instead of the usual $+$ and $\times$. One can construct $A$ as isomorphic to $B$ as an infinite enumerated set. However $A$ is not isomorphic to $B$ as a ring; for example, for $a \in A$ and $a \in B$, the expressions $a + A(b)$ and $B(a) + b$ give completely different results; hence we would not want the expression $a + b$ to be implicitly resolved to any one of above two, as the coercion mechanism would do.

Coercions also cannot be used with facade parents (see `Sets.Facade`) like in the example above.

We now look at a category of isomorphic objects:

```python
sage: C = Sets().IsomorphicObjects(); C
Category of isomorphic objects of sets

sage: C.super_categories()
[Category of subobjects of sets, Category of quotients of sets]

sage: C.all_super_categories()
[Category of isomorphic objects of sets, Category of subobjects of sets, Category of quotients of sets, Category of subquotients of sets, Category of sets, Category of sets with partial maps, Category of objects]
```

```python
>>> from sage.all import *

>>> C = Sets().IsomorphicObjects(); C
Category of isomorphic objects of sets

>>> C.super_categories()
[Category of subobjects of sets, Category of quotients of sets]

>>> C.all_super_categories()
[Category of isomorphic objects of sets, Category of subobjects of sets,]
```
Unless something specific about isomorphic objects is implemented for this category, one actually get an optimized super category:

```sage
C = Semigroups().IsomorphicObjects(); C
Join of Category of quotients of semigroups
   and Category of isomorphic objects of sets
```
>>> from sage.all import *

>>> C = Semigroups().Quotients(); C
Category of quotients of semigroups

>>> C.super_categories()
[Category of subquotients of semigroups, Category of quotients of sets]

>>> C.all_super_categories()
[Category of quotients of semigroups,
  Category of subquotients of semigroups,
  Category of semigroups,
  Category of subquotients of magmas,
  Category of magmas,
  Category of quotients of sets,
  Category of subquotients of sets,
  Category of sets,
  Category of sets with partial maps,
  Category of objects]

The caller is responsible for checking that the given category admits a well defined category of quotients:

sage: EuclideanDomains().Quotients()
Join of Category of euclidean domains
  and Category of subquotients of monoids
  and Category of quotients of semigroups

>>> from sage.all import *

>>> EuclideanDomains().Quotients()
Join of Category of euclidean domains
  and Category of subquotients of monoids
  and Category of quotients of semigroups

Subobjects()

Return the full subcategory of the objects of self constructed as subobjects.

Given a concrete category As() (i.e. a subcategory of Sets()), As().Subobjects() returns the
category of objects of As() endowed with a distinguished embedding into some other object of As().

Implementing an object of As().Subobjects() is done in the same way as for As().
Subquotients(); namely by providing an ambient space and a lift and a retract map. In the case
of a trivial embedding, the two maps will typically be identity maps that just change the parent of their
argument. See Subquotients() for detailed instructions.

See also:

- Subquotients() for background
- subobjects.SubobjectsCategory
- RegressiveCovariantFunctorialConstruction

EXAMPLES:

sage: C = Sets().Subobjects(); C
Category of subobjects of sets

sage: C.super_categories()
Unless something specific about subobjects is implemented for this category, one actually gets an optimized super category:

```
sage: C = Semigroups().Subobjects(); C
Join of Category of subquotients of semigroups
    and Category of subobjects of sets

>>> from sage.all import *

>>> C = Semigroups().Subobjects(); C
Join of Category of subquotients of semigroups
    and Category of subobjects of sets
```

The caller is responsible for checking that the given category admits a well defined category of subobjects.

**Subquotients()**

Return the full subcategory of the objects of self constructed as subquotients.

Given a concrete category self == As() (i.e. a subcategory of Sets()), As().Subquotients() returns the category of objects of As() endowed with a distinguished description as subquotient of some other object of As().

**EXAMPLES:**

```
sage: Monoids().Subquotients()
Category of subquotients of monoids

>>> from sage.all import *

>>> Monoids().Subquotients()
Category of subquotients of monoids
```

A parent $A$ in As() is further in As().Subquotients() if there is a distinguished parent $B$ in As(), called the ambient set, a subobject $B'$ of $B$, and a pair of maps:

$$l : A \rightarrow B'$$
$$r : B' \rightarrow A$$
called respectively the lifting map and retract map such that \( r \circ l \) is the identity of \( A \) and \( r \) is a morphism in \( \text{As()} \).

**Todo:** Draw the typical commutative diagram.

It follows that, for each operation \( \text{op} \) of the category, we have some property like:

\[
\text{op}_A(e) = r(\text{op}_B(l(e))), \text{ for all } e \in A
\]

This allows for implementing the operations on \( A \) from those on \( B \).

The two most common use cases are:

- **homomorphic images (or quotients),** when \( B' = B \), \( r \) is an homomorphism from \( B \) to \( A \) (typically a canonical quotient map), and \( l \) a section of it (not necessarily a homomorphism); see \( \text{Quotients()} \);
- **subobjects** (up to an isomorphism), when \( l \) is an embedding from \( A \) into \( B \); in this case, \( B' \) is typically isomorphic to \( A \) through the inverse isomorphisms \( r \) and \( l \); see \( \text{Subobjects()} \);

**Note:**

- The usual definition of “subquotient” (Wikipedia article Subquotient) does not involve the lifting map \( l \). This map is required in Sage’s context to make the definition constructive. It is only used in computations and does not affect their results. This is relatively harmless since the category is a concrete category (i.e., its objects are sets and its morphisms are set maps).

- In mathematics, especially in the context of quotients, the retract map \( r \) is often referred to as a projection map instead.

- Since \( B' \) is not specified explicitly, it is possible to abuse the framework with situations where \( B' \) is not quite a subobject and \( r \) not quite a morphism, as long as the lifting and retract maps can be used as above to compute all the operations in \( A \). Use at your own risk!

**Assumptions:**

- For any category \( \text{As()}, \text{As().Subquotients()} \) is a subcategory of \( \text{As()} \).
  
  Example: a subquotient of a group is a group (e.g., a left or right quotient of a group by a non-normal subgroup is not in this category).

- This construction is covariant: if \( \text{As()} \) is a subcategory of \( \text{Bs()} \), then \( \text{As().Subquotients()} \) is a subcategory of \( \text{Bs().Subquotients()} \).
  
  Example: if \( A \) is a subquotient of \( B \) in the category of groups, then it is also a subquotient of \( B \) in the category of monoids.

- If the user (or a program) calls \( \text{As().Subquotients()} \), then it is assumed that subquotients are well defined in this category. This is not checked, and probably never will be. Note that, if a category \( \text{As()} \) does not specify anything about its subquotients, then its subquotient category looks like this:

```python
sage: EuclideanDomains().Subquotients()
Join of Category of euclidean domains
and Category of subquotients of monoids
```

```python
>>> from sage.all import *
>>> EuclideanDomains().Subquotients()
Join of Category of euclidean domains
and Category of subquotients of monoids
```

Interface: the ambient set \( B \) of \( A \) is given by \( A.ambient() \). The subset \( B' \) needs not be specified, so the retract map is handled as a partial map from \( B \) to \( A \).
The lifting and retract map are implemented respectively as methods `A.lift(a)` and `A.retract(b)`. As a shorthand for the former, one can use alternatively `a.lift()`:

```python
sage: S = Semigroups().Subquotients().example(); S
An example of a (sub)quotient semigroup: a quotient of the left zero_
˓→semigroup
sage: S.ambient()
An example of a semigroup: the left zero semigroup
sage: S(3).lift().parent()
An example of a semigroup: the left zero semigroup
sage: S(3) * S(1) == S.retract( S(3).lift() * S(1).lift() )
True
```

See `S?` for more.

**Todo:** use a more interesting example, like $\mathbb{Z}/n\mathbb{Z}$.

See also:

- `Quotients()`, `Subobjects()`, `IsomorphicObjects()`
- `subquotients.SubquotientsCategory`
- `RegressiveCovariantFunctorialConstruction`

**Topological()**

Return the subcategory of the topological objects of `self`.

```python
class Subobjects(category, *args)
    Bases: SubobjectsCategory
    A category for subobjects of sets.
    See also:
    Sets().Subobjects()
    EXAMPLES:

    sage: Sets().Subobjects()
    Category of subobjects of sets
    sage: Sets().Subobjects().all_super_categories()
    [Category of subobjects of sets,
     Category of subquotients of sets,
     Category of sets,
     Category of sets with partial maps,
     Category of objects]
```
class ParentMethods

    Bases: object

class Subquotients(category, *args)

    Bases: SubquotientsCategory

    A category for subquotients of sets.

    See also:

    Sets().Subquotients()

    EXAMPLES:

    sage: Sets().Subquotients()
    Category of subquotients of sets
    sage: Sets().Subquotients().all_super_categories()
    [Category of subquotients of sets, Category of sets, 
     Category of sets with partial maps, 
     Category of objects]

    class ElementMethods

    Bases: object

    lift()

    Lift self to the ambient space for its parent.

    EXAMPLES:

    sage: S = Semigroups().Subquotients().example()
    sage: s = S.an_element()
    sage: s, s.parent()
    (42, An example of a (sub)quotient semigroup: 
     a quotient of the left zero semigroup)
    sage: S.lift(s), S.lift(s).parent()
    (42, An example of a semigroup: the left zero semigroup)
    sage: s.lift(), s.lift().parent()
    (42, An example of a semigroup: the left zero semigroup)
```python
>>> from sage.all import *
>>> S = Semigroups().Subquotients().example()
>>> s = S.an_element()
>>> s, s.parent()
(42, An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup)
>>> S.lift(s), S.lift(s).parent()
(42, An example of a semigroup: the left zero semigroup)
>>> s.lift(), s.lift().parent()
(42, An example of a semigroup: the left zero semigroup)
```

class ParentMethods

Bases: object

ambient()

Return the ambient space for self.

EXAMPLES:

```
>>> S = Semigroups().Subquotients().example()
>>> S.ambient()
An example of a semigroup: the left zero semigroup
```

See also:

Sets.SubcategoryMethods.Subquotients() for the specifications and lift() and retract().

lift(x)

Lift x to the ambient space for self.

INPUT:

• x – an element of self

EXAMPLES:

```
>>> S = Semigroups().Subquotients().example()
>>> s = S.an_element()
>>> s, s.parent()
(42, An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup)
>>> S.lift(s), S.lift(s).parent()
(42, An example of a semigroup: the left zero semigroup)
>>> s.lift(), s.lift().parent()
(42, An example of a semigroup: the left zero semigroup)
```
```
See also:


retract(x)
Retract x to self.

INPUT:
• x – an element of the ambient space for self

See also:


EXAMPLES:

sage: S = Semigroups().Subquotients().example()
sage: s = S.ambient().an_element()
sage: s, s.parent()
(42, An example of a semigroup: the left zero semigroup)
sage: S.retract(s), S.retract(s).parent()
(42, An example of a (sub)quotient semigroup: a quotient of the left zero semigroup)

Topological
alias of TopologicalSpaces
class WithRealizations (category, *args)
    Bases: WithRealizationsCategory
class ParentMethods
    Bases: object
class Realizations (parent_with_realization)
    Bases: Category_realization_of_parent

super_categories()
Category Framework, Release 10.4

needs sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field

```
A.Realizations().super_categories()
```

needs sage.modules
[Category of realizations of sets]

a_realization()

Return a realization of self.

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A
```

```
A.a_realization()
```

``` needs sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
```

```
from sage.all import *
```

```
A = Sets().WithRealizations().example(); A
```

```
A.a_realization()
```

``` needs sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
```

facade_for()

Return the parents self is a facade for, that is the realizations of self

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A
```

```
A.facade_for()
```

```
[The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis,
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis,
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis]
```

```
sage: # needs sage.combinat sage.modules
sage: f = A.F().an_element(); f
```

```
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

```
sage: i = A.In().an_element(); i
```

```
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
```

```
sage: o = A.Out().an_element(); o
```

```
Out[{}] + 2*Out[{1}] + 3*Out[{2}] + Out[{1, 2}]
```

```
sage: f in A, i in A, o in A
```

(True, True, True)
```python
from sage.all import *

A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field

A.facade_for()
[The subset algebra of {1, 2, 3} over Rational Field in the...
Fundamental basis,
The subset algebra of {1, 2, 3} over Rational Field in the In basis,
The subset algebra of {1, 2, 3} over Rational Field in the Out basis]

# needs sage.combinat sage.modules

A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field

f = A.F().an_element(); f
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]

i = A.In().an_element(); i
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]

o = A.Out().an_element(); o
Out[{}] + 2*Out[{1}] + 3*Out[{2}] + Out[{1, 2}]

f in A, i in A, o in A
(True, True, True)

inject_shorthands (shorthands=None, verbose=True)
Import standard shorthands into the global namespace.

INPUT:
• shorthands – a list (or iterable) of strings (default: self._shorthands) or "all" (for
  self._shorthands_all)
• verbose – boolean (default True);
  whether to print the defined shorthands

EXAMPLES:
When computing with a set with multiple realizations, like SymmetricFunctions or SubsetAlgebra, it is convenient to define shorthands for the various realizations, but cumbersome to do it by hand:

```
This method automates the process:

```
sage: # needs sage.combinat sage.modules
sage: S.inject_shorthands()
Defining e as shorthand for
    Symmetric Functions over Integer Ring in the elementary basis
Defining f as shorthand for
    Symmetric Functions over Integer Ring in the forgotten basis
Defining h as shorthand for
    Symmetric Functions over Integer Ring in the homogeneous basis
Defining m as shorthand for
    Symmetric Functions over Integer Ring in the monomial basis
Defining p as shorthand for
    Symmetric Functions over Integer Ring in the powersum basis
Defining s as shorthand for
    Symmetric Functions over Integer Ring in the Schur basis
sage: e
Symmetric Functions over Integer Ring in the elementary basis
sage: p
Symmetric Functions over Integer Ring in the powersum basis
sage: s
Symmetric Functions over Integer Ring in the Schur basis
```

Sometimes, like for symmetric functions, one can request for all shorthands to be defined, including less common ones:

```
sage: S.inject_shorthands("all") # needs lrcalc_python sage.combinat sage.modules
Defining e as shorthand for
    Symmetric Functions over Integer Ring in the elementary basis
Defining f as shorthand for
    Symmetric Functions over Integer Ring in the forgotten basis
Defining h as shorthand for
    Symmetric Functions over Integer Ring in the homogeneous basis
Defining m as shorthand for
    Symmetric Functions over Integer Ring in the monomial basis
Defining p as shorthand for
    Symmetric Functions over Integer Ring in the powersum basis
Defining s as shorthand for
    Symmetric Functions over Integer Ring in the Schur basis
```
Symmetric Functions over Integer Ring in the elementary basis
Defining \( f \) as shorthand for
Symmetric Functions over Integer Ring in the forgotten basis
Defining \( h \) as shorthand for
Symmetric Functions over Integer Ring in the homogeneous basis
Defining \( ht \) as shorthand for
Symmetric Functions over Integer Ring in the
induced trivial symmetric group character basis
Defining \( m \) as shorthand for
Symmetric Functions over Integer Ring in the monomial basis
Defining \( o \) as shorthand for
Symmetric Functions over Integer Ring in the orthogonal basis
Defining \( p \) as shorthand for
Symmetric Functions over Integer Ring in the powersum basis
Defining \( s \) as shorthand for
Symmetric Functions over Integer Ring in the Schur basis
Defining \( sp \) as shorthand for
Symmetric Functions over Integer Ring in the symplectic basis
Defining \( st \) as shorthand for
Symmetric Functions over Integer Ring in the
irreducible symmetric group character basis
Defining \( w \) as shorthand for
Symmetric Functions over Integer Ring in the Witt basis

```python
from sage.all import *
S.inject_shorthands("all")
```

The messages can be silenced by setting `verbose=False`:

```
sage: # needs sage.combinat sage.modules
sage: Q = QuasiSymmetricFunctions(ZZ)
```
sage: Q.inject_shorthands(\texttt{verbose=False})
5*F[1, 1, 1, 1] - 5*F[1, 1, 2] - 3*F[1, 2, 1] + 6*F[1, 3] +
sage: F
Quasisymmetric functions over the Integer Ring in the
Fundamental basis
sage: M
Quasisymmetric functions over the Integer Ring in the
Monomial basis

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> Q = QuasiSymmetricFunctions(ZZ)
>>> Q.inject_shorthands(\texttt{verbose=False})
>>> F[Integer(1),Integer(2),Integer(1)] + Integer(5)*M[Integer(1),
Integer(3)] + F[Integer(2)]**Integer(2)
5*F[1, 1, 1, 1] - 5*F[1, 1, 2] - 3*F[1, 2, 1] + 6*F[1, 3] +
```

```
sage: F
Quasisymmetric functions over the Integer Ring in the
Fundamental basis
```

```
sage: M
Quasisymmetric functions over the Integer Ring in the
Monomial basis
```

One can also just import a subset of the shorthands:

```python
sage: # needs sage.combinat sage.modules
sage: SQ = SymmetricFunctions(QQ)
sage: SQ.inject_shorthands(['p', 's'], \texttt{verbose=False})
sage: p
Symmetric Functions over Rational Field in the powersum basis
sage: s
Symmetric Functions over Rational Field in the Schur basis
```

```python
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> SQ = SymmetricFunctions(QQ)
>>> SQ inject_shorthands(['p', 's'], \texttt{verbose=False})
>>> p
Symmetric Functions over Rational Field in the powersum basis
```

```
sage: s
Symmetric Functions over Rational Field in the Schur basis
```

Note that \texttt{e} is left unchanged:

```python
sage: e
\texttt{# needs sage.combinat sage.modules}
Symmetric Functions over Integer Ring in the elementary basis
```

```python
>>> from sage.all import *
>>> e
```

```
sage: e
Symmetric Functions over Integer Ring in the elementary basis
```

4.143. Sets 1249
realizations()

Return all the realizations of self that self is aware of.

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A
# needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
sage: A.realizations()
# needs sage.modules
[The subset algebra of {1, 2, 3} over Rational Field in the...
  Fundamental basis,
  The subset algebra of {1, 2, 3} over Rational Field in the In basis,
  The subset algebra of {1, 2, 3} over Rational Field in the Out basis]
```

```
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
# needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
>>> A.realizations()
# needs sage.modules
[The subset algebra of {1, 2, 3} over Rational Field in the...
  Fundamental basis,
  The subset algebra of {1, 2, 3} over Rational Field in the In basis,
  The subset algebra of {1, 2, 3} over Rational Field in the Out basis]
```

Note: Constructing a parent \( P \) in the category \( A.Realizations() \) automatically adds \( P \) to this list by calling \( A._register_realization(A) \).
sage: Sets().WithRealizations().extra_super_categories()
[Category of facade sets]
sage: Sets().WithRealizations().super_categories()
[Category of facade sets]

```python
>>> from sage.all import *
>>> Sets().WithRealizations().extra_super_categories()
[Category of facade sets]
>>> Sets().WithRealizations().super_categories()
[Category of facade sets]
```

**example** *(choice=None)*

Return examples of objects of `Sets()`, as per `Category.example()`.

**EXAMPLES:**

```python
sage: Sets().example()
Set of prime numbers (basic implementation)
sage: Sets().example("inherits")
Set of prime numbers
sage: Sets().example("facade")
Set of prime numbers (facade implementation)
sage: Sets().example("wrapper")
Set of prime numbers (wrapper implementation)
```

```python
>>> from sage.all import *
>>> Sets().example()
Set of prime numbers (basic implementation)
>>> Sets().example("inherits")
Set of prime numbers
>>> Sets().example("facade")
Set of prime numbers (facade implementation)
>>> Sets().example("wrapper")
Set of prime numbers (wrapper implementation)
```

**super_categories()**

We include `SetsWithPartialMaps` between `Sets` and `Objects` so that we can define morphisms between sets that are only partially defined. This is also to have the `Homset` constructor not complain that `SetsWithPartialMaps` is not a supercategory of `Fields`, for example.

**EXAMPLES:**

```python
sage: Sets().super_categories()
[Category of sets with partial maps]
```

```python
>>> from sage.all import *
>>> Sets().super_categories()
[Category of sets with partial maps]
```

sage.categories.sets_cat.print_compare(x, y)
Helper method used in Sets.ParentMethods._test_elements_eq_symmetric(), Sets.ParentMethods._test_elements_eq_transitive().

INPUT:

• x – an element
• y – an element

EXAMPLES:

```python
sage: from sage.categories.sets_cat import print_compare
sage: print_compare(1,2)
1 != 2
sage: print_compare(1,1)
1 == 1
```

```python
>>> from sage.all import *
>>> from sage.categories.sets_cat import print_compare

>>> print_compare(Integer(1),Integer(2))
1 != 2
```

4.144 Sets With a Grading

```python
class sage.categories.sets_with_grading.SetsWithGrading
    Bases: Category

The category of sets with a grading.

A set with a grading is a set $S$ equipped with a grading by some other set $I$ (by default the set $\mathbb{N}$ of the non-negative integers):

$$S = \bigcup_{i \in I} S_i$$

where the graded components $S_i$ are (usually finite) sets. The grading function maps each element $s$ of $S$ to its grade $i$, so that $s \in S_i$.

From implementation point of view, if the graded set is enumerated then each graded component should be enumerated (there is a check in the method _test_graded_components()). The contrary needs not be true.

To implement this category, a parent must either implement graded_component() or subset(). If only subset() is implemented, the first argument must be the grading for compatibility with graded_component(). Additionally either the parent must implement grading() or its elements must implement a method grade(). See the example sage.categories.examples.sets_with_grading.NonnegativeIntegers.

Finally, if the graded set is enumerated (see EnumeratedSets) then each graded component should be enumerated. The contrary needs not be true.

EXAMPLES:

A typical example of a set with a grading is the set of non-negative integers graded by themselves:
sage: N = SetsWithGrading().example(); N
Non negative integers
sage: N.category()
Category of facade infinite sets with grading
sage: N.grading_set()
Non negative integers

```python
>> from sage.all import *
>> N = SetsWithGrading().example(); N
Non negative integers
>> N.category()
Category of facade infinite sets with grading
>> N.grading_set()
Non negative integers
```

The *grading function* is given by `N.grading`:

```python
sage: N.grading(4)
4

>> from sage.all import *
>> N.grading(Integer(4))
4
```

The graded component $N_i$ is the set with one element $i$:

```python
sage: N.graded_component(grade=5)
{5}
sage: N.graded_component(grade=42)
{42}

>> from sage.all import *
>> N.graded_component(grade=Integer(5))
{5}
>> N.graded_component(grade=Integer(42))
{42}
```

Here are some information about this category:

```python
sage: SetsWithGrading()
Category of sets with grading
sage: SetsWithGrading().super_categories()
[Category of sets]
sage: SetsWithGrading().all_super_categories()
[Category of sets with grading,
Category of sets,
Category of sets with partial maps,
Category of objects]

>> from sage.all import *
>> SetsWithGrading()
Category of sets with grading
>> SetsWithGrading().super_categories()
[Category of sets]
>> SetsWithGrading().all_super_categories()
[Category of sets with grading,
...]
```
Todo:

- This should be moved to `Sets().WithGrading()`.
- Should the grading set be a parameter for this category?
- Does the enumeration need to be compatible with the grading? Be careful that the fact that graded components are allowed to be finite or infinite make the answer complicated.

```python
class ParentMethods
    Bases: object

    generating_series()
        Default implementation for generating series.

        OUTPUT:

        A series, indexed by the grading set.

        EXAMPLES:

        sage: N = SetsWithGrading().example(); N
        Non negative integers
        sage: N.generating_series()
        1/(-z + 1)

        sage: Permutations().generating_series()  # needs sage.combinat
        1 + z + 2*z^2 + 6*z^3 + 24*z^4 + 120*z^5 + 720*z^6 + O(z^7)

        .. TODO::
            - Very likely, this should always return a lazy power series.

    graded_component(grade)
        Return the graded component of `self` with grade `grade`.

        The default implementation just calls the method `subset()` with the first argument `grade`.

        EXAMPLES:

        sage: N = SetsWithGrading().example(); N
        Non negative integers
        sage: N.graded_component(3)
        {3}

        >>> from sage.all import *
        >>> N = SetsWithGrading().example(); N
        Non negative integers
        >>> N.graded_component(Integer(3))
        {3}
```
grading(elt)

Return the grading of the element elt of self.

This default implementation calls elt.grade().

EXAMPLES:

```python
sage: N = SetsWithGrading().example(); N
Non negative integers
sage: N.grading(4)
4
```

```python
>>> from sage.all import *
>>> N = SetsWithGrading().example(); N
Non negative integers
>>> N.grading(Integer(4))
4
```

grading_set()

Return the set self is graded by. By default, this is the set of non-negative integers.

EXAMPLES:

```python
sage: SetsWithGrading().example().grading_set()
Non negative integers
```

```python
>>> from sage.all import *
>>> SetsWithGrading().example().grading_set()
Non negative integers
```

subset(*args, **options)

Return the subset of self described by the given parameters.

See also:
- graded_component()

EXAMPLES:

```python
sage: W = WeightedIntegerVectors([3,2,1]); W
# needs sage.combinat
Integer vectors weighted by [3, 2, 1]
sage: W.subset(4)
# needs sage.combinat
Integer vectors of 4 weighted by [3, 2, 1]
```

```python
>>> from sage.all import *
>>> W = WeightedIntegerVectors([Integer(3),Integer(2),Integer(1)]); W
# needs sage.combinat
Integer vectors weighted by [3, 2, 1]
>>> W.subset(Integer(4))
# needs sage.combinat
Integer vectors of 4 weighted by [3, 2, 1]
```

super_categories()

EXAMPLES:
sage: SetsWithGrading().super_categories()
[Category of sets]

>>> from sage.all import *
>>> SetsWithGrading().super_categories()
[Category of sets]

### 4.145 SetsWithPartialMaps

```python
class sage.categories.sets_with_partial_maps.SetsWithPartialMaps
    Bases: Category_singleton

The category whose objects are sets and whose morphisms are maps that are allowed to raise a ValueError on some inputs.

This category is equivalent to the category of pointed sets, via the equivalence sending an object X to X union \{error\}, a morphism f to the morphism of pointed sets that sends x to f(x) if f does not raise an error on x, or to error if it does.

EXAMPLES:
```

sage: SetsWithPartialMaps()
Category of sets with partial maps

sage: SetsWithPartialMaps().super_categories()
[Category of objects]

>>> from sage.all import *
>>> SetsWithPartialMaps()
Category of sets with partial maps

>>> SetsWithPartialMaps().super_categories()
[Category of objects]

super_categories()

EXAMPLES:
```

```
4.146 Shephard Groups

```python
class sage.categories.shephard_groups.ShephardGroups:
    Bases: Category_singleton
    The category of Shephard groups.
    EXAMPLES:
    sage: from sage.categories.shephard_groups import ShephardGroups
    sage: C = ShephardGroups(); C
    Category of shephard groups
    >>> from sage.all import *
    >>> from sage.categories.shephard_groups import ShephardGroups
    >>> C = ShephardGroups(); C
    Category of shephard groups
    super_categories():
    EXAMPLES:
    sage: from sage.categories.shephard_groups import ShephardGroups
    sage: ShephardGroups().super_categories()
    [Category of finite generalized Coxeter groups]
```

4.147 Simplicial Complexes

```python
class sage.categories.simplicial_complexes.SimplicialComplexes:
    Bases: Category_singleton
    The category of abstract simplicial complexes.
    An abstract simplicial complex $A$ is a collection of sets $X$ such that:
    - $\emptyset \in A$,
    - if $X \subseteq Y \in A$, then $X \in A$.
    Todo: Implement the category of simplicial complexes considered as CW complexes and rename this to the category of AbstractSimplicialComplexes with appropriate functors.
    EXAMPLES:
    sage: from sage.categories.simplicial_complexes import SimplicialComplexes
    sage: C = SimplicialComplexes(); C
    Category of simplicial complexes
```
```python
>>> from sage.all import *
>>> from sage.categories.simplicial_complexes import SimplicialComplexes
>>> C = SimplicialComplexes(); C
Category of simplicial complexes

class Connected(base_category)
    Bases: CategoryWithAxiom

    The category of connected simplicial complexes.

    EXAMPLES:

sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: C = SimplicialComplexes().Connected()
sage: TestSuite(C).run()

>>> from sage.all import *
>>> from sage.categories.simplicial_complexes import SimplicialComplexes
>>> C = SimplicialComplexes().Connected()

class Finite(base_category)
    Bases: CategoryWithAxiom

    Category of finite simplicial complexes.

class ParentMethods
    Bases: object

    dimension()

        Return the dimension of self.

        EXAMPLES:

        sage: S = SimplicialComplex([[1,3,4], [1,2],[2,5],[4,5]])
        # needs sage.graphs
        sage: S.dimension()
        # needs sage.graphs
        2

        class ParentMethods
            Bases: object

            faces()

                Return the faces of self.

                EXAMPLES:
```

1258 Chapter 4. Individual Categories
sage: S = SimplicialComplex([[1, 3, 4], [1, 2], [2, 5], [4, 5]])  # needs sage.graphs
sage: S.faces()  # needs sage.graphs
{-1: {()}, 0: {(1,), (2,), (3,), (4,), (5,)}, 1: {(1, 2), (1, 3), (1, 4), (2, 5), (3, 4), (4, 5)}, 2: {(1, 3, 4)}}

>>> from sage.all import *
>>> S = SimplicialComplex([[Integer(1), Integer(3), Integer(4)],
[Integer(1), Integer(2)], [Integer(2), Integer(5)], [Integer(4),
Integer(5)]]))  # needs sage.graphs
>>> S.faces()  # needs sage.graphs
{-1: {()}, 0: {(1,), (2,), (3,), (4,), (5,)}, 1: {(1, 2), (1, 3), (1, 4), (2, 5), (3, 4), (4, 5)}, 2: {(1, 3, 4)}}

facets()
Return the facets of self.

EXAMPLES:
sage: S = SimplicialComplex([[1, 3, 4], [1, 2], [2, 5], [4, 5]])  # needs sage.graphs
sage: sorted(S.facets())  # needs sage.graphs
[(1, 2), (1, 3, 4), (2, 5), (4, 5)]

class SubcategoryMethods
Bases: object

Connected()
Return the full subcategory of the connected objects of self.

EXAMPLES:
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: SimplicialComplexes().Connected()
Category of connected simplicial complexes

>>> from sage.all import *
>>> from sage.categories.simplicial_complexes import SimplicialComplexes
>>> SimplicialComplexes().Connected()
Category of connected simplicial complexes
super_categories()

Return the super categories of self.

EXAMPLES:

```
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: SimplicialComplexes().super_categories()
[Category of sets]
```

```
>>> from sage.all import *
>>> from sage.categories.simplicial_complexes import SimplicialComplexes
>>> SimplicialComplexes().super_categories()
[Category of sets]
```

## 4.148 Simplicial Sets

class `sage.categories.simplicial_sets.SimplicialSets`

Bases: `Category_singleton`

The category of simplicial sets.

A simplicial set $X$ is a collection of sets $X_i$, indexed by the non-negative integers, together with maps

\[
\begin{align*}
d_i & : X_n \to X_{n-1}, \quad 0 \leq i \leq n \quad \text{(face maps)} \\
s_j & : X_n \to X_{n+1}, \quad 0 \leq j \leq n \quad \text{(degeneracy maps)}
\end{align*}
\]

satisfying the simplicial identities:

\[
\begin{align*}
d_i d_j &= d_{j-1} d_i \quad \text{if } i < j \\
d_i s_j &= s_{j-1} d_i \quad \text{if } i < j \\
d_j s_j &= 1 = d_{j+1} s_j \\
d_i s_j &= s_{j} d_{i-1} \quad \text{if } i > j + 1 \\
s_i s_j &= s_{j+1} s_i \quad \text{if } i < j
\end{align*}
\]

Morphisms are sequences of maps $f_i : X_i \to Y_i$ which commute with the face and degeneracy maps.

EXAMPLES:

```
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: C = SimplicialSets(); C
Category of simplicial sets
```

```
>>> from sage.all import *
>>> from sage.categories.simplicial_sets import SimplicialSets
>>> C = SimplicialSets(); C
Category of simplicial sets
```

class `Finite(base_category)`

Bases: `CategoryWithAxiom`

Category of finite simplicial sets.

The objects are simplicial sets with finitely many non-degenerate simplices.
class Homsets (category, *args)
    Bases: HomsetsCategory

class Endset (base_category)
    Bases: CategoryWithAxiom

class ParentMethods
    Bases: object

    one()
    Return the identity morphism in Hom(S, S).

    EXAMPLES:

    sage: T = simplicial_sets.Torus()
    # needs sage.graphs
    sage: Hom(T, T).identity()  # needs sage.graphs
    Simplicial set endomorphism of Torus
    Defn: Identity map

>>> from sage.all import *
>>> T = simplicial_sets.Torus()  # needs sage.graphs
>>> Hom(T, T).identity()  # needs sage.graphs
Simplicial set endomorphism of Torus
    Defn: Identity map

class ParentMethods
    Bases: object

    is_finite()
    Return True if this simplicial set is finite, i.e., has a finite number of nondegenerate simplices.

    EXAMPLES:

    sage: simplicial_sets.Torus().is_finite()  # needs sage.graphs
    True
    sage: C5 = groups.misc.MultiplicativeAbelian([5])  # needs sage.graphs sage.groups
    sage: simplicial_sets.ClassifyingSpace(C5).is_finite()  # needs sage.graphs sage.groups
    False

>>> from sage.all import *
>>> simplicial_sets.Torus().is_finite()  # needs sage.graphs
    True
>>> C5 = groups.misc.MultiplicativeAbelian([Integer(5)])  # needs sage.graphs sage.groups
>>> simplicial_sets.ClassifyingSpace(C5).is_finite()  # needs sage.graphs sage.groups
    False
is_pointed()

Return True if this simplicial set is pointed, i.e., has a base point.

EXAMPLES:

```python
sage: # needs sage.graphs
sage: from sage.topology.simplicial_set import AbstractSimplex,
    SimplicialSet
sage: v = AbstractSimplex(0)
sage: w = AbstractSimplex(0)
sage: e = AbstractSimplex(1)
sage: X = SimplicialSet({e: (v, w)})
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)
sage: X.is_pointed()
False
sage: Y.is_pointed()
True
```

set_base_point (point)

Return a copy of this simplicial set in which the base point is set to point.

INPUT:
- point – a 0-simplex in this simplicial set

EXAMPLES:

```python
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.topology.simplicial_set import AbstractSimplex,
    SimplicialSet
>>> v = AbstractSimplex(Integer(0))
>>> w = AbstractSimplex(Integer(0))
>>> e = AbstractSimplex(Integer(1))
>>> X = SimplicialSet({e: (v, w)})
>>> Y = SimplicialSet({e: (v, w)}, base_point=w)
>>> X.is_pointed()
False
>>> Y.is_pointed()
True
```
class Pointed (base_category)

    Bases: CategoryWithAxiom

class Finite (base_category)

    Bases: CategoryWithAxiom

class ParentMethods

    Bases: object

fat_wedge (n)

    Return the $n$-th fat wedge of this pointed simplicial set.

    This is the subcomplex of the $n$-fold product $X^n$ consisting of those points in which at least one
    factor is the base point. Thus when $n = 2$, this is the wedge of the simplicial set with itself, but
    when $n$ is larger, the fat wedge is larger than the $n$-fold wedge.

EXAMPLES:

sage: # needs sage.graphs
sage: S1 = simplicial_sets.Sphere(1)
sage: S1.fat_wedge(0)
Point
sage: S1.fat_wedge(1)
S^1
sage: S1.fat_wedge(2).fundamental_group()  # needs sage.groups
Finitely presented group < e0, e1 | >
sage: S1.fat_wedge(4).homology()  # needs sage.modules
{0: 0, 1: Z x Z x Z x Z, 2: Z^6, 3: Z x Z x Z x Z x Z}
smash_product (*other*)

Return the smash product of this simplicial set with other.

INPUT:
- other – one or several simplicial sets

EXAMPLES:

```python
sage: # needs sage.graphs sage.groups
sage: S1 = simplicial_sets.Sphere(1)
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)
sage: X = S1.smash_product(RP2)
sage: X.homology(base_ring=GF(2)) # sagemodules
{0: Vector space of dimension 0 over Finite Field of size 2,
  1: Vector space of dimension 0 over Finite Field of size 2,
  2: Vector space of dimension 1 over Finite Field of size 2,
  3: Vector space of dimension 1 over Finite Field of size 2}
```

unset_base_point ()

Return a copy of this simplicial set in which the base point has been forgotten.

EXAMPLES:
```python
sage: from sage.topology.simplicial_set import AbstractSimplex,
   SimplicialSet
sage: v = AbstractSimplex(0, name='v_0')
sage: w = AbstractSimplex(0, name='w_0')
sage: e = AbstractSimplex(1)
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)
sage: Y.is_pointed()
True
sage: Y.base_point()
w_0
sage: Z = Y.unset_base_point()
sage: Z.is_pointed()
False
```

```python
>>> from sage.all import *
>>> from sage.topology.simplicial_set import AbstractSimplex,
   SimplicialSet
>>> v = AbstractSimplex(Integer(0), name='v_0')
>>> e = AbstractSimplex(Integer(1))
>>> S1 = SimplicialSet({e: (v, v)}, base_point=v)
>>> S1.is_pointed()
True
```

```python
class ParentMethods
   Bases: object

   base_point()
      Return this simplicial set's base point

   EXAMPLES:
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
>>> S1.is_pointed()
True
>>> S1.base_point()
*

**base_point_map** (*domain=None*)

Return a map from a one-point space to this one, with image the base point.

This raises an error if this simplicial set does not have a base point.

**INPUT:**

- *domain* – optional, default None. Use this to specify a particular one-point space as the domain.

  The default behavior is to use the `sage.topology.simplicial_set.Point()` function to use a standard one-point space.

**EXAMPLES:**

```python
sage: # needs sage.graphs
sage: T = simplicial_sets.Torus()
sage: f = T.base_point_map(); f
Simplicial set morphism:
    From: Point
    To:   Torus
    Defn: Constant map at (v_0, v_0)
sage: S3 = simplicial_sets.Sphere(3)
sage: g = S3.base_point_map()
sage: f.domain() == g.domain()
True
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
# needs sage.groups
sage: temp = simplicial_sets.Simplex(0)
sage: pt = temp.set_base_point(temp.n_cells(0)[0])
sage: h = RP3.base_point_map(domain=pt)
# needs sage.groups
sage: f.domain() == h.domain()
False
sage: C5 = groups.misc.MultiplicativeAbelian([5])
# needs sage.graphs sage.groups
sage: BC5 = simplicial_sets.ClassifyingSpace(C5)
# needs sage.graphs sage.groups
sage: BC5.base_point_map()
Simplicial set morphism:
    From: Point
    To:   Classifying space of Multiplicative Abelian group isomorphic to C5
    Defn: Constant map at 1
```

```python
>>> from sage.all import *
```
Defn: Constant map at \((v_0, v_0)\)

```python
>>> S3 = simplicial_sets.Sphere(Integer(3))
>>> g = S3.base_point_map()
>>> f.domain() == g.domain()
True
```

```python
>>> RP3 = simplicial_sets.RealProjectiveSpace(Integer(3))

# needs sage.groups
```

```python
>>> temp = simplicial_sets.Simplex(Integer(0))
>>> pt = temp.set_base_point(temp.n_cells(Integer(0))[Integer(0)])
```

```python
>>> h = RP3.base_point_map(domain=pt)
```

```python
# needs sage.groups
```

```python
>>> f.domain() == h.domain()
```

```python
False
```

```python
>>> C5 = groups.misc.MultiplicativeAbelian([Integer(5)])

# needs sage.graphs sage.groups
```

```python
>>> BC5 = simplicial_sets.ClassifyingSpace(C5)
```

```python
# needs sage.graphs sage.groups
```

```python
>>> BC5.base_point_map()
```

Simplicial set morphism:
- From: Point
- To: Classifying space of Multiplicative Abelian group isomorphic to C5

Defn: Constant map at 1

**connectivity** \((\text{max\_dim}=\text{None})\)

Return the connectivity of this pointed simplicial set.

**INPUT:**
- \(\text{max\_dim}\) – specify a maximum dimension through which to check. This is required if this simplicial set is simply connected and not finite.

The dimension of the first nonzero homotopy group. If simply connected, this is the same as the dimension of the first nonzero homology group.

**Warning:** See the warning for the \texttt{is\_simply\_connected()\ method.}

The connectivity of a contractible space is \(+\infty\).

**EXAMPLES:**

```python
sage: # needs sage.groups sage.graphs
sage: simplicial_sets.Sphere(3).connectivity()
2
```

```python
sage: simplicial_sets.Sphere(0).connectivity()
-1
```

```python
sage: K = simplicial_sets.Simplex(4)
```

```python
sage: K = K.set_base_point(K.n_cells(0)[0])
```

```python
sage: K.connectivity()
+Infinity
```

```python
sage: X = simplicial_sets.Torus().suspension(2)
```

```python
sage: X.connectivity()
2
```

(continues on next page)
**cover** *(character)*

Return the cover of the simplicial set associated to a character of the fundamental group.

The character is represented by a dictionary, that assigns an element of a finite group to each non-degenerate 1-dimensional cell. It should correspond to an epimorphism from the fundamental group.

**INPUT:**

• **character** – a dictionary

**EXAMPLES:**

```python
sage: # needs sage.graphs sage.groups
groups.misc.MultiplicativeAbelian([2])  # needs sage.graphs sage.groups
sage: BC2 = simplicial_sets.ClassifyingSpace(C2)  # needs sage.graphs sage.groups
sage: BC2.connectivity()  # needs sage.graphs sage.groups
0

>>> from sage.all import *
>>> simplicial_sets.Sphere(Integer(3)).connectivity() 2
>>> simplicial_sets.Sphere(Integer(0)).connectivity() -1
>>> K = simplicial_sets.Simplex(Integer(4))
>>> K = K.set_base_point(K.n_cells(Integer(0))[Integer(0)])
>>> K.connectivity() +Infinity
>>> X = simplicial_sets.Torus().suspension(Integer(2))
>>> X.connectivity()
2

>>> C2 = groups.misc.MultiplicativeAbelian([Integer(2)])  # needs sage.graphs sage.groups
>>> BC2 = simplicial_sets.ClassifyingSpace(C2)  # needs sage.graphs sage.groups
>>> BC2.connectivity()  # needs sage.graphs sage.groups
0
```
sage: C.homology(1)  # needs sage.modules
Z x Z x Z x Z

sage: C.fundamental_group()
Finitely presented group < e0, e1, e2, e3 | >

>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> S1 = simplicial_sets.Sphere(Integer(1))
>>> W = S1.wedge(S1)
>>> G = CyclicPermutationGroup(Integer(3))
>>> (a, b) = W.n_cells(Integer(1))
>>> C = W.cover({a : G.gen(Integer(0)), b : G.gen(Integer(0))**Integer(2)})
>>> C.face_data()
{(*, ()): None,
(*, (1,2,3)): None,
(*, (1,3,2)): None,
(sigma_1, ()): ((*, (1,2,3)), (*, ())),
(sigma_1, ()): ((*, (1,3,2)), (*, ())),
(sigma_1, (1,2,3)): ((*, (1,3,2)), (*, (1,2,3))),
(sigma_1, (1,2,3)): ((*, (1,3,2)), (*, (1,2,3))),
(sigma_1, (1,3,2)): ((*, ()), (*, (1,3,2))),
(sigma_1, (1,3,2)): ((*, (1,2,3)), (*, (1,3,2)))}

>>> C.homology(Integer(1))  # needs sage.modules
Z x Z x Z x Z

C.fundamental_group()
Finitely presented group < e0, e1, e2, e3 | >

covering_map(character)

Return the covering map associated to a character.

The character is represented by a dictionary that assigns an element of a finite group to each non-degenerate 1-dimensional cell. It should correspond to an epimorphism from the fundamental group.

INPUT:
- character—a dictionary

EXAMPLES:

sage: # needs sage.graphs sage.groups
sage: S1 = simplicial_sets.Sphere(1)
sage: W = S1.wedge(S1)
sage: G = CyclicPermutationGroup(3)
sage: a, b = W.n_cells(1)

sage: C = W.covering_map({a : G.gen(0), b : G.one()}); C
Simplicial set morphism:
From: Simplicial set with 9 non-degenerate simplices
To: Wedge: (S^1 v S^1)
Defn: [(*, ()), (*, (1,2,3)), (*, (1,3,2)), (sigma_1, ()),
    (sigma_1, ()), (sigma_1, (1,2,3)), (sigma_1, (1,2,3)),
    (sigma_1, (1,3,2)), (sigma_1, (1,3,2))]
    --> [*, *, *, sigma_1, sigma_1, sigma_1, sigma_1, sigma_1, sigma_1]

sage: C.domain()
Simplicial set with 9 non-degenerate simplices
\begin{verbatim}
>>> from sage.all import *

>>> W = S1.wedge(S1)

>>> a, b = W.n_cells(Integer(1))

>>> C = W.covering_map({a : G.gen(Integer(0)), b : G.one()}); C

Simplicial set morphism:
From: Simplicial set with 9 non-degenerate simplices
to: Wedge: (S^1 v S^1)
defn: ['(*, ()), (*, (1,2,3)), (*, (1,3,2)), (sigma_1, ()),
     (sigma_1, ()), (sigma_1, (1,2,3)), (sigma_1, (1,2,3)),
     (sigma_1, (1,3,2)), (sigma_1, (1,3,2))']

>>> C.domain()
Simplicial set with 9 non-degenerate simplices

>>> C.domain().face_data()
{(*, ()): None,
 (*, (1,2,3)): None,
 (*, (1,3,2)): None,
 (sigma_1, ()): (('*, (1,2,3)), ('*, ())),
 (sigma_1, ()): (('', ()), ('*, ())),
 (sigma_1, (1,2,3)): (('', (1,3,2)), ('*, (1,2,3)));
 (sigma_1, (1,2,3)): (('', (1,2,3)), ('*, (1,2,3)));
 (sigma_1, (1,3,2)): (('', ()), ('*, (1,3,2)));
 (sigma_1, (1,3,2)): (('', (1,3,2)), ('*, (1,3,2))))
\end{verbatim}

**fundamental_group** (*simplify=True*)

Return the fundamental group of this pointed simplicial set.

**INPUT:**
- *simplify* (bool, optional True) – if False, then return a presentation of the group in terms of generators and relations. If True, the default, simplify as much as GAP is able to.

**Algorithm:** we compute the edge-path group – see Section 19 of [Kan1958] and Wikipedia article Fundamental_group. Choose a spanning tree for the connected component of the 1-skeleton containing the base point, and then the group’s generators are given by the non-degenerate edges. There are two types of relations: $e = 1$ if $e$ is in the spanning tree, and for every 2-simplex, if its faces are $e_0$, $e_1$, and $e_2$, then we impose the relation $e_0 e_1^{-1} e_2 = 1$, where we first set $e_1 = 1$ if $e_1$ is degenerate.

**EXAMPLES:**

\begin{verbatim}
>>> S1 = simplicial_sets.Sphere(1)

# needs sage.graphs

>>> C = S1.fundamental_group(); C

Simplicial set morphism:
From: Simplicial set with 9 non-degenerate simplices
to: Wedge: (S^1 v S^1)
defn: ['(*, ()), (*, (1,2,3)), (*, (1,3,2)), (sigma_1, ()),
     (sigma_1, ()), (sigma_1, (1,2,3)), (sigma_1, (1,2,3)),
     (sigma_1, (1,3,2)), (sigma_1, (1,3,2))']

>>> C.domain()
Simplicial set with 9 non-degenerate simplices

>>> C.domain().face_data()
{(*, ()): None,
 (*, (1,2,3)): None,
 (*, (1,3,2)): None,
 (sigma_1, ()): (('*, (1,2,3)), ('*, ())),
 (sigma_1, ()): (('', ()), ('*, ())),
 (sigma_1, (1,2,3)): (('', (1,3,2)), ('*, (1,2,3)));
 (sigma_1, (1,2,3)): (('', (1,2,3)), ('*, (1,2,3)));
 (sigma_1, (1,3,2)): (('', ()), ('*, (1,3,2)));
 (sigma_1, (1,3,2)): (('', (1,3,2)), ('*, (1,3,2))))
\end{verbatim}
The fundamental group of a disjoint union of course depends on the choice of base point:

```python
sage: T = simplicial_sets.Torus()
# needs sage.graphs
sage: K = simplicial_sets.KleinBottle()
# needs sage.graphs
sage: X = T.disjoint_union(K)
# needs sage.graphs
sage: # needs sage.graphs
sage: X_0 = X.set_base_point(X.n_cells(Integer(0)))[0]
sage: X_0.fundamental_group().is_abelian()  # needs sage.groups
True
sage: X_1 = X.set_base_point(X.n_cells(Integer(0)))[1]
sage: X_1.fundamental_group().is_abelian()  # needs sage.groups
False
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
# needs sage.graphs sage.groups
sage: RP3.fundamental_group()  # needs sage.graphs sage.groups
Finitely presented group < e | e^2 >
```

```python
>>> from sage.all import *
>>> S1 = simplicial_sets.Sphere(Integer(1))
  # needs sage.graphs
>>> eight = S1.wedge(S1)
  # needs sage.graphs
>>> eight.fundamental_group()  # free group on 2 generators
Finitely presented group < e0, e1 | >
```

```python
sage: eight = S1.wedge(S1)
# needs sage.graphs
sage: eight.fundamental_group()  # free group on 2 generators
# needs sage.graphs sage.groups
Finitely presented group < e0, e1 | >
```

```python
from sage.all import *
S1 = simplicial_sets.Sphere(Integer(1))
eight = S1.wedge(S1)
eight.fundamental_group()  # free group on 2 generators
# needs sage.graphs
from sage.all import *
S1 = simplicial_sets.Sphere(Integer(1))
eight = S1.wedge(S1)
eight.fundamental_group()  # free group on 2 generators
# needs sage.graphs
```

(continues on next page)
False

```python
>>> RP3 = simplicial_sets.RealProjectiveSpace(Integer(3))
# needs sage.graphs sage.groups
>>> RP3.fundamental_group()
# needs sage.graphs sage.groups
Finitely presented group < e | e^2 >
```

Compute the fundamental group of some classifying spaces:

```python
sage: C5 = groups.misc.MultiplicativeAbelian([5])
# needs sage.graphs sage.groups
sage: BC5 = C5.nerve()
# needs sage.graphs sage.groups
sage: BC5.fundamental_group()
# needs sage.graphs sage.groups
Finitely presented group < e0 | e0^5 >

sage: Sigma3 = groups.permutation.Symmetric(3)

sage: BSigma3 = Sigma3.nerve()

sage: pi = BSigma3.fundamental_group(); pi
Finitely presented group < e1, e2 | e2^2, e1^3, (e2*e1)^2 >

sage: pi.order()
6
sage: pi.is_abelian()
False
```

```python
>>> from sage.all import *
# needs sage.graphs sage.groups
```

The sphere has a trivial fundamental group:

```python
sage: S2 = simplicial_sets.Sphere(2)
# needs sage.graphs
sage: S2.fundamental_group()
# needs sage.graphs sage.groups
Finitely presented group < | >
```
is_simply_connected()

Return True if this pointed simplicial set is simply connected.

Warning: Determining simple connectivity is not always possible, because it requires determining when a group, as given by generators and relations, is trivial. So this conceivably may give a false negative in some cases.

EXAMPLES:

```
sage: # needs sage.graphs sage.groups
t = simplicial_sets.Torus()
sage: t.is_simply_connected()
False
sage: t.suspension().is_simply_connected()
True
sage: simplicial_sets.KleinBottle().is_simply_connected()
False
sage: # needs sage.graphs
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: (S2.wedge(S3)).is_simply_connected()  # needs sage.groups
True
sage: X = S2.disjoint_union(S3)
sage: X = X.set_base_point(X.n_cells(0)[0])
sage: X.is_simply_connected()
False
sage: C3 = groups.misc.MultiplicativeAbelian([3])  # needs sage.groups
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)  # needs sage.graphs sage.groups
sage: BC3.is_simply_connected()  # needs sage.graphs sage.groups
False
```
twisted_chain_complex (twisting_operator=None, dimensions=None, augmented=False, cochain=False, verbose=False, subcomplex=None, check=False)

Return the normalized chain complex twisted by some operator.

A twisting operator is a map from the set of simplices to some algebra. The differentials are then twisted by this operator.

INPUT:

- twisting_operator – a dictionary, associating the twist of each simplex. If it is not given, the canonical one (associated to the laurent polynomial ring abelianization of the fundamental group, ignoring torsion) is used.
- dimensions – if None, compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero.
- augmented (default: False) – if True, return the augmented chain complex (that is, include a class in dimension −1 corresponding to the empty cell).
- cochain (default: False) – if True, return the cochain complex (that is, the dual of the chain complex).
- verbose (default: False) – ignored.
- subcomplex (default: None) – if present, compute the chain complex relative to this subcomplex.
- check (default: False) – If True, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

The normalized chain complex of a simplicial set is isomorphic to the chain complex obtained by molding out by degenerate simplices, and the latter is what is actually constructed here.

EXAMPLES:

```python
sage: # needs sage.graphs
sage: W = simplicial_sets.Sphere(1).wedge(simplicial_sets.Sphere(2))

sage: W.nondegenerate_simplices()  # needs sage.graphs
[* sigma_1, sigma_2]

sage: s1 = W.nondegenerate_simplices()[1]

sage: L.<t> = LaurentPolynomialRing(QQ)

sage: tw = {s1:t}

sage: ChC = W.twisted_chain_complex(tw)

sage: ChC.differential(1)
[-1 + t]
```
sage: ChC.differential(2)
[0]

```python
>>> from sage.all import *

# needs sage.graphs
>>> W = simplicial_sets.Sphere(Integer(1)).wedge(simplicial_sets.Sphere(Integer(2)))
>>> W.nondegnerate_simplices()
[*, sigma_1, sigma_2]
>>> s1 = W.nondegnerate_simplices()[Integer(1)]
>>> L = LaurentPolynomialRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> tw = {s1:t}
>>> ChC = W.twisted_chain_complex(tw)
>>> ChC.differential(Integer(1))
[-1 + t]
>>> ChC.differential(Integer(2))

sage: # needs sage.graphs
sage: X = simplicial_sets.Torus()

sage: C = X.twisted_chain_complex()

sage: C.differential(Integer(1))
[ f3 - 1 f2*f3^-1 - 1 f2 - 1]

sage: C.differential(Integer(2))
[ 1 f2*f3^-1]
[ f3 1]
[-1 -1]

sage: C.differential(Integer(3))
[]
```

```python
>>> from sage.all import *

# needs sage.graphs
>>> X = simplicial_sets.Torus()

>>> C = X.twisted_chain_complex()

>>> C.differential(Integer(1))
[ f3 - 1 f2*f3^-1 - 1 f2 - 1]

>>> C.differential(Integer(2))
[ 1 f2*f3^-1]
[ f3 1]
[-1 -1]

>>> C.differential(Integer(3))
[]
```

```python
sage: # needs sage.graphs
sage: Y = simplicial_sets.RealProjectiveSpace(2)

sage: C = Y.twisted_chain_complex()

sage: C.differential(Integer(1))
[-1 + F1]

sage: C.differential(Integer(2))
[1 + F1]

sage: C.differential(Integer(3))
[]
```
twisted_homology\( (n, \text{ reduced}=False) \)

The \( n \)-th twisted homology module of the simplicial set with respect to the abelianization of the fundamental\_group.

It is a module over a polynomial ring, including relations to make some variables the multiplicative inverses of others.

**INPUT:**
- \( n \) – a positive integer.
- \( \text{reduced} \) – (default: \( False \)) if set to \( True \), the presentation matrix will be reduced.

