1 Introduction ......................................................... 1
2 The Sage Category Framework ................................. 3
3 Maps and Morphisms ............................................. 109
4 Individual Categories ........................................... 169
5 Functorial constructions ......................................... 867
6 Examples of parents using categories ....................... 899
7 Internals .......................................................... 961
8 Indices and Tables ................................................ 973
Python Module Index .............................................. 975
Index ..................................................................... 979
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.semingroups the Semigroups() category
- sage.categories.examples.semingroups 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
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: Permutations(5).cardinality()  
  #→ optional - sage.combinat
  120
  sage: GL(2,2).cardinality()  
  #→ optional - sage.groups
  6
  sage: A = random_matrix(ZZ, 6, 3, x=7)  
  #→ optional - sage.modules
  sage: L = LatticePolytope(A.rows())  
  #→ optional - sage.geometry.polyhedron sage.modules
  sage: L.npoints()  
  #→ optional - sage.geometry.polyhedron sage.modules
  37
  # oops! # random
  ```

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

Example: binary powering:

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

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

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

and sure enough, binary powering is defined there:

```sage
sage: m._pow_int.__module__
'sage.categories.monoids'
```

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

```sage
sage: GL(2, ZZ) in Monoids()
optional - sage.modules
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:

```sage
sage: Sets()
Category of sets
sage: Groups()
Category of groups
```

(continues on next page)
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:

```
sage: G = GL(2, ZZ) # optional - sage.modules

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

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

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

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

 Altogether, our group gets algorithms from a bunch of bookshelves:

```
sage: G.categories()  # optional - sage.modules
[Category of infinite groups, Category of groups, Category of monoids, ...

Category of magmas, Category of infinite sets, ...]
```

Those can be viewed graphically:

```
sage: g = Groups().category_graph()  # optional - sage.modules sage.graphs

sage: g.set_latex_options(format="dot2tex")  # optional - sage.modules sage.graphs

sage: view(g)  # not tested  # optional - sage.modules sage.graphs sage.plot
```

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

```
sage: g.show(vertex_shape=None, figsize=20)  # optional - sage.modules sage.graphs 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:

```
sage: IntegerRing() is IntegerRing()
sage: 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:

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

One can also create elements using various methods on the parent and arithmetic of elements:

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

Unlike parents, elements in Sage are not necessarily unique.
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:

```python
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
```

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:

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

sage: ZZ
Integer Ring

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

sage: ZZ.categories()
[Join of Category of euclidean domains
   and Category of infinite enumerated sets
   and Category of metric spaces,
   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()

sage: g.set_latex_options(format="dot2tex")

sage: view(g)
# not tested
```

---

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

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

Applying an operation is generally done by calling a method:

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

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

```sage
sage: p.factor() # optional - sage.symbolic
6*(x + 1)^2
```

```sage
sage: R.<x> = PolynomialRing(QQ, sparse=True) # optional - sage.symbolic
sage: pQ = R(p) # optional - sage.symbolic
sage: type(pQ) # optional - sage.symbolic
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category.element_class'>
```

```sage
sage: pQ.factor() # optional - sage.symbolic
(6) * (x + 1)^2
```

```sage
sage: pZ = ZZ['x'](p) # optional - sage.symbolic
sage: type(pZ) # optional - sage.symbolic
<class 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>
```

```sage
sage: pZ.factor() # optional - sage.symbolic
(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:

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

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

```python
sage: i._pow_.__module__  # not tested (Issue #24275)
'sage.categories.semigroups'
sage: pQ._mul_.__module__  # not tested - sage.symbolic
'sage.rings.polynomial.polynomial_element_generic'
sage: pQ._pow_.__module__  # not tested (Issue #24275)
'sage.categories.semigroups'
```

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:

```python
sage: P = Permutations(4)  # not tested
sage: m = P.an_element()  # not tested
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.
The full hierarchy is best viewed graphically:

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

```
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:

```
sage: G = GL(2, ZZ)                               # optional - sage.modules
```

```
sage: type(G)                                     # optional - sage.modules
<class 'sage.groups.matrix_gps.linear.LinearMatrixGroup_gap_with_category'>
```

Here is a piece of the hierarchy of classes above it:

```
sage: for cls in G.__class__.mro(): print(cls)   # optional - sage.modules
<class 'sage.groups.matrix_gps.linear.LinearMatrixGroup_gap_with_category'>
...
<class 'sage.categories.groups.Groups.parent_class'>
<class 'sage.categories.monoids.Monoids.parent_class'>
<class 'sage.categories.semisimple.Algebras.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:
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”.