**EXAMPLES:**

```python
sage: # needs sage.graphs
sage: X = simplicial_sets.Sphere(1).wedge(simplicial_sets.Sphere(2))
sage: X.twisted_homology(1)
Quotient module by Submodule of Ambient free module of rank 0 over the...
\[\rightarrow\]\text{integral domain Multivariate Polynomial Ring in f1, f1inv over...}
\[\rightarrow\]\text{Integer Ring}
Generated by the rows of the matrix:
[]
sage: X.twisted_homology(2)
Quotient module by Submodule of Ambient free module of rank 1 over the...
\[\rightarrow\]\text{integral domain Multivariate Polynomial Ring in f1, f1inv over...}
\[\rightarrow\]\text{Integer Ring}
Generated by the rows of the matrix:
[f1*f1inv - 1]
```

```python
>>> from sage.all import *
>>> # needs sage.graphs
>>> Y = simplicial_sets.Torus()
```
sage: Y.twisted_homology(1)
Quotient module by Submodule of Ambient free module of rank 5 over the...
→integral domain Multivariate Polynomial Ring in f2, f2inv, f3, f3inv...
→over Integer Ring
Generated by the rows of the matrix:
[ 1 0 0 0 0]
[ 0 1 0 0 0]
[ 0 0 1 0 0]
[ 0 0 0 1 0]
[ 0 0 0 0 1]
[f2*f2inv - 1 0 0 0 0]
[ 0 f2*f2inv - 1 0 0 0]
[ 0 0 f2*f2inv - 1 0 0]
[ 0 0 0 f2*f2inv - 1 0]
[ 0 0 0 0 f2*f2inv - 1]
[f3*f3inv - 1 0 0 0 0]
[ 0 f3*f3inv - 1 0 0 0]
[ 0 0 f3*f3inv - 1 0 0]
[ 0 0 0 f3*f3inv - 1 0]
[ 0 0 0 0 f3*f3inv - 1]
sage: Y.twisted_homology(2)
Quotient module by Submodule of Ambient free module of rank 0 over the...
→integral domain Multivariate Polynomial Ring in f2, f2inv, f3, f3inv...
→over Integer Ring
Generated by the rows of the matrix:
[]
sage: Y.twisted_homology(1, reduced=True)
Quotient module by Submodule of Ambient free module of rank 5 over the...
→integral domain Multivariate Polynomial Ring in f2, f2inv, f3, f3inv...
→over Integer Ring
Generated by the rows of the matrix:
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]

>>> from sage.all import *
>>> # needs sage.graphs
>>> Y = simplicial_sets.Torus()
>>> Y.twisted_homology(Integer(1))
Quotient module by Submodule of Ambient free module of rank 5 over the...
→integral domain Multivariate Polynomial Ring in f2, f2inv, f3, f3inv...
→over Integer Ring
Generated by the rows of the matrix:
[ 1 0 0 0 0]
[ 0 1 0 0 0]
[ 0 0 1 0 0]
[ 0 0 0 1 0]
[ 0 0 0 0 1]
[f2*f2inv - 1 0 0 0 0]
[ 0 f2*f2inv - 1 0 0 0]
[ 0 0 f2*f2inv - 1 0 0]
[ 0 0 0 f2*f2inv - 1 0]
[ 0 0 0 0 f2*f2inv - 1]
[f3*f3inv - 1 0 0 0 0]
[ 0 f3*f3inv - 1 0 0 0]
[ 0 0 f3*f3inv - 1 0 0]
[ 0 0 0 f3*f3inv - 1 0]
[ 0 0 0 0 f3*f3inv - 1]

(continues on next page)
universal_cover()

Return the universal cover of the simplicial set. The fundamental group must be finite in order to ensure that the universal cover is a simplicial set of finite type.

EXAMPLES:

```python
sage: # needs sage.graphs sage.groups
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
sage: C = RP3.universal_cover(); C
Simplicial set with 8 non-degenerate simplices
sage: C.face_data()
{(1, 1): None, 
 (1, e): None, 
 (f, 1): {(1, e), (1, 1)}, 
 (f, e): {(1, 1), (1, e)}, 
 (f * f, 1): {(f, e), s_0 (1, 1), (f, 1)}, 
 (f * f, e): {(f, 1), s_0 (1, e), (f, e)}, 
 (f * f * f, 1): {(f * f, e), s_0 (f, 1), s_1 (f, 1), (f * f, 1)}, 
 (f * f * f, e): {(f * f, 1), s_0 (f, e), s_1 (f, e), (f * f, e)}
```

```python
>>> from sage.all import *
... >>> # needs sage.graphs sage.groups
... >>> RP3 = simplicial_sets.RealProjectiveSpace(Integer(3))
... >>> C = RP3.universal_cover(); C
... Simplicial set with 8 non-degenerate simplices
... >>> C.face_data()
... {(1, 1): None, 
... (1, e): None, 
... (f, 1): {(1, e), (1, 1)}, 
... (f, e): {(1, 1), (1, e)}, 
... (f * f, 1): {(f, e), s_0 (1, 1), (f, 1)}, 
... (f * f, e): {(f, 1), s_0 (1, e), (f, e)}, 
... (f * f * f, 1): {(f * f, e), s_0 (f, 1), s_1 (f, 1), (f * f, 1)}, 
... (f * f * f, e): {(f * f, 1), s_0 (f, e), s_1 (f, e), (f * f, e)},
```

(continues on next page)
universal_cover_map()

Return the universal covering map of the simplicial set.

It requires the fundamental group to be finite.

EXAMPLES:

```
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)  # needs sage.graphs sage.groups

sage: phi = RP2.universal_cover_map(); phi
Simplicial set morphism:
    From: Simplicial set with 6 non-degenerate simplices
    To: RP^2
    Defn: [(1, 1), (1, e), (f, 1), (f, e), (f * f, 1), (f * f, e)]
        --> [1, 1, f, f, f * f, f * f]
```

```
sage: phi.domain().face_data()  # needs sage.graphs sage.groups
{(1, 1): None,
 (1, e): None,
 (f, 1): ((1, e), (1, 1)),
 (f, e): ((1, 1), (1, e)),
 (f * f, 1): ((f, e), s_0 (1, 1), (f, 1)),
 (f * f, e): ((f, 1), s_0 (1, e), (f, e))}
```

```
>> from sage.all import *
>> RP2 = simplicial_sets.RealProjectiveSpace(Integer(2))  # needs sage.graphs sage.groups

>> phi = RP2.universal_cover_map(); phi  # needs sage.graphs sage.groups
Simplicial set morphism:
    From: Simplicial set with 6 non-degenerate simplices
    To: RP^2
    Defn: [(1, 1), (1, e), (f, 1), (f, e), (f * f, 1), (f * f, e)]
        --> [1, 1, f, f, f * f, f * f]
>> phi.domain().face_data()  # needs sage.graphs sage.groups
{(1, 1): None,
 (1, e): None,
 (f, 1): ((1, e), (1, 1)),
 (f, e): ((1, 1), (1, e)),
 (f * f, 1): ((f, e), s_0 (1, 1), (f, 1)),
 (f * f, e): ((f, 1), s_0 (1, e), (f, e))}
```

```
class SubcategoryMethods
    Bases: object

Pointed()
```

A simplicial set is pointed if it has a distinguished base point.

EXAMPLES:
Category Framework, Release 10.4

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: SimplicialSets().Pointed().Finite()
Category of finite pointed simplicial sets
sage: SimplicialSets().Finite().Pointed()
Category of finite pointed simplicial sets
```

```python
>>> from sage.all import *
>>> from sage.categories.simplicial_sets import SimplicialSets
>>>
SimplicialSets().Pointed().Finite()
Category of finite pointed simplicial sets
>>>
SimplicialSets().Finite().Pointed()
Category of finite pointed simplicial sets
```

### super_categories()

**EXAMPLES:**

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: SimplicialSets().super_categories()
[Category of sets]
```

```python
>>> from sage.all import *
>>> from sage.categories.simplicial_sets import SimplicialSets
>>> SimplicialSets().super_categories()
[Category of sets]
```

## 4.149 Super Algebras

**class** `sage.categories.super_algebras.SuperAlgebras(base_category)`

**Bases:** `SuperModulesCategory`

The category of super algebras.

An $R$-super algebra is an $R$-super module $A$ endowed with an $R$-algebra structure satisfying

$$A_0 A_0 \subseteq A_0, \quad A_0 A_1 \subseteq A_1, \quad A_1 A_0 \subseteq A_1, \quad A_1 A_1 \subseteq A_0$$

and $1 \in A_0$.

**EXAMPLES:**

```python
sage: Algebras(ZZ).Super()
Category of super algebras over Integer Ring
```

```python
>>> from sage.all import *
>>> Algebras(ZZ).Super()
Category of super algebras over Integer Ring
```

**class** `ParentMethods`

**Bases:** `object`

**graded_algebra()**

Return the associated graded algebra to `self`. 

```python
class ParentMethods

Bases: object

graded_algebra()

    Return the associated graded algebra to self.
```
Warning: Because a super module $M$ is naturally $\mathbb{Z}/2\mathbb{Z}$-graded, and graded modules have a natural filtration induced by the grading, if $M$ has a different filtration, then the associated graded module $\text{gr} M \neq M$. This is most apparent with super algebras, such as the differential Weyl algebra, and the multiplication may not coincide.

\texttt{tensor} (*\textit{parents}, **\textit{kwargs})

Return the tensor product of the parents.

EXAMPLES:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
ea = SteenrodAlgebra(3).an_element()
sage: M = CombinatorialFreeModule(GF(3), ['s', 't', 'u'])
sage: s = M.basis()['s']
sage: tensor([a, s])
# # # Needs sage.rings.finite_rings
2*Q_1 Q_3 P(2,1) # B['s']
\end{verbatim}

This also works when the other elements do not have a signed tensor product (Issue #31266):

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> a = SteenrodAlgebra(Integer(3)).an_element()
>>> M = CombinatorialFreeModule(GF(Integer(3)), ['s', 't', 'u'])
>>> s = M.basis()['s']
>>> tensor([a, s])  # #

class SignedTensorProducts (category, *\textit{args})

Bases: SignedTensorProductsCategory

extra_super_categories ()

EXAMPLES:
Meaning: a signed tensor product of coalgebras is a coalgebra

```python
class SubcategoryMethods
    Bases: object

Supercommutative()
    Return the full subcategory of the supercommutative objects of self.

A super algebra $M$ is supercommutative if, for all homogeneous $x, y \in M$,

$$x \cdot y = (-1)^{|x||y|} y \cdot x.$$  

REFERENCES:
Wikipedia article Supercommutative_algebra

EXAMPLES:
```
4.150 Super algebras with basis

```python
class sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis(base_category):
    Bases: SuperModulesCategory

    The category of super algebras with a distinguished basis

    EXAMPLES:

    sage: C = Algebras(ZZ).WithBasis().Super(); C
    Category of super algebras with basis over Integer Ring

    >>> from sage.all import *
    >>> C = Algebras(ZZ).WithBasis().Super(); C
    Category of super algebras with basis over Integer Ring

    class ElementMethods
        Bases: object

        supercommutator(x)

        Return the supercommutator of self and x.

        Let A be a superalgebra. The supercommutator of homogeneous elements $x, y \in A$ is defined by

        \[ [x, y] = xy - (-1)^{|x||y|}yx \]

        and extended to all elements by linearity.

        EXAMPLES:

        sage: # needs sage.modules
        sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
        sage: Cl.<x,y,z> = CliffordAlgebra(Q)
        sage: a = x*y - z
        sage: b = x - y + y*z
        sage: a.supercommutator(b)  # needs sage.modules
        -5*x*y + 8*x*z - 2*y*z - 6*x + 12*y - 5*z
        sage: a.supercommutator(Cl.one())
        0
        sage: Cl.one().supercommutator(a)
        0
        sage: Cl.zero().supercommutator(a)
        0
        sage: a.supercommutator(Cl.zero())
        0

        sage: # needs sage.modules
        sage: Q = QuadraticForm(ZZ, 2, [-1,1,-3])
        sage: Cl.<x,y> = CliffordAlgebra(Q)
        sage: [a*b-b*a for a in Cl.basis() for b in Cl.basis()]
        [0, 0, 0, 0, 0, 0, 2*x*y - 1, -x - 2*y, 0, 6*x + y, 0, x + 2*y, -6*x - y, 0]
```

Exterior algebras inherit from Clifford algebras, so supercommutators work as well. We verify the exterior algebra is supercommutative:

```python
sage: E.<x,y,z,w> = ExteriorAlgebra(QQ)  # needs sage.modules
sage: all(b1.supercommutator(b2) == 0  # needs sage.modules
    ....:     for b1 in E.basis() for b2 in E.basis())
True
```

```text
class ParentMethods
Bases: object

graded_algebra()

Return the associated graded module to self.

See AssociatedGradedAlgebra for the definition and the properties of this.
```
See also:

graded_algebra()

EXAMPLES:

```
sage: W.<x,y> = algebras.DifferentialWeyl(QQ)  # needs sage.modules
sage: W.graded_algebra()  # needs sage.combinat sage.modules
Graded Algebra of Differential Weyl algebra of polynomials in x, y over Rational Field
```

```
>>> from sage.all import *
>>> W = algebras.DifferentialWeyl(QQ, names=('x', 'y',)); (x, y,) = W._
first_ngens(2)  # needs sage.modules
>>> W.graded_algebra()  # needs sage.combinat sage.modules
Graded Algebra of Differential Weyl algebra of polynomials in x, y over Rational Field
```

### class SignedTensorProducts (category, *args)

**Bases:** `SignedTensorProductsCategory`

The category of super algebras with basis constructed by tensor product of super algebras with basis.

**extra_super_categories()**

**EXAMPLES:**

```
sage: Algebras(QQ).Super().SignedTensorProducts().extra_super_categories()
[Category of super algebras over Rational Field]
sage: Algebras(QQ).Super().SignedTensorProducts().super_categories()
[Category of signed tensor products of graded algebras over Rational Field, Category of super algebras over Rational Field]
```

```
>>> from sage.all import *
>>> Algebras(QQ).Super().SignedTensorProducts().extra_super_categories()
[Category of super algebras over Rational Field]
>>> Algebras(QQ).Super().SignedTensorProducts().super_categories()
[Category of signed tensor products of graded algebras over Rational Field, Category of super algebras over Rational Field]
```

Meaning: a signed tensor product of super algebras is a super algebra

**extra_super_categories()**

**EXAMPLES:**

```
sage: C = Algebras(ZZ).WithBasis().Super()
sage: sorted(C.super_categories(), key=str)  # indirect doctest
[Category of graded algebras with basis over Integer Ring, Category of super algebras over Integer Ring, Category of super modules with basis over Integer Ring]
```

```
>>> from sage.all import *
>>> C = Algebras(ZZ).WithBasis().Super()
```

(continues on next page)
4.151 Super Hopf algebras with basis

class sage.categories.super_hopf_algebras_with_basis.SuperHopfAlgebrasWithBasis(base_category):

Bases: SuperModulesCategory

The category of super Hopf algebras with a distinguished basis.

EXAMPLES:

sage: C = HopfAlgebras(ZZ).WithBasis().Super(); C
Category of super Hopf algebras with basis over Integer Ring

sage: sorted(C.super_categories(), key=str)
[Category of super Hopf algebras over Integer Ring,
 Category of super algebras with basis over Integer Ring,
 Category of super coalgebras with basis over Integer Ring]

>>> from sage.all import *

>>> C = HopfAlgebras(ZZ).WithBasis().Super(); C
Category of super Hopf algebras with basis over Integer Ring

>>> sorted(C.super_categories(), key=str)
[Category of super Hopf algebras over Integer Ring,
 Category of super algebras with basis over Integer Ring,
 Category of super coalgebras with basis over Integer Ring]

class ParentMethods

Bases: object

antipode()

The antipode of this Hopf algebra.

If antipode_basis() is available, this constructs the antipode morphism from self to self by extending it by linearity. Otherwise, self.antipode_by_coercion() is used, if available.

EXAMPLES:

sage: A = SteenrodAlgebra(7) # needs sage.combinat, sage.modules
sage: a = A.an_element() # needs sage.combinat, sage.modules
sage: A, A.antipode(a) # needs sage.combinat, sage.modules
(6 Q_1 Q_3 P(2,1), Q_1 Q_3 P(2,1))

>>> from sage.all import *

>>> A = SteenrodAlgebra(Integer(7)) # needs sage.combinat, sage.modules
4.152 Super Lie Conformal Algebras

AUTHORS:


```python
sage: LieConformalAlgebras(AA).Super()
# needs sage.rings.number_field
Category of super Lie conformal algebras over Algebraic Real Field

from sage.all import *

sage: LieConformalAlgebras(AA).Super()
# needs sage.rings.number_field
Category of super Lie conformal algebras over Algebraic Real Field
```

Notice that we can force to have a purely even super Lie conformal algebra:

```python
bosondict = {('a','a'): {Integer(1):{('K',Integer(0)):Integer(1)}}}

R = LieConformalAlgebra(QQ, bosondict, names=('a',),
... central_elements=('K'), super=True)

[g.is_even_odd() for g in R.gens()]
# needs sage.combinat.sage.modules
[0, 0]
```

```python
class ElementMethods
    Bases: object

    is_even_odd()
        Return 0 if this element is even and 1 if it is odd.
```
EXAMPLES:

```python
sage: R = lie_conformal_algebras.NeveuSchwarz(QQ)  #...
    # needs sage.combinat sage.modules
sage: R.inject_variables()  #...
    # needs sage.combinat sage.modules
Defining L, G, C
sage: G.is_even_odd()  #...
    # needs sage.combinat sage.modules
```

```python
>>> from sage.all import *
>>> R = lie_conformal_algebras.NeveuSchwarz(QQ)  #...
    # needs sage.combinat sage.modules
>>> R.inject_variables()  #...
    # needs sage.combinat sage.modules
Defining L, G, C
>>> G.is_even_odd()  #...
    # needs sage.combinat sage.modules
```

```python
class Graded(base_category)

    Bases: GradedModulesCategory

    The category of H-graded super Lie conformal algebras.

    EXAMPLES:

```python
sage: LieConformalAlgebras(AA).Super().Graded()  #...
    # needs sage.rings.number_field
Category of H-graded super Lie conformal algebras over Algebraic Real Field
```

```python
>>> from sage.all import *
>>> LieConformalAlgebras(AA).Super().Graded()  #...
    # needs sage.rings.number_field
Category of H-graded super Lie conformal algebras over Algebraic Real Field
```

```python
class ParentMethods

    Bases: object

    example()

        An example parent in this category.

        EXAMPLES:

```python
sage: LieConformalAlgebras(QQ).Super().example()  #...
    # needs sage.combinat sage.modules
The Neveu-Schwarz super Lie conformal algebra over Rational Field
```

```python
>>> from sage.all import *
>>> LieConformalAlgebras(QQ).Super().example()  #...
    # needs sage.combinat sage.modules
The Neveu-Schwarz super Lie conformal algebra over Rational Field
```

```python
def extra_super_categories()

    The extra super categories of self.

    EXAMPLES:
```
4.153 Super modules

class sage.categories.super_modules.SuperModules(base_category):

    Bases: SuperModulesCategory

    The category of super modules.

    An $R$-super module (where $R$ is a ring) is an $R$-module $M$ equipped with a decomposition $M = M_0 \oplus M_1$ into two $R$-submodules $M_0$ and $M_1$ (called the even part and the odd part of $M$, respectively).

    Thus, an $R$-super module automatically becomes a $\mathbb{Z}/2\mathbb{Z}$-graded $R$-module, with $M_0$ being the degree-0 component and $M_1$ being the degree-1 component.

    EXAMPLES:

    sage: Modules(ZZ).Super()
    Category of super modules over Integer Ring
    sage: Modules(ZZ).Super().super_categories()
    [Category of graded modules over Integer Ring]

    >>> from sage.all import *
    >>> Modules(ZZ).Super()
    Category of super modules over Integer Ring
    >>> Modules(ZZ).Super().super_categories()
    [Category of graded modules over Integer Ring]

    The category of super modules defines the super structure which shall be preserved by morphisms:

    sage: Modules(ZZ).Super().additional_structure()
    Category of super modules over Integer Ring

    >>> from sage.all import *
    >>> Modules(ZZ).Super().additional_structure()
    Category of super modules over Integer Ring

    class ElementMethods

        Bases: object

        is_even()
            Return if self is an even element.

        EXAMPLES:

        sage: # needs sage.combinat sage.modules
        sage: cat = Algebras(QQ).WithBasis().Super()
sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
sage: C.degree_on_basis = sum
sage: C.basis()[2,2,1].is_even()
False
sage: C.basis()[2,2].is_even()
True

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> cat = Algebras(QQ).WithBasis().Super()
>>> C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
>>> C.degree_on_basis = sum
>>> C.basis()[Integer(2),Integer(2),Integer(1)].is_even()
False
>>> C.basis()[Integer(2),Integer(2)].is_even()
True

is_even_odd()

Return 0 if self is an even element or 1 if an odd element.

Note: The default implementation assumes that the even/odd is determined by the parity of degree(). Overwrite this method if the even/odd behavior is desired to be independent.

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: cat = Algebras(QQ).WithBasis().Super()
*sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
>>> C.basis()[2,2,1].is_even_odd()
1
>>> C.basis()[2,2].is_even_odd()
0

is_odd()

Return if self is an odd element.

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: cat = Algebras(QQ).WithBasis().Super()
>>> C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
>>> C.basis()[Integer(2),Integer(2),Integer(1)].is_even_odd()
1
>>> C.basis()[Integer(2),Integer(2)].is_even_odd()
0


\[
\text{sage: C.basis()[2,2,1].is_odd()}
True
\text{sage: C.basis()[2,2].is_odd()}
False
\]

\[
\text{\texttt{from sage.all import *}}
\text{\texttt{# needs sage.combinat sage.modules}}
\text{\texttt{cat = Algebras(QQ).WithBasis().Super()}}
\text{\texttt{C = CombinatorialFreeModule(QQ, Partitions(), category=cat)}}
\text{\texttt{C.degree_on_basis = sum}}
\text{\texttt{C.basis()[Integer(2),Integer(2),Integer(1)].is_odd()}}
True
\text{\texttt{C.basis()[Integer(2),Integer(2)].is_odd()}}
False
\]

\[
\text{class ParentMethods}
\text{Bases: object}
\text{extra_super_categories()}
\text{\texttt{Adds VectorSpaces to the super categories of self if the base ring is a field.}}
\]

\[
\text{\texttt{sage: Modules(QQ).Super().extra_super_categories()}}
[\text{Category of vector spaces over Rational Field}]
\text{\texttt{sage: Modules(ZZ).Super().extra_super_categories()}}
[]
\]

\[
\text{\texttt{from sage.all import *}}
\text{\texttt{Modules(QQ).Super().extra_super_categories()}}
[\text{Category of vector spaces over Rational Field}]
\text{\texttt{Modules(ZZ).Super().extra_super_categories()}}
[]
\]

\[
\text{This makes sure that Modules(QQ).Super() returns an instance of \texttt{SuperModules} and not a join category of an instance of this class and of VectorSpaces(QQ):}
\]

\[
\text{\texttt{sage: type(Modules(QQ).Super())}}
<\text{class 'sage.categories.super_modules.SuperModules_with_category'>}
\]

\[
\text{\texttt{from sage.all import *}}
\text{\texttt{type(Modules(QQ).Super())}}
<\text{class 'sage.categories.super_modules.SuperModules_with_category'>}
\]

\[
\text{Todo: Get rid of this workaround once there is a more systematic approach for the alias Modules(QQ) -> VectorSpaces(QQ). Probably the latter should be a category with axiom, and covariant constructions should play well with axioms.}
\]

\[
\text{\texttt{super_categories()}}
\text{\texttt{EXAMPLES:}}
\]

\[
\text{\texttt{sage: Modules(ZZ).Super().super_categories()}}
[\text{Category of graded modules over Integer Ring}]
\]
>>> from sage.all import *
>>> Modules(ZZ).Super().super_categories()
[Category of graded modules over Integer Ring]

Nota bene:

```
sage: Modules(QQ).Super()
Category of super modules over Rational Field
```

```
sage: Modules(QQ).Super().super_categories()
[Category of graded modules over Rational Field]
```

```python
class sage.categories.super_modules.SuperModulesCategory(base_category)
    Bases: CovariantConstructionCategory, Category_over_base_ring

EXAMPLES:
```
sage: C = Algebras(QQ).Super()
sage: C
Category of super algebras over Rational Field
sage: C.base_category()
Category of algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of graded algebras over Rational Field,
 Category of super modules over Rational Field]
sage: AlgebrasWithBasis(QQ).Super().base_ring()
Rational Field
sage: HopfAlgebrasWithBasis(QQ).Super().base_ring()
Rational Field
```

```python
classmethod default_super_categories(cls, category, *args)
    Return the default super categories of \( F_{\text{Cat}}(A, B, \ldots) \) for \( A, B, \ldots \) parents in \( \text{Cat} \).

    INPUT:
    • cls – the category class for the functor \( F \)
• category – a category $Cat$
• *args – further arguments for the functor

OUTPUT:
A join category.

This implements the property that subcategories constructed by the set of whitelisted axioms is a subcategory.

EXAMPLES:

```python
sage: HopfAlgebras(ZZ).WithBasis().FiniteDimensional().Super()  # indirect doctest
Category of finite dimensional super Hopf algebras with basis over Integer

sage: from sage.all import *
>>> HopfAlgebras(ZZ).WithBasis().FiniteDimensional().Super()  # indirect doctest
Category of finite dimensional super Hopf algebras with basis over Integer
```

### 4.154 Super modules with basis

The category of super modules with a distinguished basis.

An $R$-super module with a distinguished basis is an $R$-super module equipped with an $R$-module basis whose elements are homogeneous.

EXAMPLES:

```python
sage: C = GradedModulesWithBasis(QQ); C
Category of graded vector spaces with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of filtered vector spaces with basis over Rational Field, Category of graded modules with basis over Rational Field, Category of graded vector spaces over Rational Field]

sage: C is ModulesWithBasis(QQ).Graded()
True

>>> from sage.all import *
>>> C = GradedModulesWithBasis(QQ); C
Category of graded vector spaces with basis over Rational Field
>>> sorted(C.super_categories(), key=str)
[Category of filtered vector spaces with basis over Rational Field, Category of graded modules with basis over Rational Field, Category of graded vector spaces over Rational Field]
>>> C is ModulesWithBasis(QQ).Graded()
True
```

```python
class ElementMethods
    Bases: object
```
even_component()

Return the even component of self.

EXAMPLES:

```python
sage: # needs sage.modules
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x*y + x - 3*y + 4
sage: a.even_component()
x*y + 4

>>> from sage.all import *
>>>
# needs sage.modules
>>> Q = QuadraticForm(QQ, Integer(2), [Integer(1),Integer(2),Integer(3)])
>>> C = CliffordAlgebra(Q, names=('x', 'y')); (x, y,) = C._first_ngens(2)
>>> a = x*y + x - Integer(3)*y + Integer(4)
>>> a.even_component()
x*y + 4
```

is_even_odd()

Return 0 if self is an even element and 1 if self is an odd element.

EXAMPLES:

```python
sage: # needs sage.modules
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x + y
sage: a.is_even_odd()
1
sage: a = x*y + 4
sage: a.is_even_odd()
0
sage: a = x + 4
sage: a.is_even_odd()
Traceback (most recent call last):
...
ValueError: element is not homogeneous

sage: E.<x,y> = ExteriorAlgebra(QQ)  # needs sage.modules
sage: (x*y).is_even_odd()  # needs sage.modules
0

>>> from sage.all import *
>>> # needs sage.modules
>>> Q = QuadraticForm(QQ, Integer(2), [Integer(1),Integer(2),Integer(3)])
>>> C = CliffordAlgebra(Q, names=('x', 'y')); (x, y,) = C._first_ngens(2)
>>> a = x + y
>>> a.is_even_odd()
1
>>> a = x*y + Integer(4)
>>> a.is_even_odd()
0
>>> a = x + Integer(4)
```
is_super_homogeneous()

Return whether this element is homogeneous, in the sense of a super module (i.e., is even or odd).

EXAMPLES:

```
sage: # needs sage.modules
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x + y
sage: a.is_super_homogeneous()
True
sage: a = x*y + 4
sage: a.is_super_homogeneous()
True
sage: a = x*y + x - 3*y + 4
sage: a.is_super_homogeneous()
False
```

The exterior algebra has a \( \mathbb{Z} \) grading, which induces the \( \mathbb{Z}/2\mathbb{Z} \) grading. However the definition of homogeneous elements differs because of the different gradings:

```
sage: # needs sage.combinat sage.modules
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: a = x*y + 4
sage: a.is_super_homogeneous()
True
sage: a.is_homogeneous()
False
```

(continues on next page)


```python
>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> a = x*y + Integer(4)
>>> a.is_super_homogeneous()
True
>>> a.is_homogeneous()
False
```

**odd_component()**

Return the odd component of self.

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x*y + x - 3*y + 4
sage: a.odd_component()
x - 3*y
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> Q = QuadraticForm(QQ, Integer(2), [Integer(1),Integer(2),Integer(3)])
>>> C = CliffordAlgebra(Q, names=('x', 'y',)); (x, y,) = C._first_ngens(2)
>>> a = x*y + x - Integer(3)*y + Integer(4)
>>> a.odd_component()
x - 3*y
```

class ParentMethods

Bases: object

## 4.155 Supercommutative Algebras

**class** `sage.categories.supercommutative_algebras.SupercommutativeAlgebras(base_category)`

*Bases: CategoryWithAxiom_over_base_ring*

The category of supercommutative algebras.

An $R$-supercommutative algebra is an $R$-super algebra $A = A_0 \oplus A_1$ endowed with an $R$-super algebra structure satisfying:

\[
x_0x'_0 = x'_0x_0, \quad x_1x'_1 = -x'_1x_1, \quad x_0x_1 = x_1x_0,
\]

for all $x_0, x'_0 \in A_0$ and $x_1, x'_1 \in A_1$.

**EXAMPLES:**

```python
sage: Algebras(ZZ).Supercommutative()
Category of supercommutative algebras over Integer Ring
```

```python
>>> from sage.all import *
>>> # Algebras(ZZ).Supercommutative()
>>> Category of supercommutative algebras over Integer Ring
```
class SignedTensorProducts (category, *args)
Bases: SignedTensorProductsCategory
eextra_super_categories ()
    Return the extra super categories of self.
    A signed tensor product of supercommutative algebras is a supercommutative algebra.
    EXAMPLES:
sage: C = Algebras(ZZ).Supercommutative().SignedTensorProducts()
sage: C.extra_super_categories()
[Category of supercommutative algebras over Integer Ring]

class WithBasis (base_category)
Bases: CategoryWithAxiom_over_base_ring
class ParentMethods
    Bases: object

4.156 Supercrystals
class sage.categories.supercrystals.SuperCrystals
Bases: Category_singleton
class Finite (base_category)
    Bases: CategoryWithAxiom_singleton
class ElementMethods
    Bases: object
    is_genuine_highest_weight (index_set=None)
        Return whether self is a genuine highest weight element.
        INPUT:
        • index_set – (optional) the index set of the (sub)crystal on which to check
        EXAMPLES:
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: for b in B.highest_weight_vectors():
    ....:     print("{} {}".format(b, b.is_genuine_highest_weight()))
        [[-2, -2, -2], [-1, -1], [1]] False
        [[-2, -2, -2], [-1, 2], [1]] False
        [[-2, -2, 2], [-1, -1], [1]] False
        sage: [b for b in B if b.is_genuine_highest_weight([-1,0])]
        [[[-2, -2, -2], [-1, -1], [1]],
         [[-2, -2, -2], [-1, -1], [2]],
         [[-2, -2, -2], [-1, 2], [2]],
         [[-2, -2, 2], [-1, -1], [2]],
         [[-2, -2, 2], [-1, 2], [2]],
         (continues on next page)
is_genuine_lowest_weight (index_set=None)

Return whether self is a genuine lowest weight element.

INPUT:
• index_set – (optional) the index set of the (sub)crystal on which to check

EXAMPLES:

sage: B = crystals.Tableaux(['A', [Integer(1),Integer(1)]], shape=[Integer(3),Integer(2),Integer(1)])
sage: for b in sorted(B.lowest_weight_vectors()): ...
    print("{} {}".format(b, b.is_genuine_lowest_weight()))
[-2, -1, 2], [-1, 2], [1] False
[-2, -1, 1], [-1, 2], [1] False
[-1, -1, 1], [1, 2], [2] True
[-1, -1, 1], [1, 2], [1] True
[-1, 1, 2], [1, 2], [2] True
[-1, 1, 2], [1, 2], [1] True
[-2, -1, 2], [-1, 2], [1] False
[-2, -1, 1], [-1, 2], [1] False
[-1, -1, 1], [1, 2], [2] True
[-1, -1, 1], [1, 2], [1] True
[-1, 1, 2], [1, 2], [2] True
[-1, 1, 2], [1, 2], [1] True

(continues on next page)
class ParentMethods
    Bases: object

    character()
    Return the character of self.

    Todo: Once the WeylCharacterRing is implemented, make this consistent with the implementation in sage.categories.classical_crystals.ClassicalCrystals.ParentMethods.character().

    EXAMPLES:
    sage: B = crystals.Letters(['A', [1, 2]])
    sage: B.character()
    B[(1, 0, 0, 0, 0)] + B[(0, 1, 0, 0, 0)] + B[(0, 0, 1, 0, 0)]
    + B[(0, 0, 0, 1, 0)] + B[(0, 0, 0, 0, 1)]
    >>> from sage.all import *
    >>> B = crystals.Letters(['A', [Integer(1), Integer(2)]])
    >>> B.character()
    B[(1, 0, 0, 0, 0)] + B[(0, 1, 0, 0, 0)] + B[(0, 0, 1, 0, 0)]
    + B[(0, 0, 0, 1, 0)] + B[(0, 0, 0, 0, 1)]

    connected_components()
    Return the connected components of self as subcrystals.

    EXAMPLES:
    sage: B = crystals.Letters(['A', [1, 2]])
    sage: B.connected_components()
    [Subcrystal of The crystal of letters for type ['A', [1, 2]]]
    sage: T = B.tensor(B)
    sage: T.connected_components()
    [Subcrystal of Full tensor product of the crystals
     [The crystal of letters for type ['A', [1, 2]],
      The crystal of letters for type ['A', [1, 2]]],
     Subcrystal of Full tensor product of the crystals
     [The crystal of letters for type ['A', [1, 2]],
      The crystal of letters for type ['A', [1, 2]]]]
    >>> from sage.all import *
    >>> B = crystals.Letters(['A', [Integer(1), Integer(2)]])
    >>> B.connected_components()
    [Subcrystal of The crystal of letters for type ['A', [1, 2]]]
connected_components_generators()  
Return the tuple of genuine highest weight elements of self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_highest_weight_vectors()
(-2,)

t = B.tensor(B)
sage: T = B.tensor(B)
sage: T.genuine_highest_weight_vectors()
(([-2, -1], [-2, -2])
	sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_highest_weight_vectors()
(([-2, -1], [-2, -2])

digraph(index_set=None)
Return the DiGraph associated to self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,3]])
sage: G = B.digraph(); G
Multi-digraph on 6 vertices
sage: Q = crystals.Letters(['Q',3])
sage: G = Q.digraph(); G
Multi-digraph on 3 vertices
sage: G.edges(sort=True)
[(1, 2, -1), (1, 2, 1), (2, 3, -2), (2, 3, 2)]

sage: from sage.all import *
sage: B = crystals.Letters(['A', [Integer(1),Integer(3)]])

(continues on next page)
G = B.digraph(); G
Multi-digraph on 6 vertices
>>>
Q = crystals.Letters(['Q', Integer(3)])
>>>
G = Q.digraph(); G
Multi-digraph on 3 vertices
>>>
G.edges(sort=True)
[(1, 2, -1), (1, 2, 1), (2, 3, -2), (2, 3, 2)]

The edges of the crystal graph are by default colored using blue for edge 1, red for edge 2, green for edge 3, and dashed with the corresponding color for barred edges. Edge 0 is dotted black:

```sage
view(G)  # optional - dot2tex graphviz, not tested (opens external window)
```

```sage
from sage.all import *

view(G)  # optional - dot2tex graphviz, not tested (opens external window)
```

genuine_highest_weight_vectors()

Return the tuple of genuine highest weight elements of `self`.

EXAMPLES:

```sage
sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_highest_weight_vectors()
(-2,)
sage: T = B.tensor(B)
sage: T.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])
sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])
```

```sage
from sage.all import *

sage: B = crystals.Letters(['A', [Integer(1),Integer(2)]]))
sage: B.genuine_highest_weight_vectors()
(-2,)
sage: T = B.tensor(B)
sage: T.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])
sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])
```

genuine_lowest_weight_vectors()

Return the tuple of genuine lowest weight elements of `self`.

EXAMPLES:

```sage
sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_lowest_weight_vectors()
(3,)
```

(continues on next page)
sage: T = B.tensor(B)
sage: T.genuine_lowest_weight_vectors()
([3, 3], [3, 2])
sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_lowest_weight_vectors()
([3, 3], [3, 2])

>>> from sage.all import *
>>> B = crystals.Letters(["A", [Integer(1),Integer(2)]])
>>> B.genuine_lowest_weight_vectors()
(3,
)

>>> T = B.tensor(B)
>>> T.genuine_lowest_weight_vectors()
([3, 3], [3, 2])

We give an example from [BKK2000] that has fake highest weight vectors:

sage: B = crystals.Tableaux(["A", [1,1]], shape=[3,2,1])
sage: B.highest_weight_vectors()
([-2, -2, -2], [-1, -1, 1], [-2, -2, -2], [-1, 2, 1], 1]

We give an example from [BKK2000] that has fake highest weight vectors:

sage: B = crystals.Tableaux(["A", [1,1]], shape=[3,2,1])
sage: B.highest_weight_vectors()
([-2, -2, -2], [-1, -1, 1],
  [-2, -2, -2], [-1, 2, 1],
  [-2, -2, -2], [-1, -1, 1])

>>> from sage.all import *
>>> B = crystals.Tableaux(["A", [Integer(1),Integer(2)]]),

(continues on next page)
Category Framework, Release 10.4

(continued from previous page)

```python
>>> B.highest_weight_vectors()
([-2, -2, -2], [-1, -1], [1]),
([-2, -2, -2], [-1, 2], [1]),
([-2, -2, 2], [-1, -1], [1])
>>> B.genuine_highest_weight_vectors()
([-2, -2, -2], [-1, -1], [1])
```

### lowest_weight_vectors()

Return the lowest weight vectors of self.

**EXAMPLES:**

```python
sage: B = crystals.Letters(['A', [1,2]])
sage: B.lowest_weight_vectors()
(3,)
```

```python
>>> from sage.all import *

>>> B = crystals.Letters(['A', [Integer(1),Integer(2)]])

```
```python
sage: B = crystals.Letters(['A', [1,2]])
sage: C = crystals.Tableaux(['A', [1,2]], shape = [2,1])
sage: T = C.tensor(B); T
Full tensor product of the crystals
[Crystal of BKK tableaux of shape [2, 1] of gl(2|3),
The crystal of letters for type ['A', [1, 2]]]
sage: S = B.tensor(C); S
Full tensor product of the crystals
[The crystal of letters for type ['A', [1, 2]],
Crystal of BKK tableaux of shape [2, 1] of gl(2|3)]
sage: G = T.digraph()
sage: H = S.digraph()
sage: G.is_isomorphic(H, edge_labels= True)
True
```

```python
class TensorProducts (category, *args)
Bases: TensorProductsCategory

The category of regular crystals constructed by tensor product of regular crystals.

extra_super_categories()

EXAMPLES:
```
sage: from sage.categories.supercrystals import SuperCrystals
sage: SuperCrystals().TensorProducts().extra_super_categories()
[Category of super crystals]
```
super_categories()

EXAMPLES:
```
sage: from sage.categories.supercrystals import SuperCrystals
sage: C = SuperCrystals()
sage: C.super_categories()
[Category of crystals]
```
4.157 Topological Spaces

class sage.categories.topological_spaces.TopologicalSpaces(category, *args)

Bases: TopologicalSpacesCategory

The category of topological spaces.

EXAMPLES:

```
sage: Sets().Topological()
Category of topological spaces
sage: Sets().Topological().super_categories()
[Category of sets]
```

The category of topological spaces defines the topological structure, which shall be preserved by morphisms:

```
sage: Sets().Topological().additional_structure()
Category of topological spaces
```

class CartesianProducts(category, *args)

Bases: CartesianProductsCategory

def extra_super_categories()

Implement the fact that a (finite) Cartesian product of topological spaces is a topological space.

EXAMPLES:

```
sage: TopologicalSpaces().CartesianProducts()
Category of topological spaces
sage: TopologicalSpaces().CartesianProducts().extra_super_categories()
[Category of topological spaces]
sage: TopologicalSpaces().CartesianProducts().super_categories()
[Category of Cartesian products of sets, Category of topological spaces]
sage: TopologicalSpaces().CartesianProducts().axioms()
frozenset()
```
```python
>>> from sage.all import *
>>> from sage.categories.topological_spaces import TopologicalSpaces
>>> C = TopologicalSpaces().Compact().CartesianProducts()
>>> C.extra_super_categories()
[Category of compact topological spaces]
>>> C.super_categories()
[Category of Cartesian products of topological spaces, Category of compact topological spaces]
>>> C.axioms()
frozenset({Compact})
```

**class Compact** *(base_category)*

Bases: `CategoryWithAxiom`

The category of compact topological spaces.

**class CartesianProducts** *(category, *args)*

Bases: `CartesianProductsCategory`

`extra_super_categories()`

Implement the fact that a (finite) Cartesian product of compact topological spaces is compact.

**EXAMPLES:**

```python
sage: from sage.categories.topological_spaces import TopologicalSpaces
sage: C = TopologicalSpaces().Compact().CartesianProducts()
```

```python
sage: C.extra_super_categories()
[Category of compact topological spaces]
```

```python
sage: C.super_categories()
[Category of Cartesian products of topological spaces, Category of compact topological spaces]
```

```python
sage: C.axioms()
frozenset({"Compact"})
```

```python
>>> from sage.all import *
>>> from sage.categories.topological_spaces import TopologicalSpaces
```

```python
>>> C = TopologicalSpaces().Compact().CartesianProducts()
```

```python
>>> C.extra_super_categories()
[Category of compact topological spaces]
```

```python
>>> C.super_categories()
[Category of Cartesian products of topological spaces, Category of compact topological spaces]
```

```python
>>> C.axioms()
frozenset({"Compact"})
```

**class Connected** *(base_category)*

Bases: `CategoryWithAxiom`

The category of connected topological spaces.

**class CartesianProducts** *(category, *args)*

Bases: `CartesianProductsCategory`

`extra_super_categories()`

Implement the fact that a (finite) Cartesian product of connected topological spaces is connected.

**EXAMPLES:**

```python
sage: from sage.categories.topological_spaces import TopologicalSpaces
sage: C = TopologicalSpaces().Compact().CartesianProducts()
```

```python
sage: C.extra_super_categories()
[Category of compact topological spaces]
```

```python
sage: C.super_categories()
[Category of Cartesian products of topological spaces, Category of compact topological spaces]
```

```python
sage: C.axioms()
frozenset({"Compact"})
```

```python
>>> from sage.all import *
>>> from sage.categories.topological_spaces import TopologicalSpaces
```

```python
>>> C = TopologicalSpaces().Compact().CartesianProducts()
```

```python
>>> C.extra_super_categories()
[Category of compact topological spaces]
```

```python
>>> C.super_categories()
[Category of Cartesian products of topological spaces, Category of compact topological spaces]
```

```python
>>> C.axioms()
frozenset({"Compact"})
```
sage: from sage.categories.topological_spaces import TopologicalSpaces
sage: C = TopologicalSpaces().Connected().CartesianProducts()

sage: C.extra_super_categories()
[Category of connected topological spaces]

sage: C.super_categories()
[Category of Cartesian products of topological spaces,
 Category of connected topological spaces]

sage: C.axioms()
frozenset({'Connected'})

>>> from sage.all import *
>>> from sage.categories.topological_spaces import TopologicalSpaces
>>>
>>> C = TopologicalSpaces().Connected().CartesianProducts()

class SubcategoryMethods
Bases: object

Compact()
Return the subcategory of the compact objects of self.

EXAMPLES:

sage: Sets().Topological().Compact()
Category of compact topological spaces

Connected()
Return the full subcategory of the connected objects of self.

EXAMPLES:

sage: Sets().Topological().Connected()
Category of connected topological spaces

class sage.categories.topological_spaces.TopologicalSpacesCategory(category, *args)
Bases: RegressiveCovariantConstructionCategory
4.158 Kac-Moody Algebras With Triangular Decomposition Basis

AUTHORS:

- Travis Scrimshaw (07-15-2017): Initial implementation

```python
class sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras(base, name=None):
   _bases: Category_over_base_ring

Category of Kac-Moody algebras with a distinguished basis that respects the triangular decomposition.

We require that the grading group is the root lattice of the appropriate Cartan type.

``` ElementMethods

Bases: object

```
part()

Return whether the element \( v \) is in the lower, zero, or upper part of \( self \).

OUTPUT:

-1 if \( v \) is in the lower part, 0 if in the zero part, or 1 if in the upper part

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebra(QQ, cartan_type="F4")
```
```
sage: L.inject_variables()
Defining e1, e2, e3, e4, f1, f2, f3, f4, h1, h2, h3, h4
sage: e1.part()
1
sage: f4.part()
-1
sage: (h2 + h3).part()
0
sage: (f1.bracket(f2) + 4*f4).part()
-1
sage: (e1 + f1).part()
Traceback (most recent call last):
...
ValueError: element is not in one part
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = LieAlgebra(QQ, cartan_type="F4")
>>> L.inject_variables()
Defining e1, e2, e3, e4, f1, f2, f3, f4, h1, h2, h3, h4
>>> e1.part()
1
>>> f4.part()
-1
>>> (h2 + h3).part()
0
>>> (f1.bracket(f2) + Integer(4)*f4).part()
-1
>>> (e1 + f1).part()
Traceback (most recent call last):
...
Traceback (most recent call last):
...
ValueError: element is not in one part
```
class ParentMethods
Bases: object
e(i=None)
    Return the generators e of self.
    INPUT:
    • i – (optional) if specified, return just the generator $e_i$
EXAMPLES:
sage: L = lie_algebras.so(QQ, 5)  # needs sage.combinat sage.modules
sage: L.e()
# needs sage.combinat sage.modules
Finite family {1: E[alpha[1]], 2: E[alpha[2]]}
sage: L.e(1)
# needs sage.combinat sage.modules
E[alpha[1]]

f(i=None)
    Return the generators f of self.
    INPUT:
    • i – (optional) if specified, return just the generator $f_i$
EXAMPLES:
sage: L = lie_algebras.so(QQ, 5)  # needs sage.combinat sage.modules
sage: L.f()
# needs sage.combinat sage.modules
Finite family {1: E[-alpha[1]], 2: E[-alpha[2]]}
sage: L.f(1)
# needs sage.combinat sage.modules
E[-alpha[1]]

>>> from sage.all import *
>>> L = lie_algebras.so(QQ, Integer(5))  # needs sage.combinat sage.modules
>>> L.f()
# needs sage.combinat sage.modules
Finite family {1: E[-alpha[1]], 2: E[-alpha[2]]}
>>> L.f(Integer(1))  # needs sage.combinat sage.modules
E[-alpha[1]]
verma_module \((la, \text{basis_key}=\text{None}, **\text{kwds})\)

Return the Verma module with highest weight \(la\) over \(self\).

**INPUT:**

- \(\text{basis_key}\) – (optional) a key function for the indexing set of the basis elements of \(self\)

**EXAMPLES:**

```
sage: # needs sage.combinat sage.modules
sage: L = lie_algebras.sl(QQ, 3)
sage: P = L.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: M = L.verma_module(La[1] + La[2])
sage: M
Verma module with highest weight \(\Lambda[1] + \Lambda[2]\) of Lie algebra of ['A', 2] in the Chevalley basis
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> P = L.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> M
Verma module with highest weight \(\Lambda[1] + \Lambda[2]\) of Lie algebra of ['A', 2] in the Chevalley basis
```

**super_categories()**

**EXAMPLES:**

```
sage: from sage.categories.triangular_kac_moody_algebras import _
   TriangularKacMoodyAlgebras
sage: TriangularKacMoodyAlgebras(QQ).super_categories()
[Join of Category of graded Lie algebras with basis over Rational Field and Category of kac moody algebras over Rational Field]
```

```
>>> from sage.all import *
>>> from sage.categories.triangular_kac_moody_algebras import _
   TriangularKacMoodyAlgebras
>>> TriangularKacMoodyAlgebras(QQ).super_categories()
[Join of Category of graded Lie algebras with basis over Rational Field and Category of kac moody algebras over Rational Field]
```
4.159 Unique factorization domains

```python
class sage.categories.unique_factorization_domains.UniqueFactorizationDomains
    Bases: Category_singleton
    
The category of (constructive) unique factorization domains.
    
    In a constructive unique factorization domain we can constructively factor members into a product of a finite number of irreducible elements.

    EXAMPLES:
```
pol = Integer(37) * (x-Integer(1))**Integer(3) * (x-Integer(2))**Integer(2) * (x-Integer(1)/Integer(3))**Integer(7) * (x-Integer(3)/Integer(7))

pol.radical()
37*x^4 - 2923/21*x^3 + 1147/7*x^2 - 1517/21*x + 74/7

Integer(Integer(10)).radical()
10

Integer(-Integer(100)).radical()
10

Integer(Integer(0)).radical()
Traceback (most recent call last):
...
ArithmeticError: radical of 0 is not defined

The next example shows how to compute the radical of a number, assuming no prime > 100000 has exponent > 1 in the factorization:

```
sage: n = 2^1000-1; n / radical(n, limit=100000)
125
```
**is_unique_factorization_domain** *(proof=True)*

Return True, since this in an object of the category of unique factorization domains.

```python
sage: UFD = UniqueFactorizationDomains()
sage: Parent(QQ, category=UFD).is_unique_factorization_domain()
True
```

**additional_structure()**

Return whether `self` is a structure category.

See also:

*Category.additional_structure()*

The category of unique factorization domains does not define additional structure: a ring morphism between unique factorization domains is a unique factorization domain morphism.

```python
sage: UniqueFactorizationDomains().additional_structure()
```

**super_categories()**

EXAMPLES:

```python
sage: UniqueFactorizationDomains().super_categories() [Category of gcd domains]
```

### 4.160 Unital algebras

**class** `sage.categories.unital_algebras.UnitalAlgebras` *(base_category)*

Bases: `CategoryWithAxiom_over_base_ring`

The category of non-associative algebras over a given base ring.

A non-associative algebra over a ring $R$ is a module over $R$ which is also a unital magma.

**Warning:** Until issue #15043 is implemented, `Algebras` is the category of associative unital algebras; thus, unlike the name suggests, `UnitalAlgebras` is not a subcategory of `Algebras` but of `MagmaticAlgebras`. 
EXAMPLES:

```sage
sage: from sage.categories.unital_algebras import UnitalAlgebras
sage: C = UnitalAlgebras(ZZ); C
Category of unital algebras over Integer Ring
```

```python
>>> from sage.all import *
>>> from sage.categories.unital_algebras import UnitalAlgebras
>>> C = UnitalAlgebras(ZZ); C
Category of unital algebras over Integer Ring
```

class CartesianProducts(category, *args)

Bases: CartesianProductsCategory

The category of unital algebras constructed as Cartesian products of unital algebras.

This construction gives the direct product of algebras. See discussion on:

- http://groups.google.fr/group/sage-devel/browse_thread/thread/35a72b1d0a2fc77a/348f42ae77a66d16#348f42ae77a66d16
- Wikipedia article Direct_product

class ParentMethods

Bases: object

one()

Return the multiplicative unit element.

EXAMPLES:

```sage
# needs sage.graphs sage.modules
sage: S2 = simplicial_complexes.Sphere(2)
sage: H = S2.cohomology_ring(QQ)
sage: C = cartesian_product([H, H])
sage: one = C.one()
sage: one
B[(0, (0, 0))] + B[(1, (0, 0))]
sage: one == one * one
True
sage: all(b == b * one for b in C.basis())
True
```

```python
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> S2 = simplicial_complexes.Sphere(Integer(2))
>>> H = S2.cohomology_ring(QQ)
>>> C = cartesian_product([H, H])
>>> one = C.one()
>>> one
B[(0, (0, 0))] + B[(1, (0, 0))]
>>> one == one * one
True
>>> all(b == b * one for b in C.basis())
True
```

extra_super_categories()

A Cartesian product of algebras is endowed with a natural unital algebra structure.

EXAMPLES:
```python
sage: from sage.categories.unital_algebras import UnitalAlgebras
sage: C = UnitalAlgebras(QQ).CartesianProducts()

sage: C.extra_super_categories()
[Category of unital algebras over Rational Field]

sage: sorted(C.super_categories(), key=str)
[Category of Cartesian products of distributive magmas and additive
 → magmas,
 Category of Cartesian products of unital magmas,
 Category of Cartesian products of vector spaces over Rational Field,
 Category of unital algebras over Rational Field]
```

```python
>>> from sage.all import *
>>> from sage.categories.unital_algebras import UnitalAlgebras
>>> C = UnitalAlgebras(QQ).CartesianProducts()

>>> C.extra_super_categories()
[Category of unital algebras over Rational Field]

>>> sorted(C.super_categories(), key=str)
[Category of Cartesian products of distributive magmas and additive
 → magmas,
 Category of Cartesian products of unital magmas,
 Category of Cartesian products of vector spaces over Rational Field,
 Category of unital algebras over Rational Field]
```

```python
class ParentMethods

    Bases: object

    from_base_ring(r)

    Return the canonical embedding of r into self.

    INPUT:
    • r - an element of self.base_ring()

    EXAMPLES:

    sage: A = AlgebrasWithBasis(QQ).example(); A
    # needs sage.combinat sage.modules
    An example of an algebra with basis:
    the free algebra on the generators ('a', 'b', 'c') over Rational Field
    sage: A.from_base_ring(1)
    # needs sage.combinat sage.modules
    B[word: ]
```

```python
>>> from sage.all import *

>>> A = AlgebrasWithBasis(QQ).example(); A
# needs sage.combinat sage.modules
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field

>>> A.from_base_ring(Integer(1))
# needs sage.combinat sage.modules
B[word: ]
```

```
```
from_base_ring()

from_base_ring_from_one_basis(r)
Implement the canonical embedding from the ground ring.

INPUT:
• r – an element of the coefficient ring

EXAMPLES:

sage: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.from_base_ring_from_one_basis(3)
3*B
sage: A.from_base_ring(3)
3*B
sage: A(3)
3*B

one()
Return the multiplicative unit element.

EXAMPLES:

sage: A = AlgebrasWithBasis(QQ).example() # needs sage.combinat sage.modules
sage: A.one_basis() # needs sage.combinat sage.modules
word:
sage: A.one() # needs sage.combinat sage.modules
B

one_basis()
When the one of an algebra with basis is an element of this basis, this optional method can return the
index of this element. This is used to provide a default implementation of one(), and an optimized
default implementation of from_base_ring().

EXAMPLES:
category:: # needs sage.combinat sage.modules
sage: A = AlgebrasWithBasis(QQ).example()
word:
sage: A.one()
B[word: ]
sage: A.from_base_ring(4)
4*B[word: ]

>>> from sage.all import *
>>> # needs sage.combinat sage.modules
>>> A = AlgebrasWithBasis(QQ).example()
>>> A.one()
B[word: ]
>>> A.from_base_ring(Integer(4))
4*B[word: ]

one_from_one_basis()

Return the one of the algebra, as per Monoids.ParentMethods.one()

By default, this is implemented from one_basis(), if available.

EXAMPLES:

sage: A = AlgebrasWithBasis(QQ).example()
word:
sage: A.one_from_one_basis()
B[word: ]
sage: A.one()
B[word: ]

4.161 Vector Bundles

class sage.categories.vector_bundles.VectorBundles(base_space, base_field, name=None)

Bases: Category_over_base_ring

The category of vector bundles over any base space and base field.

See also:

TopologicalVectorBundle

4.161. Vector Bundles
EXAMPLES:

```python
sage: M = Manifold(2, 'M', structure='top')
sage: from sage.categories.vector_bundles import VectorBundles
sage: C = VectorBundles(M, RR); C
Category of vector bundles over Real Field with 53 bits of precision
   with base space 2-dimensional topological manifold M
sage: C.super_categories()
[Category of topological spaces]
```

```python
from sage.all import *
M = Manifold(Integer(2), 'M', structure='top')
from sage.categories.vector_bundles import VectorBundles
C = VectorBundles(M, RR); C
Category of vector bundles over Real Field with 53 bits of precision
   with base space 2-dimensional topological manifold M
C.super_categories()
[Category of topological spaces]
```

class **Differentiable** *(base_category)*

Bases: `CategoryWithAxiom_over_base_ring`

The category of differentiable vector bundles.

A differentiable vector bundle is a differentiable manifold with differentiable surjective projection on a differentiable base space.

class **Smooth** *(base_category)*

Bases: `CategoryWithAxiom_over_base_ring`

The category of smooth vector bundles.

A smooth vector bundle is a smooth manifold with smooth surjective projection on a smooth base space.

class **SubcategoryMethods**

Bases: `object`

**Differentiable** ()

Return the subcategory of the differentiable objects of `self`.

EXAMPLES:

```python
sage: M = Manifold(2, 'M')
sage: from sage.categories.vector_bundles import VectorBundles
sage: VectorBundles(M, RR).Differentiable()
Category of differentiable vector bundles over Real Field with 53 bits of precision with base space 2-dimensional differentiable manifold M
```

```python
from sage.all import *
M = Manifold(Integer(2), 'M')
from sage.categories.vector_bundles import VectorBundles
VectorBundles(M, RR).Differentiable()
Category of differentiable vector bundles over Real Field with 53 bits of precision with base space 2-dimensional differentiable manifold M
```

**Smooth** ()

Return the subcategory of the smooth objects of `self`. 
EXAMPLES:

```python
sage: M = Manifold(2, 'M')
sage: from sage.categories.vector_bundles import VectorBundles
sage: VectorBundles(M, RR).Smooth()
Category of smooth vector bundles over Real Field with 53 bits of precision with base space 2-dimensional differentiable manifold M
```

```python
>>> from sage.all import *
>>> M = Manifold(Integer(2), 'M')
>>> from sage.categories.vector_bundles import VectorBundles
>>> VectorBundles(M, RR).Smooth()
Category of smooth vector bundles over Real Field with 53 bits of precision with base space 2-dimensional differentiable manifold M
```

### base_space()

Return the base space of this category.

**EXAMPLES:**

```python
sage: M = Manifold(2, 'M', structure='top')
sage: from sage.categories.vector_bundles import VectorBundles
sage: VectorBundles(M, RR).base_space()
2-dimensional topological manifold M
```

```python
>>> from sage.all import *
>>> M = Manifold(Integer(2), 'M', structure='top')
>>> from sage.categories.vector_bundles import VectorBundles
>>> VectorBundles(M, RR).base_space()
2-dimensional topological manifold M
```

### super_categories()

**EXAMPLES:**

```python
sage: M = Manifold(2, 'M')
sage: from sage.categories.vector_bundles import VectorBundles
sage: VectorBundles(M, RR).super_categories()
[Category of topological spaces]
```

```python
>>> from sage.all import *
>>> M = Manifold(Integer(2), 'M')
>>> from sage.categories.vector_bundles import VectorBundles
>>> VectorBundles(M, RR).super_categories()
[Category of topological spaces]
```
4.162 Vector Spaces

class sage.categories.vector_spaces.VectorSpaces(K)
   Bases: Category_module

   The category of (abstract) vector spaces over a given field
   ??? with an embedding in an ambient vector space ???

   EXAMPLES:

   sage: VectorSpaces(QQ)
   Category of vector spaces over Rational Field
   sage: VectorSpaces(QQ).super_categories()
   [Category of modules over Rational Field]

   >>> from sage.all import *
   >>> VectorSpaces(QQ)
   Category of vector spaces over Rational Field
   >>> VectorSpaces(QQ).super_categories()
   [Category of modules over Rational Field]

   class CartesianProducts(category, *args)
      Bases: CartesianProductsCategory

      extra_super_categories()

      The category of vector spaces is closed under Cartesian products:

      sage: C = VectorSpaces(QQ)
      sage: C.CartesianProducts()
      Category of Cartesian products of vector spaces over Rational Field
      sage: C in C.CartesianProducts().super_categories()
      True

      >>> from sage.all import *
      >>> C = VectorSpaces(QQ)
      >>> C.CartesianProducts()
      Category of Cartesian products of vector spaces over Rational Field
      >>> C in C.CartesianProducts().super_categories()
      True

   class DualObjects(category, *args)
      Bases: DualObjectsCategory

      extra_super_categories()

      Returns the dual category

      EXAMPLES:

      The category of algebras over the Rational Field is dual to the category of coalgebras over the same field:

      sage: C = VectorSpaces(QQ)
      sage: C.dual()
      Category of duals of vector spaces over Rational Field
      sage: C.dual().super_categories() # indirect doctest
      [Category of vector spaces over Rational Field]
```python
>>> from sage.all import *
>>> C = VectorSpaces(QQ)
>>> C.dual()
Category of duals of vector spaces over Rational Field
>>> C.dual().super_categories() # indirect doctest
[Category of vector spaces over Rational Field]

class ElementMethods:
    Bases: object

class Filtered(base_category):
    Bases: FilteredModulesCategory
    Category of filtered vector spaces.

class FiniteDimensional(base_category):
    Bases: CategoryWithAxiom_over_base_ring
    
    class TensorProducts(category, *args):
        Bases: TensorProductsCategory
        
        extra_super_categories()
        Implement the fact that a (finite) tensor product of finite dimensional vector spaces is a finite dimensional vector space.

        EXAMPLES:

        sage: VectorSpaces(QQ).FiniteDimensional().TensorProducts().extra_super_categories()
        [Category of finite dimensional vector spaces over Rational Field]
        sage: VectorSpaces(QQ).FiniteDimensional().TensorProducts().
        ... FiniteDimensional()
        Category of tensor products of finite dimensional vector spaces over Rational Field

        >>> from sage.all import *
        >>> VectorSpaces(QQ).FiniteDimensional().TensorProducts().extra_super_categories()
        [Category of finite dimensional vector spaces over Rational Field]
        >>> VectorSpaces(QQ).FiniteDimensional().TensorProducts().
        ... FiniteDimensional()
        Category of tensor products of finite dimensional vector spaces over Rational Field


class Graded(base_category):
    Bases: GradedModulesCategory
    Category of graded vector spaces.

class ParentMethods:
    Bases: object
    
    dimension()
    Return the dimension of this vector space.

    EXAMPLES:
```

4.162. Vector Spaces 1321
class TensorProducts(category, *args)
    Bases: TensorProductsCategory
    extra_super_categories()
    The category of vector spaces is closed under tensor products:

    sage: C = VectorSpaces(QQ)
sage: C.TensorProducts()  
Category of tensor products of vector spaces over Rational Field
sage: C in C.TensorProducts().super_categories()  
True

>>> from sage.all import *
>>> C = VectorSpaces(QQ)
>>> C.TensorProducts()  
Category of tensor products of vector spaces over Rational Field
>>> C in C.TensorProducts().super_categories()  
True

class WithBasis(base_category)
    Bases: CategoryWithAxiom_over_base_ring
class CartesianProducts(category, *args)

Bases: CartesianProductsCategory

extra_super_categories()

The category of vector spaces with basis is closed under Cartesian products:

```python
sage: C = VectorSpaces(QQ).WithBasis()
sage: C.CartesianProducts()
Category of Cartesian products of vector spaces with basis over Rational Field
sage: C in C.CartesianProducts().super_categories()
True
```

class Filtered(base_category)

Bases: FilteredModulesCategory

Category of filtered vector spaces with basis.

element(base_ring=None)

Return an example of a graded vector space with basis, as per Category.example().

EXAMPLES:

```python
sage: Modules(QQ).WithBasis().Graded().example()  #...
∧ needs sage.combinat sage.modules
An example of a graded module with basis:
the free module on partitions over Rational Field
```

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom_over_base_ring

class TensorProducts(category, *args)

Bases: TensorProductsCategory

extra_super_categories()

Implement the fact that a (finite) tensor product of finite dimensional vector spaces is a finite dimensional vector space.

EXAMPLES:

```python
sage: VectorSpaces(QQ).WithBasis().FiniteDimensional().
∧ TensorProducts().extra_super_categories()
[Category of finite dimensional vector spaces with basis over...
```
class Graded(base_category)
Bases: GradedModulesCategory

Category of graded vector spaces with basis.

g__init__(base_ring=None)
Return an example of a graded vector space with basis, as per Category.example().

EXAMPLES:

```python
sage: Modules(QQ).WithBasis().Graded().example() # needs sage.combinat sage.modules
An example of a graded module with basis:
the free module on partitions over Rational Field
```

class TensorProducts(category, *args)
Bases: TensorProductsCategory

extra_super_categories()

The category of vector spaces with basis is closed under tensor products:

```python
sage: C = VectorSpaces(QQ).WithBasis()
sage: C.TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
sage: C in C.TensorProducts().super_categories()
True
```

```python
from sage.all import *
>>> C = VectorSpaces(QQ).WithBasis()
>>> C.TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
```
is_abelian()

Return whether this category is abelian.

This is always True since the base ring is a field.

EXAMPLES:

```sage
sage: VectorSpaces(QQ).WithBasis().is_abelian()
True
```

```sage
>>> from sage.all import *
...

sage: is_abelian()
True
```

additional_structure()

Return None.

Indeed, the category of vector spaces defines no additional structure: a bimodule morphism between two vector spaces is a vector space morphism.

See also:

Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

```sage
sage: VectorSpaces(QQ).additional_structure()
```

```sage
>>> from sage.all import *
...

sage: additional_structure()
```

base_field()

Returns the base field over which the vector spaces of this category are all defined.

EXAMPLES:

```sage
sage: VectorSpaces(QQ).base_field()
Rational Field
```

```sage
>>> from sage.all import *
...

sage: base_field()
Rational Field
```

super_categories()

EXAMPLES:

```sage
sage: VectorSpaces(QQ).super_categories()
<Category of modules over Rational Field>
```
4.163 Weyl Groups

```python
>>> from sage.all import *
>>> VectorSpaces(QQ).super_categories()
[Category of modules over Rational Field]
```

```python
>>> from sage.all import *
>>> WeylGroups()
Category of Weyl groups
>>> WeylGroups().super_categories()
[Category of Coxeter groups]
```

Here are some examples:

```python
sage: WeylGroups().example()  # todo: not implemented
sage: FiniteWeylGroups().example()  
The symmetric group on \{0, \ldots, 3\}
sage: AffineWeylGroups().example()  # todo: not implemented
sage: WeylGroup(['B', 3])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
```

```python
>>> from sage.all import *
>>> WeylGroups().example()  # todo: not implemented
>>> FiniteWeylGroups().example()  
The symmetric group on \{0, \ldots, 3\}
>>> AffineWeylGroups().example()  # todo: not implemented
>>> WeylGroup(['B', Integer(3)])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
```

This one will eventually be also in this category:

```python
sage: SymmetricGroup(4)
Symmetric group of order 4! as a permutation group
```

```python
>>> from sage.all import *
>>> SymmetricGroup(Integer(4))
Symmetric group of order 4! as a permutation group
```

```python
class ElementMethods
    Bases: object
```
bruhat_lower_covers_coroots()

Return all 2-tuples \((v, \alpha)\) where \(v\) is covered by \(self\) and \(\alpha\) is the positive coroot such that \(self = v s_\alpha\) where \(s_\alpha\) is the reflection orthogonal to \(\alpha\).

ALGORITHM:

See bruhat_lower_covers() and bruhat_lower_covers_reflections() for Coxeter groups.

EXAMPLES:

```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_lower_covers_coroots()
  (s3*s2*s1, alphacheck[2]), (s3*s1*s2, alphacheck[1])]]
```

bruhat_upper_covers_coroots()

Returns all 2-tuples \((v, \alpha)\) where \(v\) is covers \(self\) and \(\alpha\) is the positive coroot such that \(self = v s_\alpha\) where \(s_\alpha\) is the reflection orthogonal to \(\alpha\).

ALGORITHM:

See bruhat_upper_covers() and bruhat_upper_covers_reflections() for Coxeter groups.

EXAMPLES:

```
sage: W = WeylGroup(['A',4], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_upper_covers_coroots()
[[(s1*s2*s3*s2*s1, alphacheck[3]),
  (s2*s3*s1*s2*s1, alphacheck[2] + alphacheck[3]),
  (s3*s4*s1*s2*s1, alphacheck[4]),
```

inversion_arrangement (side='right')

Return the inversion hyperplane arrangement of \(self\).

INPUT:

- side = 'right' (default) or 'left'
OUTPUT:

A (central) hyperplane arrangement whose hyperplanes correspond to the inversions of \texttt{self} given as roots.

The \texttt{side} parameter determines on which side to compute the inversions.

EXAMPLES:

```python
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([1, 2, 3, 1, 2])
sage: A = w.inversion_arrangement(); A
Arrangement of 5 hyperplanes of dimension 3 and rank 3
sage: A.hyperplanes()
(Hyperplane 0*a1 + 0*a2 + a3 + 0,
 Hyperplane 0*a1 + a2 + 0*a3 + 0,
 Hyperplane 0*a1 + a2 + a3 + 0,
 Hyperplane a1 + a2 + 0*a3 + 0,
 Hyperplane a1 + a2 + a3 + 0)
```

```python
>>> from sage.all import *
\>>> W = WeylGroup(['A',Integer(3)])
\>>> w = W.from_reduced_word([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2)])
\>>> A = w.inversion_arrangement(); A
Arrangement of 5 hyperplanes of dimension 3 and rank 3
\>>> A.hyperplanes()
(Hyperplane 0*a1 + 0*a2 + a3 + 0,
 Hyperplane 0*a1 + a2 + 0*a3 + 0,
 Hyperplane 0*a1 + a2 + a3 + 0,
 Hyperplane a1 + a2 + 0*a3 + 0,
 Hyperplane a1 + a2 + a3 + 0)
```

The identity element gives the empty arrangement:

```python
sage: W = WeylGroup(['A',3])
sage: W.one().inversion_arrangement()
Empty hyperplane arrangement of dimension 3
```

```python
\>>> from sage.all import *
\>>> W = WeylGroup(['A',Integer(3)])
\>>> W.one().inversion_arrangement()
Empty hyperplane arrangement of dimension 3
```

\texttt{inversions} (\texttt{side='right'}, \texttt{inversion_type='reflections'}

Return the set of inversions of \texttt{self}.

INPUT:

- \texttt{side} – 'right' (default) or 'left'
- \texttt{inversion_type} – 'reflections' (default), 'roots', or 'coroots'

OUTPUT:

For reflections, the set of reflections \(r\) in the Weyl group such that \(\texttt{self} \cdot r < \texttt{self}\). For (co)roots, the set of positive (co)roots that are sent by \texttt{self} to negative (co)roots; their associated reflections are described above.

If \texttt{side} is 'left', the inverse Weyl group element is used.

EXAMPLES:
sage: W = WeylGroup(['C',2], prefix="s")
sage: w = W.from_reduced_word([1,2])
sage: w.inversions()
[s2, s2*s1*s2]
sage: w.inversions(inversion_type = 'reflections')
[s2, s2*s1*s2]
sage: w.inversions(inversion_type = 'roots')
[alpha[2], alpha[1] + alpha[2]]
sage: w.inversions(inversion_type = 'coroots')
[alphacheck[2], alphacheck[1] + 2*alphacheck[2]]
sage: w.inversions(side = 'left')
[s1, s1*s2*s1]
sage: w.inversions(side = 'left', inversion_type = 'roots')
[alpha[1], 2*alpha[1] + alpha[2]]
sage: w.inversions(side = 'left', inversion_type = 'coroots')
[alphacheck[1], alphacheck[1] + alphacheck[2]]

>>> from sage.all import *
>>> W = WeylGroup(['C',Integer(2)], prefix="s")
>>> w = W.from_reduced_word([Integer(1),Integer(2)])
>>> w.inversions()
[s2, s2*s1*s2]
>>> w.inversions(inversion_type = 'reflections')
[s2, s2*s1*s2]
>>> w.inversions(inversion_type = 'roots')
[alpha[2], alpha[1] + alpha[2]]
>>> w.inversions(inversion_type = 'coroots')
[alphacheck[2], alphacheck[1] + 2*alphacheck[2]]
>>> w.inversions(side = 'left')
[s1, s1*s2*s1]
>>> w.inversions(side = 'left', inversion_type = 'roots')
[alpha[1], 2*alpha[1] + alpha[2]]
>>> w.inversions(side = 'left', inversion_type = 'coroots')
[alphacheck[1], alphacheck[1] + alphacheck[2]]

is_pieri_factor()

Returns whether self is a Pieri factor, as used for computing Stanley symmetric functions.

See also:

- stanley_symmetric_function()
- WeylGroups.ParentMethods.pieri_factors()

EXAMPLES:

sage: W = WeylGroup(['A',5,1])
sage: W.from_reduced_word([3,2,5]).is_pieri_factor()
True
sage: W.from_reduced_word([3,2,4,5]).is_pieri_factor()
False

sage: W = WeylGroup(['C',4,1])
sage: W.from_reduced_word([0,2,1]).is_pieri_factor()
True
sage: W.from_reduced_word([0,2,1,0]).is_pieri_factor()
False

sage: W = WeylGroup(['B',3])

(continues on next page)
left_pieri_factorizations (max_length=None)

Returns all factorizations of self as \( uv \), where \( u \) is a Pieri factor and \( v \) is an element of the Weyl group.

See also:

• WeylGroups.ParentMethods.pieri_factors()
• sage.combinat.root_system.pieri_factors

EXAMPLES:

If we take \( w = w_0 \) the maximal element of a strict parabolic subgroup of type \( A_{n_1} \times \cdots \times A_{n_k} \), then the Pieri factorizations are in correspondence with all Pieri factors, and there are \( \prod 2^{n_i} \) of them:

| sage: W = WeylGroup(['A', 4, 1]) |
| sage: W.from_reduced_word([]).left_pieri_factorizations().cardinality() | 1 |
| sage: W.from_reduced_word([1]).left_pieri_factorizations().cardinality() | 2 |
| sage: W.from_reduced_word([1,2,1]).left_pieri_factorizations().cardinality() | 4 |
| sage: W.from_reduced_word([1,2,3,1,2,1]).left_pieri_factorizations().cardinality() | 8 |
| sage: W.from_reduced_word([1,3]).left_pieri_factorizations().cardinality() | 4 |
sage: W.from_reduced_word([1,3,4,3]).left_pieri_factorizations().cardinality()
8

sage: W.from_reduced_word([2,1]).left_pieri_factorizations().cardinality()
3

sage: W.from_reduced_word([1,2]).left_pieri_factorizations().cardinality()
2

sage: [W.from_reduced_word([1,2]).left_pieri_factorizations(max_length=i).cardinality()
   for i in [-1, 0, 1, 2]]
[0, 1, 2, 2]

sage: W = WeylGroup(['C', 4, 1])

sage: w = W.from_reduced_word([0, 3, 2, 1, 0])

sage: w.left_pieri_factorizations().cardinality()
7

sage: [(u.reduced_word(), v.reduced_word())
   for (u, v) in w.left_pieri_factorizations()]
[([], [3, 2, 0, 1, 0]),
 (0, [3, 2, 1, 0]),
 ([3], [2, 0, 1, 0]),
 ([3, 0], [2, 1, 0]),
 ([3, 2], [0, 1, 0]),
 ([3, 2, 0], [1, 0]),
 ([3, 2, 0, 1], [0])]

sage: W = WeylGroup(['B', 4, 1])

sage: W.from_reduced_word([0, 2, 1, 0]).left_pieri_factorizations().cardinality()
6

```
>>> from sage.all import *

>>> W = WeylGroup(['A', Integer(4), Integer(1)])

>>> W.from_reduced_word([]).left_pieri_factorizations().cardinality()
1

>>> W.from_reduced_word([Integer(1)]).left_pieri_factorizations().cardinality()
2

>>> W.from_reduced_word([Integer(1), Integer(2), Integer(1)]).left_pieri_factorizations().cardinality()
4

>>> W.from_reduced_word([Integer(1), Integer(2), Integer(3), Integer(1),
Integer(2), Integer(1)]).left_pieri_factorizations().cardinality()
8

>>> W.from_reduced_word([Integer(1), Integer(3)]).left_pieri_factorizations().cardinality()
4

>>> W.from_reduced_word([Integer(1), Integer(3), Integer(4), Integer(3)]).
left_pieri_factorizations().cardinality()
8

>>> W.from_reduced_word([Integer(2), Integer(1)]).left_pieri_factorizations().cardinality()
3
```
>>> W.from_reduced_word([Integer(1),Integer(2)]).left_pieri_factorizations().cardinality()
2
>>> [W.from_reduced_word([Integer(1),Integer(2)]).left_pieri_factorizations(max_length=i).cardinality()... for i in [-Integer(1), Integer(0), Integer(1), Integer(2)]]
[0, 1, 2, 2]

W = WeylGroup(['C',Integer(4),Integer(1)])

w = W.from_reduced_word([Integer(0),Integer(3),Integer(2),Integer(1),Integer(0)])

w.left_pieri_factorizations().cardinality()
7

[(u.reduced_word(),v.reduced_word())... for (u,v) in w.left_pieri_factorizations()]
[([], [3, 2, 0, 1, 0]),
 ([0], [3, 2, 1, 0]),
 ([3], [2, 0, 1, 0]),
 ([3, 2], [0, 1, 0]),
 ([3, 2, 0], [1, 0]),
 ([3, 2, 0, 1], [0])]

W = WeylGroup(['B',Integer(4),Integer(1)])

W.from_reduced_word([Integer(0),Integer(2),Integer(1),Integer(0)]).left_pieri_factorizations().cardinality()
6

quantum_bruhat_successors(index_set=None, roots=False, quantum_only=False)
Return the successors of self in the quantum Bruhat graph on the parabolic quotient of the Weyl group determined by the subset of Dynkin nodes index_set.

INPUT:
• self – a Weyl group element, which is assumed to be of minimum length in its coset with respect to the parabolic subgroup
• index_set – (default: None) indicates the set of simple reflections used to generate the parabolic subgroup; the default value indicates that the subgroup is the identity
• roots – (default: False) if True, returns the list of 2-tuples (w, α) where w is a successor and α is the positive root associated with the successor relation
• quantum_only – (default: False) if True, returns only the quantum successors

EXAMPLES:

sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2])
sage: w.quantum_bruhat_successors([1], roots = True)
[(s3, alpha[2]), (s1*s2*s3*s2, alpha[3]), (s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3])]
sage: w.quantum_bruhat_successors([1,3])
[1, s2*s3*s1*s2]
sage: w.quantum_bruhat_successors(roots = True)
[(s3*s1*s2*s1, alpha[1]), (s3*s1, alpha[2]), (s1*s2*s3*s2, alpha[3]), (s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3])]
sage: w.quantum_bruhat_successors()
reflection_to_coroot()

Return the coroot associated with the reflection self.

EXAMPLES:

```python
sage: W = WeylGroup(['C',2],prefix="s")
sage: W.from_reduced_word([1,2,1]).reflection_to_coroot()
sage: W.from_reduced_word([1,2]).reflection_to_coroot()
Traceback (most recent call last):
... ValueError: s1*s2 is not a reflection
sage: W.long_element().reflection_to_coroot()
Traceback (most recent call last):
... ValueError: s2*s1*s2*s1 is not a reflection
```
reflection_to_root()

Return the root associated with the reflection self.

EXAMPLES:

```
sage: W = WeylGroup(['C', 2], prefix="s")
sage: W.from_reduced_word([1, 2, 1]).reflection_to_root()
sage: W.from_reduced_word([1, 2]).reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s1*s2 is not a reflection
sage: W.long_element().reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s2*s1*s2*s1 is not a reflection

>>> from sage.all import *

>>> W = WeylGroup(['C', Integer(2)], prefix="s")

>>> W.from_reduced_word([Integer(1), Integer(2), Integer(1)]).reflection_to_root()

>>> W.from_reduced_word([Integer(1), Integer(2)]).reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s1*s2 is not a reflection

>>> W.long_element().reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s2*s1*s2*s1 is not a reflection
```

stanley_symmetric_function()

Return the affine Stanley symmetric function indexed by self.

INPUT:

- self – an element \( w \) of a Weyl group

Returns the affine Stanley symmetric function indexed by \( w \). Stanley symmetric functions are defined as generating series of the factorizations of \( w \) into Pieri factors and weighted by a statistic on Pieri factors.

See also:

- `stanley_symmetric_function_as_polynomial()`
- `WeylGroups.ParentMethods.pieri_factors()`
- `sage.combinat.root_system.pieri_factors`

EXAMPLES:

```
sage: W = WeylGroup(['A', 3, 1])
sage: W.from_reduced_word([3, 1, 2, 0, 3, 1, 0]).stanley_symmetric_function()
```

(continues on next page)
8*m[1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 1, 1, 1, 1] + 2*m[2, 2, 1, 1, 1, 1] + m[2, 2, 2, 1, 1]

sage: A = AffinePermutationGroup(['A',3,1])
sage: A.from_reduced_word([3,1,2,0,3,1,0]).stanley_symmetric_function()
8*m[1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 1, 1, 1, 1] + 2*m[2, 2, 1, 1, 1, 1] + m[2, 2, 2, 1, 1]

sage: W = WeylGroup(['C',3,1])
sage: W.from_reduced_word([0,2,1,0]).stanley_symmetric_function()
32*m[1, 1, 1, 1] + 16*m[2, 1, 1] + 8*m[2, 2] + 4*m[3, 1]

sage: W = WeylGroup(['B',3,1])
sage: W.from_reduced_word([3,2,1]).stanley_symmetric_function()
2*m[1, 1, 1] + m[2, 1] + 1/2*m[3]

sage: W = WeylGroup(['B',4])
sage: w = W.from_reduced_word([3,2,3,1])
sage: w.stanley_symmetric_function()  # long time (6s on sage.math, 2011)
48*m[1, 1, 1, 1] + 24*m[2, 1, 1] + 12*m[2, 2] + 8*m[3, 1] + 2*m[4]

sage: a = A([-2,0,1,4,12])
sage: a.stanley_symmetric_function()
6*m[1, 1, 1, 1, 1, 1, 1, 1] + 5*m[2, 1, 1, 1, 1, 1, 1, 1]
+ 4*m[2, 2, 1, 1, 1, 1, 1, 1] + 3*m[3, 1, 1, 1, 1, 1, 1, 1]
+ 2*m[3, 2, 1, 1, 1, 1, 1, 1] + 2*m[3, 3, 1, 1, 1, 1, 1, 1]
+ 2*m[4, 1, 1, 1, 1, 1, 1, 1] + m[4, 2, 1, 1, 1, 1, 1, 1]
+ 3*m[5, 1, 1, 1, 1, 1, 1, 1] + 2*m[6, 1, 1, 1, 1, 1, 1, 1]
+ m[7, 1, 1, 1, 1, 1, 1, 1] + 2*m[8, 1, 1, 1, 1, 1, 1, 1]
+ m[9, 1, 1, 1, 1, 1, 1, 1] + 3*m[10, 1, 1, 1, 1, 1, 1, 1]
+ 2*m[11, 1, 1, 1, 1, 1, 1, 1] + m[12, 1, 1, 1, 1, 1, 1, 1]

>>> from sage.all import *

>>> W = WeylGroup(['A',3,1])
>>> W.from_reduced_word([3,1,2,0,3,1,0]).stanley_symmetric_function()
8*m[1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 1, 1, 1, 1] + 2*m[2, 2, 1, 1, 1, 1] + m[2, 2, 2, 1, 1]

>>> A = AffinePermutationGroup(['A',3,1])
>>> A.from_reduced_word([3,1,2,0,3,1,0]).stanley_symmetric_function()
8*m[1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 1, 1, 1, 1] + 2*m[2, 2, 1, 1, 1, 1] + m[2, 2, 2, 1, 1]

>>> W = WeylGroup(['C',3,1])
>>> W.from_reduced_word([0,2,1,0]).stanley_symmetric_function()
32*m[1, 1, 1, 1] + 16*m[2, 1, 1] + 8*m[2, 2] + 4*m[3, 1]

>>> W = WeylGroup(['B',3,1])
>>> W.from_reduced_word([3,2,1]).stanley_symmetric_function()
2*m[1, 1, 1] + m[2, 1] + 1/2*m[3]

>>> W = WeylGroup(['B',4])
>>> w = W.from_reduced_word([3,2,3,1])

48*m[1, 1, 1, 1] + 24*m[2, 1, 1] + 12*m[2, 2] + 8*m[3, 1] + 2*m[4]
A = AffinePermutationGroup(['A', Integer(4), Integer(1)])
a = A([-Integer(2), Integer(0), Integer(1), Integer(4), Integer(12)])
a.stanley_symmetric_function()
6*m[1, 1, 1, 1, 1, 1, 1, 1] + 5*m[2, 1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 2, 1, 1, 1, 1, 1] + 3*m[2, 2, 1, 1, 1, 1, 1, 1] + 2*m[2, 2, 2, 1, 1, 1, 1, 1] + 4*m[3, 1, 1, 1, 1, 1, 1, 1] + 3*m[3, 2, 1, 1, 1, 1, 1, 1] + 2*m[3, 3, 1, 1, 1, 1, 1, 1] + 2*m[4, 1, 1, 1, 1, 1, 1, 1] + m[4, 2, 1, 1, 1, 1, 1, 1] + m[4, 2, 2, 1, 1, 1, 1, 1] + m[4, 3, 1, 1, 1, 1, 1, 1] + m[4, 3, 2, 1, 1, 1, 1, 1] + m[4, 4, 1, 1, 1, 1, 1, 1]

One more example (Issue #14095):

sage: G = SymmetricGroup(4)
sage: w = G.from_reduced_word([3,2,3,1])
sage: w.stanley_symmetric_function()
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]

REFERENCES:
• [BH1994]
• [Lam2008]
• [LSS2009]
• [Pon2010]

stanley_symmetric_function_as_polynomial (max_length=None)

Returns a multivariate generating function for the number of factorizations of a Weyl group element into Pieri factors of decreasing length, weighted by a statistic on Pieri factors.

See also:
• stanley_symmetric_function()
• WeylGroups.ParentMethods.pieri_factors()
• sage.combinat.root_system.pieri_factors

INPUT:
• self – an element w of a Weyl group \( W \)
• max_length – a non negative integer or infinity (default: infinity)

Returns the generating series for the Pieri factorizations \( w = u_1 \cdots u_k \), where \( u_i \) is a Pieri factor for all \( i \), \( l(w) = \sum_{i=1}^{k} l(u_i) \) and \( \max_{\text{length}} \geq l(u_1) \geq \cdots \geq l(u_k) \).

A factorization \( u_1 \cdots u_k \) contributes a monomial of the form \( \prod_{i} x_l(u_i) \), with coefficient given by \( \prod_{i} 2^{c(u_i)} \), where \( c \) is a type-dependent statistic on Pieri factors, as returned by the method \( u[1].stanley_symm_poly_weight() \).

EXAMPLES:
sage: W = WeylGroup(['A', 3, 1])
sage: W.from_reduced_word([[]]).stanley_symmetric_function_as_polynomial()
1
sage: W.from_reduced_word([[]]).stanley_symmetric_function_as_polynomial()
x1
sage: W.from_reduced_word([[1]]).stanley_symmetric_function_as_→polynomial()
sage: W.from_reduced_word([2,1]).stanley_symmetric_function_as_.
→polynomial()
x1^2 + x2
sage: W.from_reduced_word([1,2,1]).stanley_symmetric_function_as_
→polynomial()
2*x1^3 + x1*x2
sage: W.from_reduced_word([1,2,1,0]).stanley_symmetric_function_as_
→polynomial()
3*x1^4 + 2*x1^2*x2 + x2^2 + x1*x3
sage: x = W.from_reduced_word([1,2,3,1,2,1,0])
sage: x.stanley_symmetric_function_as_polynomial()
# long time
22*x1^7 + 11*x1^5*x2 + 5*x1^3*x2^2 + 3*x1^4*x3 + 2*x1*x2^3 + x1^2*x2*x3
sage: y = W.from_reduced_word([3,1,2,0,3,1,0])
sage: y.stanley_symmetric_function_as_polynomial()
# long time
8*x1^7 + 4*x1^5*x2 + 2*x1^3*x2^2 + x1*x2^3
sage: W = WeylGroup(['C',3,1])
sage: W.from_reduced_word([0,2,1,0]).stanley_symmetric_function_as_
→polynomial()
32*x1^4 + 16*x1^2*x2 + 8*x2^2 + 4*x1*x3
sage: W = WeylGroup(['B',3,1])
sage: W.from_reduced_word([3,2,1]).stanley_symmetric_function_as_
→polynomial()
2*x1^3 + x1*x2 + 1/2*x3

>>> from sage.all import *

>>> W = WeylGroup(['A', Integer(3), Integer(1)])

>>> W.from_reduced_word([]).stanley_symmetric_function_as_polynomial()
1

>>> W.from_reduced_word([Integer(1)]).stanley_symmetric_function_as_
→polynomial()
x1

>>> W.from_reduced_word([Integer(1),Integer(2)]).stanley_symmetric_  
→function_as_polynomial()
x1^2

>>> W.from_reduced_word([Integer(2),Integer(1)]).stanley_symmetric_  
→function_as_polynomial()
x1^2 + x2

>>> W.from_reduced_word([Integer(1),Integer(2),Integer(1)]).stanley_  
→symmetric_function_as_polynomial()
2*x1^3 + x1*x2

>>> W.from_reduced_word([Integer(1),Integer(2),Integer(1),Integer(0)]).  
→stanley_symmetric_function_as_polynomial()
3*x1^4 + 2*x1^2*x2 + x2^2 + x1*x3

>>> x = W.from_reduced_word([Integer(1),Integer(2),Integer(3),Integer(1),  
→Integer(2),Integer(1),Integer(0)])

>>> x.stanley_symmetric_function_as_polynomial()  # long time
22*x1^7 + 11*x1^5*x2 + 5*x1^3*x2^2 + 3*x1^4*x3 + 2*x1*x2^3 + x1^2*x2*x3

>>> y = W.from_reduced_word([Integer(3),Integer(1),Integer(2),Integer(0),  
→Integer(3),Integer(1),Integer(0)])

>>> y.stanley_symmetric_function_as_polynomial()  # long time
8*x1^7 + 4*x1^5*x2 + 2*x1^3*x2^2 + x1*x2^3

>>> W = WeylGroup(['C',Integer(3),Integer(1)])
Algorithm: Induction on the left Pieri factors. Note that this induction preserves subsets of $W$ which are stable by taking right factors, and in particular Grassmanian elements.

Finite
alias of FiniteWeylGroups
class ParentMethods
    Bases: object
    bruhat_cone(x, y, side='upper', backend='cdd')
    Return the (upper or lower) Bruhat cone associated to the interval $[x, y]$.
    To a cover relation $v \prec w$ in strong Bruhat order you can assign a positive root $\beta$ given by the unique reflection $s_\beta$ such that $s_\beta v = w$.
    The upper Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $x \prec \alpha$ for all atoms $\alpha$ in the interval. The lower Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $c \prec y$ for all coatoms $c$ in the interval.
    INPUT:
    • $x$ – an element in the group $W$
    • $y$ – an element in the group $W$
    • side (default: 'upper') – must be one of the following:
      – 'upper' – return the upper Bruhat cone of the interval $[x, y]$
      – 'lower' – return the lower Bruhat cone of the interval $[x, y]$
    • backend – string (default: 'cdd'); the backend to use to create the polyhedron
    EXAMPLES:

    >>> from sage.all import *
    >>> W = WeylGroup(['A', Integer(2)])
    >>> x = W.from_reduced_word([Integer(1)])
    >>> y = W.w0

Finite
alias of FiniteWeylGroups
class ParentMethods
    Bases: object
    bruhat_cone(x, y, side='upper', backend='cdd')
    Return the (upper or lower) Bruhat cone associated to the interval $[x, y]$.
    To a cover relation $v \prec w$ in strong Bruhat order you can assign a positive root $\beta$ given by the unique reflection $s_\beta$ such that $s_\beta v = w$.
    The upper Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $x \prec \alpha$ for all atoms $\alpha$ in the interval. The lower Bruhat cone of the interval $[x, y]$ is the non-empty, polyhedral cone generated by the roots corresponding to $c \prec y$ for all coatoms $c$ in the interval.
    INPUT:
    • $x$ – an element in the group $W$
    • $y$ – an element in the group $W$
    • side (default: 'upper') – must be one of the following:
      – 'upper' – return the upper Bruhat cone of the interval $[x, y]$
      – 'lower' – return the lower Bruhat cone of the interval $[x, y]$
    • backend – string (default: 'cdd'); the backend to use to create the polyhedron
    EXAMPLES:

    >>> from sage.all import *
    >>> W = WeylGroup(['A', Integer(2)])
    >>> x = W.from_reduced_word([Integer(1)])
    >>> y = W.w0

```python
>>> W.bruhat_cone(x, y)
A 2-dimensional polyhedron in QQ^3
defined as the convex hull of 1 vertex and 2 rays

>>> W = WeylGroup(['E', Integer(6)])
>>> x = W.one()
>>> y = W.w0
>>> W.bruhat_cone(x, y, side='lower')
A 6-dimensional polyhedron in QQ^8
defined as the convex hull of 1 vertex and 6 rays
```

REFERENCES:
- [Dy1994]
- [JS2021]

**coxeter_matrix()**

Return the Coxeter matrix associated to self.

EXAMPLES:

```python
sage: G = WeylGroup(['A',3])
sage: G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

```python
>>> from sage.all import *

>>> G = WeylGroup(['A',Integer(3)])

>>> G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

**pieri_factors(*args, **keywords)**

Returns the set of Pieri factors in this Weyl group.

For any type, the set of Pieri factors forms a lower ideal in Bruhat order, generated by all the conjugates of some special element of the Weyl group. In type $A_n$, this special element is $s_n \cdots s_1$, and the conjugates are obtained by rotating around this reduced word.

These are used to compute Stanley symmetric functions.

See also:
- WeylGroups.ElementMethods.stanley_symmetric_function()
- sage.combinat.root_system.pieri_factors

EXAMPLES:

```python
sage: W = WeylGroup(['A',5,1])
sage: PF = W.pieri_factors()
sage: PF.cardinality()
63

sage: W = WeylGroup(['B',3])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF])
[[],
```

(continues on next page)
from sage.all import *

W = WeylGroup(['A',Integer(5),Integer(1)])
PF = W.pieri_factors()
PF.cardinality()
63

W = WeylGroup(['B',Integer(3)])
PF = W.pieri_factors()
sorted([w.reduced_word() for w in PF])

W = WeylGroup(['C',Integer(4),Integer(1)])
PF = W.pieri_factors()
quantum_bruhat_graph\((\text{index\_set}())\)

Return the quantum Bruhat graph of the quotient of the Weyl group by a parabolic subgroup \(W_J\).

**INPUT:**

- \(\text{index\_set}\) – (default: \()\) a tuple \(J\) of nodes of the Dynkin diagram

By default, the value for \(\text{index\_set}\) indicates that the subgroup is trivial and the quotient is the full Weyl group.

**EXAMPLES:**

```
sage: W = WeylGroup(['A',3], prefix="s")
sage: g = W.quantum_bruhat_graph((1,3))
sage: g
Parabolic Quantum Bruhat Graph of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space)
for nodes (1, 3): Digraph on 6 vertices
sage: g.vertices(sort=True)
[s2*s3*s1*s2, s3*s1*s2, s1*s2, s3*s2, s2, 1]
sage: g.edges(sort=True)
[(s2*s3*s1*s2, s2, alpha[2]),
 (s3*s1*s2, s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3]),
 (s3*s1*s2, 1, alpha[2]),
 (s1*s2, s3*s1*s2, alpha[2] + alpha[3]),
 (s3*s2, s3*s1*s2, alpha[1] + alpha[2]),
 (s2, s1*s2, alpha[1] + alpha[2]),
 (s2, s3*s2, alpha[2] + alpha[3]),
 (1, s2, alpha[2])]
sage: W = WeylGroup(['A',3,1], prefix="s")
sage: g = W.quantum_bruhat_graph()
Traceback (most recent call last):
...
ValueError: the Cartan type ['A', 3, 1] is not finite
```

```python
>>> from sage.all import *
>>> W = WeylGroup(['A',Integer(3)], prefix="s")
>>> g = W.quantum_bruhat_graph((Integer(1),Integer(3)))
>>> g
Parabolic Quantum Bruhat Graph of Weyl Group of type ['A', 3]
(as a matrix group acting on the ambient space)
for nodes (1, 3): Digraph on 6 vertices
>>> g.vertices(sort=True)
[s2*s3*s1*s2, s3*s1*s2, s1*s2, s3*s2, s2, 1]
>>> g.edges(sort=True)
[(s2*s3*s1*s2, s2, alpha[2]),
 (s3*s1*s2, s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3]),
 (s3*s1*s2, 1, alpha[2]),
 (s1*s2, s3*s1*s2, alpha[2] + alpha[3]),
 (s3*s2, s3*s1*s2, alpha[1] + alpha[2]),
 (s2, s1*s2, alpha[1] + alpha[2]),
 (s2, s3*s2, alpha[2] + alpha[3]),
 (1, s2, alpha[2])]
>>> W = WeylGroup(['A',Integer(3),Integer(1)], prefix="s")
>>> g = W.quantum_bruhat_graph()
Traceback (most recent call last):
...
ValueError: the Cartan type ['A', 3, 1] is not finite
```
Return None.

Indeed, the category of Weyl groups defines no additional structure: Weyl groups are a special class of Coxeter groups.

See also:

Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

```
sage: WeylGroups().additional_structure()
```

```
>>> from sage.all import *

>>> WeylGroups().additional_structure()
```

super_categories()

EXAMPLES:

```
sage: WeylGroups().super_categories()
[Category of Coxeter groups]
```

```
>>> from sage.all import *

>>> WeylGroups().super_categories()
[Category of Coxeter groups]
```

4.164 Technical Categories

4.164.1 Facade Sets

For background, see What is a facade set?.

```
class sage.categories.facade_sets.FacadeSets(base_category)
    Bases: CategoryWithAxiom_singleton

class ParentMethods
    Bases: object

    facade_for()

    Returns the parents this set is a facade for

    This default implementation assumes that self has an attribute _facade_for, typically initialized by Parent.__init__(). If the attribute is not present, the method raises a NotImplementedError.

    EXAMPLES:

    sage: S = Sets().Facade().example(); S
    An example of facade set: the monoid of positive integers
    sage: S.facade_for()
    (Integer Ring,)
```


Check that Issue #13801 is corrected:

```python
sage: class A(Parent):
    ....:     def __init__(self):
    ....:         Parent.__init__(self, category=Sets(), facade=True)
```

```python
sage: a = A()
sage: a.facade_for()
Traceback (most recent call last):
... 
NotImplementedError: this parent did not specify which parents it is a facade for
```

```python
sage: class A(Parent):
    ....:     def __init__(self):
    ....:         Parent.__init__(self, category=Sets(), facade=True)
```

```python
a = A()
a facaze_for()
Traceback (most recent call last):
... 
NotImplementedError: this parent did not specify which parents it is a facade for
```

**is_parent_of(element)**

Returns whether `self` is the parent of `element`

**INPUT:**

- `element` — any object

Since `self` is a facade domain, this actually tests whether the parent of `element` is any of the parent `self` is a facade for.

**EXAMPLES:**

```python
sage: S = Sets().Facade().example(); S
An example of facade set: the monoid of positive integers
sage: S.is_parent_of(1)
True
sage: S.is_parent_of(1/2)
False
```

This method differs from `__contains__()` in two ways. First, this does not take into account the fact that `self` may be a strict subset of the parent(s) it is a facade for:
Warning: this implementation does not handle facade parents of facade parents. Is this a feature we want generically?

example (choice='subset')

Returns an example of facade set, as per Category.example().

INPUT:

• choice = 'union' or 'subset' (default: 'subset').

EXAMPLES:

```python
sage: Sets().Facade().example()
An example of facade set: the monoid of positive integers
sage: Sets().Facade().example(choice='union')
An example of a facade set: the integers completed by +-infinity
sage: Sets().Facade().example(choice='subset')
An example of facade set: the monoid of positive integers
```
CHAPTER FIVE

FUNCTORIAL CONSTRUCTIONS

5.1 Covariant Functorial Constructions

A functorial construction is a collection of functors \((F_{Cat})_{Cat}\) (indexed by a collection of categories) which associate to a sequence of parents \((A, B, \ldots)\) in a category \(Cat\) a parent \(F_{Cat}(A, B, \ldots)\). Typical examples of functorial constructions are cartesian_product and tensor_product.

The category of \(F_{Cat}(A, B, \ldots)\), which only depends on \(Cat\), is called the (functorial) construction category.

A functorial construction is (category)-covariant if for every categories \(Cat\) and \(SuperCat\), the category of \(F_{Cat}(A, B, \ldots)\) is a subcategory of the category of \(F_{SuperCat}(A, B, \ldots)\) whenever \(Cat\) is a subcategory of \(SuperCat\).

A functorial construction is (category)-regressive if the category of \(F_{Cat}(A, B, \ldots)\) is a subcategory of \(Cat\).

The goal of this module is to provide generic support for covariant functorial constructions. In particular, given some parents \(A, B, \ldots\) in respective categories \(Cat_A, Cat_B, \ldots\), it provides tools for calculating the best known category for the parent \(F(A, B, \ldots)\). For examples, knowing that Cartesian products of semigroups (resp. monoids, groups) have a semigroup (resp. monoid, group) structure, and given a group \(B\) and two monoids \(A\) and \(C\) it can calculate that \(A \times B \times C\) is naturally endowed with a monoid structure.

See CovariantFunctorialConstruction, CovariantConstructionCategory and RegressiveCovariantConstructionCategory for more details.

AUTHORS:

• Nicolas M. Thiery (2010): initial revision

```
class sage.categories.covariant_functorial_construction.CovariantConstructionCategory (category, *args)
```

Bases: FunctorialConstructionCategory

Abstract class for categories \(F_{Cat}\) obtained through a covariant functorial construction

```
additional_structure()
```

Return the additional structure defined by \self. By default, a functorial construction category \(A.F()\) defines additional structure if and only if \(A\) is the category defining \(F\). The rationale is that, for a subcategory \(B\) of \(A\), the fact that \(B.F()\) morphisms shall preserve the \(F\)-specific structure is already imposed by \(A.F()\).

See also:

• Category.additional_structure().
• is_construction_defined_by_base().
EXAMPLES:

```
sage: Modules(ZZ).Graded().additional_structure()
Category of graded modules over Integer Ring
sage: Algebras(ZZ).Graded().additional_structure()
```

```python
>>> from sage.all import *
>>> Modules(ZZ).Graded().additional_structure()
Category of graded modules over Integer Ring
>>> Algebras(ZZ).Graded().additional_structure()
```

classmethod `default_super_categories` *(category, *args)*

Return the default super categories of \( F_{\text{Cat}}(A, B, \ldots) \) for \( A, B, \ldots \) parents in \( \text{Cat} \).

**INPUT:**
- `cls` — the category class for the functor \( F \)
- `category` — a category \( \text{Cat} \)
- `*args` — further arguments for the functor

**OUTPUT:** a (join) category

The default implementation is to return the join of the categories of \( F(A, B, \ldots) \) for \( A, B, \ldots \) in turn in each of the super categories of `category`.

This is implemented as a class method, in order to be able to reconstruct the functorial category associated to each of the super categories of `category`.

**EXAMPLES:**

Bialgebras are both algebras and coalgebras:

```
sage: Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field,
 Category of coalgebras over Rational Field]
```

```python
>>> from sage.all import *
>>> Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field,
 Category of coalgebras over Rational Field]
```

Hence tensor products of bialgebras are tensor products of algebras and tensor products of coalgebras:

```
sage: Bialgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of algebras over Rational Field,
 Category of tensor products of coalgebras over Rational Field]
```

```python
>>> from sage.all import *
>>> Bialgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of algebras over Rational Field,
 Category of tensor products of coalgebras over Rational Field]
```

Here is how `default_super_categories()` was called internally:

```
sage: C = sage.categories.tensor.TensorProductsCategory
sage: C.default_super_categories(Bialgebras(QQ))
Join of Category of tensor products of algebras over Rational Field
    and Category of tensor products of coalgebras over Rational Field
```
We now show a similar example, with the `Algebra` functor which takes a parameter `Q`:

```
sage: FiniteMonoids().super_categories()
[Category of monoids, Category of finite semigroups]
sage: sorted(FiniteMonoids().Algebras(QQ).super_categories(), key=str)
[Category of finite dimensional algebras with basis over Rational Field,
 Category of finite set algebras over Rational Field,
 Category of monoid algebras over Rational Field]
```

Note that neither the category of `finite` semigroup algebras nor that of monoid algebras appear in the result; this is because there is currently nothing specific implemented about them.

Here is how `default_super_categories()` was called internally:

```
sage: C = sage.categories.algebra_functor.AlgebrasCategory
sage: C.default_super_categories(FiniteMonoids(), QQ)
Join of Category of finite dimensional algebras with basis over Rational Field
   and Category of monoid algebras over Rational Field
   and Category of finite set algebras over Rational Field
```

### `is_construction_defined_by_base()`

Return whether the construction is defined by the base of `self`.

**EXAMPLES:**

The graded functorial construction is defined by the modules category. Hence this method returns `True` for graded modules and `False` for other graded `xxx` categories:

```
sage: Modules(ZZ).Graded().is_construction_defined_by_base()
True
sage: Algebras(QQ).Graded().is_construction_defined_by_base()
False
sage: Modules(ZZ).WithBasis().Graded().is_construction_defined_by_base()
False
```
This is implemented as follows: given the base category $A$ and the construction $F$ of self, that is self=$A$. $F()$, check whether no super category of $A$ has $F$ defined.

**Note:** Recall that, when $A$ does not implement the construction $F$, a join category is returned. Therefore, in such cases, this method is not available:

```
sage: Bialgebras(QQ).Graded().is_construction_defined_by_base()
Traceback (most recent call last):
  ... AttributeError: 'JoinCategory_with_category' object has no attribute 'is_construction_defined_by_base'
```

```python
>>> from sage.all import *
>>> Bialgebras(QQ).Graded().is_construction_defined_by_base()
Traceback (most recent call last):
  ... AttributeError: 'JoinCategory_with_category' object has no attribute 'is_construction_defined_by_base'
```

```
class sage.categories.covariant_functorial_construction.
CovariantFunctorialConstruction

Bases: UniqueRepresentation, SageObject

An abstract class for construction functors $F$ (eg $F =$ Cartesian product, tensor product, $\mathbb{Q}$-algebra, …) such that:

- Each category $\text{Cat}$ (eg $\text{Cat} =$ Groups()) can provide a category $F_{\text{Cat}}$ for parents constructed via this functor (eg. $F_{\text{Cat}} = \text{CartesianProductsOf}($Groups$())$).
- For every category $\text{Cat}$, $F_{\text{Cat}}$ is a subcategory of $F_{\text{SuperCat}}$ for every super category $\text{SuperCat}$ of $\text{Cat}$ (the functorial construction is (category)-covariant).
- For parents $A, B, \ldots$, respectively in the categories $\text{Cat}_A, \text{Cat}_B, \ldots$, the category of $F(A, B, \ldots)$ is $F_{\text{Cat}}$ where $\text{Cat}$ is the meet of the categories $\text{Cat}_A, \text{Cat}_B, \ldots$.

This covers two slightly different use cases:

- In the first use case, one uses directly the construction functor to create new parents:

```
sage: tensor()  # todo: not implemented (add an example)
```

```
>>> from sage.all import *
>>> tensor()  # todo: not implemented (add an example)
```

or even new elements, which indirectly constructs the corresponding parent:

```
sage: tensor(...)  # todo: not implemented
```

1348 Chapter 5. Functorial constructions
>>> from sage.all import *
>>> tensor(Ellipsis) # todo: not implemented

- In the second use case, one implements a parent, and then put it in the category $F_{\text{Cat}}$ to specify supplementary mathematical information about that parent.

The main purpose of this class is to handle automatically the trivial part of the category hierarchy. For example, $\text{CartesianProductsOf(Groups())}$ is set automatically as a subcategory of $\text{CartesianProductsOf(Monoids())}$.

In practice, each subclass of this class should provide the following attributes:

- `_functor_category` – a string which should match the name of the nested category class to be used in each category to specify information and generic operations for elements of this category.

- `_functor_name` – a string which specifies the name of the functor, and also (when relevant) of the method on parents and elements used for calling the construction.

TODO: What syntax do we want for $F_{\text{Cat}}$? For example, for the tensor product construction, which one do we want among (see chat on IRC, on 07/12/2009):

- `tensor(Cat)`
- `tensor((Cat, Cat))`
- `tensor.of((Cat, Cat))`
- `tensor.category_from_categories((Cat, Cat, Cat))`
- `Cat.TensorProducts()`

The syntax `Cat.TensorProducts()` does not support well multivariate constructions like `tensor.of([Algebras(), HopfAlgebras(), ...])`. Also it forces every category to be (somehow) aware of all the tensorial construction that could apply to it, even those which are only induced from super categories.

Note: for each functorial construction, there probably is one (or several) largest categories on which it applies. For example, the `CartesianProducts()` construction makes only sense for concrete categories, that is subcategories of `Sets()`. Maybe we want to model this one way or the other.

`category_from_categories(categories)`

Return the category of $F(A, B, ...)$ for $A, B, ...$ parents in the given categories.

INPUT:

- `self`: a functor $F$
- `categories`: a non empty tuple of categories

EXAMPLES:

```python
sage: Cat1 = Rings()
sage: Cat2 = Groups()
sage: cartesian_product.category_from_categories((Cat1, Cat1, Cat1))
Join of Category of rings and ...
      and Category of Cartesian products of monoids
      and Category of Cartesian products of commutative additive groups
sage: cartesian_product.category_from_categories((Cat1, Cat2))
Category of Cartesian products of monoids
```
```python
>>> from sage.all import *
>>> Cat1 = Rings()
>>> Cat2 = Groups()
>>> cartesian_product.category_from_categories((Cat1, Cat1, Cat1))
Join of Category of rings and ... and Category of Cartesian products of monoids and Category of Cartesian products of commutative additive groups
>>> cartesian_product.category_from_categories((Cat1, Cat2))
Category of Cartesian products of monoids
```

`category_from_category` *(category)*

Return the category of $F(A, B, ...)$ for $A, B, ...$ parents in category.

**INPUT:**

- `self`: a functor $F$
- `category`: a category

**EXAMPLES:**

```python
sage: tensor.category_from_category(ModulesWithBasis(QQ))
Category of tensor products of vector spaces with basis over Rational Field
```

# TODO: add support for parametrized functors

`category_from_parents` *(parents)*

Return the category of $F(A, B, ...)$ for $A, B, ...$ parents.

**INPUT:**

- `self`: a functor $F$
- `parents`: a list (or iterable) of parents.

**EXAMPLES:**

```python
sage: E = CombinatorialFreeModule(QQ, ["a", "b", "c"])
# needs sage.modules
sage: tensor.category_from_parents((E, E, E))
# needs sage.modules
Category of tensor products of finite dimensional vector spaces with basis over Rational Field
```

```python
>>> from sage.all import *
>>> E = CombinatorialFreeModule(QQ, ["a", "b", "c"])
# needs sage.modules
>>> tensor.category_from_parents((E, E, E))
# needs sage.modules
Category of tensor products of finite dimensional vector spaces with basis over Rational Field
```

```python
class sage.categories.covariant_functorial_construction.FunctorialConstructionCategory(category, *args)
```

Chapter 5. Functorial constructions
Bases: Category

Abstract class for categories $F_{\text{Cat}}$ obtained through a functorial construction

`base_category()`

Return the base category of the category `self`.

For any category $\mathcal{B} = F_{\text{Cat}}$ obtained through a functorial construction $F$, the call $\mathcal{B}.\text{base_category()}$ returns the category $\mathcal{Cat}$.

EXAMPLES:

```
sage: Semigroups().Quotients().base_category()
Category of semigroups
```

```
>>> from sage.all import *
>>> Semigroups().Quotients().base_category()
Category of semigroups
```

`classmethod category_of(category, *args)`

Return the image category of the functor $F_{\text{Cat}}$.

This is the main entry point for constructing the category $F_{\text{Cat}}$ of parents $F(A, B, ...)$ constructed from parents $A, B, ...$ in $\mathcal{Cat}$.

INPUT:

- `cls` – the category class for the functorial construction $F$
- `category` – a category $\mathcal{Cat}$
- `*args` – further arguments for the functor

EXAMPLES:

```
sage: C = sage.categories.tensor.TensorProductsCategory
sage: C.category_of(ModulesWithBasis(QQ))
Category of tensor products of vector spaces with basis over Rational Field
```

```
>>> from sage.all import *
>>> C = sage.categories.tensor.TensorProductsCategory
>>> C.category_of(ModulesWithBasis(QQ))
Category of tensor products of vector spaces with basis over Rational Field
```

```
sage: C = sage.categories.algebra_functor.AlgebrasCategory
sage: C.category_of(FiniteMonoids(), QQ)
Join of Category of finite dimensional algebras with basis over Rational Field and Category of monoid algebras over Rational Field and Category of finite set algebras over Rational Field
```

```
>>> from sage.all import *
>>> C = sage.categories.tensor.TensorProductsCategory
>>> C.category_of(ModulesWithBasis(QQ))
Category of tensor products of vector spaces with basis over Rational Field
```

```
>>> C = sage.categories.algebra_functor.AlgebrasCategory
>>> C.category_of(FiniteMonoids(), QQ)
Join of Category of finite dimensional algebras with basis over Rational Field and Category of monoid algebras over Rational Field and Category of finite set algebras over Rational Field
```

`extra_super_categories()`

Return the extra super categories of a construction category.

Default implementation which returns `[]`.
EXCEPTIONS:

```python
def super_categories(self):
    pass
```

Return the super categories of a construction category.

EXAMPLES:

```python
sage: Sets().Subquotients().super_categories()
[Category of sets]
sage: Semigroups().Quotients().super_categories()
[Category of subquotients of semigroups, Category of quotients of sets]

>>> from sage.all import *

Sets().Subquotients().super_categories()
[Category of sets]

Semigroups().Quotients().super_categories()
[Category of subquotients of semigroups, Category of quotients of sets]
```

class sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory

Bases: CovariantConstructionCategory

Abstract class for categories $F_{\mathcal{C}}$ obtained through a regressive covariant functorial construction

```python
classmethod default_super_categories(cls, category, *args):
    pass
```

Return the default super categories of $F_{\mathcal{C}}(A, B, ...)$ for $A, B, ...$ parents in $\mathcal{C}$.

INPUT:

- `cls` -- the category class for the functor $F$
- `category` -- a category $\mathcal{C}$
- `*args` -- further arguments for the functor

OUTPUT:

A join category.

This implements the property that an induced subcategory is a subcategory.

EXAMPLES:

A subquotient of a monoid is a monoid, and a subquotient of semigroup:

```python
sage: Monoids().Subquotients().super_categories()
[Category of monoids, Category of subquotients of semigroups]
```
5.2 Cartesian Product Functorial Construction

AUTHORS:

• Nicolas M. Thiery (2008-2010): initial revision and refactoring

class sage.categories.cartesian_product.CartesianProductFunctor(category=None)
Bases: CovariantFunctorialConstruction, MultivariateConstructionFunctor

The Cartesian product functor.

EXAMPLES:

```python
>>> from sage.all import *
>>> cartesian_product
The cartesian_product functorial construction
```

cartesian_product takes a finite collection of sets, and constructs the Cartesian product of those sets:

```python
sage: A = FiniteEnumeratedSet(['a', 'b', 'c'])
sage: B = FiniteEnumeratedSet([1, 2])
sage: C = cartesian_product([A, B]); C
The Cartesian product of ({'a', 'b', 'c'}, {1, 2})
sage: C.an_element()
('a', 1)
sage: C.list()  # todo: not implemented
[['a', 1], ['a', 2], ['b', 1], ['b', 2], ['c', 1], ['c', 2]]
```

If those sets are endowed with more structure, say they are monoids (hence in the category Monoids()), then the result is automatically endowed with its natural monoid structure:

```python
>>> from sage.all import *
>>> A = FiniteEnumeratedSet(['a', 'b', 'c'])
>>> B = FiniteEnumeratedSet([Integer(1), Integer(2)])
>>> C = cartesian_product([A, B]); C
The Cartesian product of ({'a', 'b', 'c'}, {1, 2})
>>> C.an_element()
('a', 1)
>>> C.list()  # todo: not implemented
[['a', 1], ['a', 2], ['b', 1], ['b', 2], ['c', 1], ['c', 2]]
```

An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')

```python
sage: M = Monoids().example()
sage: M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.rename('M')
sage: C = cartesian_product([M, ZZ, QQ])
sage: C
The Cartesian product of (M, Integer Ring, Rational Field)
```

(continues on next page)
```python
sage: C.an_element()
('abcd', 1, 1/2)
sage: C.an_element()^2
('abcdabcd', 1, 1/4)
sage: C.category()
Category of Cartesian products of monoids
sage: Monoids().CartesianProducts()
Category of Cartesian products of monoids

>>> from sage.all import *
>>> M = Monoids().example()
>>> M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
>>> M.rename('M')
>>> C = cartesian_product([M, ZZ, QQ])
>>> C
The Cartesian product of (M, Integer Ring, Rational Field)
>>> C.an_element()
('abcd', 1, 1/2)
>>> C.an_element()**Integer(2)
('abcdabcd', 1, 1/4)
>>> C.category()
Category of Cartesian products of monoids

>>> Monoids().CartesianProducts()
Category of Cartesian products of monoids

The Cartesian product functor is covariant: if A is a subcategory of B, then A.CartesianProducts() is a subcategory of B.CartesianProducts() (see also CovariantFunctorialConstruction):

sage: C.categories()
[Category of Cartesian products of monoids,
 Category of monoids,
 Category of Cartesian products of semigroups,
 Category of semigroups,
 Category of Cartesian products of unital magmas,
 Category of Cartesian products of magmas,
 Category of magmas,
 Category of Cartesian products of sets,
 Category of sets, ...]