### 2.1.5 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
call sage: g = class_graph(m.__class__)
    # not tested

sage: g.relabel(lambda x: x.replace("_",r"\_"))
    # not tested

sage: g.set_latex_options(format="dot2tex")
    # 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”.

### 2.1.5 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
call sage: g = class_graph(m.__class__)
    # not tested

sage: g.relabel(lambda x: x.replace("_",r"\_"))
    # not tested

sage: g.set_latex_options(format="dot2tex")
    # 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”.

### 2.1.5 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
call sage: g = class_graph(m.__class__)
    # not tested

sage: g.relabel(lambda x: x.replace("_",r"\_"))
    # not tested

sage: g.set_latex_options(format="dot2tex")
    # not tested

sage: view(g)
```
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
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
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
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'>
```
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 $C_S()$ is a *super category* of a category $D_S()$ if Sage considers any object of $D_S()$ to be an object of $C_S()$, up to an implicit application of a canonical functor from $D_S()$ to $C_S()$. This functor is normally an inclusion of categories or a forgetful functor. Reciprocally, $D_S()$ is said to be a *subcategory* of $C_S()$.

**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 github 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() #optional - 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: 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
```

**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._contains_  # 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(verbos=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:

```python
sage: S._test_associativity()
```

Here is how to access the code of this test:

```python
sage: S._test_associativity?? # not tested
```

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("+x.value +y.value")
```

And rerun the test:
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
sage: 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.
   → py(145)_test_associativity()
   ← tester assertTrue((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:

```python
sage: S = FiniteSemigroups().example(); S
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
sage: S? # not tested
```

Where do all the operations on $S$ and its elements come from?
_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'
```

_pow_int is a generic method for all finite semigroups:

```python
sage: 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
```

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'
```

_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
```

product is a mathematical method implemented by the parent:

```python
sage: S.product.__module__
'sage.categories.examples.finite_semigroups'
```

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

```python
sage: 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):

```python
sage: S.multiplication_table.__module__
'sage.categories.magmas'
```

Consider now the implementation of the semigroup:

```python
sage: S??
# not tested
```

2.1. Elements, parents, and categories in Sage: a primer 19
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 \texttt{product}. It also implements the optional method \texttt{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 \texttt{is\_idempotent} to always return \texttt{True}.

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

\begin{verbatim}
sage: FiniteSemigroups().required_methods()
{'element': {'optional': ['\_mul\_'], 'required': []},
 'parent': {'optional': ['semigroup\_generators'],
 'required': ['\_contains\_']}}
\end{verbatim}

\texttt{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:

\begin{verbatim}
sage: C = FiniteSemigroups().element_class
sage: C._mul_?
# not tested
\end{verbatim}

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

Here is the code for the finite semigroups category:

\begin{verbatim}
sage: FiniteSemigroups??
# not tested
\end{verbatim}

### 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):

\begin{verbatim}
sage: P = PermutationGroup([[1,2,3]]); P
Permutation Group with generators [[1,2,3]]
sage: P.category()
Category of finite enumerated permutation groups
\end{verbatim}

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

\begin{verbatim}
sage: P = PermutationGroup([[1,2,3]],
....: category=PermutationGroups().Finite().Commutative()); P
Permutation Group with generators [[1,2,3]]
sage: P.category()
Category of finite enumerated commutative permutation groups
\end{verbatim}

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:
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())
# optional - sage.combinat
sage: P.foo()
# optional - sage.combinat
foo
sage: p = P.an_element()
# optional - sage.combinat
sage: p.bar()
# optional - sage.combinat
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:

sage: P = PermutationGroup([[1,2,3]], category=Sets())
# todo: not implemented
# optional - sage.combinat

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, monoid algebras, …: See: *Sets.ParentMethods.algebra()*.
Let for example $A$ and $B$ be two parents, and let us construct the Cartesian product $A \times B \times B$:

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

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:

```
sage: C in Monoids()  # optional - sage.combinat sage.modules
True
```

the unit being the Cartesian product of the units of the operands:

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

```
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`. 

---

Chapter 2. The Sage Category Framework
Let us now look at the categories of `C`:

```python
sage: C.categories()  # optional - sage.combinat sage.modules
[Category of finite dimensional Cartesian products of algebras with basis over Rational Field, ...
Category of Cartesian products of algebras over Rational Field, ...
Category of Cartesian products of semigroups, Category of semigroups, ...
Category of Cartesian products of magmas, ..., Category of magmas, ...
Category of Cartesian products of additive magmas, ..., Category of additive magmas,
Category of Cartesian products of sets, Category of sets, ...]
```

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'})
```

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
sage: Magmas()
Category of magmas
sage: Magmas().Unital()
Category of unital magmas
sage: Magmas().Commutative().Unital()
Category of commutative unital magmas
sage: Magmas().Unital().Commutative()
Category of commutative unital magmas
sage: Magmas().Associative()
Category of semigroups
```
Axioms and categories with axioms

Here, `Associative`, `Unital`, `Commutative` are axioms. In general, any category $C_s$ in Sage can declare a new axiom $A$. Then, the category with axiom $C_s.A()$ models the subcategory of the objects of $C_s$ satisfying the axiom $A$. Similarly, for any subcategory $D_s$ of $C_s$, $D_s.A()$ models the subcategory of the objects of $D_s$ 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()
sage: Groups().Finite()
```

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?
```

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
sage: sorted(C.axioms())
```

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
sage: sorted(C.axioms())
```

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

```
sage: DivisionRings() & Sets().Finite()
```
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 “AdditiveAssociative” is about the properties of the addition $+$, whereas the axiom “Associative” is about the properties of the multiplication $\cdot$.

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, $\cdot$, $/$, 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 [github issue #15741](https://github.com/sagemath/sage/issues/15741) to unclutter the global name space from, for example, the many variants of the category of algebras like:

```sage
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:

```sage
sage: 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 $\cdot$:

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

...
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

or for more advanced categories:

```
sage: g = HopfAlgebras(QQ).WithBasis().Graded().Connected().category_graph()  # not tested
˓→optional - sage.graphs
sage: g.set_latex_options(format="dot2tex")  # not tested
˓→optional - sage.graphs
sage: view(g)  # not tested  # optional - 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:

```
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:

```
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 A an axiom or a covariant functorial construction, if Ds is a subcategory of Cs, then Ds.A() is a subcategory of Cs.A().
What happens if we consider reciprocally an object of $\text{Cs.A()}$ which is also in $\text{Ds}$? A finite dimensional module which is also an algebra is a finite dimensional algebra:

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

On the other hand, a graded module $O$ which is also an algebra is not necessarily a graded algebra! Indeed, the grading on $O$ may not be compatible with the product on $O$:

```python
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 $O$ seen as a module (and thus does not depend on the given category), whereas $\text{Graded}$ is a statement about the properties of $O$ 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 $\text{Cs}$ of $\text{Ds}$, $\text{Ds.A()}$ is the intersection of the categories $\text{Ds}$ and $\text{Cs.A()}$:

```python
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 $\text{Bs}$ 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, $\text{As}$ is the category of graded algebras, which are further endowed with some basis; that basis need not respect the grading. In fact $\text{As}$ is really a join category:

```python
sage: type(As)
<class 'sage.categories.category.JoinCategory_with_category'>
sage: As._repr_(as_join=True)
(continues on next page)
```
'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

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.super_categories to include C.

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

\[
\text{Cs().is_subcategory}(\text{Category.join(Cs().super_categories())})
\]

\[
\text{Cs().cmp_key} > \text{other.cmp_key for other in Cs().super_categories()}
\]

This is checked by _test_category().