[Category of Cartesian products of monoids,
 Category of monoids,
 Category of Cartesian products of semigroups,
 Category of semigroups,
 Category of Cartesian products of magmas,
 Category of unital magmas,
 Category of magmas,
 Category of Cartesian products of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```
>>> from sage.all import *
>>> C.categories()
[Category of Cartesian products of monoids,
Category of monoids,
Category of Cartesian products of semigroups,
Category of semigroups,
Category of Cartesian products of unital magmas,
Category of Cartesian products of magmas,
Category of unital magmas,
Category of magmas,
Category of Cartesian products of sets,
Category of sets, ...]

[Category of Cartesian products of monoids,
Category of monoids,
Category of Cartesian products of semigroups,
Category of semigroups,
Category of Cartesian products of magmas,
Category of unital magmas,
Category of magmas,
Category of Cartesian products of sets,
Category of sets,
Category of sets with partial maps,
Category of objects]

Hence, the role of Monoids().CartesianProducts() is solely to provide mathematical information and algorithms which are relevant to Cartesian product of monoids. For example, it specifies that the result is again a monoid, and that its multiplicative unit is the Cartesian product of the units of the underlying sets:

```
sage: C.one()
('', 1, 1)
```

Those are implemented in the nested class Monoids.CartesianProducts of Monoids(QQ). This nested class is itself a subclass of CartesianProductsCategory.

symbol = ' (+) '

class sage.categories.cartesian_product.CartesianProductsCategory (category, *args)
Bases: CovariantConstructionCategory

An abstract base class for all CartesianProducts categories.

CartesianProducts()
Return the category of (finite) Cartesian products of objects of self.

By associativity of Cartesian products, this is self (a Cartesian product of Cartesian products of A’s is a Cartesian product of A’s).

EXAMPLES:

```
sage: ModulesWithBasis(QQ).CartesianProducts().CartesianProducts()
Category of Cartesian products of vector spaces with basis over Rational Field```

5.2. Cartesian Product Functorial Construction 1355
>>> from sage.all import *
>>> ModulesWithBasis(QQ).CartesianProducts().CartesianProducts()
Category of Cartesian products of vector spaces with basis over Rational Field

base_ring()

The base ring of a Cartesian product is the base ring of the underlying category.

EXAMPLES:

```python
sage: Algebras(ZZ).CartesianProducts().base_ring()
Integer Ring
```

5.3 Tensor Product Functorial Construction

AUTHORS:

• Nicolas M. Thiéry (2008-2010): initial revision and refactorization

class sage.categories.tensor.TensorProductFunctor
Bases: CovariantFunctorialConstruction

A singleton class for the tensor functor.

This functor takes a collection of vector spaces (or modules with basis), and constructs the tensor product of those vector spaces. If this vector space is in a subcategory, say that of \textit{Algebras}(\mathbb{Q}), it is automatically endowed with its natural algebra structure, thanks to the category \textit{Algebras}(\mathbb{Q}).TensorProducts() of tensor products of algebras. For elements, it constructs the natural tensor product element in the corresponding tensor product of their parents.

The tensor functor is covariant: if \( A \) is a subcategory of \( B \), then \( A.TensorProducts() \) is a subcategory of \( B.TensorProducts() \) (see also \textit{CovariantFunctorialConstruction}). Hence, the role of \textit{Algebras}(\mathbb{Q}).TensorProducts() is solely to provide mathematical information and algorithms which are relevant to tensor product of algebras.

Those are implemented in the nested class \textit{TensorProducts} of \textit{Algebras}(\mathbb{Q}). This nested class is itself a subclass of \textit{TensorProductsCategory}.

```python
symbol = ' #'
unicode_symbol = ' ⊗'
```

class sage.categories.tensor.TensorProductsCategory(category, *args)
Bases: CovariantConstructionCategory

An abstract base class for all TensorProducts's categories

TensorProducts()

Returns the category of tensor products of objects of \textit{self}

By associativity of tensor products, this is \textit{self} (a tensor product of tensor products of \textit{Cat}'s is a tensor product of \textit{Cat}'s)

EXAMPLES:
```python
sage: ModulesWithBasis(QQ).TensorProducts().TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
```

```python
>>> from sage.all import *

from sage.all import *

>>> ModulesWithBasis(QQ).TensorProducts().TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
```

```python
base()
```

The base of a tensor product is the base (usually a ring) of the underlying category.

EXAMPLES:

```python
sage: ModulesWithBasis(ZZ).TensorProducts().base()
Integer Ring
```

```python
>>> from sage.all import *

>>> ModulesWithBasis(ZZ).TensorProducts().base()
Integer Ring
```

```python
sage.categories.tensor.tensor = The tensor functorial construction
```

The tensor product functorial construction

See `TensorProductFunctor` for more information

EXAMPLES:

```python
sage: tensor
The tensor functorial construction
```

```python
>>> from sage.all import *

>>> tensor
The tensor functorial construction
```

### 5.4 Signed Tensor Product Functorial Construction

AUTHORS:

- Travis Scrimshaw (2019-07): initial version

```python
class sage.categories.signed_tensor.SignedTensorProductFunctor
    Bases: CovariantFunctorialConstruction
```

A singleton class for the signed tensor functor.

This functor takes a collection of graded algebras (possibly with basis) and constructs the signed tensor product of those algebras. If this algebra is in a subcategory, say that of `Algebras(QQ).Graded()`, it is automatically endowed with its natural algebra structure, thanks to the category `Algebras(QQ).Graded()`. `SignedTensorProducts()` of signed tensor products of graded algebras. For elements, it constructs the natural tensor product element in the corresponding tensor product of their parents.

The signed tensor functor is covariant: if `A` is a subcategory of `B`, then `A.SignedTensorProducts()` is a subcategory of `B.SignedTensorProducts()` (see also `CovariantFunctorialConstruction`). Hence, the role of `Algebras(QQ).Graded().SignedTensorProducts()` is solely to provide mathematical information and algorithms which are relevant to signed tensor product of graded algebras.
Those are implemented in the nested class `SignedTensorProducts` of `Algebras(QQ).Graded()`. This nested class is itself a subclass of `SignedTensorProductsCategory`.

**EXAMPLES:**

```python
sage: tensor_signed
The signed tensor functorial construction
```

```python
>>> from sage.all import *
>>> tensor_signed
The signed tensor functorial construction
```

```python
symbol = '#'
unicode_symbol = '⊗'
```

class `sage.categories.signed_tensor.SignedTensorProductsCategory`(`category`, *`args`)

Bases: `CovariantConstructionCategory`

An abstract base class for all `SignedTensorProducts`'s categories.

**SignedTensorProducts()**

Return the category of signed tensor products of objects of `self`.

By associativity of signed tensor products, this is `self` (a tensor product of signed tensor products of `Cat`'s is a tensor product of `Cat`'s with the same twisting morphism)

**EXAMPLES:**

```python
sage: AlgebrasWithBasis(QQ).Graded().SignedTensorProducts().
→ SignedTensorProducts()
Category of signed tensor products of graded algebras with basis over Rational Field
```

```python
>>> from sage.all import *
>>> AlgebrasWithBasis(QQ).Graded().SignedTensorProducts().
→ SignedTensorProducts()
Category of signed tensor products of graded algebras with basis over Rational Field
```

**base()**

The base of a signed tensor product is the base (usually a ring) of the underlying category.

**EXAMPLES:**

```python
sage: AlgebrasWithBasis(ZZ).Graded().SignedTensorProducts().base()
Integer Ring
```

```python
>>> from sage.all import *
>>> AlgebrasWithBasis(ZZ).Graded().SignedTensorProducts().base()
Integer Ring
```
5.5 Dual functorial construction

AUTHORS:
- Nicolas M. Thiery (2009-2010): initial revision

```python
class sage.categories.dual.DualFunctor
    Bases: CovariantFunctorialConstruction
    A singleton class for the dual functor
    symbol = '^*

class sage.categories.dual.DualObjectsCategory(category, *args)
    Bases: CovariantConstructionCategory
```

5.6 Group algebras and beyond: the Algebra functorial construction

5.6.1 Introduction: group algebras

Let $G$ be a group and $R$ be a ring. For example:

```python
sage: G = DihedralGroup(3)
sage: R = QQ

>>> from sage.all import *
>>> G = DihedralGroup(Integer(3))
>>> R = QQ

The group algebra $A = RG$ of $G$ over $R$ is the space of formal linear combinations of elements of group with coefficients in $R$:

```python
sage: A = G.algebra(R); A
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
sage: a = A.an_element(); a
() + (1,2) + 3*(1,2,3) + 2*(1,3,2)
```

```python
>>> from sage.all import *
>>> A = G.algebra(R); A
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
>>> a = A.an_element(); a
() + (1,2) + 3*(1,2,3) + 2*(1,3,2)
```

This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication of $G$ to a multiplication on $RG$:

```python
sage: A in Algebras
True
sage: a * a
14*(()) + 5*(2,3) + 2*(1,2) + 10*(1,2,3) + 13*(1,3,2) + 5*(1,3)
```
In particular, the product of two basis elements is induced by the product of the corresponding elements of the group, and the unit of the group algebra is indexed by the unit of the group:

```
sage: (s, t) = A.algebra_generators()
sage: s*t
(1,2)
sage: A.one_basis()
()  
sage: A.one()
()
```

For the user convenience and backward compatibility, the group algebra can also be constructed with:

```
sage: GroupAlgebra(G, R)
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
```

Since Issue #18700, both constructions are strictly equivalent:

```
sage: GroupAlgebra(G, R) is G.algebra(R)
True
```

Group algebras are further endowed with a Hopf algebra structure; see below.
5.6.2 Generalizations

The above construction extends to weaker multiplicative structures than groups: magmas, semigroups, monoids. For a monoid \( S \), we obtain the monoid algebra \( RS \), which is defined exactly as above:

```sage
S = Monoids().example(); S
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')

A = S.algebra(QQ); A
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
over Rational Field

A.category()
Category of monoid algebras over Rational Field
```

This construction also extends to additive structures: magmas, semigroups, monoids, or groups:

```sage
S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')

U = S.algebra(QQ); U
Algebra of An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')
over Rational Field

U.category()
Category of monoid algebras over Rational Field
```

Despite saying “free module”, this is really an algebra, whose multiplication is induced by the addition of elements of \( S \):

```sage
U in Algebras(QQ)
True
(a,b,c,d) = S.additive_semigroup_generators()
U(a) * U(b)
B[a + b]
```

To cater uniformly for the use cases above and some others, for \( S \) a set and \( K \) a ring, we define in Sage the algebra of
"S" as the $K$-free module with basis indexed by $S$, endowed with whatever algebraic structure can be induced from that of $S$.

**Warning:** In most use cases, the result is actually an algebra, hence the name of this construction. In other cases this name is misleading:

```sage
A = Sets().example().algebra(QQ); A
Algebra of Set of prime numbers (basic implementation)
  over Rational Field
sage: A.category()
Category of set algebras over Rational Field
sage: A in Algebras(QQ)
False
```

Suggestions for a uniform, meaningful, and non misleading name are welcome!

To achieve this flexibility, the features are implemented as a *Covariant Functorial Constructions* that is essentially a hierarchy of categories each providing the relevant additional features:

```sage
A = DihedralGroup(3).algebra(QQ)
sage: A.categories()
[Category of finite group algebras over Rational Field,
 ...
 Category of group algebras over Rational Field,
 ...
 Category of monoid algebras over Rational Field,
 ...
 Category of semigroup algebras over Rational Field,
 ...
 Category of unital magma algebras over Rational Field,
 ...
 Category of magma algebras over Rational Field,
 ...
 Category of set algebras over Rational Field,
 ...
]```

```python
from sage.all import *
>>> A = DihedralGroup(Integer(3)).algebra(QQ)
```

(continues on next page)
5.6.3 Specifying the algebraic structure

Constructing the algebra of a set endowed with both an additive and a multiplicative structure is ambiguous:

```python
sage: Z3 = IntegerModRing(3)
sage: A = Z3.algebra(QQ)
Traceback (most recent call last):
...
TypeError: `S = Ring of integers modulo 3` is both
an additive and a multiplicative semigroup.
Constructing its algebra is ambiguous.
Please use, e.g., S.algebra(QQ, category=Semigroups())
```

This ambiguity can be resolved using the `category` argument of the construction:

```python
>>> from sage.all import *
>>> Z3 = IntegerModRing(Integer(3))
>>> A = Z3.algebra(QQ)
Traceback (most recent call last):
...
TypeError: `S = Ring of integers modulo 3` is both
an additive and a multiplicative semigroup.
Constructing its algebra is ambiguous.
Please use, e.g., S.algebra(QQ, category=Semigroups())
```

```python
sage: A = Z3.algebra(QQ, category=Monoids()); A
Algebra of Ring of integers modulo 3 over Rational Field
sage: A.category()
Category of finite dimensional monoid algebras over Rational Field
```

```python
sage: A = Z3.algebra(QQ, category=CommutativeAdditiveGroups()); A
Algebra of Ring of integers modulo 3 over Rational Field
sage: A.category()
Category of finite dimensional commutative additive group algebras
over Rational Field
```

```python
>>> from sage.all import *
>>> A = Z3.algebra(QQ, category=Monoids()); A
Algebra of Ring of integers modulo 3 over Rational Field
>>> A.category()
Category of finite dimensional monoid algebras over Rational Field
>>> A = Z3.algebra(QQ, category=CommutativeAdditiveGroups()); A
Algebra of Ring of integers modulo 3 over Rational Field
>>> A.category()
Category of finite dimensional commutative additive group algebras
over Rational Field
```
In general, the category argument can be used to specify which structure of $S$ shall be extended to $KS$.

5.6.4 Group algebras, continued

Let us come back to the case of a group algebra $A = RG$. It is endowed with more structure and in particular that of a Hopf algebra:

```python
sage: G = DihedralGroup(3)
sage: A = G.algebra(R); A
Algebra of Dihedral group of order 6 as a permutation group
over Rational Field
sage: A in HopfAlgebras(R).FiniteDimensional().WithBasis()
True
```

The basis elements are group-like for the coproduct: $\Delta(g) = g \otimes g$:

```python
sage: s
(1,2,3)
sage: s.coproduct()
(1,2,3) # (1,2,3)
```

The counit is the constant function 1 on the basis elements:

```python
sage: A = GroupAlgebra(DihedralGroup(6), QQ)
sage: [A.counit(g) for g in A.basis()]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The antipode is given on basis elements by $\chi(g) = g^{-1}$:

```python
sage: A = GroupAlgebra(DihedralGroup(3), QQ)
sage: s
(1,2,3)
sage: s.antipode()
(1,3,2)
```
By Maschke’s theorem, for a finite group whose cardinality does not divide the characteristic of the base field, the algebra is semisimple:

```
sage: SymmetricGroup(5).algebra(QQ) in Algebras(QQ).Semisimple()  # needs sage.combinat
True
sage: CyclicPermutationGroup(10).algebra(FiniteField(7)) in Algebras.Semisimple
True
sage: CyclicPermutationGroup(10).algebra(FiniteField(5)) in Algebras.Semisimple
False
```

5.6.5 Coercions

Let $RS$ be the algebra of some structure $S$. Then $RS$ admits the natural coercion from any other algebra $R'S'$ of some structure $S'$, as long as $R'$ coerces into $R$ and $S'$ coerces into $S$.

For example, since there is a natural inclusion from the dihedral group $D_2$ of order 4 into the symmetric group $S_4$ of order 4!, and since there is a natural map from the integers to the rationals, there is a natural map from $\mathbb{Z}[D_2]$ to $\mathbb{Q}[S_4]$:

```
sage: # needs sage.combinat
sage: A = DihedralGroup(2).algebra(ZZ)
sage: B = SymmetricGroup(4).algebra(QQ)
sage: a = A.an_element(); a
() + 2*3(3,4) + 3*1(1,2) + (1,2)(3,4)
sage: b = B.an_element(); b
() + (2,3,4) + 2*1(1,3)(2,4) + 3*1(1,4)(2,3)
sage: B(a)
() + 2*3(3,4) + 3*1(1,2) + (1,2)(3,4)
sage: B(a) * b  # a is automatically converted to an element of B
() + 2*3(3,4) + 2*2(2,3) + 2(3,4) + 3*1(1,2) + (1,2)(3,4) + (1,3,2) + 3*1(1,3,4,2) + 5*1(1,3,2,4) + 13*1(1,3,2,4) + 12*1(1,4,2,3) + 5*1(1,4)(2,3)
sage: parent(a * b)
Symmetric group algebra of order 4 over Rational Field
```

5.6. Group algebras and beyond: the Algebra functorial construction

(continues on next page)
>>> B = SymmetricGroup(Integer(4)).algebra(QQ)
>>> a = A.an_element(); a
() + 2*{(3,4)} + 3*{(1,2)} + (1,2){(3,4)}
>>> b = B.an_element(); b
() + (2,3,4) + 2*{(1,3)}{(2,4)} + 3*{(1,4)}{(2,3)}
>>> B(a)
() + 2*{(3,4)} + 3*{(1,2)} + (1,2){(3,4)}
>>> a * b
# a is automatically converted to an element of B
() + 2*{(3,4)} + 2*{(2,3)} + (2,3,4) + 3*{(1,2)} + (1,2){(3,4)} + (1,3,2) + 3*{(1,3,4,2)} + 5*{(1,3,2,4)} + 13*{(1,3,2,4)} + 12*{(1,4,2,3)} + 5*{(1,4)}{(2,3)}
>>> parent(a * b)
Symmetric group algebra of order 4 over Rational Field

There is no obvious map in the other direction, though:

```
sage: A(b)
Traceback (most recent call last):
 ...
TypeError: do not know how to make x (= () + (2,3,4) + 2*{(1,3)}{(2,4)} + 3*{(1,4)}{(2,3)}) an element of self
(=Algebra of Dihedral group of order 4 as a permutation group over Integer Ring)
```

```
>>> from sage.all import *

>>> A(b)
Traceback (most recent call last):
 ...
TypeError: do not know how to make x (= () + (2,3,4) + 2*{(1,3)}{(2,4)} + 3*{(1,4)}{(2,3)}) an element of self
(=Algebra of Dihedral group of order 4 as a permutation group over Integer Ring)
```

If \( S \) is a unital (additive) magma, then \( RS \) is a unital algebra, and thus admits a coercion from its base ring \( R \) and any ring that coerces into \( R \).

```
sage: G = DihedralGroup(2)
sage: A = G.algebra(ZZ)
sage: A(2)
2*()
```

```
>>> from sage.all import *

>>> G = DihedralGroup(Integer(2))
>>> A = G.algebra(ZZ)
>>> A(Integer(2))
2*()
```

If \( S \) is a multiplicative group, then \( RS \) admits a coercion from \( S \) and from any group which coerce into \( S \):

```
sage: g = DihedralGroup(2).gen(0); g
(3,4)
sage: A(g)
(3,4)
sage: A(2) * g
2*(3,4)
```

```
>>> from sage.all import *

>>> g = DihedralGroup(Integer(2)).gen(Integer(0)); g
```
A(g)
A(Integer(2)) * g
2*(3,4)

Note that there is an ambiguity if $S'$ is a group which coerces into both $R$ and $S$. For example) if $S$ is the additive group $(\mathbb{Z}, +)$, and $A = RS$ is its group algebra, then the integer 2 can be coerced into $A$ in two ways – via $S$, or via the base ring $R$ – and the answers are different. It that case the coercion to $R$ takes precedence. In particular, if $\mathbb{Z}$ is the ring (or group) of integers, then $\mathbb{Z}$ will coerce to any $RS$, by sending $\mathbb{Z}$ to $R$. In generic code, it is therefore recommended to always explicitly use $A$.monomial(g) to convert an element of the group into $A$.

AUTHORS:

- David Loeffler (2008-08-24): initial version
- John Palmieri (2011-07): more updates to coercion, categories, etc., group algebras constructed using CombinatorialFreeModule – see Issue #6670.
- Nicolas M. Thiéry (2010-2017), Travis Scrimshaw (2017): generalization to a covariant functorial construction for monoidal algebras, and beyond – see e.g. Issue #18700.

```python
>>> from sage.all import *
>>> from sage.categories.algebra_functor import AlgebraFunctor

>>> F = AlgebraFunctor(QQ); F
The algebra functorial construction
>>> F(DihedralGroup(Integer(3)))
Algebra of Dihedral group of order 6 as a permutation group over Rational Field

base_ring()
Return the base ring for this functor.

EXAMPLES:

```
class sage.categories.algebra_functor.AlgebrasCategory(category, *args)

Bases: CovariantConstructionCategory, Category_over_base_ring

An abstract base class for categories of monoidal algebras, groups algebras, and the like.

See also:

- Sets.ParentMethods.algebra()
- Sets.SubcategoryMethods.Algebras()
- CovariantFunctorialConstruction

INPUT:

- base_ring - a ring

EXAMPLES:

sage: C = Groups().Algebras(QQ); C
Category of group algebras over Rational Field
sage: C = Monoids().Algebras(QQ); C
Category of monoid algebras over Rational Field
sage: C._short_name()
'Algebras'
sage: latex(C)  # todo: improve that
\mathbf{Algebras}(\mathbf{Monoids})

class ParentMethods

Bases: object

coproduct_on_basis(g)

Return the coproduct of the element g of the basis.

Each basis element g is group-like. This method is used to compute the coproduct of any element.

EXAMPLES:

sage: # needs sage.combinat
sage: PF = NonDecreasingParkingFunctions(4)
sage: A = PF.algebra(ZZ); A
Algebra of Non-decreasing parking functions of size 4 over Integer Ring
sage: g = PF.an_element(); g

```python
sage: A.coproduct_on_basis(g)
B[[1, 1, 1, 1]] # B[[1, 1, 1, 1]]
```

```python
sage: a = A.an_element(); a
2*B[[1, 1, 1, 1]] + 2*B[[1, 1, 1, 2]] + 3*B[[1, 1, 1, 3]]
```

```python
sage: a.coproduct()
2*B[[1, 1, 1, 1]] # B[[1, 1, 1, 1]] +
2*B[[1, 1, 1, 2]] # B[[1, 1, 1, 2]] +
3*B[[1, 1, 1, 3]] # B[[1, 1, 1, 3]]
```

```python
>>> from sage.all import *
>>> # needs sage.combinat
>>> PF = NonDecreasingParkingFunctions(Integer(4))
>>> A = PF.algebra(ZZ); A
Algebra of Non-decreasing parking functions of size 4 over Integer Ring
>>> g = PF.an_element(); g
[1, 1, 1, 1]
>>> A.coproduct_on_basis(g)
B[[1, 1, 1, 1]] # B[[1, 1, 1, 1]]
```

```python
g = PF.an_element(); g
[1, 1, 1, 1]
```

```python
>>> a = A.an_element(); a
2*B[[1, 1, 1, 1]] + 2*B[[1, 1, 1, 2]] + 3*B[[1, 1, 1, 3]]
```

```python
>>> a.coproduct()
2*B[[1, 1, 1, 1]] # B[[1, 1, 1, 1]] +
2*B[[1, 1, 1, 2]] # B[[1, 1, 1, 2]] +
3*B[[1, 1, 1, 3]] # B[[1, 1, 1, 3]]
```

---

**class** `sage.categories.algebra_functor.GroupAlgebraFunctor(group)`

**Bases:** `ConstructionFunctor`

For a fixed group, a functor sending a commutative ring to the corresponding group algebra.

**INPUT:**

- `group` – the group associated to each group algebra under consideration

**EXAMPLES:**

```python
sage: from sage.categories.algebra_functor import GroupAlgebraFunctor
sage: F = GroupAlgebraFunctor(KleinFourGroup()); F
GroupAlgebraFunctor
sage: A = F(QQ); A
Algebra of The Klein 4 group of order 4, as a permutation group over Rational...
```

```python
>>> from sage.all import *
>>> from sage.categories.algebra_functor import GroupAlgebraFunctor
>>> F = GroupAlgebraFunctor(KleinFourGroup()); F
GroupAlgebraFunctor
>>> A = F(QQ); A
Algebra of The Klein 4 group of order 4, as a permutation group over Rational...
```

**group()**

Return the group which is associated to this functor.

**EXAMPLES:**

5.6. Group algebras and beyond: the Algebra functorial construction
5.7 Subquotient Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

```python
class sage.categories.subquotients.SubquotientsCategory(category, *args)
    Bases: RegressiveCovariantConstructionCategory
```

5.8 Quotients Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

```python
class sage.categories.quotients.QuotientsCategory(category, *args)
    Bases: RegressiveCovariantConstructionCategory
```

```
classmethod default_super_categories(category)
    Returns the default super categories of category.Quotients()
    Mathematical meaning: if A is a quotient of B in the category C, then A is also a subquotient of B in the category C.
    INPUT:
    • cls – the class QuotientsCategory
    • category – a category Cat
    OUTPUT: a (join) category
    In practice, this returns category.Subquotients(), joined together with the result of the method RegressiveCovariantConstructionCategory.default_super_categories() (that is the join of category and cat.Quotients() for each cat in the super categories of category).

EXAMPLES:

Consider category=Groups(), which has cat=Monoids() as super category. Then, a subgroup of a group G is simultaneously a subquotient of G, a group by itself, and a quotient monoid of G:
```
sage: Groups().Quotients().super_categories()
[Category of groups, Category of subquotients of monoids, Category of... quotients of semigroups]
>>> from sage.all import *
>>> Groups().Quotients().super_categories()
[Category of groups, Category of subquotients of monoids, Category of...
→quotients of semigroups]

Mind the last item above: there is indeed currently nothing implemented about quotient monoids.

This resulted from the following call:

```
sage: sage.categories.quotients.QuotientsCategory.default_super_categories(Groups())
Join of Category of groups and Category of subquotients of monoids and...
→Category of quotients of semigroups
```

```
>>> from sage.all import *
>>> sage.categories.quotients.QuotientsCategory.default_super_categories(Groups())
Join of Category of groups and Category of subquotients of monoids and...
→Category of quotients of semigroups
```

5.9 Subobjects Functorial Construction

AUTHORS:

• Nicolas M. Thiery (2010): initial revision

class sage.categories.subobjects.SubobjectsCategory(category, *args)
Bases: RegressiveCovariantConstructionCategory
classmethod default_super_categories(category)

Returns the default super categories of category.Subobjects()

Mathematical meaning: if \( A \) is a subobject of \( B \) in the category \( C \), then \( A \) is also a subquotient of \( B \) in the category \( C \).

INPUT:

• \( \text{cls} \) – the class SubobjectsCategory
• \( \text{category} \) – a category \( \text{Cat} \)

OUTPUT: a (join) category

In practice, this returns category.Subquotients(), joined together with the result of the method RegressiveCovariantConstructionCategory.default_super_categories() (that is the join of category and \(\text{cat}.\text{Subobjects}()\) for each \(\text{cat}\) in the super categories of category).

EXAMPLES:

Consider \(\text{category} = \text{Groups}()\), which has \(\text{cat} = \text{Monoids}()\) as super category. Then, a subgroup of a group \( G \) is simultaneously a subquotient of \( G \), a group by itself, and a submonoid of \( G \):

```
sage: Groups().Subobjects().super_categories()
[Category of groups, Category of subquotients of monoids, Category of...
→subobjects of sets]
```

5.9. Subobjects Functorial Construction 1371
Mind the last item above: there is indeed currently nothing implemented about submonoids.

This resulted from the following call:

```python
>>> from sage.all import *
>>> Groups().Subobjects().super_categories()
[Category of groups, Category of subquotients of monoids, Category of subobjects of sets]
```

5.10 Isomorphic Objects Functorial Construction

AUTHORS:
- Nicolas M. Thiery (2010): initial revision

```python
class sage.categories.isomorphic_objects.IsomorphicObjectsCategory(category, *args)
    Bases: RegressiveCovariantConstructionCategory
classmethod default_super_categories(category)
    Returns the default super categories of category.IsomorphicObjects()
    Mathematical meaning: if \(A\) is the image of \(B\) by an isomorphism in the category \(C\),
    then \(A\) is both a subobject of \(B\) and a quotient of \(B\) in the category \(C\).
    INPUT:
    • cls -- the class IsomorphicObjectsCategory
    • category -- a category \(Cat\)
    OUTPUT: a (join) category
    In practice, this returns category.Subobjects() and category.Quotients(),
    joined together with the result of the method RegressiveCovariantConstructionCategory.
    default_super_categories() (that is the join of category and cat.
    IsomorphicObjects() for each cat in the super categories of category).

EXAMPLES:
Consider category=Groups(), which has cat=Monoids() as super category. Then, the image of a
```
```
```python
sage: Groups().IsomorphicObjects().super_categories()
[Category of groups, Category of subquotients of monoids, Category of subobjects of sets]
```
```
Category of quotients of semigroups,
Category of isomorphic objects of sets]

Mind the last item above: there is indeed currently nothing implemented about isomorphic objects of monoids.

This resulted from the following call:

```
sage: sage.categories.isomorphic_objects.IsomorphicObjectsCategory.default_
    →super_categories(Groups())
Join of Category of groups and
Category of subquotients of monoids and
Category of quotients of semigroups and
Category of isomorphic objects of sets
```

5.11 Homset categories

```
class sage.categories.homsets.Homsets
    Bases: Category_singleton

The category of all homsets.

EXAMPLES:

sage: from sage.categories.homsets import Homsets
sage: Homsets()
Category of homsets
```

This is a subcategory of Sets():

```
sage: Homsets().super_categories()
[Category of sets]
```
By this, we assume that all homsets implemented in Sage are sets, or equivalently that we only implement locally small categories. See Wikipedia article Category_(mathematics).

Issue #17364: every homset category shall be a subcategory of the category of all homsets:

```python
sage: Schemes().Homsets().is_subcategory(Homsets())
True
sage: AdditiveMagmas().Homsets().is_subcategory(Homsets())
True
sage: AdditiveMagmas().AdditiveUnital().Homsets().is_subcategory(Homsets())
True
```

This is tested in HomsetsCategory._test_homsets_category().

```python
>>> from sage.all import *
>>> from sage.categories.homsets import Homsets
>>> Homsets().Endset()
Category of endsets
```

```python
>>> from sage.all import *
>>> from sage.categories.homsets import Homsets
>>> Homsets().Endset()
Category of endsets
```

```python
>>> from sage.all import *
>>> from sage.categories.homsets import Homsets
>>> Homsets().Endset()
Category of endsets
```

```python
>>> from sage.all import *
>>> from sage.categories.homsets import Homsets
>>> Homsets().Endset()
Category of endsets
```

```python
class ParentMethods
    Bases: object

    is_endomorphism_set()
        Return True as self is in the category of Endsets.

    EXAMPLES:
        sage: P.<t> = ZZ[]
        sage: E = End(P)
        sage: E.is_endomorphism_set()
        True
```
extra_super_categories()
Implement the fact that endsets are monoids.

See also:
CategoryWithAxiom.extra_super_categories()

EXAMPLES:

```python
sage: from sage.categories.homsets import Homsets
sage: Homsets().Endset().extra_super_categories()
[Category of monoids]
```

class ParentMethods
Bases: object

is_endomorphism_set()
Return True if the domain and codomain of self are the same object.

EXAMPLES:

```python
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().is_endomorphism_set()
False
sage: g = P.hom([2*t])
sage: g.parent().is_endomorphism_set()
True
```

class SubcategoryMethods
Bases: object

Endset()
Return the subcategory of the homsets of self that are endomorphism sets.

EXAMPLES:
super_categories()

Return the super categories of self.

EXAMPLES:

```python
sage: from sage.categories.homsets import Homsets
sage: Homsets()
Category of homsets
```

class sage.categories.homsets.HomsetsCategory(category, *args)

Bases: FunctorialConstructionCategory

base()

If this homsets category is subcategory of a category with a base, return that base.

Todo: Is this really useful?

EXAMPLES:

```python
sage: ModulesWithBasis(ZZ).Homsets().base()
Integer Ring
```

classmethod default_super_categories(category)

Return the default super categories of category.Homsets().

INPUT:

- `cls` – the category class for the functor $F$
- `category` – a category $\mathcal{C}$

OUTPUT: a category
As for the other functorial constructions, if `category` implements a nested `Homsets` class, this method is used in combination with `category.Homsets().extra_super_categories()` to compute the super categories of `category.Homsets()`.

EXAMPLES:

If `category` has one or more full super categories, then the join of their respective homsets category is returned. In this example, this join consists of a single category:

```
sage: from sage.categories.homsets import HomsetsCategory
sage: from sage.categories.additive_groups import AdditiveGroups
sage: C = AdditiveGroups()
sage: C.full_super_categories()
[Category of additive inverse additive unital additive magmas,
 Category of additive monoids]
sage: H = HomsetsCategory.default_super_categories(C); H
Category of homsets of additive monoids
sage: type(H)
<class 'sage.categories.additive_monoids.AdditiveMonoids.Homsets_with_category
˓→'>
```

and, given that nothing specific is currently implemented for homsets of additive groups, `H` is directly the category thereof:

```
sage: C.Homsets()
Category of homsets of additive monoids
```

Similarly for rings: a ring homset is just a homset of unital magmas and additive magmas:

```
sage: Rings().Homsets()
Category of homsets of unital magmas and additive unital additive magmas
```

Otherwise, if `category` implements a nested class `Homsets`, this method returns the category of all homsets:
```python
>>> from sage.all import *
>>> AdditiveMagmas().Homsets().super_categories()
[Category of additive magmas, Category of homsets]

the other coming from category.Homsets().extra_super_categories():

```python
>>> from sage.all import *
>>> AdditiveMagmas().Homsets().extra_super_categories()
[Category of additive magmas]
```

Finally, as a last resort, this method returns a stub category modelling the homsets of this category:

```python
sage: hasattr(Posets, "Homsets")
False
sage: H = HomsetsCategory.default_super_categories(Posets()); H
Category of homsets of posets
sage: type(H)
<class 'sage.categories.posets.PosetsOf_with_category'>

```
Default class for homsets of a category.

This is used when a category $C$ defines some additional structure but not a homset category of its own. Indeed, unlike for covariant functorial constructions, we cannot represent the homset category of $C$ by just the join of the homset categories of its super categories.

EXAMPLES:

```
sage: C = (Magmas() & Posets()).Homsets(); C
category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
```

```
>>> from sage.all import *

```

```
sage: C = (Magmas() & Posets()).Homsets(); C
category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
```

```
super_categories()

Return the super categories of self.

A stub homset category admits a single super category, namely the category of all homsets.

EXAMPLES:

```
sage: C = (Magmas() & Posets()).Homsets(); C
category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
sage: C.super_categories()
[Category of homsets]
```

```
>>> from sage.all import *

```

```
sage: C = (Magmas() & Posets()).Homsets(); C
category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
>>> C.super_categories()
[Category of homsets]
```

### 5.12 Realizations Covariant Functorial Construction

See also:

- `Sets().WithRealizations` for an introduction to realizations and with realizations.
- `sage.categories.covariant_functorial_construction` for an introduction to covariant functorial constructions.
- `sage.categories.examples.with_realizations` for an example.

```python
class sage.categories.realizations.Category_realization_of_parent(parent_with_realization):

    Bases: Category_over_base, BindableClass

    An abstract base class for categories of all realizations of a given parent
```

```python
72x125
5.12. Realizations Covariant Functorial Construction 1379
```

```python
72x125
```
INPUT:

- parent_with_realization – a parent

See also:
Sets().WithRealizations

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A
# needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
```

```
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
# needs sage.modules
The subset algebra of {1, 2, 3} over Rational Field
```

The role of this base class is to implement some technical goodies, like the binding A.Realizations() when a subclass Realizations is implemented as a nested class in A (see the code of the example):

```
sage: C = A.Realizations(); C
# needs sage.modules
Category of realizations of
The subset algebra of {1, 2, 3} over Rational Field
```

```
>>> from sage.all import *
>>> C = A.Realizations(); C
# needs sage.modules
Category of realizations of
The subset algebra of {1, 2, 3} over Rational Field
```

as well as the name for that category.

```
sage.categories.realizations.Realizations(self)
Return the category of realizations of the parent self or of objects of the category self
```

INPUT:

- self – a parent or a concrete category

Note: this function is actually inserted as a method in the class Category (see Realizations()). It is defined here for code locality reasons.

EXAMPLES:

The category of realizations of some algebra:

```
sage: Algebras(QQ).Realizations()
Join of Category of algebras over Rational Field
and Category of realizations of unital magmas
```

```
>>> from sage.all import *
>>> Algebras(QQ).Realizations()
Join of Category of algebras over Rational Field
and Category of realizations of unital magmas
```
The category of realizations of a given algebra:

```python
sage: A = Sets().WithRealizations().example(); A
↓ needs sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.Realizations()
↓ needs sage.modules
Category of realizations of
The subset algebra of \{1, 2, 3\} over Rational Field
sage: C = GradedHopfAlgebrasWithBasis(QQ).Realizations(); C
Join of Category of graded Hopf algebras with basis over Rational Field
and Category of realizations of Hopf algebras over Rational Field
sage: C.super_categories()
[Category of graded Hopf algebras with basis over Rational Field,
Category of realizations of Hopf algebras over Rational Field]
sage: TestSuite(C).run()
```

See also:

- `Sets().WithRealizations`
- `ClasscallMetaclass`

**Todo:** Add an optional argument to allow for:

```python
sage: Realizations(A, category=Blahs()) # todo: not implemented
```

```python
>>> from sage.all import *
>>> Realizations(A, category=Blahs()) # todo: not implemented
```

```python
class sage.categories.realizations.RealizationsCategory({category, *args)
    Bases: RegressiveCovariantFunctorialConstructionCategory
    An abstract base class for all categories of realizations category
```
Relization are implemented as \texttt{RegressiveCovariantConstructionCategory}. See there for the documentation of how the various bindings such as \texttt{Sets().Realizations()} and \texttt{P.Realizations()}, where \texttt{P} is a parent, work.

See also:

\texttt{Sets().WithRealizations}

### 5.13 With Realizations Covariant Functorial Construction

See also:

- \texttt{Sets().WithRealizations} for an introduction to \textit{realizations} and \textit{with realizations}.
- \texttt{sage.categories.covariant_functorial_construction} for an introduction to covariant functorial constructions.

```python
sage.categories.with_realizations.WithRealizations(self)
```

Return the category of parents in \texttt{self} endowed with multiple realizations.

**INPUT:**

- \texttt{self} – a category

See also:

- The documentation and code (\texttt{sage.categories.examples.with_realizations}) of \texttt{Sets().WithRealizations().example()} for more on how to use and implement a parent with several realizations.
- Various use cases:
  - \texttt{SymmetricFunctions}
  - \texttt{QuasiSymmetricFunctions}
  - \texttt{NonCommutativeSymmetricFunctions}
  - \texttt{SymmetricFunctionsNonCommutingVariables}
  - \texttt{DescentAlgebra}
  - \texttt{algebras.Moebius}
  - \texttt{IwahoriHeckeAlgebra}
  - \texttt{ExtendedAffineWeylGroup}
- The \texttt{Implementing Algebraic Structures} thematic tutorial.
- \texttt{sage.categories.realizations}

**Note:** this \textit{function} is actually inserted as a \textit{method} in the class \texttt{Category} (see \texttt{WithRealizations()}). It is defined here for code locality reasons.

**EXAMPLES:**

```python
sage: Sets().WithRealizations()
Category of sets with realizations
```
Parent with realizations

Let us now explain the concept of realizations. A parent with realizations is a facade parent (see \texttt{Sets.Facade}) admitting multiple concrete realizations where its elements are represented. Consider for example an algebra $A$ which admits several natural bases:

```python
from sage.all import *
Sets().WithRealizations()
Category of sets with realizations
```

```
Sets().WithRealizations().example()
Category of sets with realizations
```

```
A = Sets().WithRealizations().example(); A
Ring Morphism:
From: The subset algebra of {1, 2, 3} over Rational Field
In the Out basis
To: The subset algebra of {1, 2, 3} over Rational Field
In the In basis
Generic morphism:
From: The subset algebra of {1, 2, 3} over Rational Field
In the Out basis
To: The subset algebra of {1, 2, 3} over Rational Field
In the In basis
sage: A = Sets().WithRealizations().example(); A
Ring Morphism:
From: The subset algebra of {1, 2, 3} over Rational Field
In the Out basis
To: The subset algebra of {1, 2, 3} over Rational Field
In the In basis
Generic morphism:
From: The subset algebra of {1, 2, 3} over Rational Field
In the Out basis
To: The subset algebra of {1, 2, 3} over Rational Field
In the In basis
sage: A = Sets().WithRealizations().example(); A
```

For each such basis $B$ one implements a parent $P_B$ which realizes $A$ with its elements represented by expanding them on the basis $B$:

```python
# needs sage.modules
sage: A = Sets().WithRealizations().example(); A
```

```python
A.F()
The subset algebra of (1, 2, 3) over Rational Field in the Fundamental basis
sage: A.Out()
The subset algebra of (1, 2, 3) over Rational Field in the Out basis
sage: A.In()
The subset algebra of (1, 2, 3) over Rational Field in the In basis
sage: A.an_element()
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

```python
# needs sage.modules
sage: A = Sets().WithRealizations().example(); A
```

```python
A.F()
The subset algebra of (1, 2, 3) over Rational Field in the Fundamental basis
sage: A.Out()
The subset algebra of (1, 2, 3) over Rational Field in the Out basis
sage: A.In()
The subset algebra of (1, 2, 3) over Rational Field in the In basis
sage: A.an_element()
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

If $B$ and $B'$ are two bases, then the change of basis from $B$ to $B'$ is implemented by a canonical coercion between $P_B$ and $P_{B'}$:

```python
# needs sage.modules
sage: F = A.F(); In = A.In(); Out = A.Out()
sage: i = In.an_element(); i
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
sage: F(i)
7*F[{}] + 3*F[{1}] + 4*F[{2}] + F[{1, 2}]
sage: F.coerce_map_from(Out)
Generic morphism:
From: The subset algebra of (1, 2, 3) over Rational Field in the Out basis
```

(continues on next page)
To: The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> F = A.F(); In = A.In(); Out = A.Out()
>>> i = In.an_element();
In[\{\}\] + 2*In[\{1\}] + 3*In[\{2\}] + In[\{1, 2\}]
>>> F(i)
7*F[\{\}\] + 3*F[\{1\}] + 4*F[\{2\}] + F[\{1, 2\}]
>>> F.coerce_map_from(Out)
Generic morphism:
   From: The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis
   To: The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
allowing for mixed arithmetic:
```
sage: (1 + Out.from_set(1)) * In.from_set(2,3)
```

In our example, there are three realizations:

```python
sage: A.realizations()
[The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis,
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis,
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis]
```

Instead of manually defining the shorthands \(F, In,\) and \(Out\), as above one can just do:

```python
sage: A.inject_shorthands()
```

Defining \(F\) as shorthand for
   The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
Defining \(In\) as shorthand for
   The subset algebra of \{1, 2, 3\} over Rational Field in the In basis
Defining \(Out\) as shorthand for
   The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis
Rationale

Besides some goodies described below, the role of $A$ is threefold:

- To provide, as illustrated above, a single entry point for the algebra as a whole: documentation, access to its properties and different realizations, etc.

- To provide a natural location for the initialization of the bases and the coercions between, and other methods that are common to all bases.

- To let other objects refer to $A$ while allowing elements to be represented in any of the realizations.

We now illustrate this second point by defining the polynomial ring with coefficients in $A$:

```sage
sage: P = A['x']; P
Univariate Polynomial Ring in x over The subset algebra of {1, 2, 3} over Rational Field
sage: x = P.gen()
```

In the following examples, the coefficients turn out to be all represented in the $F$ basis:

```sage
sage: P.one()
F[{}]
sage: (P.an_element() + 1)^2
F[{}]*x^2 + 2*F[{}]*x + F[{}]
```

However we can create a polynomial with mixed coefficients, and compute with it:
Note how each coefficient involves a single basis which need not be that of the other coefficients. Which basis is used depends on how coercion happened during mixed arithmetic and needs not be deterministic.

One can easily coerce all coefficient to a given basis with:

```
sage: p.map_coefficients(In)
```

```
(−4*I[{}]) + 2*I[{1}] + 4*I[{2}] + 2*I[{3}] 
− 2*I[{1, 2}] − I[{1, 3}] − 2*I[{2, 3}] + I[{1, 2, 3}])*x²
+ I[{1}]*x + I[{}]
```

Alas, the natural notation for constructing such polynomials does not yet work:

```
sage: I[{1}] * x
```

```
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'The subset algebra of {1, 2, 3} over Rational Field in the In basis'
and 'Univariate Polynomial Ring in x over
The subset algebra of {1, 2, 3} over Rational Field'
```

(continues on next page)
The category of realizations of $A$

The set of all realizations of $A$, together with the coercion morphisms is a category (whose class inherits from `Category_realization_of_parent`):

```python
sage: A.Realizations()  # needs sage.modules
Category of realizations of The subset algebra of {1, 2, 3} over Rational Field
```

The various parent realizing $A$ belong to this category:

```python
sage: A.F() in A.Realizations()  # needs sage.modules
True
```

$A$ itself is in the category of algebras with realizations:

```python
sage: A in Algebras(QQ).WithRealizations()  # needs sage.modules
True
```

The (mostly technical) `WithRealizations` categories are the analogs of the `*WithSeveralBases` categories in MuPAD-Combinat. They provide support tools for handling the different realizations and the morphisms between them.

Typically, `VectorSpaces(QQ).FiniteDimensional().WithRealizations()` will eventually be in charge, whenever a coercion $\phi : A \rightarrow B$ is registered, to register $\phi^{-1}$ as coercion $B \rightarrow A$ if there is none defined yet. To achieve this, `FiniteDimensionalVectorSpaces` would provide a nested class `WithRealizations` implementing the appropriate logic.
WithRealizations is a **recessive covariant functorial construction**. On our example, this simply means that $A$ is automatically in the category of rings with realizations (covariance):

```
sage: A in Rings().WithRealizations()  #...
needs sage.modules
True
```

and in the category of algebras (recessiveness):

```
sage: A in Algebras(QQ)  #...
needs sage.modules
True
```

Note: For $C$ a category, $C$.WithRealizations() in fact calls sage.categories.with_realizations.WithRealizations($C$). The later is responsible for building the hierarchy of the categories with realizations in parallel to that of their base categories, optimizing away those categories that do not provide a WithRealizations nested class. See sage.categories.covariant_functorial_construction for the technical details.

Note: Design question: currently WithRealizations is a recessive construction. That is self.WithRealizations() is a subcategory of self by default:

```
sage: Algebras(QQ).WithRealizations().super_categories()  
[Category of algebras over Rational Field,  
Category of monoids with realizations,  
Category of additive unital additive magmas with realizations]
```

```
>>> from sage.all import *  
>>> Algebras(QQ).WithRealizations().super_categories()  
[Category of algebras over Rational Field,  
Category of monoids with realizations,  
Category of additive unital additive magmas with realizations]
```

Is this always desirable? For example, AlgebrasWithBasis(QQ).WithRealizations() should certainly be a subcategory of Algebras(QQ), but not of AlgebrasWithBasis(QQ). This is because AlgebrasWithBasis(QQ) is specifying something about the concrete realization.

```python
class sage.categories.with_realizations.WithRealizationsCategory(category, *args):
    Bases: RegressiveCovariantConstructionCategory

    An abstract base class for all categories of parents with multiple realizations.
```
See also:

Sets().WithRealizations

The role of this base class is to implement some technical goodies, such as the name for that category.
EXAMPLES OF PARENTS USING CATEGORIES

6.1 Examples of algebras with basis

sage.categories.examples.algebras_with_basis.Example
alias of FreeAlgebra
class sage.categories.examples.algebras_with_basis.FreeAlgebra(R, alphabet=('a', 'b', 'c'))

Bases: CombinatorialFreeModule
An example of an algebra with basis: the free algebra
This class illustrates a minimal implementation of an algebra with basis.
algebra_generators()
Return the generators of this algebra, as per algebra_generators()

EXAMPLES:

```python
sage: A = AlgebrasWithBasis(QQ).example(); A  # needs sage.modules
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over Rational Field
sage: A.algebra_generators()  # needs sage.modules
Family (B[word: a], B[word: b], B[word: c])
```

one_basis()
Returns the empty word, which index the one of this algebra, as per AlgebrasWithBasis.
ParentMethods.one_basis().

EXAMPLES:

```python
>>> from sage.all import *
>>> A = AlgebrasWithBasis(QQ).example(); A  # needs sage.modules
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over Rational Field
>>> A.algebra_generators()  # needs sage.modules
Family (B[word: a], B[word: b], B[word: c])
```

```python
sage: A = AlgebrasWithBasis(QQ).example() # needs sage.modules.sage: A.one_basis() # needs sage.modules
sage: A.one() # needs sage.modules
B[word: ]
```
product_on_basis(\(w_1, w_2\))

Product of basis elements, as per `AlgebrasWithBasis.ParentMethods.product_on_basis()`.

EXAMPLES:

```python
sage: # needs sage.modules
sage: A = AlgebrasWithBasis(QQ).example()

sage: A.product_on_basis(AlgebrasWithBasis(QQ).Example(), AlgebrasWithBasis(QQ).Example())

\[B[\text{word: abbc}] - 2*B[\text{word: abc}] + B[\text{word: ac}]\]
```

6.2 Examples of commutative additive monoids

sage.categories.examples.commutative_additive_monoids.Example

alias of `FreeCommutativeAdditiveMonoid`

class `sage.categories.examples.commutative_additive_monoids.FreeCommutativeAdditiveMonoid`

An example of a commutative additive monoid: the free commutative monoid generated by (\(a, b, c, d\))

This class illustrates a minimal implementation of a commutative monoid.

EXAMPLES:

```python
sage: S = CommutativeAdditiveMonoids().example(); S

An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')

sage: S.category()

Category of commutative additive monoids
```

```python
>>> from sage.all import *

>>> S = CommutativeAdditiveMonoids().example(); S

An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')
```
This is the free semigroup generated by:

```python
sage: S.additive_semigroup_generators()
Family (a, b, c, d)
```

with product rule given by $a \times b = a$ for all $a, b$:

```python
sage: (a,b,c,d) = S.additive_semigroup_generators()
```

We conclude by running systematic tests on this commutative monoid:

```python
sage: TestSuite(S).run(backend = 'True')
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
    running ._test_category() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_nonzero_equal() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass
```

(continues on next page)
class Element (parent, iterable)
    Bases: Element

zero()
    Returns the zero of this additive monoid, as per CommutativeAdditiveMonoids.
    ParentMethods.zero().

    EXAMPLES:

    sage: M = CommutativeAdditiveMonoids().example(); M
    An example of a commutative monoid: the free commutative monoid generated by (˓→'a', 'b', 'c', 'd')
    sage: M.zero()
    0

    >>> from sage.all import *
    >>> M = CommutativeAdditiveMonoids().example(); M
    An example of a commutative monoid: the free commutative monoid generated by (˓→'a', 'b', 'c', 'd')
    >>> M.zero()
    0

### 6.3 Examples of commutative additive semigroups

sage.categories.examples.commutative_additive_semigroups.Example
    alias of FreeCommutativeAdditiveSemigroup

class sage.categories.examples.commutative_additive_semigroups.FreeCommutativeAdditiveSemigroup
Bases: `UniqueRepresentation, Parent`

An example of a commutative additive monoid: the free commutative monoid

This class illustrates a minimal implementation of a commutative additive monoid.

**EXAMPLES:**

```python
sage: S = CommutativeAdditiveSemigroups().example(); S
An example of a commutative semigroup: the free commutative semigroup generated by ('a', 'b', 'c', 'd')

sage: S.category()
Category of commutative additive semigroups
```

This is the free semigroup generated by:

```python
sage: S.additive_semigroup_generators()
Family (a, b, c, d)
```

with product rule given by $a \times b = a$ for all $a, b$:

```python
sage: (a,b,c,d) = S.additive_semigroup_generators()
```

We conclude by running systematic tests on this commutative monoid:

```python
sage: TestSuite(S).run( verbose = True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
    Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
    pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
```

(continues on next page)
class Element (parent, iterable)

    Bases: ElementWrapper

    EXAMPLES:

    sage: F = CommutativeAdditiveSemigroups().example()
    sage: x = F.element_class(F, [('a',4), ('b', 0), ('a', 2), ('c', 1), ('d', 5)])
    sage: x
    2*a + c + 5*d
    sage: x.value
    {'a': 2, 'b': 0, 'c': 1, 'd': 5}
    sage: x.parent()
    An example of a commutative semigroup: the free commutative semigroup
    generated by ('a', 'b', 'c', 'd')

    >>> from sage.all import *
    >>> F = CommutativeAdditiveSemigroups().example()
    >>> x = F.element_class(F, [('a',Integer(4)), ('b', Integer(0)), ('a',
    Integer(2)), ('c', Integer(1)), ('d', Integer(5)))
    >>> x
    2*a + c + 5*d
    >>> x.value
an_element()
Returns an element of the semigroup.

EXAMPLES:

```sage
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.an_element()
a + 2*b + 3*c + 4*d
```

summation(x, y)
Returns the product of \( x \) and \( y \) in the semigroup, as per \texttt{CommutativeAdditiveSemigroups.ParentMethods.summation()}.

EXAMPLES:

```sage
sage: F = CommutativeAdditiveSemigroups().example()
sage: (a,b,c,d) = F.additive_semigroup_generators()
sage: F.summation(a,b)
a + b
sage: (a+b) + (a+c)
2*a + b + c
```
6.4 Examples of Coxeter groups

6.5 Example of a crystal

class sage.categories.examples.crystals.HighestWeightCrystalOfTypeA(n=3):
    Bases: UniqueRepresentation, Parent

    An example of a crystal: the highest weight crystal of type $A_n$ of highest weight $\omega_1$.

    The purpose of this class is to provide a minimal template for implementing crystals. See CrystalOfLetters for a full featured and optimized implementation.

    EXAMPLES:

    sage: C = Crystals().example()
    sage: C
    Highest weight crystal of type A_3 of highest weight omega_1
    sage: C.category()
    Category of classical crystals

    The elements of this crystal are in the set $\{1,\ldots,n + 1\}$:

    sage: C.list()
    [1, 2, 3, 4]
    sage: C.module_generators[0]
    1

    The crystal operators themselves correspond to the elementary transpositions:

    sage: b = C.module_generators[0]
    sage: b.f(1)
    2

(continues on next page)
sage: b.f(1).e(1) == b
True

>>> from sage.all import *
>>> b = C.module_generators[Integer(0)]
>>> b.f(Integer(1))
2
>>> b.f(Integer(1)).e(Integer(1)) == b
True

Only the following basic operations are implemented:

- cartan_type()
- module_generators
- Element.e()
- Element.f()

All the other usual crystal operations are inherited from the categories; for example:

sage: C.cardinality()
4

>>> from sage.all import *
>>> C.cardinality()
4

class Element

Bases: ElementWrapper

\(e(i)\)

Returns the action of \(e_i\) on \(self\).

EXAMPLES:

sage: C = Crystals().example(4)
sage: [(c,i,c.e(i)) for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1], [3, 2, 2], [4, 3, 3], [5, 4, 4]]

\(f(i)\)

Returns the action of \(f_i\) on \(self\).

EXAMPLES:

sage: C = Crystals().example(4)
sage: [(c,i,c.f(i)) for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2], [2, 2, 3], [3, 3, 4], [4, 4, 5]]
```python
>>> from sage.all import *
>>> C = Crystals().example(Integer(4))
>>> [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[[1, 1, 2], [2, 2, 3], [3, 3, 4], [4, 4, 5]]

class sage.categories.examples.crystals.NaiveCrystal
    Bases: UniqueRepresentation, Parent

This is an example of a “crystal” which does not come from any kind of representation, designed primarily to test the Stembridge local rules with. The crystal has vertices labeled 0 through 5, with 0 the highest weight.

The code here could also possibly be generalized to create a class that automatically builds a crystal from an edge-colored digraph, if someone feels adventurous.

Currently, only the methods highest_weight_vector(), e(), and f() are guaranteed to work.

EXAMPLES:

sage: C = Crystals().example(choice='naive')
sage: C.highest_weight_vector()
0

class Element
    Bases: ElementWrapper

    e(i)

    Returns the action of $e_i$ on self.

    EXAMPLES:

sage: C = Crystals().example(choice='naive')
sage: [[c,i,c.e(i)] for i in C.index_set() for c in [C(j) for j in range(0,5)] if c.e(i) is not None]
[[[1, 1, 0], [2, 1, 1], [3, 1, 2], [5, 1, 3], [4, 2, 0], [5, 2, 4]]

f(i)

    Returns the action of $f_i$ on self.

    EXAMPLES:

sage: C = Crystals().example(choice='naive')
sage: [[c,i,c.f(i)] for i in C.index_set() for c in [C(j) for j in range(0,5)] if c.f(i) is not None]
[[[0, 1, 1], [1, 1, 2], [2, 1, 3], [3, 1, 5], [0, 2, 4], [4, 2, 5]]
```
6.6 Examples of CW complexes

sage.categories.examples.cw_complexes.Example
alias of Surface

class sage.categories.examples.cw_complexes.Surface(bdy=(1, 2, 1, 2))

   Bases: UniqueRepresentation, Parent

   An example of a CW complex: a (2-dimensional) surface.
   This class illustrates a minimal implementation of a CW complex.

   EXAMPLES:

   sage: from sage.categories.cw_complexes import CWComplexes
   sage: X = CWComplexes().example(); X
   An example of a CW complex: the surface given by the boundary map (1, 2, 1, 2)
   sage: X.category()  
   Category of finite finite dimensional CW complexes

   We conclude by running systematic tests on this manifold:

   sage: TestSuite(X).run() 

   >>> from sage.all import *
   >>> from sage.categories.cw_complexes import CWComplexes
   >>> X = CWComplexes().example(); X
   An example of a CW complex: the surface given by the boundary map (1, 2, 1, 2)
   >>> X.category()  
   Category of finite finite dimensional CW complexes

   class Element (parent, dim, name)

   Bases: Element

   A cell in a CW complex.

   dimension()
       Return the dimension of self.

   EXAMPLES:

   sage: from sage.categories.cw_complexes import CWComplexes
   sage: X = CWComplexes().example()
   sage: f = X.an_element()
```python
sage: f = X.an_element()
>>> f.dimension()
2
```

### an_element()

Return an element of the CW complex, as per `Sets.ParentMethods.an_element()`.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.an_element()
2-cell f
```

### cells()

Return the cells of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: C = X.cells()
sage: sorted((d, C[d]) for d in C.keys())
[(0, (0-cell v,)),
 (1, (0-cell e1, 0-cell e2)),
 (2, (2-cell f,))]
```
6.7 Example of facade set

class sage.categories.examples.facade_sets.IntegersCompletion
    Bases: UniqueRepresentation, Parent

    An example of a facade parent: the set of integers completed with $+\infty$ and $-\infty$

    This class illustrates a minimal implementation of a facade parent that models the union of several other parents.

    EXAMPLES:

    sage: S = Sets().Facade().example("union"); S
    An example of a facade set: the integers completed by $\pm\infty$

>>> from sage.all import *

>>> S = Sets().Facade().example("union"); S
An example of a facade set: the integers completed by $\pm\infty$

class sage.categories.examples.facade_sets.PositiveIntegerMonoid
    Bases: UniqueRepresentation, Parent

    An example of a facade parent: the positive integers viewed as a multiplicative monoid

    This class illustrates a minimal implementation of a facade parent which models a subset of a set.

    EXAMPLES:

    sage: S = Sets().Facade().example(); S
    An example of facade set: the monoid of positive integers

>>> from sage.all import *

>>> S = Sets().Facade().example(); S
An example of facade set: the monoid of positive integers

6.8 Examples of finite Coxeter groups

class sage.categories.examples.finite_coxeter_groups.DihedralGroup(n=5)
    Bases: UniqueRepresentation, Parent

    An example of finite Coxeter group: the $n$-th dihedral group of order $2n$.

    The purpose of this class is to provide a minimal template for implementing finite Coxeter groups. See DihedralGroup for a full featured and optimized implementation.

    EXAMPLES:

    sage: G = FiniteCoxeterGroups().example()

>>> from sage.all import *

>>> G = FiniteCoxeterGroups().example()

This group is generated by two simple reflections $s_1$ and $s_2$ subject to the relation $(s_1s_2)^n = 1$:
An element is represented by its reduced word (a tuple of elements of \textit{self.index\_set()}):

\begin{verbatim}
sage: G.an_element()
(1, 2)
sage: list(G)
[(),
 (1,),
 (2,),
 (1, 2),
 (2, 1),
 (1, 2, 1),
 (2, 1, 2),
 (1, 2, 1, 2),
 (2, 2, 1, 1),
 (1, 2, 2, 1)]
\end{verbatim}

This reduced word is unique, except for the longest element where the chosen reduced word is \((1, 2, 1, 2, \ldots)\):

\begin{verbatim}
sage: G.long_element()
(1, 2, 1, 2, 1)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> G.an_element()
(1, 2)

>>> list(G)
[(),
 (1,),
 (2,),
 (1, 2),
 (2, 1),
 (1, 2, 1),
 (2, 1, 2),
 (1, 2, 1, 2),
 (2, 1, 2, 1),
 (1, 2, 1, 2, 1)]
\end{verbatim}
class Element

Bases: ElementWrapper

apply_simple_reflection_right(i)

Implements CoxeterGroups.ElementMethods.apply_simple_reflection().

EXAMPLES:

```python
sage: D5 = CoxeterGroups().example(5)
sage: [i^2 for i in D5]  # indirect doctest
[(), (), (), (1, 2, 1, 2), (2, 1, 2, 1), (), (), (2, 1), (1, 2), ()]
sage: [i^5 for i in D5]  # indirect doctest
[(), (1,), (2,), (), (), (1, 2, 1), (2, 1, 2), (), (), (1, 2, 1, 2, 1)]
```

has_right_descent(i, positive=False, side='right')

Implements SemiGroups.ElementMethods.has_right_descent().

EXAMPLES:

```python
sage: D6 = CoxeterGroups().example(6)
sage: s = D6.simple_reflections()
sage: s[1].has_descent(1)
True
sage: s[1].has_descent(2)
False
sage: D6.one().has_descent(1)
False
sage: D6.one().has_descent(2)
False
```

(continues on next page)
wrapped_class
  alias of tuple

coxeter_matrix()
  Return the Coxeter matrix of self.
  EXAMPLES:

sage: FiniteCoxeterGroups().example(6).coxeter_matrix()
[1 6]
[6 1]

>>> from sage.all import *
>>> FiniteCoxeterGroups().example(Integer(6)).coxeter_matrix()
[1 6]
[6 1]

degrees()
  Return the degrees of self.
  EXAMPLES:

sage: FiniteCoxeterGroups().example(6).degrees()
(2, 6)

>>> from sage.all import *
>>> FiniteCoxeterGroups().example(Integer(6)).degrees()
(2, 6)

index_set()
  Implements CoxeterGroups.ParentMethods.index_set().
  EXAMPLES:

sage: D4 = FiniteCoxeterGroups().example(4)
sage: D4.index_set()
(1, 2)

>>> from sage.all import *
>>> D4 = FiniteCoxeterGroups().example(Integer(4))
>>> D4.index_set()
(1, 2)

one()
  Implements Monoids.ParentMethods.one().
  EXAMPLES:

sage: D6 = FiniteCoxeterGroups().example(6)
sage: D6.one()
()
6.9 Example of a finite dimensional algebra with basis

An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver.

This class illustrates a minimal implementation of a finite dimensional algebra with basis. See `sage.quivers.algebra.PathAlgebra` for a full-featured implementation of path algebras.

```python
algebra_generators()
```

Return algebra generators for this algebra.

See also:

Algebras.ParentMethods.algebra_generators().

EXAMPLES:

```python
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: A.algebra_generators()
Finite family {'x': x, 'y': y, 'a': a, 'b': b}
```

```python
one()
```

Return the unit of this algebra.

See also:

AlgebrasWithBasis.ParentMethods.one_basis().

EXAMPLES:

```python
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
```

```python
sage: A.one()
x + y
```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
>>> A.one()
x + y

**product_on_basis**(*w1*, *w2*)

Return the product of the two basis elements indexed by *w1* and *w2*.

**See also:**

AlgebrasWithBasis.ParentMethods.product_on_basis().

**EXAMPLES:**

```
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
```

Here is the multiplication table for the algebra:

```
sage: matrix([[p*q for q in A.basis()] for p in A.basis()])
[x 0 a b]
[0 y 0 0]
[0 a 0 0]
[0 b 0 0]
```

Here we take some products of linear combinations of basis elements:

```
sage: x, y, a, b = A.basis()
sage: a * (1-b)^2 * x
0
sage: x*a + b*y
a + b
sage: x*x
x
sage: x*y
0
sage: x*a*y
a
```
6.10 Examples of a finite dimensional Lie algebra with basis

```python
class sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra(R, n=None, M=None, ambient=None):
    Bases: Parent, UniqueRepresentation

    An example of a finite dimensional Lie algebra with basis: the abelian Lie algebra.

    Let $R$ be a commutative ring, and $M$ an $R$-module. The *abelian Lie algebra* on $M$ is the $R$-Lie algebra obtained by endowing $M$ with the trivial Lie bracket ($[a, b] = 0$ for all $a, b \in M$).

    This class illustrates a minimal implementation of a finite dimensional Lie algebra with basis.

    **INPUT:**
    - $R$ – base ring
    - $n$ – (optional) a nonnegative integer (default: None)
    - $M$ – an $R$-module (default: the free $R$-module of rank $n$) to serve as the ground space for the Lie algebra
    - $ambient$ – (optional) a Lie algebra; if this is set, then the resulting Lie algebra is declared a Lie subalgebra of $ambient$

    **OUTPUT:**
    The abelian Lie algebra on $M$.
```

```python
class Element(parent, value):
    Bases: Element

    Initialize self.

    **EXAMPLES:**
    ```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: TestSuite(a).run()
```
```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> TestSuite(a).run()
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.lift()
needs sage.combinat sage.libs.singular
˓→ 2*b0 + 2*b1 + 3*b2

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> elt = Integer(2)*a + Integer(2)*b + Integer(3)*c
>>> elt.lift()
needs sage.combinat sage.libs.singular
˓→ 2*b0 + 2*b1 + 3*b2
```

```
monomial_coefficients(copy=True)

Return the monomial coefficients of self.

EXAMPLES:
```
```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.monomial_coefficients()
(0: 2, 1: 2, 2: 3)
```
```
```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> elt = Integer(2)*a + Integer(2)*b + Integer(3)*c
>>> elt.monomial_coefficients()
(0: 2, 1: 2, 2: 3)
```
```
```
to_vector(order=None, sparse=False)

Return self as a vector in self.parent().module().

See the docstring of the latter method for the meaning of this.

EXAMPLES:
```
```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.to_vector()
(2, 2, 3)
```
```
```
>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> elt = Integer(2)*a + Integer(2)*b + Integer(3)*c
>>> elt.to_vector()
(2, 2, 3)
```
```
```
ambient()

Return the ambient Lie algebra of self.

EXAMPLES:
```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.ambient() == L
True

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> S = L.subalgebra([Integer(2)*a+b, b + c])
>>> S.ambient() == L
True

basis()

Return the basis of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}

basis_matrix()

Return the basis matrix of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> L.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

from_vector(v, order=None)

Return the element of self corresponding to the vector v in self.module().

Implement this if you implement module(); see the documentation of sage.categories.lie_algebras.LieAlgebras.module() for how this is to be done.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
(continues on next page)
sage: parent(u) is L
True

>>> from sage.all import *
>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> u = L.from_vector(vector(QQ, (Integer(1), Integer(0), Integer(0)))); u
(1, 0, 0)
>>> parent(u) is L
True

gens()

Return the generators of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))

ideal(gens)

Return the Lie subalgebra of self generated by the elements of the iterable gens.
This currently requires the ground ring $R$ to be a field.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([2*a+b, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1  1]

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([Integer(2)*a+b, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1  1]

is_ideal(A)

Return if self is an ideal of the ambient space A.

EXAMPLES:
leading_monomials()  
Return the set of leading monomials of the basis of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: I = L.ideal([2*a + b, b + c])
sage: I.leading_monomials()
((1, 0, 0), (0, 1, 0))

lie_algebra_generators()  
Return the basis of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}
\begin{verbatim}
>>> L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}

\textbf{\texttt{lift(x)}}

Return the lift of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.gens()
sage: L.lift(a)
b0
sage: L.lift(b).parent() \texttt{is} L.universal_enveloping_algebra()
True
sage: I = L.ideal([a + 2*b, b + 3*c])
sage: I.lift(I.basis()[0])
(1, 0, -6)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()

>>> L.lift(b).parent() \texttt{is} L.universal_enveloping_algebra()
True

>>> I = L.ideal([a + \texttt{Integer}(2)*b, b + \texttt{Integer}(3)*c])

>>> I.lift(I.basis()[\texttt{Integer}(0)])
(1, 0, -6)
\end{verbatim}

\textbf{\texttt{module()}}

Return an \(R\)-module which is isomorphic to the underlying \(R\)-module of \texttt{self}.

See \texttt{sage.categories.lie_algebras.LieAlgebras.module()} for an explanation.

In this particular example, this returns the module \(M\) that was used to construct \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()

>>> L.module()
Vector space of dimension 3 over Rational Field
\end{verbatim}
\end{verbatim}
a, b, c = L.lie_algebra_generators()
S = L.subalgebra([Integer(2)*a+b, b + c])
S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]

subalgebra(gens)
Return the Lie subalgebra of self generated by the elements of the iterable gens.
This currently requires the ground ring $R$ to be a field.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([2*a+b, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1 1]

from sage.all import *
L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
a, b, c = L.lie_algebra_generators()
L.subalgebra([2*a+b, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1 1]

universal_enveloping_algebra()
Return the universal enveloping algebra of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.universal_enveloping_algebra()
Noncommutative Multivariate Polynomial Ring in b0, b1, b2
over Rational Field, nc-relations: {}

from sage.all import *
L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
L.universal_enveloping_algebra()
Noncommutative Multivariate Polynomial Ring in b0, b1, b2
over Rational Field, nc-relations: {}

zero()
Return the zero element.

EXAMPLES:
6.11 Examples of finite enumerated sets

class sage.categories.examples.finite_enumerated_sets.Example
    Bases: UniqueRepresentation, Parent

An example of a finite enumerated set: \{1, 2, 3\}

This class provides a minimal implementation of a finite enumerated set.

See \texttt{FiniteEnumeratedSet} for a full featured implementation.

**EXAMPLES:**

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.cardinality()
3
sage: C.list()
[1, 2, 3]
sage: C.an_element()
1
```

This checks that the different methods of the enumerated set \(C\) return consistent results:

```python
sage: TestSuite(C).run(verbos = True)
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
    Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
```
>>> from sage.all import *
>>> TestSuite(C).run(verbosity = True)
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ... Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_nonzero_equal() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_enumerated_set_contains() ... pass
running ._test_enumerated_set_iter_cardinality() ... pass
running ._test_enumerated_set_iter_list() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_some_elements() ... pass

6.11. Examples of finite enumerated sets 1417
class sage.categories.examples.finiteEnumeratedSets.IsomorphicObjectOfFiniteEnumeratedSet

Bases: UniqueRepresentation, Parent

ambient()

Returns the ambient space for self, as per Sets.Subquotients.ParentMethods.ambient().

EXAMPLES:

sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.ambient()
An example of a finite enumerated set: {1,2,3}

>>> from sage.all import *
>>> C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
>>> C.ambient()
An example of a finite enumerated set: {1,2,3}

lift(x)

INPUT:

• x – an element of self

Lifts x to the ambient space for self, as per Sets.Subquotients.ParentMethods.lift().

EXAMPLES:

sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.lift(9)
3

>>> from sage.all import *
>>> C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
(continues on next page)
\begin{verbatim}
>>> C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2, \rightarrow 3}
>>> C.retract(3)
9
\end{verbatim}

6.12 Examples of finite monoids

sage.categories.examples.finite_monoids.Example
alias of IntegerModMonoid
class sage.categories.examples.finite_monoids.IntegerModMonoid(n=12)
    Bases: UniqueRepresentation, Parent
    An example of a finite monoid: the integers mod \( n \)
    This class illustrates a minimal implementation of a finite monoid.
    EXAMPLES:

\begin{verbatim}
    sage: S = FiniteMonoids().example(); S
    An example of a finite multiplicative monoid: the integers modulo 12
    sage: S.category()
    Category of finitely generated finite enumerated monoids

    from sage.all import *
    >>> S = FiniteMonoids().example(); S
    An example of a finite multiplicative monoid: the integers modulo 12
    >>> S.category()
    Category of finitely generated finite enumerated monoids
\end{verbatim}

We conclude by running systematic tests on this monoid:
sage: TestSuite(S).run(verbose = True)
running ._test_an_element() ... pass
running ._test_associativity() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ...
   Running the test suite of self.an_element()
   running ._test_category() ... pass
   running ._test_eq() ... pass
   running ._test_new() ... pass
   running ._test_not_implemented_methods() ... pass
   running ._test_pickling() ... pass
   pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_enumerated_set_contains() ... pass
running ._test_enumerated_set_iter_cardinality() ... pass
running ._test_enumerated_set_iter_list() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_one() ... pass
running ._test_pickling() ... pass
running ._test_prod() ... pass
running ._test_some_elements() ... pass

>>> from sage.all import *
>>> TestSuite(S).run(verbose = True)
running ._test_an_element() ... pass
running ._test_associativity() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ...
   Running the test suite of self.an_element()
   running ._test_category() ... pass
   running ._test_eq() ... pass
   running ._test_new() ... pass
   running ._test_not_implemented_methods() ... pass
   running ._test_pickling() ... pass
   pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_enumerated_set_contains() ... pass
running ._test_enumerated_set_iter_cardinality() ... pass
running ._test_enumerated_set_iter_list() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_one() ... pass
running ._test_pickling() ... pass
running ._test_prod() ... pass

(continues on next page)
class Element
    Bases: ElementWrapper

wrapped_class
    alias of Integer

an_element()
    Returns an element of the monoid, as per \texttt{Sets.ParentMethods.an_element()}. EXAMPLES:

    \begin{verbatim}
    sage: M = FiniteMonoids().example()
    sage: M.an_element()
    6
    >>> from sage.all import *
    >>> M = FiniteMonoids().example()
    >>> M.an_element()
    6
    \end{verbatim}

one()
    Return the one of the monoid, as per \texttt{Monoids.ParentMethods.one()}. EXAMPLES:

    \begin{verbatim}
    sage: M = FiniteMonoids().example()
    sage: M.one()
    1
    >>> from sage.all import *
    >>> M = FiniteMonoids().example()
    >>> M.one()
    1
    \end{verbatim}

product(x, y)
    Return the product of two elements \( x \) and \( y \) of the monoid, as per \texttt{Semigroups.ParentMethods.product()}. EXAMPLES:

    \begin{verbatim}
    sage: M = FiniteMonoids().example()
    sage: M.product(M(3), M(5))
    3
    >>> from sage.all import *
    >>> M = FiniteMonoids().example()
    >>> M.product(M(Integer(3)), M(Integer(5)))
    3
    \end{verbatim}

semigroup_generators()
    Returns a set of generators for self, as per \texttt{Semigroups.ParentMethods.semigroup_generators()}. Currently this returns all integers mod \( n \), which is of course far from optimal!

6.12. Examples of finite monoids
6.13 Examples of finite semigroups

sage.categories.examples.finite_semigroups.
Example
alias of LeftRegularBand
class sage.categories.examples.finite_semigroups.
LeftRegularBand(alphabet=('a', 'b', 'c', 'd'))

Bases: UniqueRepresentation, Parent

An example of a finite semigroup

This class provides a minimal implementation of a finite semigroup.

EXAMPLES:

sage: S = FiniteSemigroups().example(); S
An example of a finite semigroup:
the left regular band generated by ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> S = FiniteSemigroups().example(); S
An example of a finite semigroup:
the left regular band generated by ('a', 'b', 'c', 'd')

This is the semigroup generated by:

sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> S.semigroup_generators()
Family ('a', 'b', 'c', 'd')

such that $x^2 = x$ and $xyx = xy$ for any $x$ and $y$ in $S$:

sage: S('dab')
'dab'
sage: S('dab') * S('acb')
'dabc'

(continues on next page)
It follows that the elements of $S$ are strings without repetitions over the alphabet $a, b, c, d$:

```
sage: sorted(S.list())
['a', 'ab', 'abc', 'abcd', 'abd', 'abdc', 'ac', 'acb', 'acbd', 'acd',
 'acdb', 'ad', 'adb', 'adbc', 'adc', 'adcb', 'b', 'ba', 'bac',
 'bacd', 'bad', 'badc', 'bc', 'bca', 'bcad', 'bcd', 'bca', 'bd',
 'bda', 'bdac', 'bdc', 'bdca', 'c', 'ca', 'cab', 'cabd', 'cad',
 'cadb', 'cb', 'cba', 'cbad', 'cbd', 'cbad', 'cd', 'cda', 'cdab',
 'cdba', 'cdba', 'd', 'da', 'dab', 'dabc', 'dac', 'dacb', 'db',
 'dba', 'dbac', 'dbc', 'dbca', 'dc', 'dca', 'dcb', 'dcab', 'dcb', 'dcba']
```

It also follows that there are finitely many of them:

```
sage: S.cardinality()
64
```

Indeed:

```
sage: 4 * ( 1 + 3 * (1 + 2 * (1 + 1)))
64
```

As expected, all the elements of $S$ are idempotents:

```
sage: all( x.is_idempotent() for x in S )
True
```

Now, let us look at the structure of the semigroup:

6.13. Examples of finite semigroups
We conclude by running systematic tests on this semigroup:

```
sage: TestSuite(S).run(.verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
    Running the test suite of self.an_element()
    running ._test_category() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

(continues on next page)
```python
class Element
    Bases: ElementWrapper

    wrapped_class
        alias of str

an_element()
    Returns an element of the semigroup.

    EXAMPLES:

        sage: S = FiniteSemigroups().example()
        sage: S.an_element()
        'cdab'

        sage: S = FiniteSemigroups().example(('b'))
        sage: S.an_element()
        'b'

>>> from sage.all import *
>>> S = FiniteSemigroups().example()
>>> S.an_element()
'cdab'

>>> S = FiniteSemigroups().example(('b'))
>>> S.an_element()
'b'

product(x, y)
    Returns the product of two elements of the semigroup.

    EXAMPLES:

        sage: S = FiniteSemigroups().example()
        sage: S('a') * S('b')
        'ab'
        sage: S('a') * S('b') * S('a')
        'ab'
        sage: S('a') * S('a')
        'a'
```

6.13. Examples of finite semigroups
>>> from sage.all import *

>>> S = FiniteSemigroups().example()
>>> S('a') * S('b')
'ab'

>>> S('a') * S('b') * S('a')
'ab'

>>> S('a') * S('a')
'a'

\textbf{semigroup\_generators()}\footnote{Returns the generators of the semigroup.}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: S = FiniteSemigroups().example(alphabet=('x','y'))
sage: S.semigroup\_generators()
Family ('x', 'y')
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> S = FiniteSemigroups().example(alphabet=('x','y'))

>>> S.semigroup\_generators()
Family ('x', 'y')
\end{verbatim}

\section*{6.14 Examples of finite Weyl groups}

\textbf{sage.categories.examples.finite\_weyl\_groups.\texttt{Example}}

\textbf{alias of SymmetricGroup}

\textbf{class sage.categories.examples.finite\_weyl\_groups.SymmetricGroup\texttt{(n=4)}}

\textbf{Bases: UniqueRepresentation, Parent}

An example of finite Weyl group: the symmetric group, with elements in list notation.

The purpose of this class is to provide a minimal template for implementing finite Weyl groups. See \texttt{SymmetricGroup} for a full featured and optimized implementation.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: S = FiniteWeylGroups().example()
sage: S
The symmetric group on \{0, ..., 3\}
sage: S.category()
Category of finite irreducible Weyl groups
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> S = FiniteWeylGroups().example()

>>> S
The symmetric group on \{0, ..., 3\}

>>> S.category()
Category of finite irreducible Weyl groups
\end{verbatim}

The elements of this group are permutations of the set \{0,\ldots,3\}: \ldots
The group itself is generated by the elementary transpositions:

```
sage: S.simple_reflections()
Finite family {0: (1, 0, 2, 3), 1: (0, 2, 1, 3), 2: (0, 1, 3, 2)}
```

Only the following basic operations are implemented:

- `one()`
- `product()`
- `simple_reflection()`
- `cartan_type()`
- `Element.has_right_descent()`.

All the other usual Weyl group operations are inherited from the categories:

```
sage: S.cardinality()
24
sage: S.long_element()
(3, 2, 1, 0)
sage: S.cayley_graph(side="left").plot()  # needs sage.graphs sage.plot
Graphics object consisting of 120 graphics primitives
```

Alternatively, one could have implemented `sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.apply_simple_reflection()` instead of `simple_reflection()` and `product()`. See `CoxeterGroups().example()`.

class Element
   Bases: ElementWrapper
has_right_descent$(i)$

Implements $\text{CoxeterGroups.ElementMethods.has_right_descent()}$.

**EXAMPLES:**

```python
sage: S = FiniteWeylGroups().example()
sage: s = S.simple_reflections()
sage: (s[1] * s[2]).has_descent(2)
True
sage: S._test_has_descent()
```

```python
>>> from sage.all import *
>>> S = FiniteWeylGroups().example()
>>> s = S.simple_reflections()
>>> (s[Integer(1)] * s[Integer(2)]).has_descent(Integer(2))
True
>>> S._test_has_descent()
```

cartan_type()

Return the Cartan type of self.

**EXAMPLES:**

```python
sage: FiniteWeylGroups().example().cartan_type()  # needs sage.modules
['A', 3] relabelled by (1: 0, 2: 1, 3: 2)
```

```python
>>> from sage.all import *
>>> FiniteWeylGroups().example().cartan_type()  # needs sage.modules
['A', 3] relabelled by (1: 0, 2: 1, 3: 2)
```

degrees()

Return the degrees of self.

**EXAMPLES:**

```python
sage: W = FiniteWeylGroups().example()
sage: W.degrees()
(2, 3, 4)
```

```python
>>> from sage.all import *
>>> W = FiniteWeylGroups().example()
>>> W.degrees()
(2, 3, 4)
```

index_set()

Implements $\text{CoxeterGroups.ParentMethods.index_set()}$.

**EXAMPLES:**

```python
sage: FiniteWeylGroups().example().index_set()
[0, 1, 2]
```

```python
>>> from sage.all import *
>>> FiniteWeylGroups().example().index_set()
[0, 1, 2]
```
one()

Implements Monoids.ParentMethods.one().

EXAMPLES:

```python
sage: FiniteWeylGroups().example().one()
(0, 1, 2, 3)
```

```python
>>> from sage.all import *
>>> FiniteWeylGroups().example().one()
(0, 1, 2, 3)
```

product(x, y)

Implements Semigroups.ParentMethods.product().

EXAMPLES:

```python
sage: s = FiniteWeylGroups().example().simple_reflections()
(0, 2, 3, 1)
```

```python
>>> from sage.all import *
>>> s = FiniteWeylGroups().example().simple_reflections()
>>> s[Integer(1)] * s[Integer(2)]
(0, 2, 3, 1)
```

simple_reflection(i)

Implements CoxeterGroups.ParentMethods.simple_reflection() by returning the transposition \((i, i + 1)\).

EXAMPLES:

```python
sage: FiniteWeylGroups().example().simple_reflection(2)
(0, 1, 3, 2)
```

```python
>>> from sage.all import *
>>> FiniteWeylGroups().example().simple_reflection(Integer(2))
(0, 1, 3, 2)
```

6.15 Examples of graded connected Hopf algebras with basis

sage.categories.examples.graded_connected_hopf_algebras_with_basis

This class illustrates an implementation of a graded Hopf algebra with basis that has one primitive generator of degree 1 and basis elements indexed by non-negative integers.

This Hopf algebra example differs from what topologists refer to as a graded Hopf algebra because the twist operation in the tensor rule satisfies

\[(\mu \otimes \mu) \circ (id \otimes \tau \otimes id) \circ (\Delta \otimes \Delta) = \Delta \circ \mu\]

where \(\tau(x \otimes y) = y \otimes x\).
\texttt{coproduct\_on\_basis}(i)

The coproduct of a basis element.

\[ \Delta(P_i) = \sum_{j=0}^{i} P_{i-j} \otimes P_j \]

INPUT:

\begin{itemize}
  \item \(i\) – a non-negative integer
\end{itemize}

OUTPUT:

\begin{itemize}
  \item an element of the tensor square of \texttt{self}
\end{itemize}

degree\_on\_basis\(\)(i)

The degree of a non-negative integer is itself.

INPUT:

\begin{itemize}
  \item \(i\) – a non-negative integer
\end{itemize}

OUTPUT:

\begin{itemize}
  \item a non-negative integer
\end{itemize}

texttt{one\_basis}()

Returns 0, which index the unit of the Hopf algebra.

OUTPUT:

\begin{itemize}
  \item the non-negative integer 0
\end{itemize}

EXAMPLES:

\begin{verbatim}
sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
sage: H.one_basis()
0
sage: H.one()
P0
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
>>> H.one_basis()
0
>>> H.one()
P0
\end{verbatim}

\texttt{product\_on\_basis}(i, j)

The product of two basis elements.

The product of elements of degree \(i\) and \(j\) is an element of degree \(i+j\).

INPUT:

\begin{itemize}
  \item \(i, j\) – non-negative integers
\end{itemize}

OUTPUT:

\begin{itemize}
  \item a basis element indexed by \(i+j\)
6.16 Examples of graded modules with basis

sage.categories.examples.graded_modules_with_basis.Example
alias of GradedPartitionModule
class sage.categories.examples.graded_modules_with_basis.GradedPartitionModule(base_ring)
Bases: CombinatorialFreeModule

This class illustrates an implementation of a graded module with basis: the free module over partitions.

INPUT:

- \( R \) – base ring

The implementation involves the following:

- A choice of how to represent elements. In this case, the basis elements are partitions. The algebra is constructed as a CombinatorialFreeModule on the set of partitions, so it inherits all of the methods for such objects, and has operations like addition already defined.

```
sage: A = GradedModulesWithBasis(QQ).example()
˓→ needs sage.modules
```

```
>>> from sage.all import *
>>> A = GradedModulesWithBasis(QQ).example() #˓→ needs sage.modules
```

- A basis function - this module is graded by the non-negative integers, so there is a function defined in this module, creatively called basis(), which takes an integer \( d \) as input and returns a family of partitions representing a basis for the algebra in degree \( d \).

```
sage: A.basis(2)
˓→ needs sage.modules
Lazy family (Term map from Partitions to An example of a graded module with basis: the free module on partitions over Rational Field(i))_{i in Partitions of the integer 2}
sage: A.basis(6)[Partition([3,2,1])]
˓→ needs sage.modules
P[3, 2, 1]
```

```
>>> from sage.all import *
>>> A.basis(Integer(2)) #˓→ needs sage.modules
Lazy family (Term map from Partitions to An example of a graded module with basis: the free module on partitions over Rational Field(i))_{i in Partitions of the integer 2}
>>> A.basis(Integer(6))[Partition([Integer(3),Integer(2),Integer(1)])] #˓→ needs sage.modules
P[3, 2, 1]
```

- If the algebra is called \( A \), then its basis function is stored as \( A.basis \). Thus the function can be used to find a basis for the degree \( d \) piece: essentially, just call \( A.basis(d) \). More precisely, call \( x \) for each \( x \) in \( A.basis(d) \).

```
sage: [m for m in A.basis(4)] # needs sage.modules
[P[4], P[3, 1], P[2, 2], P[2, 1, 1], P[1, 1, 1, 1]]
```
>>> from sage.all import *
>>> [m for m in A.basis(Integer(4))]
-> # needs sage.modules
[P[4], P[3, 1], P[2, 2], P[2, 1, 1], P[1, 1, 1, 1]]

• For dealing with basis elements: degree_on_basis(), and _repr_term(). The first of these defines the degree of any monomial, and then the degree method for elements – see the next item – uses it to compute the degree for a linear combination of monomials. The last of these determines the print representation for monomials, which automatically produces the print representation for general elements.

```python
sage: A.degree_on_basis(Partition([4, 3]))
-> # needs sage.modules
7
sage: A._repr_term(Partition([4, 3]))
-> # needs sage.modules
'P[4, 3]'
```

• There is a class for elements, which inherits from IndexedFreeModuleElement. An element is determined by a dictionary whose keys are partitions and whose corresponding values are the coefficients. The class implements two things: an is_homogeneous method and a degree method.

```python
sage: p = A.monomial(Partition([3, 2, 1])); p
-> # needs sage.modules
P[3, 2, 1]
sage: p.is_homogeneous()
-> # needs sage.modules
True
sage: p.degree()
-> # needs sage.modules
6
```

```python
>>> from sage.all import *
>>> p = A.monomial(Partition([Integer(3), Integer(2), Integer(1)])); p
-> # needs sage.modules
P[3, 2, 1]
>>> p.is_homogeneous()
-> # needs sage.modules
True
>>> p.degree()
-> # needs sage.modules
6
```

**basis**(d=None)

Return the basis for (the d-th homogeneous component of) self.

**INPUT:**

- d – (default: None) nonnegative integer or None

**OUTPUT:**

If $d$ is None, returns the basis of the module. Otherwise, returns the basis of the homogeneous component of degree $d$ (i.e., the subfamily of the basis of the whole module which consists only of the basis vectors lying in $F_d \setminus \bigcup_{i<d} F_i$).

The basis is always returned as a family.

EXAMPLES:

```
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Lazy family (Term map from Partitions to An example of a
filtered module with basis: the free module on partitions
over Integer Ring(i))_{i in Partitions of the integer 4}
```

Without arguments, the full basis is returned:

```
sage: A.basis()
Lazy family (Term map from Partitions to An example of a
filtered module with basis: the free module on partitions
over Integer Ring(i))_{i in Partitions}
```

Checking this method on a filtered algebra. Note that this will typically raise a `NotImplementedError` when this feature is not implemented.

```
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Traceback (most recent call last):
... 
NotImplementedError: infinite set
```
Without arguments, the full basis is returned:

```
sage: A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring(i))_{i in Free abelian monoid indexed by {'x', 'y', 'z'}}
```

```
>>> from sage.all import *
>>> A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring(i))_{i in Free abelian monoid indexed by {'x', 'y', 'z'}}
```

An example with a graded algebra:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.basis()
Lazy family (Term map from Subsets of {0,1} to The exterior algebra of rank 2 over Rational Field(i))_{i in Subsets of {0,1}}
```

```
>>> from sage.all import *
>>>
>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> E.basis()
Lazy family (Term map from Subsets of {0,1} to The exterior algebra of rank 2 over Rational Field(i))_{i in Subsets of {0,1}}
```

degree_on_basis($t$)

The degree of the element determined by the partition $t$ in this graded module.

**INPUT:**

- $t$ – the index of an element of the basis of this module, i.e. a partition

**OUTPUT:** an integer, the degree of the corresponding basis element

**EXAMPLES:**

```
sage: # needs sage.modules
sage: A = GradedModulesWithBasis(QQ).example()
sage: A.degree_on_basis(Partition((2,1)))
3
```

```
sage: A = GradedModulesWithBasis(QQ).example()
sage: A.degree_on_basis(Partition((4,2,1,1,1)))
10
```

```
sage: type(A.degree_on_basis(Partition((1,1))))
<class 'sage.rings.integer.Integer'>
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> A = GradedModulesWithBasis(QQ).example()
>>> A.degree_on_basis(Partition((Integer(2),Integer(1))))
3
```
6.17 Examples of graphs

class sage.categories.examples.graphs.Cycle(n=5)
    Bases: UniqueRepresentation, Parent

    An example of a graph: the cycle of length $n$.
    This class illustrates a minimal implementation of a graph.

    EXAMPLES:

    sage: from sage.categories.graphs import Graphs
    sage: C = Graphs().example(); C
    An example of a graph: the 5-cycle
    sage: C.category()
    Category of graphs

    >>> from sage.all import *
    >>> from sage.categories.graphs import Graphs
    >>> C = Graphs().example(); C
    An example of a graph: the 5-cycle
    >>> C.category()
    Category of graphs

    We conclude by running systematic tests on this graph:
    sage: TestSuite(C).run()
    >>> from sage.all import *
    >>> TestSuite(C).run()

    class Element
        Bases: ElementWrapper
dimension()

        Return the dimension of self.

        EXAMPLES:

        sage: from sage.categories.graphs import Graphs
        sage: C = Graphs().example()
        sage: e = C.edges()[0]
        sage: e.dimension()
        2
        sage: v = C.vertices()[0]
        sage: v.dimension()
        1
```python
>>> from sage.all import *
>>> from sage.categories.graphs import Graphs
>>> C = Graphs().example()
>>> e = C.edges()[Integer(0)]
>>> e.dimension()
2
>>> v = C.vertices()[Integer(0)]
>>> v.dimension()
1
```

**an_element()**

Return an element of the graph, as per `Sets.ParentMethods.an_element()`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
g sage: C = Graphs().example()
sage: C.an_element()
0
```

**edges()**

Return the edges of self.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
g sage: C = Graphs().example()
g sage: C.edges()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

**vertices()**

Return the vertices of self.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
g sage: C = Graphs().example()
g sage: C.vertices()
[0, 1, 2, 3, 4]
```

1436 Chapter 6. Examples of parents using categories
6.18 Examples of Hopf algebras with basis

```python
class sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra(R, G):
    Bases: CombinatorialFreeModule

    An example of a Hopf algebra with basis: the group algebra of a group

    This class illustrates a minimal implementation of a Hopf algebra with basis.

    algebra_generators()
        Return the generators of this algebra, as per algebra_generators().

        They correspond to the generators of the group.

        EXAMPLES:

        sage: A = HopfAlgebrasWithBasis(QQ).example(); A
        An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a permutation group over Rational Field
        sage: A.algebra_generators()
        Finite family {(1,2,3): B[(1,2,3)], (1,3): B[(1,3)]}

        >>> from sage.all import *
        >>> A = HopfAlgebrasWithBasis(QQ).example(); A
        >>> (a, b) = A._group.gens()
        >>> A.antipode_on_basis(a)
        B[(1,3,2)]

    antipode_on_basis(g)
        Antipode, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.

        It is given, on basis elements, by \( \nu(g) = g^{-1} \)

        EXAMPLES:

        sage: A = HopfAlgebrasWithBasis(QQ).example()
        sage: (a, b) = A._group.gens()
        sage: A.antipode_on_basis(a)
        B[(1,3,2)]

        >>> from sage.all import *
        >>> A = HopfAlgebrasWithBasis(QQ).example()
        >>> (a, b) = A._group.gens()
        >>> A.antipode_on_basis(a)
        B[(1,3,2)]

    coproduct_on_basis(g)
        Coproduct, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.

        The basis elements are group like: \( \Delta(g) = g \otimes g \).
```

6.18. Examples of Hopf algebras with basis 1437
EXAMPLES:

```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: A.coproduct_on_basis(a)
B[(1,2,3)] # B[(1,2,3)]
```

```python
>>> from sage.all import *
>>> A = HopfAlgebrasWithBasis(QQ).example()
>>> (a, b) = A._group.gens()
>>> A.coproduct_on_basis(a)
B[(1,2,3)] # B[(1,2,3)]
```

counit_on_basis(g)

Counit, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.
counit_on_basis().

The counit on the basis elements is 1.

EXAMPLES:

```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: A.counit_on_basis(a)
1
```

```python
>>> from sage.all import *
>>> A = HopfAlgebrasWithBasis(QQ).example()
>>> (a, b) = A._group.gens()
>>> A.counit_on_basis(a)
1
```

one_basis()

Returns the one of the group, which index the one of this algebra, as per AlgebrasWithBasis.
ParentMethods.one_basis().

EXAMPLES:

```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: A.one_basis()

```

```python
>>> from sage.all import *
>>> A = HopfAlgebrasWithBasis(QQ).example()
>>> A.counit_on_basis(a)
1
```

```python
>>> from sage.all import *
>>> A = HopfAlgebrasWithBasis(QQ).example()
>>> A.counit_on_basis(a)
1
```

product_on_basis(g1,g2)

Product, on basis elements, as per AlgebrasWithBasis.ParentMethods.
product_on_basis().

The product of two basis elements is induced by the product of the corresponding elements of the group.

EXAMPLES:
6.19 Examples of infinite enumerated sets

sage.categories.examples.infinite_enumerated_sets.

Example

alias of NonNegativeIntegers

class sage.categories.examples.infinite_enumerated_sets.

NonNegativeIntegers

Bases: UniqueRepresentation, Parent

An example of infinite enumerated set: the non negative integers

This class provides a minimal implementation of an infinite enumerated set.

EXAMPLES:

sage: NN = InfiniteEnumeratedSets().example()
sage: NN
An example of an infinite enumerated set: the non negative integers
sage: NN.cardinality()
+Infinity
sage: NN.list()
Traceback (most recent call last):
...
NotImplementedError: cannot list an infinite set
sage: NN.element_class
<class 'sage.rings.integer.Integer'>
sage: it = iter(NN)
sage: [next(it), next(it), next(it), next(it), next(it)]
[0, 1, 2, 3, 4]
sage: x = next(it); type(x)
<Integer>

6.19. Examples of infinite enumerated sets 1439
An example of an infinite enumerated set: the non negative integers

```python
>>> NN = InfiniteEnumeratedSets().example()
>>> NN
An example of an infinite enumerated set: the non negative integers
```

```python
>>> NN.cardinality()
+Infinity
```

```python
Traceback (most recent call last):
...
NotImplementedError: cannot list an infinite set
```

```python
>>> NN.element_class
<class 'sage.rings.integer.Integer'>
```

```python
>>> it = iter(NN)
```

```python
>>> [next(it), next(it), next(it), next(it), next(it)]
[0, 1, 2, 3, 4]
```

```python
>>> x = next(it); type(x)
<class 'sage.rings.integer.Integer'>
```

```python
>>> x.parent()
Integer Ring
```

```python
>>> x+Integer(3)
8
```

```python
>>> NN(Integer(15))
15
```

```python
>>> NN.first()
0
```

This checks that the different methods of $\mathcal{N}$ return consistent results:

```python
sage: TestSuite(NN).run(verify = True)
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
    Running the test suite of self.an_element()
    running ._test_category() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_nonzero_equal() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    pass
    running ._test_elements_eq_reflexive() . . . pass
    running ._test_elements_eq_symmetric() . . . pass
    running ._test_elements_eq_transitive() . . . pass
    running ._test_elements_neq() . . . pass
    running ._test_enumerated_set_contains() . . . pass
    running ._test_enumerated_set_iter_cardinality() . . . pass
    running ._test_enumerated_set_iter_list() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    running ._test_some_elements() . . . pass
```

```python
(continues on next page)
```
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ...
    Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_nonzero_equal() ... pass
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_enumerated_set_contains() ... pass
running ._test_enumerated_set_iter_cardinality() ... pass
running ._test_enumerated_set_iter_list() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_some_elements() ... pass

Element
    alias of Integer

an_element()

    EXAMPLES:

    sage: InfiniteEnumeratedSets().example().an_element()
    42

    >>> from sage.all import *
    >>> InfiniteEnumeratedSets().example().an_element()
    42

next(a)

    EXAMPLES:

    sage: NN = InfiniteEnumeratedSets().example()
sage: NN.next(3)
    4

    >>> from sage.all import *
    >>> NN = InfiniteEnumeratedSets().example()
    >>> NN.next(Integer(3))
    4

6.19. Examples of infinite enumerated sets
6.20 Examples of a Lie algebra

sage.categories.examples.lie_algebras.Example
alias of LieAlgebraFromAssociative
class sage.categories.examples.lie_algebras.LieAlgebraFromAssociative(gens)
    Bases: Parent, UniqueRepresentation

An example of a Lie algebra: a Lie algebra generated by a set of elements of an associative algebra.

This class illustrates a minimal implementation of a Lie algebra.

Let $R$ be a commutative ring, and $A$ an associative $R$-algebra. The Lie algebra $A$ (sometimes denoted $A^-$) is defined to be the $R$-module $A$ with Lie bracket given by the commutator in $A$: that is, $[a, b] := ab - ba$ for all $a, b \in A$.

What this class implements is not precisely $A^-$, however; it is the Lie subalgebra of $A^-$ generated by the elements of the iterable $gens$. This specific implementation does not provide a reasonable containment test (i.e., it does not allow you to check if a given element $a$ of $A^-$ belongs to this Lie subalgebra); it, however, allows computing inside it.

INPUT:

• $gens$ – a nonempty iterable consisting of elements of an associative algebra $A$

OUTPUT:

The Lie subalgebra of $A^-$ generated by the elements of $gens$

EXAMPLES:

We create a model of $sl_2$ using matrices:

```
sage: gens = [matrix([[0,1],[0,0]]), matrix([[0,0],[1,0]]), matrix([[1,0],[0,-1]])]
sage: for g in gens:
    ....:     g.set_immutable()
sage: L = LieAlgebras(QQ).example(gens)
sage: e,f,h = L.lie_algebra_generators()
sage: e.bracket(f) == h
True
sage: h.bracket(e) == 2*e
True
sage: h.bracket(f) == -2*f
True
```

```ruby
>>> from sage.all import *
>>> gens = [matrix([[Integer(0),Integer(1)],[Integer(0),Integer(0)]]),
          matrix([[Integer(0),Integer(0)],[Integer(1),Integer(0)]]),
          matrix([[Integer(1),
                   -Integer(0)],[Integer(0),-Integer(1)]]))
>>> for g in gens:
    ...     g.set_immutable()
>>> L = LieAlgebras(QQ).example(gens)
>>> e,f,h = L.lie_algebra_generators()
>>> e.bracket(f) == h
True
>>> h.bracket(e) == Integer(2)*e
True
>>> h.bracket(f) == -Integer(2)*f
True
```
class Element
Bases: ElementWrapper
Wrap an element as a Lie algebra element.

lie_algebra_generators()
Return the generators of self as a Lie algebra.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).example()
# needs sage.combinat sage.groups
sage: L.lie_algebra_generators()
# needs sage.combinat sage.groups
Family ([2, 1, 3], [2, 3, 1])
```

zero()
Return the element 0.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).example()
# needs sage.combinat sage.groups
sage: L.zero()
# needs sage.combinat sage.groups
0
```

### 6.21 Examples of a Lie algebra with basis

class sage.categories.examples.lie_algebras_with_basis.AbelianLieAlgebra(R, gens)
Bases: CombinatorialFreeModule
An example of a Lie algebra: the abelian Lie algebra.
This class illustrates a minimal implementation of a Lie algebra with a distinguished basis.

class Element
Bases: IndexedFreeModuleElement
lift()

Return the lift of self to the universal enveloping algebra.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: elt = L.an_element()
sage: elt.lift()
3*P[F[2]] + 2*P[F[1]] + 2*P[F[]]
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).WithBasis().example()
>>> elt = L.an_element()
>>> elt.lift()
3*P[F[2]] + 2*P[F[1]] + 2*P[F[]]
```

bracket_on_basis(x, y)

Return the Lie bracket on basis elements indexed by \(x\) and \(y\).

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.bracket_on_basis(Partition([4,1]), Partition([2,2,1]))
0
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).WithBasis().example()
>>> L.bracket_on_basis(Partition([Integer(4),Integer(1)]),
\text{\ldots} Partition([Integer(2),Integer(2),Integer(1)]))
0
```

lie_algebra_generators()

Return the generators of self as a Lie algebra.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.lie_algebra_generators()
Lazy family (Term map from Partitions to
An example of a Lie algebra: the abelian Lie algebra on the
generators indexed by Partitions over Rational
Field(i))_{i in Partitions}
```

```python
>>> from sage.all import *
>>> L = LieAlgebras(QQ).WithBasis().example()
>>> L.lie_algebra_generators()
Lazy family (Term map from Partitions to
An example of a Lie algebra: the abelian Lie algebra on the
generators indexed by Partitions over Rational
Field(i))_{i in Partitions}
```
Bases: CombinatorialFreeModule

Polynomial ring whose generators are indexed by an arbitrary set.

Todo: Currently this is just used as the universal enveloping algebra for the example of the abelian Lie algebra. This should be factored out into a more complete class.

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: UEA.algebra_generators()
Lazy family (algebra generator map(i))_{i in Partitions}
```

one_basis()

Return the index of element 1.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: I = UEA._indices
sage: UEA.one_basis()
1
sage: UEA.one_basis().parent()
Free abelian monoid indexed by Partitions
```

product_on_basis(x, y)

Return the product of the monomials indexed by x and y.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: I = UEA._indices
sage: UEA.product_on_basis(I.an_element(), I.an_element())
```
6.22 Examples of magmas

sage.categories.examples.magmas.Example
alias of FreeMagma
class sage.categories.examples.magmas.FreeMagma(alphabet=('a', 'b', 'c', 'd'))
    Bases: UniqueRepresentation, Parent

An example of magma.

The purpose of this class is to provide a minimal template for implementing a magma.

EXAMPLES:

sage: M = Magmas().example(); M
An example of a magma: the free magma generated by ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> M = Magmas().example(); M
An example of a magma: the free magma generated by ('a', 'b', 'c', 'd')

This is the free magma generated by:

sage: M.magma_generators()
Family ('a', 'b', 'c', 'd')
sage: a, b, c, d = M.magma_generators()

and with a non-associative product given by:

sage: a * (b * c) * (d * a * b)
'((a*(b*c))*((d*a)*b))'
sage: a * (b * c) == (a * b) * c
False

>>> from sage.all import *
>>> a * (b * c) * (d * a * b)
'((a*(b*c))*((d*a)*b))'
>>> a * (b * c) == (a * b) * c
False

class Element
    Bases: ElementWrapper
The class for elements of the free magma.

\texttt{wrapped_class}

alias of \texttt{str}

\texttt{an_element()}

Return an element of the magma.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = Magmas().example()
sage: F.an_element()
'(((a*b)*c)*d)'
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> F = Magmas().example()
>>> F.an_element()
'(((a*b)*c)*d)'
\end{verbatim}

\texttt{magma_generators()}

Return the generators of the magma.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = Magmas().example()
sage: F.magma_generators()
Family (a, b, c, d)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> F = Magmas().example()
>>> F.magma_generators()
Family (a, b, c, d)
\end{verbatim}

\texttt{product(x, y)}

Return the product of \(x\) and \(y\) in the magma, as per \texttt{Magmas.ParentMethods.product()}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = Magmas().example()
sage: F('a') * F.an_element()
'(a*(((a*b)*c)*d))'
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> F = Magmas().example()
>>> F('a') * F.an_element()
'(a*(((a*b)*c)*d))'
\end{verbatim}
6.23 Examples of manifolds

sage.categories.examples.manifolds.Example
alias of Plane

class sage.categories.examples.manifolds.Plane(n=3, base_ring=None)
Bases: UniqueRepresentation, Parent
An example of a manifold: the $n$-dimensional plane.
This class illustrates a minimal implementation of a manifold.

EXAMPLES:

```
sage: from sage.categories.manifolds import Manifolds
sage: M = Manifolds(QQ).example(); M
An example of a Rational Field manifold: the 3-dimensional plane
sage: M.category()
Category of manifolds over Rational Field
```

We conclude by running systematic tests on this manifold:

```
sage: TestSuite(M).run()
```

Element
alias of ElementWrapper

an_element()
Return an element of the manifold, as per `Sets.ParentMethods.an_element()`.

EXAMPLES:

```
sage: from sage.categories.manifolds import Manifolds
sage: M = Manifolds(QQ).example()
sage: M.an_element()
(0, 0, 0)
```

```
dimension()

Return the dimension of self.

EXAMPLES:

```python
dimension()

Return the dimension of self.

EXAMPLES:

```

6.24 Examples of monoids

sage.categories.examples.monoids.Example

alias of FreeMonoid
class sage.categories.examples.monoids.FreeMonoid(alphabet=('a', 'b', 'c', 'd'))

Bases: FreeSemigroup

An example of a monoid: the free monoid

This class illustrates a minimal implementation of a monoid. For a full featured implementation of free monoids,
see FreeMonoid().

EXAMPLES:

```python
dimension()

Return the dimension of self.

EXAMPLES:

```

This is the free semigroup generated by:

```python
dimension()

Return the dimension of self.

EXAMPLES:

```

with product rule given by concatenation of words:
and unit given by the empty word:

```
sage: S.one()
```

We conclude by running systematic tests on this monoid:

```
sage: TestSuite(S).run(quiet = False, verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
   Running the test suite of self.an_element()
   running ._test_category() . . . pass
   running ._test_eq() . . . pass
   running ._test_new() . . . pass
   running ._test_not_implemented_methods() . . . pass
   running ._test_pickling() . . . pass
   pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

(continues on next page)
running ._test_not_implemented_methods() ... pass
  running ._test_pickling() ... pass
  pass
  running ._test_elements_eq_reflexive() ... pass
  running ._test_elements_eq_symmetric() ... pass
  running ._test_elements_eq_transitive() ... pass
  running ._test_elements_neq() ... pass
  running ._test_eq() ... pass
  running ._test_new() ... pass
  running ._test_not_implemented_methods() ... pass
  running ._test_one() ... pass
  running ._test_pickling() ... pass
  running ._test_prod() ... pass
  running ._test_some_elements() ... pass

class Element
  Bases: ElementWrapper

    wrapped_class
      alias of str

monoid_generators()
  Return the generators of this monoid.

  EXAMPLES:

    sage: M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    sage: M.monoid_generators()
    Finite family {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd'}
    sage: a,b,c,d = M.monoid_generators()
    sage: a*d*c*b
    'adcb'

    >>> from sage.all import *
    >>> M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    >>> M.monoid_generators()
    Finite family {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd'}
    >>> a,b,c,d = M.monoid_generators()
    >>> a*d*c*b
    'adcb'

one()
  Returns the one of the monoid, as per Monoids.ParentMethods.one().