In several cases, the category C is directly provided with a generic implementation of super_categories; a typical example is when C implements an axiom or a functorial construction; in such a case, C may implement C.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 Magsas.Commutative.Algebras just 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 ParentMethods (or ElementMethods). The methods of that class will automatically become methods of the objects (or the elements). For instance:

```sage
P.<x,y> = ZZ[]
P.prod([x,y,2])
2*x*y
```

```sage
P.prod.__module__
'sage.categories.monoids'
P.prod.__func__ is raw_getattr(Monoids().ParentMethods, "prod")
True
```

We recommend to study the code of one example:

```sage
C = CommutativeAdditiveMonoids()
```

```sage
C
# not tested
```
On the order of super categories

The generic method `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 `foo` in `ParentMethods`, then `P` will use `C_1`’s version of `foo` if and only if `C_1` appears in `C.all_super_categories()` before `C_2`.

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 `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 `foo` in this category. See the method `some_elements` in the code of the category `FiniteCoxeterGroups` for an example.

Since github issue #11943, `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 `C.all_super_categories()`, `C.parent_class.mro()` and `C.element_class.mro()` are guaranteed to be consistent.

Since github 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 `sage.misc.c3_controlled`. A visible effect is that the order in which categories are specified in `C.super_categories()`, or in a join category, no longer influences the result of `C.all_super_categories()`.

Subcategory hook (advanced optimization feature)

The default implementation of the method `C.is_subcategory(D)` is to look up whether `D` appears in `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 can not be subcategories of each other.

When such a short-path is known, one can implement a method `_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 `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 `issubclass(C.parent_class,D.parent_class)`, which is very often giving the right answer:

```
sage: Rings()._subcategory_hook_(Algebras(QQ))
True
sage: HopfAlgebras(QQ)._subcategory_hook_(Algebras(QQ))
False
sage: Algebras(QQ)._subcategory_hook_(HopfAlgebras(QQ))
True
```
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 `x.category()` returns the category to which `x` belongs. If `C` is a category and `x` is any object, `C(x)` tries to make an object in `C` from `x`. Checking if `x` belongs to `C` is done as usually by `x in C`.

See `Category` and `sage.categories.primer` for more details.

EXAMPLES:

We create a couple of categories:

```python
sage: Sets()
Category of sets
sage: GSets(AbelianGroup([2, 4, 9]))
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
```

Let's request the category of some objects:

```python
sage: V = VectorSpace(RationalField(), 3)
Category of finite dimensional vector spaces with basis
over (number fields and quotient fields and metric spaces)
```

Let's check some memberships:
For parametrized categories one can use the following shorthand:

```python
sage: V in VectorSpaces
# optional - sage.modules
True
sage: G in VectorSpaces
# optional - 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 = VectorSpace(RationalField(), 3)
# optional - sage.modules
sage: v = V.gen(1)
# optional - sage.modules
sage: v.category()
# optional - sage.modules
Category of elements of Vector space of dimension 3 over Rational Field
```

```python
class sage.categories.category.Category(s=\texttt{None})
    Bases: \texttt{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:

```python
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
    ....:         pass
    ....:     
    ....:     class ElementMethods: # holds the generic operations on elements
    ....:         def gcd(x,y):
    ....:             # Euclid algorithms
    ....:             pass
    ....:     
    ....:     class MorphismMethods: # holds the generic operations on morphisms
    ....:         # TODO: find a good example of an operation
    ....:         pass
    ....:
```

Note that the nested class 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 ParentMethods nested classes, and stored in the attributes parent_class. Then, a parent in a category $C$ receives the appropriate operations from all the super categories by usual class inheritance from C.parent_class.

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

**EXAMPLES:**

We define a hierarchy of four categories As(), Bs(), Cs(), 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

\begin{verbatim}
sage: from sage.categories.category import Category
c sage: from sage.misc.lazy_attribute import lazy_attribute
c sage: class As (Category):
....:     def super_categories(self):
....:         return []
....:     class ParentMethods:
....:         def fA(self):
....:             return "A"
....:             f = fA

c sage: class Bs (Category):
....:     def super_categories(self):
....:         return [As()]
....:     class ParentMethods:
....:         def fB(self):
....:             return "B"

c sage: class Cs (Category):
....:     def super_categories(self):
....:         return [As()]
....:     class ParentMethods:
....:         def fC(self):
....:             return "C"
....:             f = fC

c sage: class Ds (Category):
....:     def super_categories(self):
....:         return [Bs(), Cs()]
....:     class ParentMethods:
....:         def fD(self):
....:             return "D"
\end{verbatim}

Categories should always have unique representation; by \texttt{github 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:

\begin{verbatim}
sage: import gc
c sage: idAs = id(As())
c sage: _ = gc.collect()
c sage: n == id(As())
False
c sage: a = As()
c sage: id(a) == id(As())
\end{verbatim}

(continues on next page)
We construct a parent in the category \texttt{Ds()} (that, is an instance of \texttt{Ds().parent_class}), and check that it has access to all the methods provided by all the categories, with the appropriate inheritance order:

\begin{verbatim}
  sage: D = Ds().parent_class()
  sage: [ D.fA(), D.fB(), D.fc(), D.fD() ]
  ['A', 'B', 'C', 'D']
  sage: D.f()
  'C'
\end{verbatim}

\begin{verbatim}
  sage: C = Cs().parent_class()
  sage: [ C.fA(), C.fc() ]
  ['A', 'C']
  sage: C.f()
  'C'
\end{verbatim}

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

\begin{verbatim}
  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'>]
\end{verbatim}

\begin{verbatim}
  sage: Bs().parent_class
  <class '__main__.Bs.parent_class'>
  sage: Bs().parent_class.__bases__
  (<class '__main__.As.parent_class'>),
  sage: Bs().parent_class.mro()
  [<class '__main__.Bs.parent_class'>, <class '__main__.As.parent_class'>, <... __main__.As.parent_class'>, <... 'object'>]
\end{verbatim}

\begin{verbatim}
  sage: Cs().parent_class
  <class '__main__.Cs.parent_class'>
  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'>, <... __main__.As.parent_class'>, <... __main__.As.parent_class'>, <... 'object'>]
\end{verbatim}

\begin{verbatim}
  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'>, <... __main__.As.parent_class'>, <... '__main__.As.parent_class'>, <... 'object'>]
\end{verbatim}
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):

```python
sage: Algebras(QQ).parent_class is Algebras(ZZ).parent_class
False
sage: 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):

```python
sage: Algebras(GF(5)).parent_class is Algebras(GF(7)).parent_class
True
sage: F = FractionField(ZZ['t'])

sage: Coalgebras(F).parent_class is Coalgebras(FractionField(F['x'])).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()

sage: D.__class__
<class '__main__.myparent_with_category'>

sage: D.__class__.__bases__
(<class '__main__.myparent_with_category'>,
 <class '__main__.Ds.parent_class'>)

sage: D.__class__.mro()
[<class '__main__.myparent_with_category'>,
 <class '__main__.myparent'>,
 <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'>]

sage: D.fA()
'A'
sage: D.fB()
'B'
```

(continues on next page)
C
sage: D.fD()
'D'
sage: D.f()
'C'
sage: 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'>]

__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:

sage: 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:

sage: 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 github 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:

```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]
```

_all_super_categories_proper()

All the proper super categories of this category.

Since github 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:

```python
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]
```

_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:

\texttt{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]
```

\texttt{_make\_named\_class(name, method\_provider, cache=False, picklable=True)}

Construction of the parent/element/... class of \texttt{self}.

INPUT:

- \texttt{name} – a string; the name of the class as an attribute of \texttt{self}. E.g. “parent\_class”
- \texttt{method\_provider} – a string; the name of an attribute of \texttt{self} that provides methods for the new class (in addition to those coming from the super categories). E.g. “ParentMethods”
- \texttt{cache} – a boolean or \texttt{ignore\_reduction} (default: \texttt{False}) (passed down to \texttt{dynamic\_class}; for internal use only)
- \texttt{picklable} – a boolean (default: \texttt{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 \texttt{self}'s super\_categories, and methods taken from the class \texttt{getattr(self,method\_provider)}.

Note:

- In this default implementation, the reduction data of the named class makes it depend on \texttt{self}. Since the result is going to be stored in a lazy attribute of \texttt{self} anyway, we may as well disable the caching in \texttt{dynamic\_class} (hence the default value \texttt{cache=False}).
- \texttt{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 \texttt{_super\_categories\_for\_classes()} . This is to guarantee that Python will build consistent method resolution orders. For background, see \texttt{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.parent_class'>,
 <class 'sage.categories.semirings.Semirings.parent_class'>)
```

Note that, by default, the result is not cached:

```python
sage: 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'>
```

Reduction for pickling also assumes the existence of this lazy attribute:

```python
sage: PC._reduction
(<built-in function getattr>, (Category of rings, 'parent_class'))
sage: loads(dumps(PC)) is Rings().parent_class
True
```

```
_srepr_()

Return the print representation of this category.

EXAMPLES:

```python
sage: Sets() # indirect doctest
Category of sets
```

```
_srepr_object_names()

Return the name of the objects of this category.