  EXAMPLES:

    sage: M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    sage: M.one()
    ''

    >>> from sage.all import *
    >>> M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    >>> M.one()
    ''

(continues on next page)
6.25 Examples of posets

```python
class sage.categories.examples.posets.FiniteSetsOrderedByInclusion
    Bases: UniqueRepresentation, Parent

An example of a poset: finite sets ordered by inclusion
This class provides a minimal implementation of a poset

EXAMPLES:

```
sage: P = Posets().example(); P
An example of a poset: sets ordered by inclusion

```
class Element

Bases: ElementWrapper

wrapped_class

alias of Set_object Enumerated

an_element()

Returns an element of this poset

EXAMPLES:

```sage
B = Posets().example()
sage: B.an_element()
{1, 4, 6}
```

```python
>>> from sage.all import *
>>> B = Posets().example()
>>> B.an_element()
{1, 4, 6}
```

le(x, y)

Returns whether \( x \) is a subset of \( y \)

EXAMPLES:

```sage
P = Posets().example()
sage: P.le( P(Set([1,3])), P(Set([1,2,3])) )
True
sage: P.le( P(Set([1,3])), P(Set([1,3])) )
True
sage: P.le( P(Set([1,2])), P(Set([1,3])) )
False
```

```python
>>> from sage.all import *
>>> P = Posets().example()
>>> P.le( P(Set([Integer(1),Integer(3)]), P(Set([Integer(1),Integer(2),
   →Integer(3)]))) )
True
>>> P.le( P(Set([Integer(1),Integer(3)]), P(Set([Integer(1),Integer(3)]))) )
True
```
class sage.categories.examples.posets.PositiveIntegersOrderedByDivisibilityFacade
Bases: UniqueRepresentation, Parent
An example of a facade poset: the positive integers ordered by divisibility
This class provides a minimal implementation of a facade poset

EXAMPLES:

    sage: P = Posets().example("facade"); P
    An example of a facade poset: the positive integers ordered by divisibility
    sage: 5
    5
    sage: 0
    Traceback (most recent call last):
      ...
    ValueError: Can't coerce '0' in any parent 'An example of a facade poset: the... positive integers ordered by divisibility' is a facade for
    sage: 3 in P
    True
    sage: 0 in P
    False

>>> from sage.all import *

>>> P = Posets().example("facade"); P
An example of a facade poset: the positive integers ordered by divisibility

>>> P(Integer(5))
5
>>> P(Integer(0))
Traceback (most recent call last):
  ...
ValueError: Can't coerce '0' in any parent 'An example of a facade poset: the... positive integers ordered by divisibility' is a facade for

>>> Integer(3) in P
True
>>> Integer(0) in P
False

class element_class (X, category=None)
Bases: Set_objectEnumerated, parent_class
A finite enumerated set.
le (x, y)
Returns whether \( x \) is divisible by \( y \)

EXAMPLES:
6.26 Examples of semigroups

class sage.categories.examples.semigroups.FreeSemigroup(alphabet=('a', 'b', 'c', 'd'))

Bases: UniqueRepresentation, Parent

An example of semigroup.

The purpose of this class is to provide a minimal template for implementing a semigroup.

EXAMPLES:

sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')

This is the free semigroup generated by:

sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')

>>> from sage.all import *
>>> S.semigroup_generators()
Family ('a', 'b', 'c', 'd')

and with product given by concatenation:

sage: S('dab') * S('acb')
dabacb

>>> from sage.all import *
>>> S('dab') * S('acb')
dabacb
class Element

    Bases: ElementWrapper

    The class for elements of the free semigroup.

    wrapped_class

        alias of str

    an_element()

        Returns an element of the semigroup.

        EXAMPLES:

        sage: F = Semigroups().example('free')
        sage: F.an_element()
        'abcd'

        >>> from sage.all import *
        >>> F = Semigroups().example('free')
        >>> F.an_element()
        'abcd'

    product(x, y)

        Returns the product of \( x \) and \( y \) in the semigroup, as per Semigroups.ParentMethods.product().

        EXAMPLES:

        sage: F = Semigroups().example('free')
        sage: F.an_element() * F('a')^5
        'abcdaaaaa'

        >>> from sage.all import *
        >>> F = Semigroups().example('free')
        >>> F.an_element() * F('a')**Integer(5)
        'abcdaaaaa'

    semigroup_generators()

        Returns the generators of the semigroup.

        EXAMPLES:

        sage: F = Semigroups().example('free')
        sage: F.semigroup_generators()
        Family (a, b, c, d)

        >>> from sage.all import *
        >>> F = Semigroups().example('free')
        >>> F.semigroup_generators()
        Family (a, b, c, d)

class sage.categories.examples.semigroups.IncompleteSubquotientSemigroup (category=None)

    Bases: UniqueRepresentation, Parent

    An incompletely implemented subquotient semigroup, for testing purposes

    EXAMPLES:
sage: S = sage.categories.examples.semigroups.IncompleteSubquotientSemigroup()
sage: S
A subquotient of An example of a semigroup: the left zero semigroup

```python
>>> from sage.all import *

>>> S = sage.categories.examples.semigroups.IncompleteSubquotientSemigroup()

>>> S
A subquotient of An example of a semigroup: the left zero semigroup
```

class Element

Bases: ElementWrapper

ambient()

Returns the ambient semigroup.

EXAMPLES:

```python
sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
An example of a semigroup: the left zero semigroup
```

```python
>>> from sage.all import *

>>> S = Semigroups().Subquotients().example()

>>> S.ambient()
An example of a semigroup: the left zero semigroup
```

class sage.categories.examples.semigroups.LeftZeroSemigroup

Bases: UniqueRepresentation, Parent

An example of a semigroup.

This class illustrates a minimal implementation of a semigroup.

EXAMPLES:

```python
sage: S = Semigroups().example(); S
An example of a semigroup: the left zero semigroup

>>> from sage.all import *

>>> S = Semigroups().example(); S
An example of a semigroup: the left zero semigroup
```

This is the semigroup that contains all sorts of objects:

```python
sage: S.some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

```python
>>> from sage.all import *

>>> S.some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

with product rule given by \(a \times b = a\) for all \(a, b\):

```python
sage: S('hello') * S('world')
'hello'
sage: S(3)*S(1)*S(2)
```

(continues on next page)
class Element
Bases: ElementWrapper

is_idempotent()
Trivial implementation of Semigroups.Element.is_idempotent since all elements of this
semigroup are idempotent!

EXAMPLES:

sage: S = Semigroups().example()
sage: S.an_element().is_idempotent()
True
sage: S(17).is_idempotent()
True

an_element()
Returns an element of the semigroup.

EXAMPLES:

sage: Semigroups().example().an_element()
42

product(x, y)
Returns the product of x and y in the semigroup, as per Semigroups.ParentMethods.product().

EXAMPLES:

sage: S = Semigroups().example()
sage: S('hello') * S('world')
'hello'
sage: S(3)*S(1)*S(2)
3
`some_elements()`

Returns a list of some elements of the semigroup.

**EXAMPLES:**

```
sage: Semigroups().example().some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

```
>>> from sage.all import *

>>> Semigroups().example().some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

```python
class sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup(category=None)

Bases: UniqueRepresentation, Parent

Example of a quotient semigroup

**EXAMPLES:**

```
sage: S = Semigroups().Subquotients().example(); S
An example of a (sub)quotient semigroup: a quotient of the left zero semigroup
```

```
>>> from sage.all import *

>>> S = Semigroups().Subquotients().example(); S
An example of a (sub)quotient semigroup: a quotient of the left zero semigroup
```

This is the quotient of:

```
sage: S.ambient()
An example of a semigroup: the left zero semigroup
```

```
>>> from sage.all import *

>>> S.ambient()
An example of a semigroup: the left zero semigroup
```

obtained by setting $x = 42$ for any $x \geq 42$:

```
sage: S(100)
42
sage: S(100) == S(42)
True
```

```
>>> from sage.all import *

>>> S(Integer(100))
42
>>> S(Integer(100)) == S(Integer(42))
True
```

The product is inherited from the ambient semigroup:

```
>>> from sage.all import *

>>> S('hello') * S('world')
'hello'
```

```
>>> S(Integer(3))*S(Integer(1))*S(Integer(2))
3
```
sage: S(1)*S(2) == S(1)
True

```python
>>> from sage.all import *

>>> S(Integer(1))*S(Integer(2)) == S(Integer(1))
True
```

class Element

Bases: ElementWrapper

ambient()

Returns the ambient semigroup.

EXAMPLES:

```python
sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
```

An example of a semigroup: the left zero semigroup

```python
>>> from sage.all import *

>>> S = Semigroups().Subquotients().example()

>>> S.ambient()
```

An example of a semigroup: the left zero semigroup

an_element()

Returns an element of the semigroup.

EXAMPLES:

```python
sage: S = Semigroups().Subquotients().example()
sage: S.an_element()
42
```

```python
>>> from sage.all import *

>>> S = Semigroups().Subquotients().example()

>>> S.an_element()
```

42

lift(x)

Lift the element x into the ambient semigroup.

INPUT:

- x – an element of self.

OUTPUT:

- an element of self.ambient().

EXAMPLES:

```python
sage: S = Semigroups().Subquotients().example()
sage: x = S.an_element(); x
42
```

```python
sage: S.lift(x)
42
```

```python
sage: S.lift(x) in S.ambient()
```

(continues on next page)
True
sage: y = S.ambient()(100); y
100
sage: S.lift(S(y))
42

```python
>>> from sage.all import *
>>> S = Semigroups().Subquotients().example()
>>> x = S.an_element(); x
42
>>> S.lift(x)
42
>>> S.lift(x) in S.ambient()
True
>>> y = S.ambient()(Integer(100)); y
100
>>> S.lift(S(y))
42
```

**retract** *(x)*

Returns the retract \( x \) onto an element of this semigroup.

**INPUT:**

- \( x \) – an element of the ambient semigroup (\( \text{self.ambient()} \)).

**OUTPUT:**

- an element of \( \text{self} \).

**EXAMPLES:**

```python
sage: S = Semigroups().Subquotients().example()
sage: L = S.ambient()
sage: S.retract(L(17))
17
sage: S.retract(L(42))
42
sage: S.retract(L(Integer(171)))
42
```

```python
>>> from sage.all import *
>>> S = Semigroups().Subquotients().example()
>>> L = S.ambient()
>>> S.retract(L(Integer(17)))
17
>>> S.retract(L(Integer(42)))
42
>>> S.retract(L(Integer(171)))
42
```

**some_elements** ()

Returns a list of some elements of the semigroup.

**EXAMPLES:**

```python
```
the_answer()

Returns the Answer to Life, the Universe, and Everything as an element of this semigroup.

EXAMPLES:

```python
sage: S = Semigroups().Subquotients().example()
sage: S.the_answer()
42
```

6.27 Examples of semigroups in cython

class sage.categories.examples.semigroups_cython.IdempotentSemigroups

Bases: Category

class ElementMethods

Bases: object

is_idempotent()

EXAMPLES:

```python
sage: from sage.categories.examples.semigroups_cython import...
    ←LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: S(2).is_idempotent()
True
```

```python
>>> from sage.all import *
>>> from sage.categories.examples.semigroups_cython import...
    ←LeftZeroSemigroup
>>> S = LeftZeroSemigroup()
>>> S(Integer(2)).is_idempotent()
True
```

super_categories()

EXAMPLES:

```python
sage: from sage.categories.examples.semigroups_cython import...
    ←IdempotentSemigroups
```

(continues on next page)
class `sage.categories.examples.semigroups_cython.LeftZeroSemigroup`

Bases: `LeftZeroSemigroup`

An example of a semigroup

This class illustrates a minimal implementation of a semi-group where the element class is an extension type, and still gets code from the category. The category itself must be a Python class though.

This is purely a proof of concept. The code obviously needs refactorisation!

Comments:

• one cannot play ugly class surgery tricks (as with `_mul_parent`). available operations should really be declared to the coercion model!

EXAMPLES:

```
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup(); S
An example of a semigroup: the left zero semigroup

>>> from sage.all import *
>>> from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
>>> S = LeftZeroSemigroup(); S
An example of a semigroup: the left zero semigroup
```

This is the semigroup which contains all sort of objects:

```
sage: S.some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

```
>>> from sage.all import *
>>> S.some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

with product rule given by \(a \times b = a\) for all \(a, b\).

```
sage: S('hello') * S('world')
'hello'
sage: S(3) * S(1) * S(2)
3
sage: S(3)^12312321312321
3
sage: TestSuite(S).run(verbosity = True)
running ._test_an_element() . . . pass
```

(continues on next page)
That’s really the only method which is obtained from the category …
```python
sage: S(42).is_idempotent
<bound method IdempotentSemigroups.ElementMethods.is_idempotent of 42>

sage: S(42).is_idempotent()
True

sage: S(42)._pow_int
<bound method IdempotentSemigroups.ElementMethods._pow_int of 42>

sage: S(42)^10
42

sage: S(42).is_idempotent
<bound method IdempotentSemigroups.ElementMethods.is_idempotent of 42>

sage: S(42).is_idempotent()
True

>>> from sage.all import *

>>> S(Integer(42)).is_idempotent
<bound method IdempotentSemigroups.ElementMethods.is_idempotent of 42>

>>> S(Integer(42)).is_idempotent()
True

>>> S(Integer(42))._pow_int
<bound method IdempotentSemigroups.ElementMethods._pow_int of 42>

>>> S(Integer(42))**Integer(10)
42

>>> S(Integer(42)).is_idempotent
<bound method IdempotentSemigroups.ElementMethods.is_idempotent of 42>

>>> S(Integer(42)).is_idempotent()
True
```

**Element**

alias of `LeftZeroSemigroupElement`

class `sage.categories.examples.semigroups_cython.LeftZeroSemigroupElement`

Bases: `Element`

EXAMPLES:

```python
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: x = S(3)
sage: TestSuite(x).run()

>>> from sage.all import *

>>> from sage.categories.examples.semigroups_cython import LeftZeroSemigroup

>>> S = LeftZeroSemigroup()

>>> x = S(Integer(3))

>>> TestSuite(x).run()
```
6.28 Examples of sets

```
class sage.categories.examples.sets_cat.PrimeNumbers
    Bases: UniqueRepresentation, Parent

An example of parent in the category of sets: the set of prime numbers.
The elements are represented as plain integers in $\mathbb{Z}$ (façade implementation).

This is a minimal implementation. For more advanced examples of implementations, see also:

```sage
P = Sets().example("facade")
P = Sets().example("inherits")
P = Sets().example("wrapper")
```

EXAMPLES:

```
sage: P = Sets().example()
sage: P(12)
Traceback (most recent call last):
  ...  
AssertionError: 12 is not a prime number
```

```sage
a = P.an_element()
asage: a.parent()
Integer Ring
```

```sage
x = P(13); x
13
```

```sage
type(x)
<class 'sage.rings.integer.Integer'>
```

```sage:x.parent()
Integer Ring
```

```sage: 13 in P
True
```

```sage: 12 in P
False
```

```sage:y = x+1; y
14
```

```sage: type(y)
<class 'sage.rings.integer.Integer'>
```

```sage: TestSuite(P).run(verbosity=True)
```

(continues on next page)
>>> from sage.all import *

>>> P = Sets().example()
>>> P(Integer(12))
Traceback (most recent call last):
  ...  
AssertionError: 12 is not a prime number

>>> a = P.an_element()
>>> a.parent()
Integer Ring

>>> x = P(Integer(13)); x
13
>>> type(x)
<class 'sage.rings.integer.Integer'>

>>> x.parent()
Integer Ring

>>> Integer(13) in P
True

>>> Integer(12) in P
False

>>> y = x+Integer(1); y
14
>>> type(y)
<class 'sage.rings.integer.Integer'>

>>> TestSuite(P).run(verbose=True)
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ... 
Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_nonzero_equal() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass

(continues on next page)
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_some_elements() ... pass

an_element()

Implements Sets.ParentMethods.an_element().

element_class

alias of Integer
class sage.categories.examples.sets_cat.PrimeNumbers_Abstract

Bases: UniqueRepresentation, Parent

This class shows how to write a parent while keeping the choice of the datastructure for the children open. Different
class with fixed datastructure will then be constructed by inheriting from PrimeNumbers_Abstract.

This is used by:
sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: P = Sets().example("wrapper")

>>> from sage.all import *
>>> P = Sets().example("facade")
>>> P = Sets().example("inherits")
>>> P = Sets().example("wrapper")
class Element

Bases: Element

is_prime()

Return whether self is a prime number.

EXAMPLES:
sage: P = Sets().example("inherits")
sage: x = P.an_element()
sage: P.an_element().is_prime()
True

>>> from sage.all import *
>>> P = Sets().example("inherits")
>>> x = P.an_element()
>>> P.an_element().is_prime()
True

next()

Return the next prime number.

EXAMPLES:
sage: P = Sets().example("inherits")
sage: p = P.an_element(); p
47
sage: p.next()
53
```python
>>> from sage.all import *
>>> P = Sets().example("inherits")
>>> p = P.an_element(); p
47
>>> p.next()
53
```

**Note:** This method is not meant to implement the protocol iterator, and thus not subject to Python 2 vs Python 3 incompatibilities.

### an_element()

### next(i)
Return the next prime number.

**EXAMPLES:**

```
sage: P = Sets().example("inherits")
sage: x = P.next(P.an_element()); x
53
sage: x.parent()
Set of prime numbers
```

### some_elements()
Return some prime numbers.

**EXAMPLES:**

```
sage: P = Sets().example("inherits")
sage: P.some_elements()
[47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

### class sage.categories.examples.sets_cat.PrimeNumbers_Facade
Bases: `PrimeNumbers_Abstract`

An example of parent in the category of sets: the set of prime numbers.

In this alternative implementation, the elements are represented as plain integers in \( \mathbb{Z} \) (facade implementation).

**EXAMPLES:**

6.28. Examples of sets
sage: P = Sets().example("facade")
sage: P(12)
Traceback (most recent call last):...
ValueError: 12 is not a prime number
sage: a = P.an_element()
sage: a.parent()
Integer Ring
sage: x = P(13); x
13
sage: type(x)
<class 'sage.rings.integer.Integer'>
sage: x.parent()
Integer Ring
sage: 13 in P
True
sage: 12 in P
False
sage: y = x+1; y
14
sage: type(y)
<class 'sage.rings.integer.Integer'>
sage: z = P.next(x); z
17
sage: type(z)
<class 'sage.rings.integer.Integer'>
>>> from sage.all import *
>>> P = Sets().example("facade")
>>> P(Integer(12))
Traceback (most recent call last):...
ValueError: 12 is not a prime number
>>> a = P.an_element()
>>> a.parent()
Integer Ring
>>> x = P(Integer(13)); x
13
>>> type(x)
<class 'sage.rings.integer.Integer'>
>>> x.parent()
Integer Ring
>>> Integer(13) in P
True
>>> Integer(12) in P
False
>>> y = x+Integer(1); y
14
>>> type(y)
<class 'sage.rings.integer.Integer'>
>>> z = P.next(x); z
17
>>> type(z)
(continues on next page)
The disadvantage of this implementation is that the elements do not know that they are prime, so that prime testing is slow:

```python
sage: pf = Sets().example("facade").an_element()
sage: timeit("pf.is_prime()") # random
625 loops, best of 3: 4.1 us per loop

>>> from sage.all import *
>>> pf = Sets().example("facade").an_element()
>>> timeit("pf.is_prime()") # random
625 loops, best of 3: 4.1 us per loop
```

compared to the other implementations where prime testing is only done if needed during the construction of the element, and later on the elements “know” that they are prime:

```python
sage: pw = Sets().example("wrapper").an_element()
sage: timeit("pw.is_prime()") # random
625 loops, best of 3: 859 ns per loop

sage: pi = Sets().example("inherits").an_element()
sage: timeit("pw.is_prime()") # random
625 loops, best of 3: 854 ns per loop

>>> from sage.all import *
>>> pw = Sets().example("wrapper").an_element()
>>> timeit("pw.is_prime()") # random
625 loops, best of 3: 859 ns per loop

>>> pi = Sets().example("inherits").an_element()
>>> timeit("pw.is_prime()") # random
625 loops, best of 3: 854 ns per loop
```

Note also that the next method for the elements does not exist:

```python
sage: pf.next()
Traceback (most recent call last):
  ... AttributeError: 'sage.rings.integer.Integer' object has no attribute 'next'
```

```python
>>> from sage.all import *
>>> pf.next()
Traceback (most recent call last):
  ... AttributeError: 'sage.rings.integer.Integer' object has no attribute 'next'
```

unlike in the other implementations:

```python
sage: pw.next()
53
sage: pi.next()
53
```
>>> from sage.all import *
>>> pw.next()
53
>>> pi.next()
53

element_class
    alias of Integer
class sage.categories.examples.sets_cat.PrimeNumbers_Inherits
    Bases: PrimeNumbers_Abstract

An example of parent in the category of sets: the set of prime numbers. In this implementation, the element are stored as object of a new class which inherits from the class Integer (technically IntegerWrapper).

EXAMPLES:

```
sage: P = Sets().example("inherits")
sage: P
Set of prime numbers
sage: P(12)
Traceback (most recent call last):
  ...
ValueError: 12 is not a prime number
sage: a = P.an_element()
sage: a
Set of prime numbers
sage: x = P(13); x
13
sage: x.is_prime()
True
sage: type(x)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'>
sage: x.parent()
Set of prime numbers
sage: P(13) in P
True
sage: y = x+1; y
14
sage: type(y)
<class 'sage.rings.integer.Integer'>
sage: y.parent()
Integer Ring
sage: type(P(13)+P(17))
<class 'sage.rings.integer.Integer'>
sage: type(P(2)+P(3))
<class 'sage.rings.integer.Integer'>
sage: z = P.next(x); z
17
sage: type(z)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'>
sage: z.parent()
Set of prime numbers
sage: TestSuite(P).run(Verbose=True)
```
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_construction() ... pass
running ._test_elements() ...
    Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_elements_neq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_some_elements() ... pass

>>> from sage.all import *
>>> P = Sets().example("inherits")
>>> P
Set of prime numbers
>>> P(Integer(12))
Traceback (most recent call last):
  ...
ValueError: 12 is not a prime number
>>> a = P.an_element()
>>> a.parent()
Set of prime numbers
>>> x = P(Integer(13)); x
13
>>> x.is_prime()
True
>>> type(x)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.
    element_class'>
>>> x.parent()
Set of prime numbers
>>> P(Integer(13)) in P
True
>>> y = x+Integer(1); y
14
>>> type(y)
<class 'sage.rings.integer.Integer'>
>>> y.parent()
Integer Ring
>>> type(P(Integer(13))+P(Integer(17)))
<class 'sage.rings.integer.Integer'>
>>> type(P(Integer(2))+P(Integer(3)))
<class 'sage.rings.integer.Integer'>
>>> z = P.next(x); z
(continues on next page)
```python
>>> type(z)
class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'

>>> z.parent()
Set of prime numbers

>>> TestSuite(P).run(verbosetrue=True)
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_construction() . . . pass
running ._test_elements() . . .
   Running the test suite of self.an_element()
   running ._test_category() . . . pass
   running ._test_eq() . . . pass
   running ._test_new() . . . pass
   running ._test_not_implemented_methods() . . . pass
   running ._test_pickling() . . . pass
   running ._test_elements_eq_reflexive() . . . pass
   running ._test_elements_eq_symmetric() . . . pass
   running ._test_elements_eq_transitive() . . . pass
   running ._test_elements_neq() . . . pass
   running ._test_eq() . . . pass
   running ._test_new() . . . pass
   running ._test_not_implemented_methods() . . . pass
   running ._test_pickling() . . . pass
   running ._test_some_elements() . . . pass
```

See also:

```python
sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: P = Sets().example("wrapper")

>>> from sage.all import *

>>> P = Sets().example("facade")
>>> P = Sets().example("inherits")
>>> P = Sets().example("wrapper")
```

class Element (parent, p)
Bases: IntegerWrapper, Element

class sage.categories.examples.sets_cat.PrimeNumbers_Wrapper
Bases: PrimeNumbers_Abstract

An example of parent in the category of sets: the set of prime numbers.

In this second alternative implementation, the prime integer are stored as a attribute of a sage object by inheriting from ElementWrapper. In this case we need to ensure conversion and coercion from this parent and its element to ZZ and Integer.

EXAMPLES:

```python
sage: P = Sets().example("wrapper")
sage: P(12)
```

(continues on next page)
Traceback (most recent call last):
...
ValueError: 12 is not a prime number
sage: a = P.an_element()
set of prime numbers (wrapper implementation)
sage: x = P(13); x
13
sage: type(x)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Wrapper_with_category.element_class'>
sage: x.parent()
set of prime numbers (wrapper implementation)
sage: 13 in P
True
sage: 12 in P
False
sage: y = x+1; y
14
sage: type(y)
<class 'sage.rings.integer.Integer'>
sage: z = P.next(x); z
17
sage: type(z)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Wrapper_with_category.element_class'>
sage: z.parent()
set of prime numbers (wrapper implementation)

>>> from sage.all import *
>>> P = Sets().example("wrapper")
>>> P(Integer(12))
Traceback (most recent call last):
...
ValueError: 12 is not a prime number
>>> a = P.an_element()
>>> a.parent()
set of prime numbers (wrapper implementation)
>>> x = P(Integer(13)); x
13
>>> type(x)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Wrapper_with_category.element_class'>
>>> x.parent()
set of prime numbers (wrapper implementation)
>>> Integer(13) in P
True
>>> Integer(12) in P
False
>>> y = x+Integer(1); y
14
>>> type(y)
<class 'sage.rings.integer.Integer'>
>>> z = P.next(x); z

17
>>> type(z)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Wrapper_with_category.
˓→element_class'>
>>> z.parent()
Set of prime numbers (wrapper implementation)

class Element
    Bases: ElementWrapper, Element

ElementWrapper
    alias of ElementWrapper

6.29 Example of a set with grading

classes ...

EXAMPLES:

    sage: E = SetsWithGrading().example(); E
    Non negative integers
    sage: E in Sets().Infinite()
    True
    sage: E.graded_component(0)
    {0}
    sage: E.graded_component(100)
    {100}

an_element()
    Return 0.

EXAMPLES:

    sage: SetsWithGrading().example().an_element()
    0
>>> from sage.all import *
>>> SetsWithGrading().example().an_element()
0

generating_series(var='z')

Return $1/(1-z)$.

EXAMPLES:

```python
sage: N = SetsWithGrading().example(); N
Non negative integers
sage: f = N.generating_series(); f
1/(-z + 1)
sage: LaurentSeriesRing(ZZ,'z')(f)
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + z^7 + z^8 + z^9 + z^10 + z^11 + z^12 + ...
   -z^13 + z^14 + z^15 + z^16 + z^17 + z^18 + z^19 + O(z^20)
```

graded_component(grade)

Return the component with grade `grade`.

EXAMPLES:

```python
sage: N = SetsWithGrading().example()
sage: N.graded_component(65)
{65}
```

grading(elt)

Return the grade of `elt`.

EXAMPLES:

```python
sage: N = SetsWithGrading().example()
sage: N.grading(10)
10
```

6.29. Example of a set with grading
6.30 Examples of parents endowed with multiple realizations

\texttt{class sage.categories.examples.with_realizations.SubsetAlgebra (R, S)}

Bases: \texttt{UniqueRepresentation,Parent}

An example of parent endowed with several realizations

We consider an algebra $A(S)$ whose bases are indexed by the subsets $s$ of a given set $S$. We consider three natural basis of this algebra: $F$, $\text{In}$, and $\text{Out}$. In the first basis, the product is given by the union of the indexing sets. That is, for any $s, t \subseteq S$

$$F_s F_t = F_{s \cup t}$$

The $\text{In}$ basis and $\text{Out}$ basis are defined respectively by:

$$\text{In}_s = \sum_{t \supseteq s} F_t \quad \text{and} \quad F_s = \sum_{t \subseteq s} \text{Out}_t$$

Each such basis gives a realization of $A$, where the elements are represented by their expansion in this basis.

This parent, and its code, demonstrate how to implement this algebra and its three realizations, with coercions and mixed arithmetic between them.

See also:

• \texttt{Sets().WithRealizations}
• the \texttt{Implementing Algebraic Structures} thematic tutorial.

EXAMPLES:

\texttt{sage: A = Sets().WithRealizations().example(); A}
The subset algebra of $\{1, 2, 3\}$ over Rational Field
\texttt{sage: A.base_ring()}
Rational Field

\begin{verbatim}
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
The subset algebra of $\{1, 2, 3\}$ over Rational Field
>>> A.base_ring()
Rational Field
\end{verbatim}

The three bases of $A$:

\texttt{sage: F = A.F(); F}
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the Fundamental basis
\texttt{sage: In = A.In(); In}
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the In basis
\texttt{sage: Out = A.Out(); Out}
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the Out basis

\begin{verbatim}
>>> from sage.all import *
>>> F = A.F(); F
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the Fundamental basis
>>> In = A.In(); In
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the In basis
>>> Out = A.Out(); Out
The subset algebra of $\{1, 2, 3\}$ over Rational Field in the Out basis
\end{verbatim}
One can quickly define all the bases using the following shortcut:

```python
sage: A.inject_shorthands()
Defining F as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the Fundamental basis
Defining In as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the In basis
Defining Out as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the Out basis

>>> from sage.all import *
>>> A.inject_shorthands()
Defining F as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the Fundamental basis
Defining In as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the In basis
Defining Out as shorthand for The subset algebra of {1, 2, 3} over Rational Field.
in the Out basis

Accessing the basis elements is done with `basis()` method:

```python
sage: F.basis().list()
[F[{}], F[{1}], F[{2}], F[{3}], F[{1, 2}], F[{1, 3}], F[{2, 3}], F[{1, 2, 3}]]

```python
>>> from sage.all import *
```python
>>> F.basis().list()
[F[{}], F[{1}], F[{2}], F[{3}], F[{1, 2}], F[{1, 3}], F[{2, 3}], F[{1, 2, 3}]]

To access a particular basis element, you can use the `from_set()` method:

```python
sage: F.from_set(2,3)
F[{2, 3}]
sage: In.from_set(1,3)
In[{1, 3}]

```python
>>> from sage.all import *
```python
>>> F.from_set(Integer(2),Integer(3))
F[{2, 3}]
```python
>>> In.from_set(Integer(1),Integer(3))
In[{1, 3}]

or as a convenient shorthand, one can use the following notation:

```python
sage: F[2,3]
F[{2, 3}]
sage: In[1,3]
In[{1, 3}]

```python
>>> from sage.all import *
```python
>>> F[Integer(2),Integer(3)]
F[{2, 3}]
```python
>>> In[Integer(1),Integer(3)]
In[{1, 3}]

Some conversions:
We can now mix expressions:

```python
sage: (1 + Out[1]) * In[2,3]
    4*Out[2,3] + 4*Out[1,2,3]
```

```python
>>> from sage.all import *
>>> (Integer(1) + Out[Integer(1)]) * In[Integer(2),Integer(3)]
    4*Out[2,3] + 4*Out[1,2,3]
```

class Bases (parent_with_realization)

The category of the realizations of the subset algebra

class ParentMethods

Bases: object

from_set (*args)

Construct the monomial indexed by the set containing the elements passed as arguments.

EXAMPLES:

```python
sage: In = Sets().WithRealizations().example().In(); In
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: In.from_set(2,3)
In[2, 3]
```

Chapter 6. Examples of parents using categories
As a shorthand, one can construct elements using the following notation:

```
sage: In[2,3]
In[{2, 3}]
```

```
>>> from sage.all import *

>>> In[Integer(2),Integer(3)]
In[{2, 3}]
```

```
one()
```

Returns the unit of this algebra.

This default implementation takes the unit in the fundamental basis, and coerces it in `self`.

**EXAMPLES:**

```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: In = A.In(); Out = A.Out()
sage: In.one()
In[{}]
sage: Out.one()
Out[{}] + Out[{1}] + Out[{2}] + Out[{3}] + Out[{1, 2}] + Out[{1, 3}] +
--+Out[{2, 3}] + Out[{1, 2, 3}]
```

```
super_categories()
```

**EXAMPLES:**

```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: C = A.Bases(); C
Category of bases of The subset algebra of {1, 2, 3} over Rational Field
sage: C.super_categories()
[Category of realizations of The subset algebra of {1, 2, 3} over Rational Field,
 Join of Category of algebras with basis over Rational Field and
 Category of commutative algebras over Rational Field and
 Category of realizations of unital magmas]
```

6.30. Examples of parents endowed with multiple realizations
Category Framework, Release 10.4

```python
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
>>> C = A.Bases(); C
Category of bases of The subset algebra of {1, 2, 3} over Rational Field
>>> C.super_categories()
[Category of realizations of The subset algebra of {1, 2, 3} over Rational Field, Join of Category of algebras with basis over Rational Field and Category of commutative algebras over Rational Field and Category of realizations of unital magmas]
```

\( F \)
alias of \( \text{Fundamental} \)

class \( \text{Fundamental} \)(\( A \))

Bases: \( \text{CombinatorialFreeModule}, \text{BindableClass} \)

The Subset algebra, in the fundamental basis

INPUT:

\( \bullet \) \( A \) – a parent with realization in \( \text{SubsetAlgebra} \)

EXAMPLES:

```python
sage: A = Sets().WithRealizations().example()
sage: A.F()
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
sage: A.Fundamental()
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
```

\( \text{one}() \)

Return the multiplicative unit element.

EXAMPLES:

```python
sage: A = AlgebrasWithBasis(QQ).example()
# needs sage.combinat sage.modules
sage: A.one_basis()
# needs sage.combinat sage.modules
word:
sage: A.one()
# needs sage.combinat sage.modules
B[word: ]
```

(continues on next page)
one_basis()

Returns the index of the basis element which is equal to ‘1’.

EXAMPLES:

```python
sage: F = Sets().WithRealizations().example().F(); F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental
  basis
sage: F.one_basis()
{}
sage: F.one()
F[{}]
```

product_on_basis(left, right)

Product of basis elements, as per `AlgebrasWithBasis.ParentMethods.product_on_basis()`.

INPUT:
- `left, right` – sets indexing basis elements

EXAMPLES:

```python
sage: F = Sets().WithRealizations().example().F(); F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental
  basis
sage: S = F.basis().keys(); S
Subsets of {1, 2, 3}
sage: F.product_on_basis(S([[]]), S([[]]))
F[{}]
sage: F.product_on_basis(S([1]), S([3]))
F[{1, 3}]
sage: F.product_on_basis(S([1,2]), S([2,3]))
F[{1, 2, 3}]
```

```python
>>> from sage.all import *
>>> F = Sets().WithRealizations().example().F(); F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental
  basis
>>> F.one_basis()
{}
>>> F.one()
F[{}]
```

```python
>>> from sage.all import *
>>> F = Sets().WithRealizations().example().F(); F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental
  basis
>>> S = F.basis().keys(); S
Subsets of {1, 2, 3}
>>> F.product_on_basis(S([]), S([]))
F[{}]
>>> F.product_on_basis(S([1]), S([3]))
F[{1, 3}]
>>> F.product_on_basis(S([1,2]), S([2,3]))
F[{1, 2, 3}]
```
class In(A)

Bases: CombinatorialFreeModule, BindableClass

The Subset Algebra, in the In basis

INPUT:

• A – a parent with realization in SubsetAlgebra

EXAMPLES:

sage: A = Sets().WithRealizations().example()
sage: A.In()
The subset algebra of {1, 2, 3} over Rational Field in the In basis

>>> from sage.all import *
>>> A = Sets().WithRealizations().example()
>>> A.In()
The subset algebra of {1, 2, 3} over Rational Field in the In basis

class Out(A)

Bases: CombinatorialFreeModule, BindableClass

The Subset Algebra, in the Out basis

INPUT:

• A – a parent with realization in SubsetAlgebra

EXAMPLES:

sage: A = Sets().WithRealizations().example()
sage: A.Out()
The subset algebra of {1, 2, 3} over Rational Field in the Out basis

>>> from sage.all import *
>>> A = Sets().WithRealizations().example()
>>> A.Out()
The subset algebra of {1, 2, 3} over Rational Field in the Out basis

a_realization()

Returns the default realization of self

EXAMPLES:

sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: A.a_realization()
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis

>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A

(continues on next page)
The subset algebra of \{1, 2, 3\} over Rational Field

```python
>>> A.a_realization()
```

The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis

**base_set()**

**EXAMPLES:**

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.base_set()
{1, 2, 3}
```

```python
>>> from sage.all import *
```

```python
>>> A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
>>> A.base_set()
{1, 2, 3}
```

**indices()**

The objects that index the basis elements of this algebra.

**EXAMPLES:**

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.indices()
Subsets of \{1, 2, 3\}
```

```python
>>> from sage.all import *
```

```python
>>> A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
>>> A.indices()
Subsets of \{1, 2, 3\}
```

**indices_key(x)**

A key function on a set which gives a linear extension of the inclusion order.

**INPUT:**

- `x` - set

**EXAMPLES:**

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
sage: sorted(A.indices(), key=A.indices_key)
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

```python
>>> from sage.all import *
```

```python
>>> A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
>>> A.indices()
Subsets of \{1, 2, 3\}
```

```python
>>> from sage.all import *
```

```python
>>> A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
>>> A.indices()
Subsets of \{1, 2, 3\}
```

**supsets(set)**

Returns all the subsets of \( S \) containing \( set \).
INPUT:

• set – a subset of the base set \( S \) of self

EXAMPLES:

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.supsets(Set((2,)))
\{\{1, 2, 3\}, \{2, 3\}, \{1, 2\}, \{2\}}
```

```python
>>> from sage.all import *
>>> A = Sets().WithRealizations().example(); A
The subset algebra of \{1, 2, 3\} over Rational Field
>>> A.supsets(Set((Integer(2),)))
\{\{1, 2, 3\}, \{2, 3\}, \{1, 2\}, \{2\}}
```
7.1 Specific category classes

This is placed in a separate file from categories.py to avoid circular imports (as morphisms must be very low in the hierarchy with the new coercion model).

```python
class sage.categories.category_types.AbelianCategory
    Bases: Category
    is_abelian()
    Return True as self is an abelian category.
    EXAMPLES:
    sage: CommutativeAdditiveGroups().is_abelian()
    True
    >>> from sage.all import *
    >>> CommutativeAdditiveGroups().is_abelian()
    True

class sage.categories.category_types.Category_ideal(ambient, name=None)
    Bases: Category_in_ambient
    classmethod an_instance()
    Return an instance of this class.
    EXAMPLES:
    sage: AlgebraIdeals.an_instance()
    Category of algebra ideals in Univariate Polynomial Ring in x over Rational
    Field
    >>> from sage.all import *
    >>> AlgebraIdeals.an_instance()
    Category of algebra ideals in Univariate Polynomial Ring in x over Rational
    Field
	ring()
    Return the ambient ring used to describe objects self.
    EXAMPLES:
```
```python
sage: C = Ideals(IntegerRing())
sage: C.ring()
Integer Ring

>>> from sage.all import *
>>> C = Ideals(IntegerRing())
>>> C.ring()
Integer Ring
```

class sage.categories.category_types.Category_in_ambient(ambient, name=None)

Bases: Category

Initialize self.

The parameter name is ignored.

EXAMPLES:

```python
sage: C = Ideals(IntegerRing())
sage: TestSuite(C).run()
```

ambient()

Return the ambient object in which objects of this category are embedded.

EXAMPLES:

```python
sage: C = Ideals(IntegerRing())
sage: C.ambient()
Integer Ring
```

class sage.categories.category_types.Category_module(base, name=None)

Bases: AbelianCategory, Category_over_base_ring

Ambient category of objects of base

class sage.categories.category_types.Category_over_base(base, name=None)

Bases: CategoryWithParameters

A base class for categories over some base object

INPUT:

- base -- a category \( C \) or an object of such a category

Assumption: the classes for the parents, elements, morphisms, of self should only depend on \( C \). See Issue #11935 for details.

EXAMPLES:
```python
sage: Algebras(GF(2)).element_class is Algebras(GF(3)).element_class
True
sage: C = GF(2).category()
sage: Algebras(GF(2)).parent_class is Algebras(C).parent_class
True
sage: C = ZZ.category()
sage: Algebras(ZZ).element_class is Algebras(C).element_class
True

>>> from sage.all import *

>>> Algebras(GF(Integer(2))).element_class is Algebras(GF(Integer(3))).element_class
True

>>> C = GF(Integer(2)).category()

>>> Algebras(GF(Integer(2))).parent_class is Algebras(C).parent_class
True

>>> C = ZZ.category()

>>> Algebras(ZZ).element_class is Algebras(C).element_class
True
```

**classmethod an_instance()**

Returns an instance of this class

**EXAMPLES:**

```python
sage: Algebras.an_instance()
Category of algebras over Rational Field

>>> from sage.all import *

>>> Algebras.an_instance()
Category of algebras over Rational Field
```

**base()**

Return the base over which elements of this category are defined.

**EXAMPLES:**

```python
sage: C = Algebras(QQ)
sage: C.base()
Rational Field

>>> from sage.all import *

>>> C = Algebras(QQ)

>>> C.base()
Rational Field
```

**class** `sage.categories.category_types.Category_over_base_ring` *(base, name=None)*

**Bases:** `Category_over_base`

Initialize `self`

**EXAMPLES:**

7.1. Specific category classes 1489
sage: C = Algebras(GF(2)); C
Category of algebras over Finite Field of size 2
sage: TestSuite(C).run()

>>> from sage.all import *
>>> C = Algebras(GF(Integer(2))); C
Category of algebras over Finite Field of size 2
>>> TestSuite(C).run()

**base_ring()**

Return the base ring over which elements of this category are defined.

EXAMPLES:

```
sage: C = Algebras(GF(2))
sage: C.base_ring()
Finite Field of size 2
```

```python
>>> from sage.all import *
>>> C = Algebras(GF(Integer(2)))
```

```
>>> C.base_ring()
Finite Field of size 2
```

**class sage.categories.category_types.**

```
class sage.categories.category_types.Elements(object)

Bases: Category

The category of all elements of a given parent.

EXAMPLES:

```
sage: a = IntegerRing()(5)
sage: C = a.category(); C
Category of elements of Integer Ring
sage: a in C
True
sage: 2/3 in C
False
sage: loads(C.dumps()) == C
True
```

```python
>>> from sage.all import *
>>> a = IntegerRing()(Integer(5))
```

```
>>> C = a.category(); C
Category of elements of Integer Ring
>>> a in C
True
>>> Integer(2)/Integer(3) in C
False
>>> loads(C.dumps()) == C
True
```

**classmethod an_instance()**

Returns an instance of this class

EXAMPLES:
7.2 Singleton categories

class
sage.categories.category_singleton.Category_contains_method_by_parent_class

Bases: object

Returns whether \( x \) is an object in this category.

More specifically, returns \( \text{True} \) if and only if \( x \) has a category which is a subcategory of this one.

EXAMPLES:

\[
\begin{align*}
\textit{sage}: & \text{ ZZ in Sets() } \\
& \text{ True }
\end{align*}
\]

\[
\begin{align*}
\text{\textgreater\textgreater\textgreater from sage.all import } & * \\
\text{\textgreater\textgreater Elements.an_instance() } \\
& \text{Category of elements of Rational Field }
\end{align*}
\]

\[
\begin{align*}
\textit{sage}: & \text{ Elements(ZZ).object() } \\
& \text{Integer Ring }
\end{align*}
\]

\[
\begin{align*}
\text{\textgreater\textgreater from sage.all import } & * \\
\text{\textgreater\textgreater Elements(ZZ).object() } \\
& \text{Integer Ring }
\end{align*}
\]

\[
\begin{align*}
\textit{sage}: & \text{ Elements(ZZ).super_categories() } \\
& [\text{Category of objects}] \\
\end{align*}
\]

\[
\begin{align*}
\text{\textgreater\textgreater from sage.all import } & * \\
\text{\textgreater\textgreater Elements(ZZ).super_categories() } \\
& [\text{Category of objects}] \\
\end{align*}
\]

**Todo:** Check that this is what we want.

7.2 Singleton categories

class
sage.categories.category_singleton.Category_singleton

Bases: Category

A base class for implementing singleton category
A singleton category is a category whose class takes no parameters like \texttt{Fields()} or \texttt{Rings()}. See also the Singleton design pattern.

This is a subclass of \texttt{Category}, with a couple optimizations for singleton categories.

The main purpose is to make the idioms:

\begin{verbatim}
sage: QQ in Fields()
True
sage: ZZ in Fields()
False
\end{verbatim}

as fast as possible, and in particular competitive to calling a constant Python method, in order to foster its systematic use throughout the Sage library. Such tests are time critical, in particular when creating a lot of polynomial rings over small fields like in the elliptic curve code.

EXAMPLES:

\begin{verbatim}
sage: from sage.categories.category_singleton import Category_singleton
class MyRings(Category):
    def super_categories(self):
        return Rings().super_categories()
sage: class MyRingsSingleton(Category_singleton):
    def super_categories(self):
        return Rings().super_categories()

>>> from sage.all import *
>>> QQ in Fields()
True
>>> ZZ in Fields()
False
\end{verbatim}

We create three rings. One of them is contained in the usual category of rings, one in the category of “my rings” and the third in the category of “my rings singleton”:

\begin{verbatim}
sage: R = QQ['x,y']
sage: R1 = Parent(category = MyRings())
sage: R2 = Parent(category = MyRingsSingleton())
sage: R in MyRings()
False
sage: R1 in MyRings()
True
sage: R1 in MyRingsSingleton()
False
sage: R2 in MyRings()
False
sage: R2 in MyRingsSingleton()
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> R = QQ['x,y']
>>> R1 = Parent(category = MyRings())
\end{verbatim}

(continues on next page)
One sees that containment tests for the singleton class is a lot faster than for a usual class:

```
sage: # not tested
timeit("R in MyRings()", number=10000)
10000 loops, best of 3: 7.12 µs per loop
```

```
sage: timeit("R1 in MyRings()", number=10000)
10000 loops, best of 3: 6.98 µs per loop
```

```
sage: timeit("R in MyRingsSingleton()", number=10000)
10000 loops, best of 3: 3.08 µs per loop
```

```
sage: timeit("R2 in MyRingsSingleton()", number=10000)
10000 loops, best of 3: 2.99 µs per loop
```

So this is an improvement, but not yet competitive with a pure Cython method:

```
sage: timeit("R.is_ring()", number=10000)  # not tested
10000 loops, best of 3: 383 ns per loop
```

```
sage: timeit("R.is_ring()", number=Integer(10000))  # not tested
10000 loops, best of 3: 383 ns per loop
```

However, it is competitive with a Python method. Actually it is faster, if one stores the category in a variable:

```
sage: _Rings = Rings()
sage: R3 = Parent(category = _Rings)
sage: R3.is_ring.__module__
'sage.categories.rings'
sage: timeit("R3.is_ring()", number=10000)  # not tested
10000 loops, best of 3: 2.64 µs per loop
```

```
sage: timeit("R3 in Rings()", number=10000)  # not tested
10000 loops, best of 3: 3.01 µs per loop
```

```
sage: timeit("R3 in _Rings", number=10000)  # not tested
10000 loops, best of 3: 652 ns per loop
```

7.2. Singleton categories
>>> from sage.all import *
>>> _Rings = Rings()
>>> R3 = Parent(category = _Rings)
>>> R3.is_ring.__module__  
'sage.categories.rings'
>>> %timeit "R3.is_ring()", number=Integer(10000)  
# not tested
10000 loops, best of 3: 2.64 µs per loop
>>> %timeit "R3 in Rings()", number=Integer(10000)  
# not tested
10000 loops, best of 3: 3.01 µs per loop
>>> %timeit "R3 in _Rings", number=Integer(10000)  
# not tested
10000 loops, best of 3: 652 ns per loop

This might not be easy to further optimize, since the time is consumed in many different spots:

sage: %timeit "MyRingsSingleton.__classcall__()", number=10000)  
# not tested
10000 loops, best of 3: 306 ns per loop
sage: X = MyRingsSingleton()
  
sage: timeit("R in X ", number=10000)  
# not tested
10000 loops, best of 3: 699 ns per loop
sage: c = MyRingsSingleton().__contains__
  
sage: timeit("c(R)", number = 10000)  
# not tested
10000 loops, best of 3: 661 ns per loop

>>> from sage.all import *
>>> %timeit "MyRingsSingleton.__classcall__()", number=Integer(10000))  
# not tested
10000 loops, best of 3: 306 ns per loop

>>> X = MyRingsSingleton()

>>> %timeit("R in X ", number=Integer(10000))  
# not tested
10000 loops, best of 3: 699 ns per loop

>>> c = MyRingsSingleton().__contains__

>>> %timeit("c(R)", number = Integer(10000))  
# not tested
10000 loops, best of 3: 661 ns per loop

Warning: A singleton concrete class \( A \) should not have a subclass \( B \) (necessarily concrete). Otherwise, creating an instance \( a \) of \( A \) and an instance \( b \) of \( B \) would break the singleton principle: \( A \) would have two instances \( a \) and \( b \).

With the current implementation only direct subclasses of \texttt{Category\_singleton} are supported:

```python
sage: class MyRingsSingleton(Category_singleton):
    ....:    def super_categories(self): return Rings().super_categories()
sage: class Disaster(MyRingsSingleton): pass
sage: Disaster()
Traceback (most recent call last):
...
AssertionError: <class '__main__.Disaster'> is not a direct subclass of <class '
'sage.categories.category_singleton.Category\_singleton'>
```
However, it is acceptable for a direct subclass $R$ of $\texttt{Category\_singleton}$ to create its unique instance as an instance of a subclass of itself (in which case, its the subclass of $R$ which is concrete, not $R$ itself). This is used for example to plug in extra category code via a dynamic subclass:

```
from sage.categories.category_singleton import Category_singleton

sage: class R(Category_singleton):
    ... def super_categories(self): return [Sets()]

sage: R()
Category of r
```

```
R() is R()
True
```

```
from sage.all import *

from sage.categories.category_singleton import Category_singleton

class R(Category_singleton):
    ... def super_categories(self): return [Sets()]

R() is R().__class__
True
```

```
7.2. Singleton categories
1495
```
In that case, \( R \) is an abstract class and has a single concrete subclass, so this does not break the Singleton design pattern.

See also:

```
Category.__classcall__(), Category.__init__()```

**Note:** The \_test_category test is failing because \textsc{MyRingsSingleton}() is not a subcategory of the join of its super categories:

```sage
sage: C = MyRingsSingleton()
sage: C.super_categories()
[Category of rngs, Category of semirings]
sage: Rngs() & Semirings()
Category of rings
sage: C.is_subcategory(Rings())
False
```

```sage
>>> from sage.all import *

>>> C = MyRingsSingleton()

>>> C.super_categories()
[Category of rngs, Category of semirings]

>>> Rngs() & Semirings()
Category of rings

>>> C.is_subcategory(Rings())
False
```

Oh well; it's not really relevant for those tests.

### 7.3 Fast functions for the category framework

**AUTHOR:**

- Simon King (initial version)

```
class sage.categories.category_cy_helper.AxiomContainer
    Bases: dict

    A fast container for axioms.
```

This is derived from \texttt{dict}. A key is the name of an axiom. The corresponding value is the “rank” of this axiom, that is used to order the axioms in \texttt{canonicalize_axioms}().
EXAMPLES:

```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: isinstance(all_axioms, sage.categories.category_with_axiom.AxiomContainer)
True
```

```python
>>> from sage.all import *
>>> all_axioms = sage.categories.category_with_axiom.all_axioms
>>> isinstance(all_axioms, sage.categories.category_with_axiom.AxiomContainer)
True
```

**add(axiom)**

Add a new axiom name, of the next rank.

**EXAMPLES:**

```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: m = max(all_axioms.values())
sage: all_axioms.add('Awesome')
sage: all_axioms['Awesome'] == m + 1
True
```

```python
>>> from sage.all import *
>>> all_axioms = sage.categories.category_with_axiom.all_axioms
>>> m = max(all_axioms.values())
>>> all_axioms.add('Awesome')
>>> all_axioms['Awesome'] == m + Integer(1)
True
```

To avoid side effects, we remove the added axiom:

```python
sage: del all_axioms['Awesome']
```

```python
>>> from sage.all import *
>>> del all_axioms['Awesome']
```

`sage.categories.category_cy_helper.canonicalize_axioms(all_axioms, axioms)`

Canonicalize a set of axioms.

**INPUT:**

- `all_axioms` – all available axioms
- `axioms` – a set (or iterable) of axioms

**Note:** AxiomContainer provides a fast container for axioms, and the collection of axioms is stored in `sage.categories.category_with_axiom`. In order to avoid circular imports, we expect that the collection of all axioms is provided as an argument to this auxiliary function.

**OUTPUT:**

A set of axioms as a tuple sorted according to the order of the tuple `all_axioms` in `sage.categories.category_with_axiom`.

**EXAMPLES:**
Category Framework, Release 10.4

```python
sage: from sage.categories.category_with_axiom import canonicalize_axioms, all_axioms
sage: canonicalize_axioms(all_axioms, ['Commutative', 'Connected', 'WithBasis', 'Finite'])
('Finite', 'Connected', 'WithBasis', 'Commutative')
sage: canonicalize_axioms(all_axioms, ['Commutative', 'Connected', 'Commutative', 'WithBasis', 'Finite'])
('Finite', 'Connected', 'WithBasis', 'Commutative')
```

```python
category_sort_key(category)
Return category._cmp_key.
This helper function is used for sorting lists of categories.
It is semantically equivalent to operator.attrgetter("_cmp_key"), but currently faster.
```

```python
sage: from sage.categories.category_cy_helper import category_sort_key
sage: category_sort_key(Rings()) is Rings()._cmp_key
True
```

```python
category_sort_key(Rings()) is Rings()._cmp_key
True
```

```python
get_axiom_index(all_axioms, axiom)
Helper function: Return the rank of an axiom.
```

```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: from sage.categories.category_cy_helper import get_axiom_index
sage: get_axiom_index(all_axioms, 'AdditiveCommutative') == all_axioms['AdditiveCommutative']
True
```

```python
>>> from sage.all import *
>>> from sage.categories.category_cy_helper import category_sort_key
>>> category_sort_key(Rings()) is Rings()._cmp_key
True
```

```python
>>> from sage.all import *
>>> all_axioms = sage.categories.category_with_axiom.all_axioms
>>> from sage.categories.category_cy_helper import get_axiom_index
>>> get_axiom_index(all_axioms, 'AdditiveCommutative') == all_axioms['AdditiveCommutative']
True
```
sage.categories.category_cy_helper.join_as_tuple\( (\text{categories, axioms, ignore\_axioms}) \)

Helper for \texttt{join()}.

INPUT:

- \texttt{categories} – tuple of categories to be joined,
- \texttt{axioms} – tuple of strings; the names of some supplementary axioms.
- \texttt{ignore\_axioms} – tuple of pairs (\texttt{cat, axiom}), such that \texttt{axiom} will not be applied to \texttt{cat}, should \texttt{cat} occur in the algorithm.

EXAMPLES:

\begin{Verbatim}
\texttt{sage: from sage.categories.category_cy_helper import join\_as\_tuple}
\texttt{sage: T = (Coalgebras(QQ), Sets().Finite(), Algebras(ZZ), SimplicialComplexes())}
\texttt{sage: join\_as\_tuple(T,(),())}
(Category of algebras over Integer Ring,}
\texttt{Category of finite monoids,}
\texttt{Category of finite additive groups,}
\texttt{Category of coalgebras over Rational Field,}
\texttt{Category of finite simplicial complexes)}
\texttt{sage: join\_as\_tuple(T,('WithBasis',),())}
\texttt{(Category of algebras with basis over Integer Ring,}
\texttt{Category of finite monoids,}
\texttt{Category of coalgebras with basis over Rational Field,}
\texttt{Category of finite additive groups,}
\texttt{Category of finite simplicial complexes)}
\texttt{sage: join\_as\_tuple(T,(),((Monoids(),'Finite'),))}
\texttt{(Category of algebras over Integer Ring,}
\texttt{Category of finite additive groups,}
\texttt{Category of coalgebras over Rational Field,}
\texttt{Category of finite simplicial complexes)}
\end{Verbatim}
### 7.4 Coercion methods for categories

The purpose of this Cython module is to hold special coercion methods, which are inserted by their respective categories.

### 7.5 Poor Man’s map

```python
class sage.categories.poor_man_map.PoorManMap(function, domain=None, codomain=None, name=None)
```

**Bases:** `SageObject`

A class for maps between sets which are not (yet) modeled by parents

Could possibly disappear when all combinatorial classes / enumerated sets will be parents

**INPUT:**

- `function` – a callable or an iterable of callables. This represents the underlying function used to implement this map. If it is an iterable, then the callables will be composed to implement this map.
- `domain` – the domain of this map or `None` if the domain is not known or should remain unspecified
- `codomain` – the codomain of this map or `None` if the codomain is not known or should remain unspecified
- `name` – a name for this map or `None` if this map has no particular name

**EXAMPLES:**

```python
sage: from sage.categories.poor_man_map import PoorManMap
sage: f = PoorManMap(factorial, domain=(1, 2, 3), codomain=(1, 2, 6))
sage: f
A map from (1, 2, 3) to (1, 2, 6)
sage: f(3)
6
```

The composition of several functions can be created by passing in a tuple of functions:

```python
sage: i = PoorManMap((factorial, sqrt), domain=(1, 4, 9), codomain=(1, 2, 6))
```

However, the same effect can also be achieved by just composing maps:

```python
sage: g = PoorManMap(factorial, domain=(1, 2, 3), codomain=(1, 2, 6))
sage: h = PoorManMap(sqrt, domain=(1, 4, 9), codomain=(1, 2, 3))
sage: i == g*h
True
```
```python
>>> from sage.all import *

>>> g = PoorManMap(factorial, domain=(Integer(1), Integer(2), Integer(3)), → codomain=(Integer(1), Integer(2), Integer(6)))

>>> h = PoorManMap(sqrt, domain=(Integer(1), Integer(4), Integer(9)), → codomain=(Integer(1), Integer(2), Integer(3)))

>>> i == g*h
True
```

codomain()

Returns the codomain of self

EXAMPLES:

```python
sage: from sage.categories.poor_man_map import PoorManMap

sage: PoorManMap(lambda x: x+1, domain=(1,2,3), codomain=(2,3,4)).codomain()
(2, 3, 4)
```

domain()

Returns the domain of self

EXAMPLES:

```python
sage: from sage.categories.poor_man_map import PoorManMap

sage: PoorManMap(lambda x: x+Integer(1), domain=(Integer(1),Integer(2), → Integer(3)), codomain=(Integer(2),Integer(3),Integer(4))).codomain()
(2, 3, 4)
```
• Index
• Module Index
• Search Page
PYTHON MODULE INDEX

C
sage.categories.action, 249
sage.categories.additive_groups, 254
sage.categories.additive_magmas, 257
sage.categories.additive_monoids, 274
sage.categories.additive_semigroups, 276
sage.categories.affine_weyl_groups, 279
sage.categories.algebra_functor, 1359
sage.categories.algebra_modules, 285
sage.categories.algebra_ideals, 284
sage.categories.algebra_modules, 285
sage.categories.algebras, 286
sage.categories.algebras_with_basis, 291
sage.categories.all, 1
sage.categories.aperiodic_semigroups, 299
sage.categories.associative_algebras, 300
sage.categories.bialgebras, 300
sage.categories.bialgebras_with_basis, 302
sage.categories.bimodules, 309
sage.categories.cartesian_product, 1353
sage.categories.category, 43
sage.categories.category_cy_helper, 1496
sage.categories.category_singleton, 1491
sage.categories.category_types, 1487
sage.categories.category_with_axiom, 101
sage.categories.classical_crystals, 311
sage.categories.coalgebras, 317
sage.categories.coalgebras_with_basis, 326
sage.categories.coercion_methods, 1500
sage.categories.commutative_additive_groups, 329
sage.categories.commutative_additive_monoids, 331
sage.categories.commutative_additive_semigroups, 332
sage.categories.commutative_algebra_ideals, 333
sage.categories.commutative_algebras, 334
sage.categories.commutative_ring_ideals, 335
sage.categories.commutative_rings, 335
sage.categories.complete_discrete_valuation, 349
sage.categories.complex_reflection_groups, 356
sage.categories.complex_reflection_or_generalized_coxeter_groups, 359
sage.categories.covariant_functorial_construction, 1345
sage.categories.coxeter_group_algebras, 388
sage.categories.coxeter_groups, 392
sage.categories.crystals, 456
sage.categories.cw_complexes, 494
sage.categories.discrete_valuation, 498
sage.categories.distributive_magmas_and_additive_magmas, 502
sage.categories.division_rings, 503
sage.categories.domains, 505
sage.categories.dual, 1359
sage.categories.enumerated_sets, 506
sage.categories.euclidean_domains, 516
sage.categories.examples.algebras_with_basis, 1391
sage.categories.examples.commutative_additive_monoids, 1392
sage.categories.examples.commutative_additive_semigroups, 1394
sage.categories.examples.coxeter_groups, 1398
sage.categories.examples.crystals, 1398
sage.categories.examples.cw_complexes, 1401
sage.categories.examples.facade_sets, 1403
sage.categories.examples.finite_coxeter_groups, 1403
sage.categories.examples.finite_dimensional_algebras_with_basis,
1407  
sage.categories.examples.finite_dimen-
sional_lie_algebras_with_basis,  
1409  
sage.categories.examples.finite_enumer-
ated_sets, 1416  
sage.categories.examples.finite_monoids, 1419  
sage.categories.examples.finite_semi-
groups, 1422  
sage.categories.examples.finite_weyl_groups, 1426  
sage.categories.examples_graded_connected_hopf_algebras_with_basis,  
1429  
sage.categories.examples_graded_modules_with_basis, 1431  
sage.categories.examples_graphs, 1435  
sage.categories.examples_hopf_algebras_with_basis, 1437  
sage.categories.examples_infinite_enumerated_sets, 1439  
sage.categories.examples_lie_algebras, 1442  
sage.categories.examples_lie_alge-
bras_with_basis, 1443  
sage.categories.examples_magnas, 1446  
sage.categories.examples_manifolds, 1448  
sage.categories.examples_monoids, 1449  
sage.categories.examples_posets, 1452  
sage.categories.examples_semgroups,  
1455  
sage.categories.examples_semi-
groups_cython, 1462  
sage.categories.examples_sets_cat, 1466  
sage.categories.examples_sets_with_grading, 1476  
sage.categories.examples_with_realiza-
tions, 1478  
sage.categories.facade_sets, 1342  
sage.categories.fields, 519  
sage.categories.filtered_algebras, 527  
sage.categories.filtered_alge-
bras_with_basis, 528  
sage.categories.filtered_modules, 541  
sage.categories.filtered_modules_with_basis, 543  
sage.categories.finite_complex_reflec-
tion_groups, 571  
sage.categories.finite_coxeter_groups,  
601  
sage.categories.finite_crystals, 623  
sage.categories.finite_dimensional_al-
gebras_with_basis, 624  
sage.categories.finite_dimensional_bialgebras_with_basis,  
658  
sage.categories.finite_dimensional_coalgebras_with_basis,  
659  
sage.categories.finite_dimensional_graded_lie_alge-
bras_with_basis, 659  
sage.categories.finite_dimensional_hopf_algebras_with_basis,  
662  
sage.categories.finite_dimensional_lie_algebras_with_basis,  
663  
sage.categories.finite_dimensional_modules_with_basis, 708  
sage.categories.finite_dimensional_nilpotent_lie_alge-
bras_with_basis, 730  
sage.categories.finite_dimensional_semisimple_alge-
bras_with_basis, 732  
sage.categories.finite_enumerated_sets, 736  
sage.categories.finite_fields, 746  
sage.categories.finite_groups, 747  
sage.categories.finite_lattice_posets,  
751  
sage.categories.finite_monoids, 756  
sage.categories.finite_permutation_groups, 762  
sage.categories.finite_posets, 768  
sage.categories.finite_semigroups, 804  
sage.categories.finite_sets, 806  
sage.categories.finite_weyl_groups, 808  
sage.categories.finitely_generated_lambdaBracketAlgebras,  
809  
sage.categories.finitely_generated_ideal_conformal_algebras,  
811  
sage.categories.finitely_generated_magmas, 813  
sage.categories.finitely_generated_semigroups, 814  
sage.categories.function_fields, 818  
sage.categories.functor, 147  
sage.categories.g_sets, 818  
sage.categories.gcd_domains, 819  
sage.categories.generalized_coxeter_groups, 820  
sage.categories.gradle_alge-
bras, 822
sage.categories.sets_with_partial_maps, 1256
sage.categories.shephard_groups, 1257
sage.categories.signed_tensor, 1357
sage.categories.simplicial_complexes, 1257
sage.categories.simplicial_sets, 1260
sage.categories.subobjects, 1371
sage.categories.subquotients, 1370
sage.categories.super_algebras, 1280
sage.categories.super_algebras_with_basis, 1283
sage.categories.super_hopf_algebras_with_basis, 1286
sage.categories.super_lie_conformal_algebras, 1287
sage.categories.super_modules, 1289
sage.categories.super_modules_with_basis, 1293
sage.categories.supercrystals, 1296
sage.categories.supercrystals, 1297
sage.categories.tensor, 1356
sage.categories.topological_spaces, 1305
sage.categories.triangular_kac_moody_algebras, 1308
sage.categories.tutorial, 154
sage.categories.unique_factorization_domains, 1311
sage.categories.unital_algebras, 1313
sage.categories.vector_bundles, 1317
sage.categories.vector_spaces, 1320
sage.categories.weyl_groups, 1326
sage.categories.with_realizations, 1382
Non-alphabetical

__classcall__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 134
__classcall__() (sage.categories.category.Category static method), 63
__classget__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 135
__init__() (sage.categories.category_with_axiom.CategoryWithAxiom method), 136
__init__() (sage.categories.category.Category method), 64
_all_super_categories() (sage.categories.category.Category method), 53
_all_super_categories_proper() (sage.categories.category.Category method), 55
_make_named_class() (sage.categories.category.Category method), 56
_make_named_class() (sage.categories.category.CategoryWithParameters.Category method), 94
__repr__() (sage.categories.category.Category method), 58
__repr__() (sage.categories.category.JoinCategory.Category method), 97
__repr_object_names__() (sage.categories.category_with_axiom.CategoryWithAxiom.method), 136
__repr_object_names__() (sage.categories.category.Category method), 58
__repr_object_names__() (sage.categories.category.JoinCategory.Category method), 97
__repr_object_names_static__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 137
_set_of_super_categories() (sage.categories.category.Category method), 55
_sort() (sage.categories.category.Category static method), 61
_sort_uniq() (sage.categories.category.Category static method), 63
_super_categories() (sage.categories.category.Category method), 53
_super_categories_for_classes() (sage.categories.category.Category method), 54
_test_category() (sage.categories.category.Category method), 58
_test_category_with_axiom() (sage.categories.category_with_axiom.CategoryWithAxiom method), 138
_with_axiom() (sage.categories.category.Category method), 59
_with_axiom_as_tuple() (sage.categories.category.Category method), 60
_without_axioms() (sage.categories.category_with_axiom.CategoryWithAxiom method), 138
_without_axioms() (sage.categories.category.Category method), 61
_without_axioms() (sage.categories.category.JoinCategory.Category method), 97

A

a_realization() (sage.categories.examples.with_realizations.SubsetAlgebra method), 1484
a_realization() (sage.categories.sets_cat.Sets.WithRealizations.ParentMethods method), 1245
AbelianCategory (class in sage.categories.category_types), 1487
AbelianLieAlgebra (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 1409
AbelianLieAlgebra (class in sage.categories.examples.lie_algebras_with_basis), 1443
AbelianLieAlgebra.Element (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 1409
AbelianLieAlgebra.Element (class in sage.categories.examples.lie_algebras_with_basis), 1443
AbelianVarieties (class in sage.categories.schemes), 1180
AbelianVarieties.Homsets (class in sage.categories.schemes), 1181
AbelianVarieties.Homsets.Endset (class in sage.categories.schemes), 1181
abs()  (sage.categories.metric_spaces.MetricSpaces.ElementMethods method), 1007
absolute_chain()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 394
absolute_chain_reflections()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 394
absolute_covers()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 396
absolute_le()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 397
absolute_length()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 397
absolute_length()  (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ElementMethods method), 602
absolute_order_ideal()  (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 575
absolute_poset()  (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 577
act()  (sage.categories.action.Action method), 250
Action  (class in sage.categories.action), 250
ActionEndomorphism  (class in sage.categories.action), 251
actor()  (sage.categories.action.Action method), 251
adams_operator()  (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ElementMethods method), 303
add()  (sage.categories.category_cy_helper.AxiomContainer method), 1497
addition_table()  (sage.categories.additive_magmas.AdditiveMagmas.ParentMethods method), 267
additional_structure()  (sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital method), 264
additional_structure()  (sage.categories.affine_weyl_groups.AffineWeylGroups method), 283
additional_structure()  (sage.categories.bialgebras.Bialgebras method), 302
additional_structure()  (sage.categories.bimodules.Bimodules method), 310
additional_structure()  (sage.categories.category_with_axiom.CategoryWithAxiom method), 139
additional_structure()  (sage.categories.categorype.Category method), 72
additional_structure()  (sage.categories.category.JoinCategory method), 98
additional_structure()  (sage.categories.classical_crystals.ClassicalCrystals method), 316
additional_structure()  (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 358
additional_structure()  (sage.categories.complex_reflection_groups.ComplexReflectionGroups.Irreducible.ParentMethods method), 515
additional_structure()  (sage.categories.coxeter_groups.CoxeterGroups method), 455
additional_structure()  (sage.categories.enumerated_sets.EnumeratedSets method), 575
additional_structure()  (sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups method), 821
additional_structure()  (sage.categories.hardest_weight_crystals.HardestWeightCrystals method), 884
additional_structure()  (sage.categories.lie_groups.LieGroups method), 943
additional_structure()  (sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas method), 991
additional_structure()  (sage.categories.magmas.Magmas.Unital method), 988
additional_structure()  (sage.categories.magmatic_algebras.MagmaticAlgebras method), 996
additional_structure()  (sage.categories.manifolds.Manifolds method), 1003
additional_structure()  (sage.categories.modules Modules method), 1029
additional_structure()  (sage.categories.objects.Objects method), 1097
additional_structure()  (sage.categories.principal_ideal_domains.PrincipalIdealDomains method), 1120
additional_structure()  (sage.categories.regular_crystals.RegularCrystals method), 1154
additional_structure()  (sage.categories.unique_factorization_domains.UniqueFactorizationDomains method), 1313
additional_structure()  (sage.categories.vector_spaces.VectorSpaces method), 1325
additional_structure()  (sage.categories.weyl_groups.WeylGroups method), 1341
additive_order()  (sage.categories.commutative_additive_groups.CommutativeAdditive-
AdditiveMagmas.AdditiveCommutative

AdditiveMagmas.AdditiveCommutative()


AdditiveGroups.Finite.Algebras

AdditiveGroups.Finite

AdditiveGroups.Algebras.ParentMethods

AdditiveGroups.Algebras

AdditiveGroups.Algebras.

AdditiveGroups.AlgebrasConfigurer

AdditiveGroups.AlgebrasConfigurer()

AdditiveGroups.AlgebrasConfigurer.CartesianProducts


AdditiveGroups.AlgebrasConfigurer.DistributiveMagmas


AdditiveGroups.AlgebrasConfigurer.DistributiveMagas
Algebras (class in sage.categories.algebras), 286
Algebras (sage.categories.coxeter_groups.CoxeterGroups attribute), 394
Algebras (sage.categories.groups.Groups attribute), 858
Algebras() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1229
Algebras.CartesianProducts (class in sage.categories.algebras), 286
AlgebrasCategory (class in sage.categories.algebra_functor), 1368
AlgebrasCategory.ParentMethods (class in sage.categories.algebra_functor), 1368
Algebras.DualObjects (class in sage.categories.algebras), 287
Algebras.ElementMethods (class in sage.categories.algebras), 288
Algebras.ParentMethods (class in sage.categories.algebras), 288
Algebras.Quotients (class in sage.categories.algebras), 289
Algebras.Quotients.ParentMethods (class in sage.categories.algebra_functor), 1368
Algebras.SubcategoryMethods (class in sage.categories.algebras), 290
Algebras.TensorCategory (class in sage.categories.algebra_functor), 1368
Algebras.TensorProducts (class in sage.categories.algebras), 291
Algebras.TensorProducts.ElementMethods (class in sage.categories.algebras), 291
Algebras.TensorProducts.ParentMethods (class in sage.categories.algebras), 291
AlgebrasWithBasis (class in sage.categories.algebra_functor), 291
AlgebrasWithBasis.CartesianProducts (class in sage.categories.algebra_functor), 294
AlgebrasWithBasis.CartesianProducts.ParentMethods (class in sage.categories.algebra_functor), 294
AlgebrasWithBasis.ElementMethods (class in sage.categories.algebra_functor), 295
AlgebrasWithBasis.ParentMethods (class in sage.categories.algebra_functor), 296
AlgebrasWithBasis.TensorCategory (class in sage.categories.algebra_functor), 297
AlgebrasWithBasis.TensorProducts.ElementMethods (class in sage.categories.algebra_functor), 297
AlgebrasWithBasis.TensorProducts.ParentMethods (class in sage.categories.algebra_functor), 297
all_paths_to_highest_weight() (sage.categories.crystals.Crystals.ElementMethods method), 465
all_super_categories() (sage.categories.category.Category method), 75
AlmostComplex() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 1000
ambient() (sage.categories.category_types.Category_in_ambient method), 1488
ambient() (sage.categories.example.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 1410
ambient() (sage.categories.example.finite_organized_sets.IsomorphicObjectOfFiniteEnumeratedSet method), 1418
ambient() (sage.categories.example.semigroupsIncompleteSubquotientSemigroup method), 1457
ambient() (sage.categories.example.semigroups.QuotientOfLeftZeroSemigroup method), 1460
ambient() (sage.categories.example.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.Subobjects.ParentMethods method), 705
ambient() (sage.categories.example.sets_cat.Subquotients.ParentMethods method), 1243
ambient() (sage.categories.crystals.Crystals.ParentMethods method), 476
ambient() (sage.categories.example.commutative_additive_semigroups.FreeCommutativeAdditiveSemigroup method), 1397
ambient() (sage.categories.example.complexes.Surface method), 1402
ambient() (sage.categories.example.finite_monoids.IntegerModMonoid method), 1421
ambient() (sage.categories.example.finite_semigroups.LeftRegularBand method), 1425
ambient() (sage.categories.example.graphs.Cycle method), 1436
ambient() (sage.categories.example.infinite_organized_sets.NonNegativeIntegers method), 1441
ambient() (sage.categories.example.magma.FreeMagma method), 1447
ambient() (sage.categories.example.manifolds.Plane method), 1448
ambient() (sage.categories.example.posets.FiniteSetsOrderedByInclusion method), 1453
ambient() (sage.categories.example.semigroups.FreeSemigroup method), 1456
ambient() (sage.categories.example.semigroups.LeftZeroSemigroup method), 1458
ambient() (sage.categories.example.semigroups.QuotientOfLeftZeroSemigroup method), 1460
ambient() (sage.categories.example.sets_cat.PrimeNumbers method), 1468
ambient() (sage.categories.example.sets_cat.PrimeNumbers method), 1468
meNumbers_Abstract method), 1469
an_element() (sage.categories.examples.sets_with_grading.NonNegativeIntegers method), 1476
an_element() (sage.categories.sets_cat.Sets.ParentMethods method), 1225
an_instance() (sage.categories.algebra_modules.AlgebraModules_class method), 285
an_instance() (sage.categories.bimodules.Bimodules_class method), 310
an_instance() (sage.categories.category_types.Category_ideal class method), 1487
an_instance() (sage.categories.category_types.Category_over_base class method), 1489
an_instance() (sage.categories.category_types.Elements_class method), 1490
an_instance() (sage.categories.category.Category, 77
an_instance() (sage.categories.g_sets.GSets_class method), 819
an_instance() (sage.categories.groupoid.Groupoid_class method), 837
Analytic() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 1001
annihilator() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ParentMethods method), 720
annihilator_basis() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ParentMethods method), 721
antichains() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 769
antipode() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.ParentMethods method), 893
antipode() (sage.categories.hopf_algebras.HopfAlgebras.ElementMethods method), 886
antipode() (sage.categories.super_hopf_algebras_with_basis.SuperHopfAlgebrasWithBasis.ParentMethods method), 1286
antipode_on_basis() (sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra method), 1437
antipode_on_basis() (sage.categories.graded_hopf_algebras_with_basis.GradedHopfAlgebrasWithBasis.Connected.ParentMethods method), 832
antipode_on_basis() (sage.categories.group_algebras.GroupAlgebras.ParentMethods method), 852
antipode_on_basis() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.ParentMethods method), 894
Aperiodic (sage.categories.semi_groups.Semigroups.attribute), 1189
Aperiodic() (sage.categories.semi_groups.Semigroups.SubcategoryMethods method), 1200
AperiodicSemigroups (class in sage.categories.aperiodic_semigroups), 299
apply_conjugation_by_simple_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 361
apply_demazure_product() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 399
apply_multilinear_morphism() (sage.categories.modules_with_basis.ModulesWithBasis.TensorProducts.ElementMethods method), 1077
apply_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 361
apply_simple_projection() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 399
apply_simple_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 363
apply_simple_reflection_left() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 365
apply_simple_reflection_right() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 366
apply_simple_reflection_right() (sage.categories.examples.finite_coxeter_groups.DihedralGroup.Element method), 1405
apply_simple_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 367
as_finite_dimensional_algebra() (sage.cate-
categories.finite_dimensional_lie_algebras_with_basis
FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 665

Associative (sage.categories.magmas.Magmas attribute), 968

Associative (sage.categories.magmatic_algebras.MagmaticAlgebras attribute), 992

Associative (sage.categories.magmatic_algebras.Magmas.SubcategoryMethods method), 979

AssociativeAlgebras (class in sage.categories.associative_algebras), 300

axiom() (in module sage.categories.category_with_axiom), 143

axiom_of_nested_class() (in module sage.categories.category_with_axiom), 144

AxiomContainer (class in sage.categories.category_with_axiom), 139

axioms() (sage.categories.category_with_axiom.CategoryWithAxiom method), 77

B

b_sharp() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 948

baker_campbell_hausdorff() (sage.categories.associative_algebras.SubcategoryMethods method), 916

Bars (class in sage.categories.category_with_axiom), 132

base() (sage.categories.category_types.CategoryWithAxiom method), 1489

base() (sage.categories.homsets.HomsetsCategory method), 1376

base() (sage.categories.signed_tensor.SignedTensorProductsCategory method), 1358

base() (sage.categories.tensor.TensorProductsCategory method), 1357

base_category() (sage.categories.associated_algebras.MonomialAlgebrasWithBasis method), 141

base_category() (sage.categories.category_with_axiom.CategoryWithAxiom method), 141

base_category() (sage.categories.covariant_functorial_construction.FunctorialConstructionCategory method), 1351

base_category_class_and_axiom() (in module sage.categories.category_with_axiom), 145

base_change_matrix() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 581

base_curve() (sage.categories.schemes.Jacobians.ParentMethods method), 1182

base_field() (sage.categories.algebras.QuantumAlgebras method), 1011

base_field() (sage.categories.associated_algebras.ModularAbelianVarieties method), 1011

base_field() (sage.categories.associated_algebras.VectorSpaces method), 1325

base_point() (sage.categories.schemes.SimplicialSets.Pointed.ParentMethods method), 1265

base_point_map() (sage.categories.schemes.SimplicialSets.Pointed.ParentMethods method), 1266

base_ring() (sage.categories.algebra_functor.AlgebraFunctor method), 1367

base_ring() (sage.categories.cartesian_product.CartesianProductsCategory method), 1356

base_ring() (sage.categories.category_types.Category_over_base_ring method), 1490

base_ring() (sage.categories.modules.Modules.HomsetsCategory method), 1018


base_ring() (sage.categories.modules.Modules.SubcategoryMethods method), 1025

base_scheme() (sage.categories.schemes.AbelianVarietiesWithBasis method), 1181

base_scheme() (sage.categories.schemes.Jacobians method), 1182

base_scheme() (sage.categories.schemes.Schemes_over_base method), 1184

base_set() (sage.categories.examples.with_realizations.SubsetAlgebra method), 1485

base_space() (sage.categories.tensor.TensorBundles method), 1319

basis() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 1411

basis() (sage.categories.examples.graded_modules_with_basis.GradedPartitionModule method), 1432

basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 556

basis() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1055

basis_matrix() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 1411

basis_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.Subobjects.ParentMethods method), 705

bch() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 918

bhz_poset() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 606

Bialgebras (class in sage.categories.bialgebras), 300

Bialgebras.ElementMethods (class in sage.cate-
Bialgebras (class in sage.categories.bialgebras), 301
Bialgebras.Super (class in sage.categories.bialgebras), 301
BialgebrasWithBasis (class in sage.categories.bialgebras_with_basis), 302
BialgebrasWithBasis.ElementMethods (class in sage.categories.bialgebras_with_basis), 303
BialgebrasWithBasis.ParentMethods (class in sage.categories.bialgebras_with_basis), 307
Bimodules (class in sage.categories.bimodules), 309
Bimodules.ElementMethods (class in sage.categories.bimodules), 310
Bimodules.ParentMethods (class in sage.categories.bimodules), 310
binary_factorizations() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 400
birational_free_labelling() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 769
birational_rowmotion() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 775
birational_toggle() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 778
birational_toggles() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 783
BlackBoxConstructionFunctor (class in sage.categories.pushout), 202
Blahs (class in sage.categories.category_with_axiom), 133
Blahs.Commutative (class in sage.categories.category_with_axiom), 133
Blahs.Connected (class in sage.categories.category_with_axiom), 133
Blahs.FiniteDimensional (class in sage.categories.category_with_axiom), 133
Blahs.Flying (class in sage.categories.category_with_axiom), 133
Blahs.SubcategoryMethods (class in sage.categories.category_with_axiom), 134
Blahs.Unital (class in sage.categories.category_with_axiom), 134
Blahs.Unital.Blue (class in sage.categories.category_with_axiom), 134
Blue() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 134
Blue_extra_super_categories() (sage.categories.category_with_axiom.Blahs method), 133
bracket() (sage.categories.lie_algebras.LieAlgebras.ElementMethods method), 911
bracket() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 919
bracket() (sage.categories.rings.Rings.ParentMethods method), 1165
bracket_on_basis() (sage.categories.examples.lie_algebras_with_basis.AbelianLieAlgebra method), 1444
bracket_on_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 932
braid_group_as_finitely_presented_group() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 432
braid_orbit() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 433
braid_orbit_iter() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 435
braid_relations() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 435
bruhat_cone() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 1338
bruhat_graph() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 436
bruhat_interval() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 437
bruhat_interval_poset() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 438
bruhat_le() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 401
bruhat_lower_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 402
bruhat_lower_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 1326
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 404
bruhat_poset() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 607
bruhat_upper_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 404
bruhat_upper_covers() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ElementMethods method), 602
bruhat_upper_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 1327
bruhat_upper_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 405

C

CallMorphism (class in sage.categories.morphism), 187
cambrian_lattice() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 608
canonical_matrix() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 406
canonical_representation() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 438
canonicalize_axioms() (in module sage.categories.category_cy_helper), 1497
cardinality() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods method), 313
cardinality() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 582
cardinality() (sage.categories.finiteEnumeratedSets.FiniteEnumeratedSets.ParentMethods method), 742
cardinality() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), 748
cardinality() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 949
cardinality() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1056
cartan_type() (sage.categories.crystals.CrystalMorphism method), 460
cartan_type() (sage.categories.crystals.Crystals.ElementMethods method), 466
cartan_type() (sage.categories.crystals.Crystals.ParentMethods method), 476
cartan_type() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 1428
cartan_type() (sage.categories.kac_moody_algebras.KacMoodyAlgebras.ParentMethods method), 900
cartan_type() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 1132
cartan_type() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.TensorProducts.ParentMethods method), 1133
cartesian_factors() (sage.categories.sets_cat.Sets.CartesianProducts.ElementMethods method), 1214
cartesian_product() (sage.categories.sets_cat.Sets.ElementMethods method), 1220
cartesian_product() (sage.categories.sets_cat.Sets.ParentMethods method), 1226
cartesian_projection() (sage.categories.sets_cat.Sets.CartesianProducts.ElementMethods method), 1215
CartesianProduct (sage.categories.posets.Posets.ParentMethods attribute), 1103
CartesianProduct (sage.categories.sets_cat.Sets.ParentMethods attribute), 1223
CartesianProductFunctor (class in sage.categories.cartesian_product), 1353
CartesianProducts() (sage.categories.cartesian_product.CartesianProductsCategory method), 1355
CartesianProducts() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1230
CartesianProductsCategory (class in sage.categories.cartesian_product), 1355
casimir_element() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 665
Category Framework, Release 10.4

Category (class in sage.categories.category), 46
category() (sage.categories.category.Category method), 78
category() (sage.categories.morphism.Morphism method), 189
Category_contains_method_by_parent_class (class in sage.categories.category_singleton), 1491
category_for() (sage.categories.map.Map method), 166
category_from_categories() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 1349
category_from_category() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 1350
category_from_parents() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 1350
category_graph() (in module sage.categories.category), 98
category_graph() (sage.categories.category.Category method), 78
Category_ideal (class in sage.categories.category_types), 1487
Category_in_ambient (class in sage.categories.category_types), 1488
Category_module (class in sage.categories.category_types), 1488
category_of() (sage.categories.covariant_functorial_construction.FunctorialConstructionCategory class method), 1351
Category_over_base (class in sage.categories.category_types), 1488
Category_over_base_ring (class in sage.categories.category_types), 1489
Category_realization_of_parent (class in sage.categories.realizations.realizations), 1379
category_sample() (in module sage.categories.category), 99
Category_singleton (class in sage.categories.category_singleton), 1491
category_sort_key() (in module sage.categories.category_cy_helper), 1498
CategoryWithAxiom (class in sage.categories.category_with_axiom), 134
CategoryWithAxiom_over_base_ring (class in sage.categories.category_with_axiom), 142
CategoryWithAxiom_singleton (class in sage.categories.category_with_axiom), 142
CategoryWithParameters (class in sage.categories.category), 93
cayley_graph() (sage.categories.semigroups.Semigroups.ParentMethods method), 1190
cayley_graph_disabled() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), 749
cayley_table() (sage.categories.groups.Groups.ParentMethods method), 862
cell_module() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 626
cells() (sage.categories.cw_complexes.CWComplexes.ParentMethods method), 496
cells() (sage.categories.examples.cw_complexes.Surface method), 1402
cells() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 627
Cellular() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.SubcategoryMethods method), 658
cellular_basis() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ElementMethods method), 628
Cellular_involution() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ElementMethods method), 628
Cellular_involution() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ElementMethods method), 628
connected_components_generators() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 877
connected_components_generators() (sage.categories.simplicial_sets.SimplicialSets.Finite.ParentMethods method), 1300
construction() (sage.categories.sets_cat.Sets.ParentMethods method), 1227
construction_tower() (in module sage.categories.pushout), 233
ConstructionFunctor (class in sage.categories.pushout), 206
content() (sage.categories.principal_ideal_domains.PrincipalIdealDomains.ParentMethods method), 1117
convolution_power_of_id() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ElementMethods method), 303
convolution_product() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ElementMethods method), 304
convolution_product() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ParentMethods method), 307
coproduct() (sage.categories.coalgebras.CoalgebrasWithBasis.ParentMethods method), 327
coproduct() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ElementMethods method), 318
coproduct() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ParentMethods method), 319
coproduct() (sage.categories.coalgebras.Coalgebras.WithRealizations.ParentMethods method), 324
coproduct_by_coercion() (sage.categories.coalgebras.examples.hopf_algebras_with_basis.MyGroupAlgebra method), 1438
coproduct_by_coercion() (sage.categories.coalgebras.examples.hopf_algebras_with_basis.MyGroupAlgebra method), 1437
coproduct_on_basis() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ParentMethods method), 853
coset representative() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 406
counit() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ParentMethods method), 328
counit() (sage.categories.coalgebras.Coalgebras.ElementMethods method), 318
counit() (sage.categories.coalgebras.Coalgebras.ParentMethods method), 319
counit() (sage.categories.coalgebras.Coalgebras.graded_connected_hopf_algebras_with_basis.MyGroupAlgebra method), 1429
counit() (sage.categories.coalgebras.graded_connected_hopf_algebras_with_basis.MyGroupAlgebra method), 1437
counit() (sage.categories.coalgebras.graded_connected_hopf_algebras_with_basis.MyGroupAlgebra method), 1437
counit() (sage.categories.coalgebras.graded_connected_hopf_algebras_with_basis.MyGroupAlgebra method), 1437
CovariantConstructionCategory (class in sage.categories.covariant_functorial_construction), 1345
CovariantFunctorialConstruction (class in sage.categories.covariant_functorial_construction), 1348
cover() (sage.categories.simplicial_sets.SimplicialSets.Pointed.ParentMethods method), 1268
cover_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 408
covered_reflections_subgroup() (sage.categories.finite_coxeter_groups.FiniteCoxeter-
default_super_categories() (sage.categories.covariant_factorial_construction.RegressiveCovariantConstructionCategory class method), 1352

default_super_categories() (sage.categories.graded_modules.GradedModulesCategory class method), 838

default_super_categories() (sage.categories.homsets.HomsetsCategory class method), 1376

default_super_categories() (sage.categories.isomorphic_objects.IsomorphicObjectsCategory class method), 1372

default_super_categories() (sage.categories.metric_spaces.MetricSpacesCategory class method), 1009

default_super_categories() (sage.categories.quotients.QuotientsCategory class method), 1370

default_super_categories() (sage.categories.subobjects.SubobjectsCategory class method), 1371

default_super_categories() (sage.categories.super_modules.SuperModulesCategory class method), 1292

degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 544

degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.Subobjects.ElementMethods method), 568

degree() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ElementMethods method), 839

degree_negation() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ElementMethods method), 840

degree_negation() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ParentMethods method), 546

degree_on_basis() (sage.categories.examples.graded_connected_hopf_algebras_with_basis.GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator method), 1430

degree_on_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 544

degree_on_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.Subobjects.ParentMethods method), 570

degree_on_basis() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.Stratified.ParentMethods method), 661

degree_on_basis() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.Quotients.ParentMethods method), 845

degree_on_basis() (sage.categories.lambda_bracket_algebras_with_basis.LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra.Graded.ParentMethods method), 907

degrees() (sage.categories.examples.finite_coxeter_groups.DihedralGroup method), 1406

degrees() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 1428

degrees() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 584

degrees() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 613

demazure_character() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods method), 314


demazure_lusztig_operator_on_basis() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.ParentMethods method), 389

demazure_lusztig_operators() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.ParentMethods method), 390

demazure_operator() (sage.categories.regular_crystals.RegularCrystals.ParentMethods method), 1149

demazure_operator_simple() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1143

demazure_product() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 442

demazure_subcrystal() (sage.categories.regular_crystals.RegularCrystals.ParentMethods method), 1151


Example (in module sage.categories.examples.commutative_additive_monoids), 1392
Example (in module sage.categories.examples.commutative_additive_semigroups), 1394
Example (in module sage.categories.examples.cw_complexes), 1401
Example (in module sage.categories.examples.finite_coxeter_groups), 1407
Example (in module sage.categories.examples.finite_dimensional_algebras_with_basis), 1407
Example (in module sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 1416
Example (in module sage.categories.examples.finite_monoids), 1419
Example (in module sage.categories.examples.finite_semigroups), 1422
Example (in module sage.categories.examples.finite_weyl_groups), 1426
Example (in module sage.categories.examples.graduated_connected_hopf_algebras_with_basis), 1429
Example (in module sage.categories.examples.graduated_modules_with_basis), 1431
Example (in module sage.categories.examples.graphs), 1436
Example (in module sage.categories.examples.infinite_enumerated_sets), 1439
Example (in module sage.categories.examples.lie_algebras), 1442
Example (in module sage.categories.examples.lie_algebras_with_basis), 1444
Example (in module sage.categories.examples.magmas), 1446
Example (in module sage.categories.examples.manifolds), 1448
Example (in module sage.categories.examples.monoids), 1449
Example (in module sage.categories.examples.sets_with_grading), 1476
example() (sage.categories.algebras_with_basis.AlgebrasWithBasis method), 299
example() (sage.categories.classical_crystals.ClassicalCrystals method), 79
example() (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 358
example() (sage.categories.crystals.Crystals method), 493
example() (sage.categories.facade_sets.FacadeSets method), 1344
example() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups method), 600
example() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible method), 581
example() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated method), 600
example() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated.Irreducible method), 596
example() (sage.categories.finite_crystals.FiniteCrystals method), 624
example() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis method), 707
example() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.IsomorphicObjects method), 742
example() (sage.categories.finite_groups.FiniteGroups method), 750
example() (sage.categories.finite_permutation_groups.FinitePermutationGroups method), 768
example() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups method), 817
example() (sage.categories.graduated_hopf_algebras_with_basis.GradedHopfAlgebrasWithBasis method), 833
example() (sage.categories.graduated_hopf_algebras_with_basis.GradedHopfAlgebrasWithBasis.Connected method), 833
example() (sage.categories.group_algebras.GroupAlgebras method), 856
example() (sage.categories.groups.Groups method), 871
example() (sage.categories.highest_weight_crystals.HighestWeightCrystals method), 885
example() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis method), 895
example() (sage.categories.kac_moody_algebras.KacMoodyAlgebras method), 901
<table>
<thead>
<tr>
<th>Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>example() (sage.categories.lie_algebras_with_basics.LieAlgebras method), 934</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.additive_groups.AdditiveGroups.Finite.Algebras method), 256</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.additive_semi_groups.AdditiveSemigroups.Algebras method), 278</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.additive_semi_groups.AdditiveSemigroups.CartesianProducts method), 278</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.additive_semi_groups.AdditiveSemigroups.Homsets method), 279</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.algebras_with_basis.AlgebrasWithBasis.CartesianProducts method), 295</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.algebras_with_basis.AlgebrasWithBasis.TensorProducts method), 298</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.algebras_with_basis.AlgebrasWithBasis.CartesianProducts method), 287</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.algebras_with_basis.AlgebrasWithBasis.DualObjects method), 287</td>
<td></td>
</tr>
<tr>
<td>extend_codomain() (sage.categories.map.Map method), 167</td>
<td></td>
</tr>
<tr>
<td>extend_domain() (sage.categories.map.Map method), 168</td>
<td></td>
</tr>
<tr>
<td>extend_to_fraction_field() (sage.categories.rings.Rings.MorphismMethods method), 1161</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.algebras.Algebras.TensorProducts method), 291</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.aperiodic_semigroups.AperiodicSemigroups method), 299</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.category_with_axiom.CategoryWithAxiom method), 141</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.clasical_crystals.ClassicalCrystals.TensorProducts method), 316</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.Super method), 329</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.coalgebras.Coalgebras.DualObjects method), 317</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.coalgebras.Coalgebras.Super method), 323</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.coalgebras.Coalgebras.TensorProducts method), 324</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.commutative_algebras.CotensorProducts method), 334</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.commutative_rings.CotensorProducts.CartesianProducts method), 336</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.commutative_rings.CotensorProducts.Finite method), 339</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.covariant Functorial_construction.FunctorialConstructionCategory method), 1351</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.crystals.Crystals.TensorProducts method), 493</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.cw_complexes.CWComplexes.Finite method), 496</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.distributive_magmas_and_additive_magmas.DivisiveMagmasAndAdditiveMagmas.TensorProducts method), 503</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.division_rings.DivisionRings method), 504</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.fields.Fields method), 527</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.filtered_modules.FilteredModules method), 542</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_crystals.FiniteCrystals.FiniteCosetSpaces method), 623</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_crystals.FiniteCrystals.TensorProducts method), 624</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.VectorSpaces method), 632</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.TensorProducts method), 730</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.CartesianProducts method), 741</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_fields.FiniteFields method), 746</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_groups.FiniteGroups.Algebras method), 747</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_permutation_groups.FinitePermutationGroups method), 768</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_sets.FiniteSets.Algebras method), 807</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finite_sets.FiniteSets.Subquotients method), 808</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.finitely_generated_semiplanar_groups.FinitelyGeneratedSemigroups method), 817</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups.Finite method), 821</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis.SignedTensorProducts method), 827</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.graded_algebras.SignedTensorProducts method), 823</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.graded_coalgebras_with_basis.GradedCoalgebrasWithBasis.SignedTensorProducts method), 830</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.graded_coalgebras.SignedTensorProducts method), 834</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.graded_lie_algebras.GradedLieAlgebras.Stratified.FiniteDimensional method), 834</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.poset.Partitions method), 843</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.poset.Partitions.Subsets method), 847</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.return_TransitiveMethod() method), 847</td>
<td></td>
</tr>
<tr>
<td>Category Framework, Release 10.4</td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.groups.Groups.CartesianProducts method), 859</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.hecke_modules.HeckeModules.Homsets method), 873</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.highest_weight_crystals.HighestWeightCrystals.TensorProducts method), 884</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.hopf_algebras_with_basis.HopfAlgebras.TensorProducts method), 894</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.lie_algebras.LieAlgebras.FinitelyPresented method), 888</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.lie_algebras.LieAlgebras.FiniteDimensional method), 914</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.TensorProducts method), 961</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.magnas_and_additive_magnas.MagnasAndAdditiveMagnas.CartesianProducts method), 990</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.magnas.Magnas.Commutative.Algebras method), 967</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.magnas.Magnas.TensorProducts method), 969</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.manifolds.Manifolds.AlmostComplex method), 998</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.manifolds.Manifolds.Analytic method), 999</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.manifolds.Manifolds.Smooth method), 1000</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.metric_spaces.MetricSpaces.CartesianProducts method), 1005</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules_with_basis.ModulesWithBasis.ModulesWithBasis.tensor_product method), 1033</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules_with_basis.ModulesWithBasis.ModulesWithBasis.tensor_product娣.erron Basis method), 1033</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules_with_basis.ModulesWithBasis.ModulesWithBasis.tensor_product娣.erron Basis method), 1080</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.CartesianProducts method), 1013</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.FiniteDimensional method), 1014</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.FiniteDimensional method), 1014</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.FinitelyPresented method), 1015</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.Homsets method), 1018</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.modules.Modules.TensorProducts method), 1029</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.monoids.Monoids.Algebras method), 1085</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.monoids.Monoids.CartesianProducts method), 1085</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.TensorProducts method), 1134</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.TensorProducts method), 1141</td>
<td></td>
</tr>
<tr>
<td>extra_super_categories() (sage.categories.r_trivial_semigroups.RTrivialSemigroups method), 1180</td>
<td></td>
</tr>
</tbody>
</table>
extra_super_categories() (sage.categories.regular_crystals.RegularCrystals.TensorProducts method), 1153
extra_super_categories() (sage.categories.regular_supercrystals.RegularSuperCrystals.TensorProducts method), 1156
extra_super_categories() (sage.categories.schemes.AbelianVarieties.Homsets method), 1181
extra_super_categories() (sage.categories.schemes.AbelianVarieties.Homsets.Endset method), 1181
extra_super_categories() (sage.categories.semi_groups.Semigroups.Algebras method), 1189
extra_super_categories() (sage.categories.semi_groups.Semigroups.CartesianProducts method), 1190
extra_super_categories() (sage.categories.sets_cat.Sets.Algebras method), 1213
extra_super_categories() (sage.categories.sets_cat.Sets.CartesianProducts method), 1220
extra_super_categories() (sage.categories.sets_cat.Sets.WithRealizations method), 1250
extra_super_categories() (sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis method), 1285
extra_super_categories() (sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis.SignedTensorProducts method), 1285
extra_super_categories() (sage.categories.super_algebras.SuperAlgebras method), 1282
extra_super_categories() (sage.categories.super_algebras.SuperAlgebras.SignedTensorProducts method), 1281
extra_super_categories() (sage.categories.super_algebras.SuperAlgebras.SuperLieConformalAlgebras method), 1288
extra_super_categories() (sage.categories.super_modules.SuperModules method), 1291
extra_super_categories() (sage.categories.supercrystals.SuperCrystals.TensorProducts method), 1304
extra_super_categories() (sage.categories.topological_spaces.TopologicalSpaces.CartesianProducts method), 1305
extra_super_categories() (sage.categories.topological_spaces.TopologicalSpaces.Calculations method), 1306
extra_super_categories() (sage.categories.topological_spaces.TopologicalSpaces.CartesianProducts method), 1306
Finite (sage.categories.weyl_groups.WeylGroups attribute), 1338
Finite () (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1234
Finite_extra_super_categories() (sage.categories.division_rings.DivisionRings method), 504
Finite_extra_super_categories() (sage.categories.h_trivial_semigroups.HTrivialSemigroups method), 895
FiniteComplexReflectionGroups (class in sage.categories.finite_complex_reflection_groups), 571
FiniteComplexReflectionGroups.ElementMethods (class in sage.categories.finite_complex_reflection_groups), 572
FiniteComplexReflectionGroups.Irreducible (class in sage.categories.finite_complex_reflection_groups), 575
FiniteComplexReflectionGroups.Irreducible.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 575
FiniteComplexReflectionGroups.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 581
FiniteComplexReflectionGroups.SubcategoryMethods (class in sage.categories.finite_complex_reflection_groups), 590
FiniteComplexReflectionGroups.WellGenerated (class in sage.categories.finite_complex_reflection_groups), 591
FiniteComplexReflectionGroups.WellGenerated.Irreducible (class in sage.categories.finite_complex_reflection_groups), 591
FiniteComplexReflectionGroups.WellGenerated.Irreducible.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 591
FiniteComplexReflectionGroups.WellGenerated.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 596
FiniteCoxeterGroups (class in sage.categories.finite_coxeter_groups), 601
FiniteCoxeterGroups.ElementMethods (class in sage.categories.finite_coxeter_groups), 601
FiniteCoxeterGroups.ParentMethods (class in sage.categories.finite_coxeter_groups), 606
FiniteCrystals (class in sage.categories.finite_crystals), 623
FiniteCrystals.TensorProducts (class in sage.categories.finite_crystals), 624
FiniteDimensional (sage.categories.graded_lie_algebras_with_basis.GradedLieAlgebrasWithBasis attribute), 296
FiniteDimensional (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis attribute), 893
FiniteDimensional (sage.categories.modules_with_basis.ModulesWithBasis attribute), 1054
FiniteDimensional() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 134
FiniteDimensional() (sage.categories.cw_complexes.CWComplexes.SubcategoryMethods method), 497
FiniteDimensional() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 641
bras_with_basis), 658
FiniteDimensionalBialgebrasWithBasis() (in module sage.categories.finite_dimensional_bialgebras_with_basis), 658
FiniteDimensionalCoalgebrasWithBasis() (in module sage.categories.finite_dimensional_coalgebras_with_basis), 659
FiniteDimensionalGradedLieAlgebrasWithBasis (class in sage.categories.finite_dimensional_graded_lie_algebras_with_basis), 659
FiniteDimensionalGradedLieAlgebrasWithBasis.Stratified (class in sage.categories.finite_dimensional_graded_lie_algebras_with_basis), 661
FiniteDimensionalHopfAlgebrasWithBasis (class in sage.categories.finite_dimensional_hopf_algebras_with_basis), 662
FiniteDimensionalHopfAlgebrasWithBasis.ElementMethods (class in sage.categories.finite_dimensional_hopf_algebras_with_basis), 662
FiniteDimensionalLieAlgebrasWithBasis (class in sage.categories.finite_dimensional_lie_algebras_with_basis), 663
FiniteDimensionalLieAlgebrasWithBasis.ElementMethods (class in sage.categories.finite_dimensional_lie_algebras_with_basis), 663
FiniteDimensionalLieAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_lie_algebras_with_basis), 665
FiniteDimensionalLieAlgebrasWithBasis.Subobjects (class in sage.categories.finite_dimensional_lie_algebras_with_basis), 705
FiniteDimensionalModulesWithBasis (class in sage.categories.finite_dimensional_modules_with_basis), 708
FiniteDimensionalModulesWithBasis.ElementMethods (class in sage.categories.finite_dimensional_modules_with_basis), 708
FiniteDimensionalModulesWithBasis.Homsets (class in sage.categories.finite_dimensional_modules_with_basis), 709
FiniteDimensionalModulesWithBasis.Homsets.Endset (class in sage.categories.finite_dimensional_modules_with_basis), 709
FiniteDimensionalModulesWithBasis.Homsets.Endset.ElementMethods (class in sage.categories.finite_dimensional_modules_with_basis), 709
FiniteDimensionalModulesWithBasis.MorphismMethods (class in sage.categories.finite_dimensional_modules_with_basis), 716
FiniteDimensionalModulesWithBasis.ParentMethods (class in sage.categories.finite_dimensional_modules_with_basis), 720
FiniteDimensionalNilpotentLieAlgebrasWithBasis (class in sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis), 730
FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis), 730
FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis), 730
FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis), 730
FiniteDimensionalSemisimpleAlgebrasWithBasis (class in sage.categories.finite_dimensional_semisimple_algebras_with_basis), 732
FiniteDimensionalSemisimpleAlgebrasWithBasis.Commutative (class in sage.categories.finite_dimensional_semisimple_algebras_with_basis), 733
FiniteDimensionalSemisimpleAlgebrasWithBasis.Commutative.ParentMethods (class in sage.categories.finite_dimensional_semisimple_algebras_with_basis), 733
FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_semisimple_algebras_with_basis), 733
FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods (class in sage.categories.finite_dimensional_semisimple_algebras_with_basis), 733
FiniteEnumeratedSets (class in sage.categories.finite_enumerated_sets), 734
FiniteEnumeratedSets.CartesianProducts (class in sage.categories.finite_enumerated_sets), 736
FiniteEnumeratedSets.CartesianProducts.ParentMethods (class in sage.categories.finite_enumerated_sets), 737

FiniteEnumeratedSets.IsomorphicObjects (class in sage.categories.finite_enumerated_sets), 741

FiniteEnumeratedSets.IsomorphicObjects.ParentMethods (class in sage.categories.finite_enumerated_sets), 741

FiniteEnumeratedSets.ParentMethods (class in sage.categories.finite_enumerated_sets), 742

FiniteFields (class in sage.categories.finite_fields), 746

FiniteFields.ElementMethods (class in sage.categories.finite_fields), 746

FiniteFields.ParentMethods (class in sage.categories.finite_fields), 748

FiniteGroups (class in sage.categories.finite_groups), 747

FiniteGroups.Algebras (class in sage.categories.finite_groups), 747

FiniteGroups.Algebras.ParentMethods (class in sage.categories.finite_groups), 747

FiniteGroups.ElementMethods (class in sage.categories.finite_groups), 748

FiniteGroups.ParentMethods (class in sage.categories.finite_groups), 748

FiniteLatticePosets (class in sage.categories.finite_lattice_posets), 751

FiniteLatticePosets.ParentMethods (class in sage.categories.finite_lattice_posets), 751

FinitelyGeneratedLambdaBracketAlgebras (class in sage.categories.finitely_generated_lambda_bracket_algebras), 809

FinitelyGeneratedLambdaBracketAlgebras.Graded (class in sage.categories.finitely_generated_lambda_bracket_algebras), 809

FinitelyGeneratedLambdaBracketAlgebras.ParentMethods (class in sage.categories.finitely_generated_lambda_bracket_algebras), 809

FinitelyGeneratedLieConformalAlgebras (class in sage.categories.finitely_generated_lie_conformal_algebras), 811

FinitelyGeneratedLieConformalAlgebras.Graded (class in sage.categories.finitely_generated_lie_conformal_algebras), 811

FinitelyGeneratedLieConformalAlgebras.ParentMethods (class in sage.categories.finitely_generated_lie_conformal_algebras), 812

FinitelyGeneratedLieConformalAlgebras.Super (class in sage.categories.finitely_generated_lie_conformal_algebras), 812

FinitelyGeneratedLieConformalAlgebras.Super.Graded (class in sage.categories.finitely_generated_lie_conformal_algebras), 812

FinitelyGeneratedMagmas (class in sage.categories.finitely_generated_magmas), 813

FinitelyGeneratedMagmas.ParentMethods (class in sage.categories.finitely_generated_magmas), 813

FinitelyGeneratedSemigroups (class in sage.categories.finitely_generated_semigroups), 814

FinitelyGeneratedSemigroups.Finite (class in sage.categories.finitely_generated_semigroups), 814

FinitelyGeneratedSemigroups.Finite.ParentMethods (class in sage.categories.finitely_generated_semigroups), 814

FinitelyGeneratedSemigroups.ParentMethods (class in sage.categories.finitely_generated_semigroups), 815

FinitelyPresented () (class in sage.categories.finitely_presented_groups), 1190

FiniteMonoids (class in sage.categories.finite_monoids), 756
get_axiom_index() (in module sage.categories.category_cy_helper), 1498
GradedAlgebras (class in sage.categories.graded_algebras), 1185
GradedAlgebras (class in sage.categories.graded_algebras), 1254
GradedAlgebras (class in sage.categories.graded_algebras), 822
GradedAlgebras.ElementMethods (class in sage.categories.graded_algebras), 822
GradedAlgebras.ParentMethods (class in sage.categories.graded_algebras), 822
GradedAlgebras.SignedTensorProducts (class in sage.categories.graded_algebras), 823
GradedAlgebras.SubcategoryMethods (class in sage.categories.graded_algebras), 823
GradedAlgebrasWithBasis (class in sage.categories.graded_algebras), 824
GradedAlgebrasWithBasis.ElementMethods (class in sage.categories.graded_algebras), 824
GradedAlgebrasWithBasis.ParentMethods (class in sage.categories.graded_algebras), 824
GradedAlgebrasWithBasis.SignedTensorProducts (class in sage.categories.graded_algebras), 826
GradedAlgebrasWithBasis.SignedTensorProducts.ParentMethods (class in sage.categories.graded_algebras), 826
GradedBialgebras() (in module sage.categories.graded_bialgebras), 828
GradedBialgebrasWithBasis() (in module sage.categories.graded_bialgebras_with_basis), 828
GradedCoalgebras (class in sage.categories.graded_coalgebras), 829
GradedCoalgebras.SignedTensorProducts (class in sage.categories.graded_coalgebras), 829
GradedCoalgebras.SubcategoryMethods (class in sage.categories.graded_coalgebras), 829
GradedCoalgebrasWithBasis (class in sage.categories.graded_coalgebras), 830
GradedCoalgebrasWithBasis.SignedTensorProducts (class in sage.categories.graded_coalgebras), 830
GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator (class in sage.categories.examples.graded_connected_hopf_algebra_with_basis), 1429
GradedHopfAlgebras() (in module sage.categories.graded_hopf_algebras), 831
GradedHopfAlgebrasWithBasis (class in sage.categories.graded_hopf_algebras_with_basis), 831
GradedHopfAlgebrasWithBasis.Connected

Index 1539
Groups.CartesianProducts (class in sage.categories.groups), 858
Groups.CartesianProducts.ParentMethods (class in sage.categories.groups), 858
Groups.Commutative (class in sage.categories.groups), 860
Groups.ElementMethods (class in sage.categories.groups), 861
Groups.ParentMethods (class in sage.categories.groups), 862
Groups.Topological (class in sage.categories.groups), 871
GSets (class in sage.categories.g_sets), 818
gt () (sage.categories.posets.Posets.ParentMethods method), 1104

H

has_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 412
has_full_support() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 413
has_left_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 413
has_right_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 414
has_right_descent() (sage.categories.examples.finite_coxeter_groups.DihedralGroup.Element method), 1405
has_right_descent() (sage.categories.examples.finite_weyl_groups.SymmetricGroup.Element method), 1427
has_standard_involution() (sage.categories.algebras.Algebras.ParentMethods method), 288
HeckeModules (class in sage.categories.hecke_modules), 872
HeckeModules.Homsets (class in sage.categories.hecke_modules), 873
HeckeModules.Homsets.ParentMethods (class in sage.categories.hecke_modules), 873
HeckeModules.ParentMethods (class in sage.categories.hecke_modules), 873
highest_weight_vector() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 879
highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.Finite.ParentMethods method), 1302
highest_weight_vectors_iterator() (sage.categories.highest_weight_crystals.HighestWeightCrystals.TensorProducts.ParentMethods method), 884
HighestWeightCrystalHomset (class in sage.categories.highest_weight_crystals), 874
HighestWeightCrystalMorphism (class in sage.categories.highest_weight_crystals), 874
HighestWeightCrystalOfTypeA (class in sage.categories.examples.crystals), 1398
HighestWeightCrystalOfTypeA.Element (class in sage.categories.examples.crystals), 1399
HighestWeightCrystals (class in sage.categories.highest_weight_crystals), 874
HighestWeightCrystals.ElementMethods (class in sage.categories.highest_weight_crystals), 875
HighestWeightCrystals.ParentMethods (class in sage.categories.highest_weight_crystals), 877
HighestWeightCrystals.TensorProducts (class in sage.categories.highest_weight_crystals), 883
HighestWeightCrystals.TensorProducts.ParentMethods (class in sage.categories.highest_weight_crystals), 883
hilbert_series() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.FiniteDimensional.ParentMethods method), 555
hilbert_series() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 559
hochschild_complex() (sage.categories.algebras_with_basis.AlgebrasWithBasis.ParentMethods method), 296
holomorph() (sage.categories.groups.Groups.ParentMethods method), 869
Hom() (in module sage.categories.homset), 176
hom() (in module sage.categories.homset), 185
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 546
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 559
homogeneous_component_as submodule() (sage.categories.finite_dimensional_graded_algebras_with_basis.FiniteDimensionalGradedAlgebrasWithBasis.ParentMethods method), 660
ideal_monoid() (sage.categories.rings.Rings.ParentMethods method), 1168
ideal submodule() (sage.categories.finite-dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 639
idempotent_lift() (sage.categories.finite-dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 639
idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.ParentMethods method), 805
IdempotentSemigroups (class in sage.categories.examples.semigroups_cython), 1462
IdempotentSemigroups.ElementMethods (class in sage.categories.examples.semigroups_cython), 1462
identity() (sage.categories.homset.Homset method), 182
IdentityConstructionFunctor (class in sage.categories.pushout), 214
IdentityFunctor() (in module sage.categories.functor), 152
IdentityFunctor_generic (class in sage.categories.functor), 152
IdentityMorphism (class in sage.categories.morphism), 188
im_gens() (sage.categories.crystals.CrystalMorphismByGenerators method), 462
image() (sage.categories.crystals.CrystalMorphismByGenerators method), 463
image() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 716
image() (sage.categories.sets_cat.Sets.MorphismMethods method), 1222
image_basis() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 716
IncompleteSubquotientSemigroup (class in sage.categories.examples.semigroups), 1456
IncompleteSubquotientSemigroup.Element (class in sage.categories.examples.semigroups), 1457
index() (sage.categories.lambda_bracket_algebras_with_basis.LambdaBracketAlgebrasWithBasis.ElementMethods method), 906
index_set() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 375
index_set() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 444
index_set() (sage.categories.crystals.Crystals.ElementMethods method), 468
index_set() (sage.categories.crystals.Crystals.ParentMethods method), 486
index_set() (sage.categories.examples.finite_coxeter_groups.DihedralGroup method), 1406
index_set() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 1428
index_set() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 1132
IndexedPolynomialRing (class in sage.categories.examples.semigroups), 1444
indices() (sage.categories.examples.with_realizations.SubsetAlgebra method), 1485
indices_key() (sage.categories.examples.with_realizations.SubsetAlgebra method), 1485
induced_graded_map() (sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis.ParentMethods method), 530
induced_graded_map() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 560
Infinite (sage.categories.enumerated_sets.EnumeratedSets attribute), 507
InfiniteEnumeratedSets (class in sage.categories.infinite_enumerated_sets), 896
InfiniteEnumeratedSets.ParentMethods (class in sage.categories.infinite_enumerated_sets), 897
InfinitePolynomialFunctor (class in sage.categories.pushout), 214
inner_derivations_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 682
IntegerModMonoid (class in sage.categories.examples.finite_monoids), 1419
IntegerModMonoid.Element (class in sage.categories.examples.finite_monoids), 1421
IntegersCompletion (class in sage.categories.examples.fractional_fields), 1403
IntegralDomains (class in sage.categories.integral_domains), 898
IntegralDomains.ElementMethods (class in sage.categories.integral_domains), 898
IntegralDomains.ParentMethods (class in
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersection()</td>
<td>sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods</td>
<td>1058</td>
</tr>
<tr>
<td>invariant_module()</td>
<td>sage.categories.finite-dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ParentMethods</td>
<td>727</td>
</tr>
<tr>
<td>Inverse()</td>
<td>sage.categories.monoids.Monoids</td>
<td>1089</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.category_types.AbelianCategory</td>
<td>1487</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.category.Category</td>
<td>81</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods</td>
<td>682</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.lie_algebras.LieAlgebras.ParentMethods</td>
<td>922</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.modules_with_basis.ModulesWithBasis</td>
<td>1080</td>
</tr>
<tr>
<td>is_abelian()</td>
<td>sage.categories.vector_spaces.VectorSpaces.WithBasis</td>
<td>1325</td>
</tr>
<tr>
<td>is_affine_grassmannian()</td>
<td>sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods</td>
<td>281</td>
</tr>
<tr>
<td>is_antichain_of_poset()</td>
<td>sage.categories.posets.Posets.ParentMethods</td>
<td>1104</td>
</tr>
<tr>
<td>is_CATEGORY()</td>
<td>in module sage.categories.category</td>
<td>100</td>
</tr>
<tr>
<td>is_central()</td>
<td>sage.categories.monoids.Monoids.Algebras.ElementMethods</td>
<td>1082</td>
</tr>
<tr>
<td>is_chain_of_poset()</td>
<td>sage.categories.posets.Posets.ParentMethods</td>
<td>1106</td>
</tr>
<tr>
<td>is_commutative()</td>
<td>sage.categories.commutative_rings.CommutativeRings.ElementMethods</td>
<td>345</td>
</tr>
<tr>
<td>is_commutative()</td>
<td>sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods</td>
<td>640</td>
</tr>
<tr>
<td>is_commutative()</td>
<td>sage.categories.lie_algebras.LieAlgebras.ParentMethods</td>
<td>923</td>
</tr>
<tr>
<td>is_commutative()</td>
<td>sage.categories.magmas.Magmas.Commutative.ParentMethods</td>
<td>971</td>
</tr>
<tr>
<td>is_commutative()</td>
<td>sage.categories.rings.Rings.ParentMethods</td>
<td>1169</td>
</tr>
<tr>
<td>is_connected()</td>
<td>sage.categories.crystals.Crystals.ParentMethods</td>
<td>1169</td>
</tr>
<tr>
<td>is_construction_defined_by_base()</td>
<td>(sage.categories.covariant_functorial_construction.CovariantConstructionCategory)</td>
<td>1347</td>
</tr>
<tr>
<td>is_coxeter_element()</td>
<td>sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ElementMethods</td>
<td>605</td>
</tr>
<tr>
<td>is_coxeter_sortable()</td>
<td>sage.categories.coxeter_groups.CoxeterGroups.ElementMethods</td>
<td></td>
</tr>
</tbody>
</table>
allLieAlgebrasWithBasis.ParentMethods method), 684
is_simply_connected() (sage.categoriessimplicial_sets.SimplicialSets.Pointed.ParentMethods
method), 1273
is_solvable() (sage.categories.finite_dimensional_ideal_algebras_with_basis.FiniteDimensionalIdealAlgebrasWithBasis.ParentMethods method), 685
is_solvable() (sage.categories.ideal_algebras.IdealAlgebras.ParentMethods method), 924
is_strict() (sage.categories.crystals.Crystals.MorphismMethods method), 475
is_subcategory() (sage.categories.category.Category method), 83
is_subcategory() (sage.categories.category.JoinCategory method), 98
is_super_homogeneous() (sage.categories.super_modules_with_basis.SuperModulesWithBasis.ElementMethods method), 1295
is_surjective() (sage.categories.crystals.Crystals.MorphismMethods method), 460
is_surjective() (sage.categories.map.FormalCompositeMap method), 161
is_surjective() (sage.categories.map.Map method), 169
is_surjective() (sage.categories.morphism.IdentityMorphism method), 188
is_surjective() (sage.categories.morphism.SetIsomorphism method), 194
is_unique_factorization_domain() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains.ParentMethods method), 1312
is_unit() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 500
is_unit() (sage.categories.fields.Fields.ElementMethods method), 522
is_unit() (sage.categories.rings.Rings.ElementMethods method), 1161
is_well_generated() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 585
is_well_generated() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated.ParentMethods method), 598
is_zero() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1036
is_zero() (sage.categories.rings.Rings.ParentMethods method), 1171
IsomorphicObjectOfFiniteEnumeratedSet (class in sage.categories.examples.finite_enumerated_sets), 1417
IsomorphicObjects() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1235
IsomorphicObjectsCategory (class in sage.categories.isomorphic_objects), 1372
isotypic-projective-modules() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 644
iterator_range() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 508
iterator_range() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ParentMethods method), 742
J
j_classes() (sage.categories.finite_semigroups.FiniteSemigroups.ParentMethods method), 805
j_classes_of_idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.ParentMethods method), 805
j_transversal_of_idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.ParentMethods method), 806
Jacobians (class in sage.categories.schemes), 1182
Jacobians.ParentMethods (class in sage.categories.schemes), 1182
join() (sage.categories.category.Category static method), 85
join() (sage.categories.lattice_posets.LatticePosets.ParentMethods method), 908
join_as_tuple() (in module sage.categories.category_cy_helper), 1499
join_irreducibles() (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 754
join_irreducibles_poset() (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 754
JoinCategory (class in sage.categories.category), 96
JTrivial (sage.categories.semi_groups.Semigroups attribute), 1190
JTrivial() (sage.categories.magnas.Magnas.SubcategoryMethods method), 982
JTrivial() (sage.categories.semi_groups.Semigroups.SubcategoryMethods method), 1201
JTrivialSemigroups (class in sage.categories.j_trivial_semigroups), 900
K
K() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.WithBasis.ElementMethods method), 1134
K_on_basis() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.With-Basis.TensorProducts.ParentMethods method), 1137

KacMoodyAlgebras (class in sage.categories.kac_moody_algebras), 900

KacMoodyAlgebras.ParentMethods (class in sage.categories.kac_moody_algebras), 900

kazhdan_lusztig_cell() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 417

kazhdan_lusztig_cells() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 445

kernel() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 717

killing_form() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 685

killing_form() (sage.categories.lie_algebras.LieAlgebras.ElementMethods method), 913

killing_form() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 925

killing_form_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 686

killing_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 686

KirillovReshetikhinCrystals (class in sage.categories.loop_crystals), 944

KirillovReshetikhinCrystals.ElementMethods (class in sage.categories.loop_crystals), 944

KirillovReshetikhinCrystals.ParentMethods (class in sage.categories.loop_crystals), 945

KirillovReshetikhinCrystals.TensorProducts (class in sage.categories.loop_crystals), 954

KirillovReshetikhinCrystals.TensorProducts.ElementMethods (class in sage.categories.loop_crystals), 954

KirillovReshetikhinCrystals.TensorProducts.ParentMethods (class in sage.categories.loop_crystals), 958

KroneckerQuiverPathAlgebra (class in sage.categories.examples.finite_dimensional_algebras_with_basis), 1407

L

Lambda() (sage.categories.crystals.Crystals.ParentMethods method), 476

LambdaBracketAlgebras (class in sage.categories.lambda_bracket_algebras), 902

LambdaBracketAlgebras.ElementMethods (class in sage.categories.lambda_bracket_algebras), 902

LambdaBracketAlgebras.ParentMethods (class in sage.categories.lambda_bracket_algebras), 904

LambdaBracketAlgebras.SubcategoryMethods (class in sage.categories.lambda_bracket_algebras), 904

LambdaBracketAlgebrasWithBasis (class in sage.categories.lambda_bracket_algebras_with_basis), 905

LambdaBracketAlgebrasWithBasis.ElementMethods (class in sage.categories.lambda_bracket_algebras_with_basis), 906

LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra (class in sage.categories.lambda_bracket_algebras_with_basis), 906

LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra.Graded (class in sage.categories.lambda_bracket_algebras_with_basis), 907

LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra.Graded.ParentMethods (class in sage.categories.lambda_bracket_algebras_with_basis), 907

last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.CartesianProducts.ParentMethods method), 738

last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ParentMethods method), 743

latex() (sage.categories.crystals.Crystals.ParentMethods method), 486

latex_file() (sage.categories.crystals.Crystals.ParentMethods method), 487

LatticePosets (class in sage.categories.lattice_posets), 908

LatticePosets.ElementMethods (class in sage.categories.lattice_posets), 908
<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>LieAlgebrasWithBasis.ParentMethods</td>
<td>(class in sage.categories.lie_algebras_with_basis), 932</td>
</tr>
<tr>
<td>LieConformalAlgebras</td>
<td>(class in sage.categories.lie_conformal_algebras), 936</td>
</tr>
<tr>
<td>LieConformalAlgebras.ElementMethods</td>
<td>(class in sage.categories.lie_conformal_algebras), 937</td>
</tr>
<tr>
<td>LieConformalAlgebras.ParentMethods</td>
<td>(class in sage.categories.lie_conformal_algebras), 938</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 940</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.FiniteGenericallyAsLambdaBracketAlgebra</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 940</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.FiniteGenericallyAsLambdaBracketAlgebra.Graded</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 941</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.FiniteGenericallyAsLambdaBracketAlgebra.Super</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 941</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.Graded</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 942</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.Super</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 942</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.Super.Graded</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 942</td>
</tr>
<tr>
<td>LieConformalAlgebrasWithBasis.Super.ParentMethods</td>
<td>(class in sage.categories.lie_conformal_algebras_with_basis), 943</td>
</tr>
<tr>
<td>LieGroups</td>
<td>(class in sage.categories.lie_groups), 943</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.examples.finite_dimensional_lie_algebras_with_basis, AbelianLieAlgebra method), 1414</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.examples.finite_dimensional_lie_algebras_with_basis, AbelianLieAlgebra.Element method), 1409</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.examples.finite_enumerated_sets.IsomorphicObjectOfFiniteEnumeratedSet method), 1418</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.examples.lie_algebras_with_basis, AbelianLieAlgebra.Element method), 1443</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup method), 1460</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.lie_algebras_with_basis, LieAlgebrasWithBasis.ElementMethods method), 931</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.lie_algebras_with_basis, LieAlgebras.ElementMethods method), 913</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.lie_algebras_with_basis, LieAlgebras.ParentMethods method), 926</td>
</tr>
<tr>
<td>lift()</td>
<td>(sage.categories.sets_cat.Sets.Subquotients.ElementMethods method), 1242</td>
</tr>
<tr>
<td>lift_to_precision()</td>
<td>(sage.categories.completion_complete_discretevaluation, CompleteDiscreteValuationRings.ElementMethods method), 354</td>
</tr>
<tr>
<td>LiftMorphism</td>
<td>(class in sage.categories.lie_algebras), 931</td>
</tr>
<tr>
<td>linear_combination()</td>
<td>(sage.categories.modules.Modules.ParentMethods method), 1018</td>
</tr>
<tr>
<td>list()</td>
<td>(sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 509</td>
</tr>
<tr>
<td>list()</td>
<td>(sage.categories.infinite_enumerated_sets.InfiniteEnumeratedSets.ParentMethods method), 897</td>
</tr>
<tr>
<td>local_energy_function()</td>
<td>(sage.categories.loop_crystals, KirillovReshetikhinCrystals.ParentMethods method), 952</td>
</tr>
<tr>
<td>LocalEnergyFunction</td>
<td>(class in sage.categories.loop_crystals), 961</td>
</tr>
<tr>
<td>long_element()</td>
<td>(sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 614</td>
</tr>
<tr>
<td>LoopCrystals</td>
<td>(class in sage.categories.loop_crystals), 962</td>
</tr>
<tr>
<td>LoopCrystals.ParentMethods</td>
<td>(class in sage.categories.loop_crystals), 963</td>
</tr>
<tr>
<td>lower_central_series()</td>
<td>(sage.categories.finite_dimensional_lie_algebras_with_basis, FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 687</td>
</tr>
<tr>
<td>lower_cover_reflections()</td>
<td>(sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 421</td>
</tr>
<tr>
<td>lower_covers()</td>
<td>(sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 421</td>
</tr>
<tr>
<td>lower_covers()</td>
<td>(sage.categories.posets.PoSet Любители методов), 1111</td>
</tr>
<tr>
<td>lowest_weight_vectors()</td>
<td>(sage.categories.highest_weight_crystals, HighestWeightCrystals.ParentMethods method), 879</td>
</tr>
<tr>
<td>lowest_weight_vectors()</td>
<td>(sage.categories.supercrystals, SuperCrystals.Finite.ParentMethods method), 1303</td>
</tr>
<tr>
<td>lt()</td>
<td>(sage.categories.posets.PoSet Любителеi методов), 1111</td>
</tr>
</tbody>
</table>
Manifolds (class in sage.categories.manifolds), 998
Manifolds.AlmostComplex (class in sage.categories.manifolds), 998
Manifolds.Analytic (class in sage.categories.manifolds), 998
Manifolds.Connected (class in sage.categories.manifolds), 999
Manifolds.Differentiable (class in sage.categories.manifolds), 999
Manifolds.FiniteDimensional (class in sage.categories.manifolds), 999
Manifolds.ParentMethods (class in sage.categories.manifolds), 1000
Manifolds.SMOOTH (class in sage.categories.manifolds), 1000
Manifolds.SubcategoryMethods (class in sage.categories.manifolds), 1000
Map (class in sage.categories.map), 165
map () (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 510
map_coefficients () (sage.categories.modules_with_ba-asis.ModulesWithBasis.ElementMethods method), 1042
map_item () (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1044
map_support () (sage.categories.modules_with_ba-asis.ModulesWithBasis.ElementMethods method), 1045
map_support_skip_none () (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1046
matrix () (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ElementMethods method), 1004
MatrixAlgebras (class in sage.categories.matrix_algebras), 1003
MatrixFunctor (class in sage.categories.pushout), 220
maximal_degree () (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 551
maximal_degree () (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.Subobjects.ElementMethods method), 570
maximal_vector () (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 952
meet () (sage.categories.category.Category static method), 88
meet () (sage.categories.lattice_posets.LatticePosets.ParentMethods method), 909
meet_irreducibles () (sage.categories.finite_lati-
meet_irreducibles_poset () (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 755
merge () (sage.categories.pushout.AlgebraicClosureFunc-
tor method), 195
merge () (sage.categories.pushout.AlgebraicExtensionFunc-
tor method), 199
merge () (sage.categories.pushout.CompletionFunctor method), 204
merge () (sage.categories.pushout.ConstructionFunctor method), 210
merge () (sage.categories.pushout.InfinitePolynomial-
Functor method), 218
merge () (sage.categories.pushout.LaurentPolynomial-
Functor method), 219
merge () (sage.categories.pushout.MatrixFunctor method), 221
merge () (sage.categories.pushout.MultivariatePolynomialFunctor method), 223
merge () (sage.categories.pushout.PermutationGroupFunctor method), 225
merge () (sage.categories.pushout.PolynomialFunctor method), 226
merge () (sage.categories.pushout.QuotientFunctor method), 228
merge () (sage.categories.pushout.SubspaceFunctor method), 229
merge () (sage.categories.pushout.SubspaceFunctor method), 231
metapost () (sage.categories.crystals.Crystals.ParentMethods method), 487
Metric (sage.categories.sets_cat.Sets attribute), 1222
metric () (sage.categories.metric_spaces.MetricSpaces.ParentMethods method), 1008
Metric() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1237
metric_function () (sage.categories.metric_spaces.MetricSpaces.ParentMethods method), 1008
MetricSpaces (class in sage.categories.metric_spaces), 1004
MetricSpaces.CartesianProducts (class in sage.categories.metric_spaces), 1005
MetricSpaces.CartesianProducts.ParentMethods (class in sage.categories.metric_spaces), 1005
MetricSpacesCategory (class in sage.categories.metric_spaces), 1009
MetricSpaces.Complete (class in sage.categories.metric_spaces), 1006
MetricSpaces.Complete.CartesianProducts (class in sage.categories.metric_spaces),
MetricSpaces.ElementMethods (class in sage.categories.metric_spaces), 1007
MetricSpaces.Homsets (class in sage.categories.metric_spaces), 1007
MetricSpaces.Homsets.ElementMethods (class in sage.categories.metric_spaces), 1008
MetricSpaces.ParentMethods (class in sage.categories.metric_spaces), 1008
MetricSpaces.SubcategoryMethods (class in sage.categories.metric_spaces), 1009
MetricSpaces.WithRealizations (class in sage.categories.metric_spaces), 1009
MetricSpaces.WithRealizations.ParentMethods (class in sage.categories.metric_spaces), 1009
milnor_fiber_poset() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 586
min_demazure_product_greater() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 422
minimal_polynomial() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.Homsets.Endset.ElementMethods method), 712
minpoly() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.Homsets.Endset.ElementMethods method), 714
ModularAbelianVarieties (class in sage.categories.modular_abelian_varieties), 1010
ModularAbelianVarieties.Homsets (class in sage.categories.modular_abelian_varieties), 1010
ModularAbelianVarieties.Homsets.Endset (class in sage.categories.modular_abelian_varieties), 1010
module
sage.categories.action, 249
sage.categories.additive_groups, 254
sage.categories.additive_magmas, 257
sage.categories.additive_monoids, 274
sage.categories.additive_semigroups, 276
sage.categories.affine_weyl_groups, 279
sage.categories.algebra_functor, 1359
sage.categories.algebra_ideals, 284
sage.categories.algebra_modules, 285
sage.categories.algebras, 286
sage.categories.algebras_with_basis, 291
sage.categories.all, 1
sage.categories.aperiodic_semigroups, 299
sage.categories.associative_algebras, 300
sage.categories.bialgebras, 300
sage.categories.bialgebras_with_basis, 302
sage.categories.bimodules, 309
sage.categories.cartesian_product, 1353
sage.categories.category, 43
sage.categories.category_cy_helper, 1496
sage.categories.category_singleton, 1491
sage.categories.category_types, 1487
sage.categories.category_with_axiom, 101
sage.categories.classical_crystals, 311
sage.categories.coalgebras, 317
sage.categories.coalgebras_with_basis, 326
sage.categories.coercion_methods, 1500
sage.categories.commutative_additive_groups, 329
sage.categories.commutative_additive_monoids, 331
sage.categories.commutative_additive_semigroups, 332
sage.categories.commutative_algebraIdeals, 333
sage.categories.commutative_algebras, 334
sage.categories.commutative_ringIdeals, 335
sage.categories.commutative_rings, 335
sage.categories.complete_discrete_valuation, 349
sage.categories.complex_reflection_groups, 356
sage.categories.complex_reflection_or_generalized_coxeter_groups, 359
sage.categories.covariant_functorial_construction, 1345
sage.categories.coxeter_group_algebras, 388
sage.categories.coxeter_groups, 392
sage.categories.crystals, 456
sage.categories.cw_complexes, 494
sage.categories.discrete_valuation, 498
sage.categories.distributive_magnas_and_additive_magnas, 502
sage.categories.division_rings, 503
sage.categories.domains, 505
sage.categories.dual, 1359
sage.categories.enumerated_sets, 506
sage.categories.euclidean_domains, 516
sage.categories.examples.algebras_with_basis, 1391
sage.categories.examples.commutative_additive_monoids, 1392
sage.categories.examples.commutative_additive_semigroups, 1394
sage.categories.examples.coxeter_groups, 1398
sage.categories.examples.crystals, 1398
sage.categories.examples.cw_complexes, 1401
sage.categories.examples.facade_sets, 1403
sage.categories.examples.finite_coxeter_groups, 1403
sage.categories.examples.finite_dimensional_algebras_with_basis, 1407
sage.categories.examples.finite_dimensional_lie_algebras_with_basis, 1409
sage.categories.examples.finite.enumerated_sets, 1416
sage.categories.examples.finite_monoids, 1419
sage.categories.examples.finite_semigroups, 1422
sage.categories.examples.finite_weyl_groups, 1426
sage.categories.examples graded_connected_hopf_algebras_with_basis, 1429
sage.categories.examplesgraded_modules_with_basis, 1431
sage.categories.examples.graphs, 1435
sage.categories.examples.hopf_algebras_with_basis, 1437
sage.categories.examples.infinite_enumerated_sets, 1439
sage.categories.examples.lie_algebras, 1442
sage.categories.examples.lie_algebras_with_basis, 1443
sage.categories.examples.magmas, 1446
sage.categories.examples.manifolds, 1448
sage.categories.examples.monoids, 1449
sage.categories.examples.posets, 1452
sage.categories.examples.semi_groups, 1455
sage.categories.examples.semi_groups_cython, 1462
sage.categories.examples.sets_cat, 1466
sage.categories.examples.stacks, 1468
sage.categories.examples.supports, 1469
sage.categories.examples.with_realizations, 1478
sage.categories.facade_sets, 1342
sage.categories.fields, 519
sage.categories.filtered_algebras, 527
sage.categories.filtered_algebras_with_basis, 528
sage.categories.filtered_modules, 541
sage.categories.filtered_modules_with_basis, 543
sage.categories.finite_complex_reflection_groups, 571
sage.categories.finite_coxeter_groups, 601
sage.categories.finite_crystals, 623
sage.categories.finite_dimensional_algebras_with_basis, 624
sage.categories.finite_dimensional_bialgebras_with_basis, 658
sage.categories.finite_dimensional_coalgebras_with_basis, 659
sage.categories.finite_dimensional_graded_lie_algebras_with_basis, 662
sage.categories.finite_dimensional_graded_lie_rings_with_basis, 663
monomial_coefficients() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1046

monomial_or_zero_if_none() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1047

monomials() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1047

Morphism (class in sage.categories.morphism), 189

morphism() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 689

morphism_class() (sage.categories.category.Category method), 88

multiplication_table() (sage.categories.magmas.Magmas.ParentMethods method), 972


MultiPolynomialFunctor (class in sage.categories.pushout), 222

MultivariateConstructionFunctor (class in sage.categories.pushout), 224

MyGroupAlgebra (class in sage.categories.examples.hopf_algebras_with_basis), 1437

N

NaiveCrystal (class in sage.categories.examples.crystals), 1400

NaiveCrystal.Element (class in sage.categories.examples.crystals), 1400

natural_map() (sage.categories.homset.Homset method), 183

nerve() (sage.categories.finite_monoids.FiniteMonoids.ParentMethods method), 757

next() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 511

next() (sage.categories.examples.infinite_enumerated_sets.NonNegativeIntegers.ElementAbstract method), 1469

next() (sage.categories.examples.sets_cat.PrimeNumbers_Abstract method), 1468

ngens() (sage.categories.finitely_generated_lambda_bracket_algebras.FinitelyGeneratedLambdaBracketAlgebras.ParentMethods method), 810


Nilpotent (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis attribute), 665

Nilpotent() (sage.categories.lie_algebras.LieAlgebras.SubcategoryMethods method), 929

nilradical() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 691

nilradical_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 691

noncrossing_partition_lattice() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 579

NonNegativeIntegers (class in sage.categories.examples.infinite_enumerated_sets), 1439

NonNegativeIntegers (class in sage.categories.examples.sets_with_grading), 1476

normalizer() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 694

normalizer_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 694

NoZeroDivisors (sage.categories.rings.Rings attribute), 1165

NoZeroDivisors() (sage.categories.rings.Rings.SubcategoryMethods method), 1178

nproduct() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.ElementMethods method), 903


number_of_irreducible_components() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 378

number_of_reflection_hyperplanes() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 587

number_of_reflections() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 588

sis() (sage.categories.algebras_with_basis.AlgebrasWithBasis.CartesianProducts.ParentMethods method), 294
one_from_one_basis() (sage.categories.unital_algebras.UnitalAlgebras.WithBasis.ParentMethods method), 1317
op (sage.categories.action.Action attribute), 251
operation() (sage.categories.action.Action method), 251
or_subcategory() (sage.categories.category.Category method), 89
order() (sage.categories.groups.Groups.CartesianProducts.ParentMethods method), 859
order_filter() (sage.categories.posets.Posets.ParentMethods method), 1112
order_filter_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 789
order_ideal() (sage.categories.posets.Posets.ParentMethods method), 1112
order_ideal_complement_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 789
order_ideal_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 791
order_ideal_toggle() (sage.categories.posets.Posets.ParentMethods method), 1113
order_ideal_toggles() (sage.categories.posets.Posets.ParentMethods method), 1114
order_ideals_lattice() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 792
orthogonal_idempotents_central_mod_radical() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 645
over() (sage.categories.commutative_rings.CommutativeRings.ParentMethods method), 345
parent_class() (sage.categories.category.Category method), 90
part() (sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras.ElementMethods method), 1308
partial_fraction_decomposition() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 1125
PartiallyOrderedMonoids (class in sage.categories.partially_ordered_monoids), 1098
PartiallyOrderedMonoids.ElementMethods (class in sage.categories.partially_ordered_monoids), 1098
PartiallyOrderedMonoids.ParentMethods (class in sage.categories.partially_ordered_monoids), 1098
pbw_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 934
peirce_decomposition() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 647
peirce_summand() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 649
permutahedron() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 616
PermutationGroupFunctor (class in sage.categories.pushout), 224
PermutationGroups (class in sage.categories.permutation_groups), 1099
Phi() (sage.categories.crystals.Crystals.ElementMethods method), 465
phi() (sage.categories.crystals.Crystals.ElementMethods method), 469
phi() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1145
phi_minus_epsilon() (sage.categories.crystals.Crystals.ElementMethods method), 470
pieri_factors() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 1339
Plane (class in sage.categories.examples.manifolds), 1448
plot() (sage.categories.crystals.Crystals.ParentMethods method), 488
plot3d() (sage.categories.crystals.Crystals.ParentMethods method), 489
poincare_birkhoff_witt_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 934
product_on_basis() (sage.categories.additive_semigroups.AdditiveSemigroups.Algebras.ParentMethods method), 277
product_on_basis() (sage.categories.algebras_with_basis.AlgebrasWithBasis.TensorProducts.ParentMethods method), 298
product_on_basis() (sage.categories.examples.algebras_with_basis.FreeAlgebra method), 1391
product_on_basis() (sage.categories.examples.finite_dimensional_algebras_with_basis.KroneckerQuiverPathAlgebra method), 1408
product_on_basis() (sage.categories.examples.graded_connected_hopf_algebras_with_basis.GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator method), 1430
product_on_basis() (sage.categories.exceptions.algebras_with_basis.FreeAlgebra method), 138
product_on_basis() (sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis.SignedTensorProducts.ParentMethods method), 827
product_on_basis() (sage.categories.semigroups.Semigroups.Algebras.ParentMethods method), 1187
product_space() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 695
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods method), 765
profile_polynomial() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods method), 766
profile_series() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods method), 767
projection() (sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis.ParentMethods method), 539
projection() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 567
pseudo_order() (sage.categories.finite_monoids.FiniteMonoids.ElementMethods method), 756
pushforward() (sage.categories.morphism.Morphism method), 191
pushout() (module sage.categories.pushout), 234
pushout() (sage.categories.pushout.ConstructionFunctor method), 211
pushout_lattice() (module sage.categories.pushout), 247
Q
q() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 1133
q_dimension() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 880
q_dimension() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 953
quantum_bruhat_graph() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 1340
quantum_bruhat_successors() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 1332
QuantumGroupRepresentations (class in sage.categories.quantum_group_representations), 1132
QuantumGroupRepresentations.ParentMethods (class in sage.categories.quantum_group_representations), 1132
QuantumGroupRepresentations.TensorProducts (class in sage.categories.quantum_group_representations), 1133
QuantumGroupRepresentations.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 1133
QuantumGroupRepresentations.WithBasis (class in sage.categories.quantum_group_representations), 1134
QuantumGroupRepresentations.WithBasis.ElementMethods (class in sage.categories.quantum_group_representations), 1134
QuantumGroupRepresentations.WithBasis.ParentMethods (class in sage.categories.quantum_group_representations), 1136
QuantumGroupRepresentations.WithBasis.TensorProducts (class in sage.categories.quantum_group_representations), 1137
QuantumGroupRepresentations.WithBasis.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 1137
radical() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 653
radical() (sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods method), 736
radical() (sage.categories.semisimple_algebras.SemisimpleAlgebras.ParentMethods method), 1208
random_element() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 512
random_element() (sage.categories.finiteEnumeratedSets.FiniteEnumeratedSets.ParentMethods method), 744
random_element() (sage.categories.infiniteEnumeratedSets.InfiniteEnumeratedSets.ParentMethods method), 897
random_element() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1069
rank (sage.categories.pushout.AlgebraicClosureFunctor attribute), 195
rank (sage.categories.pushout.AlgebraicExtensionFunctor attribute), 202
rank (sage.categories.pushout.BlackBoxConstruction Functor attribute), 203
rank (sage.categories.pushout.CompletionFunctor attribute), 205
rank (sage.categories.pushout.FractionField attribute), 214
rank (sage.categories.pushout.IdentityConstructionFunctor attribute), 214
rank (sage.categories.pushout.InfinitePolynomialFunctor attribute), 218
rank (sage.categories.pushout.LaurentPolynomialFunctor attribute), 220
rank (sage.categories.pushout.MatrixFunctor attribute), 222
rank (sage.categories.pushout.MultiPolynomialFunctor attribute), 224
rank (sage.categories.pushout.PermutationGroupFunctor attribute), 225
rank (sage.categories.pushout.PolynomialFunctor at-
reflection()  (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 379
reflection_index_set()  (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 380
reflection_length()  (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 368
reflection_length()  (sage.categories.complex_reflection_groups.ComplexReflectionGroups.ElementMethods method), 427
reflection_length()  (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ElementMethods method), 573
reflection_representation()  (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 449
reflection_to_coroot()  (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 1333
reflection_to_root()  (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 1334
reflections()  (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 381
reflections_from_w0()  (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 617
register_as_coercion()  (sage.categories.morphism.Morphism method), 191
register_as_conversion()  (sage.categories.morphism.Morphism method), 192
RegressiveCovariantConstructionCategory  (class in sage.categories.covariant_functorial_construction), 1352
regular_representation()  (sage.categories.semi_groups.Semigroups.Algebras.ParentMethods method), 1188
regular_representation()  (sage.categories.semi_groups.Semigroups.ParentMethods method), 1196
RegularCrystals  (class in sage.categories.regular_crystals), 1143
RegularCrystals.ElementMethods  (class in sage.categories.regular_crystals), 1143
RegularCrystals.MorphismMethods  (class in sage.categories.regular_crystals), 1149
RegularCrystals.ParentMethods  (class in sage.categories.regular_crystals), 1149
RegularCrystals.TensorProducts (class in sage.categories.regular_crystals), 1153
RegularLoopCrystals (class in sage.categories.loop_crystals), 964
RegularLoopCrystals.ElementMethods (class in sage.categories.loop_crystals), 964
RegularSuperCrystals (class in sage.categories.regular_supercrystals), 1155
RegularSuperCrystals.ElementMethods (class in sage.categories.regular_supercrystals), 1155
RegularSuperCrystals.TensorProducts (class in sage.categories.regular_supercrystals), 1155
representation() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 927
representation() (sage.categories.semigroups.Semigroups.ParentMethods method), 1196
required_methods() (sage.categories.category.Category method), 91
retract() (sage.categories.examples.finite.enumerated_sets.IsomorphicObjectOfFiniteEnumeratedSet method), 1419
retract() (sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup method), 1461
reversed() (sage.categories.homset.Homset method), 184
rhodes_radical_congruence() (sage.categories.finite_monoids.FiniteMonoids.ParentMethods method), 761
right_base_ring() (sage.categories.bimodules.Bimodules.method), 311
right_domain() (sage.categories.action.Action method), 251
right_precomposition (sage.categories.action.PrecomposedAction attribute), 254
RightModules (class in sage.categories.right_modules), 1157
RightModules.ElementMethods (class in sage.categories.right_modules), 1157
RightModules.ParentMethods (class in sage.categories.right_modules), 1157
ring() (sage.categories.category_types.Category_ideal method), 1487
RingIdeals (class in sage.categories.ring_ideals), 1158
Rings (class in sage.categories.rings), 1159
Rings.ElementMethods (class in sage.categories.rings), 1160
Rings.MorphismMethods (class in sage.categories.rings), 1161
Rings.ParentMethods (class in sage.categories.rings), 1165
Rings.SubcategoryMethods (class in sage.categories.rings), 1178
Rngs (class in sage.categories.rngs), 1179
rowmotion() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 797
rowmotion_orbit_iter() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 797
rowmotion_orbits() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 799
rowmotion_orbits_plots() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 800
RTrivial (sage.categories.semi groups.Semigroups attribute), 1200
RTrivial() (sage.categories.semi groups.SubcategoryMethods method), 1203
RTrivial_extra_super_categories() (sage.categories.l_trivial_semigroups.LTrivialSemigroups method), 965
RTrivialSemigroups (class in sage.categories.r_trivial_semigroups), 1180
S
s () (sage.categories.crystals.Crystals.ElementMethods method), 470
s () (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 954
sage.categories.action module, 249
categories.additive_groups module, 254
categories.additive_magmas module, 257
categories.additive_monoids module, 274
categories.additive_semigroups module, 276
categories.affine_weyl_groups module, 279
categories.algebra_functor module, 1359
categories.algebra_ideals
module, 284
sage.categories.algebra_modules
    module, 285
sage.categories.algebras
    module, 286
sage.categories.algebras_with_basis
    module, 291
sage.categories.all
    module, 1
sage.categories.aperiodic_semigroups
    module, 299
sage.categories.associative_algebras
    module, 300
sage.categories.bialgebras
    module, 300
sage.categories.bialgebras_with_basis
    module, 302
sage.categories.bimodules
    module, 309
sage.categories.cartesian_product
    module, 1353
sage.categories.category
    module, 43
sage.categories.category_CY_helper
    module, 1496
sage.categories.category_singleton
    module, 1491
sage.categories.category_types
    module, 1487
sage.categories.category_with_axiom
    module, 101
sage.categories.classical_crystals
    module, 311
sage.categories.coalgebras
    module, 317
sage.categories.coalgebras_with_basis
    module, 326
sage.categories.coercion_methods
    module, 1500
sage.categories.commutative_additive_groups
    module, 329
sage.categories.commutative_additive_monoids
    module, 331
sage.categories.commutative_additive_semigroups
    module, 332
sage.categories.commutative_algebra_ideals
    module, 333
sage.categories.commutative_algebras
    module, 334
sage.categories.commutative_ring_ideals
    module, 335
sage.categories.commutative_rings
    module, 335
sage.categories.complete_discrete_valuation
    module, 349
sage.categories.complex_reflection_groups
    module, 356
sage.categories.complex_reflection_or_generalized_coxeter_groups
    module, 359
sage.categories.covariant_functorial_construction
    module, 1345
sage.categories.coxeter_group_algebras
    module, 388
sage.categories.coxeter_groups
    module, 392
sage.categories.crystals
    module, 456
sage.categories.cw_complexes
    module, 494
sage.categories.discrete_valuation
    module, 498
sage.categories.distributive_magma_and_additive_magmas
    module, 502
sage.categories.division_rings
    module, 503
sage.categories.domains
    module, 505
sage.categories.dual
    module, 1359
sage.categories.enumerated_sets
    module, 506
sage.categories.euclidean_domains
    module, 516
sage.categories.examples.algebras_with_basis
    module, 1391
sage.categories.examples.commutative_additive_monoids
    module, 1392
sage.categories.examples.commutative_additive_semigroups
    module, 1394
sage.categories.examples.coxeter_groups
    module, 1398
sage.categories.examples.crystals
module, 708
sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis module, 730
sage.categories.finite_dimensional_semisimple_algebras_with_basis module, 732
sage.categories.finiteEnumerated_sets module, 736
sage.categories.finiteFields module, 746
sage.categories.finiteGroups module, 747
sage.categories.finiteLatticePosets module, 751
sage.categories.finiteMonoids module, 756
sage.categories.finitePermutationGroups module, 762
sage.categories.finitePosets module, 768
sage.categories.finiteSemigroups module, 804
sage.categories.finiteSets module, 806
sage.categories.finiteWeylGroups module, 808
sage.categories.finitelyGenerated_lambdaBracketAlgebras module, 809
sage.categories.finitelyGenerated_lieConformalAlgebras module, 811
sage.categories.finitelyGenerated_magmas module, 813
sage.categories.finitelyGeneratedSemigroups module, 814
sage.categories.functionFields module, 818
sage.categories.functor module, 147
sage.categories.g_sets module, 818
sage.categories.gcd_domains module, 819
sage.categories.generalizedCoxeterGroups module, 820
sage.categories.graded_algebras module, 822
sage.categories.graded_algebras_with_basis module, 824
sage.categories.graded_bialgebras module, 828
sage.categories.graded_bialgebras_with_basis module, 828
sage.categories.graded_coalgebras module, 829
sage.categories.graded_coalgebras_with_basis module, 830
sage.categories.graded_hopf_algebras module, 831
sage.categories.graded_hopf_algebras_with_basis module, 831
sage.categories.graded_lie_algebras module, 833
sage.categories.graded_lie_algebras_with_basis module, 835
sage.categories.graded_lie_conformal_algebras module, 835
sage.categories.graded_modules module, 837
sage.categories.graded_modules_with_basis module, 839
sage.categories.graphs module, 846
sage.categories.groupAlgebras module, 849
sage.categories.groupoid module, 857
sage.categories.groups module, 858
sage.categories.h_trivialSemigroups module, 895
sage.categories.hecke_modules module, 872
sage.categories.highest_weight_cryystals module, 874
sage.categories.homset module, 174
sage.categories.homsets module, 1373
sage.categories.hopf_algebras module, 885
sage.categories.hopf_algebras_with_basis module, 1004
sage.categories.infinite enumerated_sets module, 898
sage.categories.integral_domains module, 1372
sage.categories.isomorphic_objects module, 900
sage.categories.j_trivial_semigroups module, 900
sage.categories.kac_moody_algebras module, 900
sage.categories.l_trivial_semigroups module, 900
sage.categories.lambda_bracket_algebras module, 902
sage.categories.lambda_bracket_algebras_with_basis module, 905
sage.categories.lattice_posets module, 908
sage.categories.left_modules module, 909
sage.categories.lie_algebras module, 910
sage.categories.lie_algebras_with_basis module, 931
sage.categories.lie_conformal_algebras module, 935
sage.categories.lie_conformal_algebras_with_basis module, 940
sage.categories.lie_groups module, 943
sage.categories.loop_crystals module, 944
sage.categories.magas module, 966
sage.categories.magas_and_additive_magas module, 989
sage.categories.magnetic_algebras module, 992
sage.categories.manifolds module, 997
sage.categories.map module, 157
sage.categories.matrix_algebras module, 1003
sage.categories.metric_spaces module, 1004
sage.categories.modular_abelian_varieties module, 1010
sage.categories.modules module, 1011
sage.categories.modules_with_basis module, 1030
sage.categories.monoid_algebras module, 1081
sage.categories.monoids module, 1081
sage.categories.morphism module, 187
sage.categories.number_fields module, 1092
sage.categories.objects module, 1096
sage.categories.partially_ordered_monoids module, 1098
sage.categories.permutation_groups module, 1099
sage.categories.pointed_sets module, 1100
sage.categories.polyhedra module, 1100
sage.categories.poor_man_map module, 1500
sage.categories.posets module, 1101
sage.categories.primer module, 3
sage.categories.principal_ideal_domains module, 1116
sage.categories.pushout module, 195
sage.categories.quantum_group_representation module, 1132
sage.categories.quotient_fields module, 1121
sage.categories.quotients module, 1370
sage.categories.r_trivial_semigroups module, 1180
sage.categories.realizations module, 1379
sage.categories.regular_crystals module, 1132
sage.categories.regular_supercrystals module, 1155
sage.categories.right_modules module, 1555
module, 1157
sage.categories.ring_ideals module, 1158
sage.categories.rings module, 1159
sage.categories.rings module, 1179
sage.categories.schemes module, 1180
sage.categories.semigroups module, 1185
sage.categories.semirings module, 1205
sage.categories.semisimple_algebras module, 1206
sage.categories.sets_cat module, 1208
sage.categories.sets_with_grading module, 1252
sage.categories.sets_with_partial_maps module, 1256
sage.categories.shephard_groups module, 1257
sage.categories.tensor module, 1356
sage.categories.topological_spaces module, 1305
sage.categories.triangular_kac_moody_algebras module, 1308
sage.categories.tutorial module, 154
sage.categories.unique_factorization_domains module, 1311
sage.categories.unital_algebras module, 1313
sage.categories.vector_bundles module, 1317
sage.categories.vector_spaces module, 1320
sage.categories.weyl Groups module, 1326
sage.categories.with_realizations module, 1382

scaling_factors() (sage.categories.crystals.CrystalMorphism method), 461
Schemes (class in sage.categories.schemes), 1183
Schemes_over_base (class in sage.categories.schemes), 1184
Section (class in sage.categories.map), 172
section() (sage.categories.map.FormalCompositeMap method), 163
section() (sage.categories.map.Map method), 172
section() (sage.categories.morphism.IdentityMorphism method), 189
section() (sage.categories.morphism.SetIsomorphism method), 194
semigroup_generators() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 870
semigroup_generators() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 382
semigroup_generators() (sage.categories.finite_monoids.IntegerModMonoid method), 1421
semigroup_generators() (sage.categories.finite_monoids.LeftRegularBand method), 1426
semigroup_generators() (sage.categories.finite_semigroups.FreeSemigroup method), 1456
semigroup_generators() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), Index
SimplicialComplexes (class in sage.categories.simplicial_complexes), 1257
SimplicialComplexes.Finite (class in sage.categories.simplicial_complexes), 1258
SimplicialComplexes.Finite.ParentMethods (class in sage.categories.simplicial_complexes), 1258
SimplicialComplexes.Finite.ParentMethods (class in sage.categories.simplicial_complexes), 1258
SimplicialComplexes.SubcategoryMethods (class in sage.categories.simplicial_complexes), 1259
SimplicialSets (class in sage.categories.simplicial_sets), 1260
SimplicialSets.Finite (class in sage.categories.simplicial_sets), 1260
SimplicialSets.Homsets (class in sage.categories.simplicial_sets), 1260
SimplicialSets.Homsets.Endset (class in sage.categories.simplicial_sets), 1261
SimplicialSets.Homsets.Endset.ParentMethods (class in sage.categories.simplicial_sets), 1261
SimplicialSets.ParentMethods (class in sage.categories.simplicial_sets), 1261
SimplicialSets.Pointed (class in sage.categories.simplicial_sets), 1263
SimplicialSets.Pointed.Finite (class in sage.categories.simplicial_sets), 1263
SimplicialSets.Pointed.Finite.ParentMethods (class in sage.categories.simplicial_sets), 1263
SimplicialSets.Pointed.ParentMethods (class in sage.categories.simplicial_sets), 1265
SimplicialSets.SubcategoryMethods (class in sage.categories.simplicial_sets), 1279
Smooth (class in sage.categories.manifolds.Manifolds.SubcategoryMethods), 1002
Smooth (class in sage.categories.vector_bundles.VectorBundles.SubcategoryMethods), 1318
solvable_radical (class in sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods), 698
solvable_radical_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 698
some_elements (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 386
some_elements() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 513

stembridgeDelta_rise() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1147

stembridgeDelta_depth() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1147

stembridgeDelta_rise() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1147

stembridgeTriple() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1147

step() (sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods method), 732


Stratified() (sage.categories.graded_lie_algebras.GradedLieAlgebras.SubcategoryMethods method), 834

string_parameters() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ElementMethods method), 875

structure() (sage.categories.category.Category method), 91

structure_coefficients() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 810

subalgebra() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebras.ParentMethods method), 1415

subalgebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 701

subalgebra() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 928

subcategory_class() (sage.categories.category.Category method), 92

subcrystal() (sage.categories.crystals.Crystals.ElementMethods method), 470

subcrystal() (sage.categories.crystals.Crystals.ElementMethods method), 489

submodule() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ParentMethods method), 842

submodule() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1070

submonoid() (sage.categories.monoids.Monoids.ParentMethods method), 1089

Subobjects() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1238

SubobjectsCategory (class in sage.categories.subobj...
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>super_categories()</td>
<td>(sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups method), 387</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.covariant_functorial_construction.FunctorialConstructionCategory method), 1352</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.coquiver.CoxeterGroups method), 455</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.crystals.Crystals method), 494</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.cw_complexes.CWComplexes method), 497</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.discrete_valuation.DiscreteValuationFields method), 499</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.discrete_valuation.DiscreteValuationRings method), 502</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.domainsDomains method), 505</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.enumerated_sets.EnumeratedSets method), 515</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.euclidean_domains.EuclideanDomains method), 518</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.examples.semi_groups_cython.IdempotentSemigroups method), 1462</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.examples.with_realizations.SubsetAlgebra.Bases method), 1481</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.function_fields.FunctionFields method), 818</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.g_sets.GSets method), 819</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.gcd_domains.GcdDomains method), 820</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups method), 822</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.graded_hopf_algebras_with_basis.GradedHopfAlgebrasWithBasis,WithRealizations method), 833</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.graphs.Graphs method), 848</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.groupoid.Groupoid method), 857</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.hecke_modules.HeckeModules method), 873</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.highest_weight_crystals.HighestWeightCrystals method), 885</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.homsets.Homsets method), 1376</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.homsets.Homsets method), 1379</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.hopf_algebras.HopfAlgebras method), 889</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.kac_moody_algebras.KacMoodyAlgebras method), 901</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.lambdaBracket_algebras.LambdaBracketAlgebras method), 905</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.lattice_posets.LatticePosets method), 909</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.lie_algebras.LieAlgebras method), 930</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.lie_conformal_algebras.LieConformalAlgebras method), 939</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.lie_groups.LieGroups method), 944</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.loop_crystals.KirillovReshetikhinCrystals method), 961</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.loop_crystals.LoopCrystals method), 964</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.loop_crystals.RegularLoopCrystals method), 965</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.magma_and_additive_magmas.MagmasAndAdditiveMagmas method), 991</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.magnmas.Magnmas method), 989</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.monomic_algebras.MagmaticAlgebras method), 997</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.manifolds.ComplexManifolds method), 997</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.manifolds.Manifolds method), 1003</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.matrix_algebras.MatrixAlgebras method), 1004</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.modular_abelian_varieties.ModularAbelianVarieties method), 1011</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.modules.Modules method), 1029</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.number_fields.NumberFields method), 1095</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.objects.Objects method), 1098</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.permutation_groups.PermutationGroups method), 1099</td>
</tr>
<tr>
<td>super_categories()</td>
<td>(sage.categories.sets_of method), 1379</td>
</tr>
</tbody>
</table>
term() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 1076

terms() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1049

TestObjects (class in sage.categories.category_with_axiom), 142

TestObjects.Commutative (class in sage.categories.category_with_axiom), 142

TestObjects.Commutative.Facade (class in sage.categories.category_with_axiom), 142

TestObjects.Commutative.Finite (class in sage.categories.category_with_axiom), 142

TestObjects.Commutative.FiniteDimensional (class in sage.categories.category_with_axiom), 142

TestObjects.FiniteDimensional (class in sage.categories.category_with_axiom), 142

TestObjects.FiniteDimensional.Finite (class in sage.categories.category_with_axiom), 142

TestObjects.FiniteDimensional.Unital (class in sage.categories.category_with_axiom), 142

TestObjects.Unital (class in sage.categories.category_with_axiom), 143

the_answer() (sage.categories.examplesemigroups.QuotientOfLeftZeroSemigroup method), 1462

then() (sage.categories.map.FormalCompositeMap method), 164

to_graded_conversion() (sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis.ParentMethods method), 540

to_graded_conversion() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 568

to_highest_weight() (sage.categories.crystals.Crystals.ElementMethods method), 472

to_lowest_weight() (sage.categories.crystals.Crystals.ElementMethods method), 473

to_matrix() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ElementMethods method), 574


to_matrix() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ElementMethods method), 633

to_module_generator() (sage.categories.crystals.CrystalMorphismByGenerators method), 463

to_vector() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 1410


to_vector() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 664

to_vector() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ElementMethods method), 931


to_vector() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ElementMethods method), 914

toggling_orbit_iter() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 800


toggling_orbits() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 802


toggling_orbits_plots() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 803

Topological (sage.categories.sets_cat.Sets.attribute), 1244

Topological() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 1241

TopologicalSpaces (class in sage.categories.topological_spaces), 1305

TopologicalSpaces.CartesianProducts
TopologicalSpacesCategory (class in sage.categories.topological_spaces), 1305
TopologicalSpaces.Compact (class in sage.categories.topological_spaces), 1306
TopologicalSpaces.Connected (class in sage.categories.topological_spaces), 1306
TopologicalSpaces.SubcategoryMethods (class in sage.categories.topological_spaces), 1307
trace() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.Endset.ElementMethods method), 715
tailing_coefficient() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 715
tailing_item() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1051
tailing_monomial() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1051
tailing_support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 1051
twisted_homology() (sage.categories.simplicial_sets.SimplicialSets.Pointed.ParentMethods method), 1276
twisted_invariant_module() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ParentMethods method), 729
type_to_parent() (in module sage.categories.pushout), 247

U
uncamelize() (in module sage.categories.category_with_axiom), 146
unicode_symbol (sage.categories.tensor.TensorProductFunctor attribute), 1356
UniqueFactorizationDomains (class in sage.categories.unique_factorization_domains), 1311
UniqueFactorizationDomains.ElementMethods (class in sage.categories.unique_factorization_domains), 1311
UniqueFactorizationDomains.ParentMethods (class in sage.categories.unique_factorization_domains), 1312
Unital (sage.categories.associative_algebras.AssociativeAlgebras attribute), 300
Unital (sage.categories.magmaic_algebras.MagmaicAlgebras attribute), 993
Unital (sage.categories.rings.Rings attribute), 1179
Unital (sage.categories.semigroups.Semigroups attribute), 1205
Unital() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 134
Unital() (sage.categories.magma.Magmas.SubcategoryMethods method), 982
Unital extra super_categories() (sage.categories.category_with_axiom.Bars method), 132
UnitalAlgebras (class in sage.categories.unital_algebras), 1313
UnitalAlgebras.CartesianProducts (class in sage.categories.unital_algebras), 1314
UnitalAlgebras.CartesianProducts.ParentMethods (class in sage.categories.unital_algebras), 1314
UnitalAlgebras.ParentMethods (class in sage.categories.unital_algebras), 1315
UnitalAlgebras.WithBasis (class in sage.categories.unital_algebras), 1315
UnitalAlgebras.WithBasis.ParentMethods (class in sage.categories.unital_algebras), 1315
universal_commutative_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 702
universal_cover() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationFields.ElementMethods method), 1264
universal_cover() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationRings.ElementMethods method), 351
universal_enveloping_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 498
universal_enveloping_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 500
universal_enveloping_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 1115
universal_enveloping_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 1264
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 500
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1328
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1331
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1336
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1341
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1342
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1360
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1364
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1365
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1370
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 1371
VectorSpaces.ParentMethods (class in sage.categories.vector_spaces), 1321
VectorSpaces.TensorProducts (class in sage.categories.vector_spaces), 1322
VectorSpaces.WithBasis (class in sage.categories.vector_spaces), 1322
VectorSpaces.WithBasis.CartesianProducts (class in sage.categories.vector_spaces), 1322
VectorSpaces.WithBasis.Filtered (class in sage.categories.vector_spaces), 1323
VectorSpaces.WithBasis.FiniteDimensional (class in sage.categories.vector_spaces), 1323
VectorSpaces.WithBasis.FiniteDimensional.TensorProducts (class in sage.categories.vector_spaces), 1324
verma_module() (sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras.ParentMethods method), 1310
vertices() (sage.categories.examples.graphs.Cycle method), 1436
vertices() (sage.categories.graphs.Graphs.ParentMethods method), 848
virtualization() (sage.categories.crystals.CrystalMorphism method), 462

W
w0 () (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 619
weak_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 429
weak_lattice() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 619
weak_le() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 431
weak_order_ideal() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 453
weak_poset() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 621
weight() (sage.categories.crystals.Crystals.ElementMethods method), 474
weight() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 1148
weight_lattice_realization() (sage.categories.crystals.Crystals.ParentMethods method), 492

weight_lattice_realization() (sage.categories.loop_crystals.LoopCrystals.ParentMethods method), 963
WellGenerated() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.SubcategoryMethods method), 590
weyl_group() (sage.categories.kac_moody_algebras.KacMoodyAlgebras.ParentMethods method), 900
WeylGroups (class in sage.categories.weyl_groups), 1326
WeylGroups.ElementMethods (class in sage.categories.weyl_groups), 1326
WeylGroups.ParentMethods (class in sage.categories.weyl_groups), 1338
WithBasis (sage.categories.algebras.Algebras attribute), 291
WithBasis (sage.categories.bialgebras.Bialgebras attribute), 302
WithBasis (sage.categories.coalgebras.Coalgebras attribute), 324
WithBasis (sage.categories.hopf_algebras.HopfAlgebras attribute), 888
WithBasis (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras attribute), 905
WithBasis (sage.categories.lie_algebras.LieAlgebras attribute), 930
WithBasis (sage.categories.lie_algebras.LieAlgebras.FiniteDimensional attribute), 914
WithBasis (sage.categories.lie_conformal_algebras.LieConformalAlgebras attribute), 938
WithBasis (sage.categories.modules.Modules attribute), 1029
WithBasis (sage.categories.semisimple_algebras.SemisimpleAlgebras.FiniteDimensional attribute), 1207
WithBasis() (sage.categories.modules.Modules.SubcategoryMethods method), 1025
WithRealizations() (in module sage.categories.with_realizations), 1382
WithRealizations() (sage.categories.category.Category method), 66
WithRealizationsCategory (class in sage.categories.with_realizations), 1388
wrapped_class (sage.categories.examples.finite_coxeter_groups.DihedralGroup.Element attribute), 1406
wrapped_class (sage.categories.examples.finite_monoids.IntegerModMonoid.Element attribute), 1421
wrapped_class (sage.categories.examples.finite_semigroups.LeftRegularBand.Element attribute), 1425
wrapped_class (sage.categories.examplesмагmas.FreeMagma.Element attribute), 1447
wrapped_class (sage.categories.examples.monoids.FreeMonoid.Element attribute), 1451
wrapped_class (sage.categories.examples.posets.FiniteSetsOrderedByInclusion.Element attribute), 1453
wrapped_class (sage.categories.examples.semi grupos.FreeSemigroup.Element attribute), 1456

X
xgcd() (sage.categories.fields.Fields.ElementMethods method), 524
xgcd() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 1129

Z
zero() (sage.categories.examples.commutative_additive_monoids.FreeCommutativeAdditiveMonoid method), 1394
zero() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 1415
zero() (sage.categories.examples.lie_algebras.LieAlgebraFromAssociative method), 1443
zeta_function() (sage.categories.number_fields.NumberFields.ParentMethods method), 1093