EXAMPLES:

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

```
_test_category(**options)

Run generic tests on this category

See also:

TestSuite.

EXAMPLES:

```
Let us now write a couple broken categories:

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

```python
sage: class MyObjects(Category):
    ....: def super_categories(self):
    ....:     return tuple()
```

```python
sage: MyObjects()._test_category()
Traceback (most recent call last):
...:
AssertionError: Category of my objects.super_categories() should return a list
```

```python
sage: class MyObjects(Category):
    ....: def super_categories(self):
    ....:     return []
```

```python
sage: 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 \texttt{self} satisfying the given \texttt{axiom}.

**INPUT:**

- \texttt{axiom} -- a string, the name of an axiom

**EXAMPLES:**

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

```python
sage: type(Magmas().Finite().Commutative())
<class 'sage.categories.category.JoinCategory_with_category'>
sage: Magmas().Finite().Commutative().super_categories()
[Category of commutative magmas, Category of finite sets]
sage: C = Algebras(QQ).WithBasis().Commutative()
sage: C is Algebras(QQ).Commutative().WithBasis()
True
```

When \texttt{axiom} is not defined for \texttt{self}, \texttt{self} is returned:

```python
sage: 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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: Sets()._with_axiom_as_tuple('Finite')</td>
<td>(Category of finite sets,)</td>
</tr>
<tr>
<td>sage: Magmas()._with_axiom_as_tuple('Finite')</td>
<td>(Category of magmas, Category of finite sets)</td>
</tr>
<tr>
<td>sage: Rings().Division()._with_axiom_as_tuple('Finite')</td>
<td>(Category of division rings,</td>
</tr>
<tr>
<td></td>
<td>Category of finite monoids,</td>
</tr>
<tr>
<td></td>
<td>Category of commutative magmas,</td>
</tr>
<tr>
<td></td>
<td>Category of finite additive groups)</td>
</tr>
<tr>
<td>sage: HopfAlgebras(QQ)._with_axiom_as_tuple('FiniteDimensional')</td>
<td>(Category of hopf algebras over Rational Field,</td>
</tr>
<tr>
<td></td>
<td>Category of finite dimensional vector spaces over Rational Field)</td>
</tr>
</tbody>
</table>

__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.

EXAMPLES:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: Sets()._without_axioms()</td>
<td>Category of sets</td>
</tr>
<tr>
<td>sage: Semigroups()._without_axioms()</td>
<td>Category of magmas</td>
</tr>
<tr>
<td>sage: Algebras(QQ).Commutative().WithBasis()._without_axioms()</td>
<td>Category of magmatic algebras over Rational Field</td>
</tr>
<tr>
<td>sage: Algebras(QQ).Commutative().WithBasis()._without_axioms(named=True)</td>
<td>Category of algebras over Rational Field</td>
</tr>
</tbody>
</table>

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:**

```python
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(), ...:
                                             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:**

```python
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(...) be a category. Since [github issue #12895](https://github.com/sagemath/sage/issues/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(...) actually calls properly Cs(...) and gives back C.

**See also:**

subcategory_class()

**EXAMPLES:**

```sage
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
```

**__init__(s=None)**

Initialize this category.

**EXAMPLES:**

```sage
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'>
```

**Note:** Specifying the name of this category by passing a string is deprecated. If the default name (built from the name of the class) is not adequate, please use __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.

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
```

The category of realizations of a given algebra:

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

Todo: Add an optional argument to allow for:

```
sage: Realizations(A, category=Blahs()) # todo: not implemented
```

**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`
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:

```
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 SetsFacade) admitting multiple concrete realizations where its elements are represented. Consider for example an algebra $A$ which admits several natural bases:

```
sage: A = Sets().WithRealizations().example(); A
˓→ # optional - sage.combinat sage.modules
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 $B$:

```
sage: A.F()
˓→ # optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
```

```
sage: A.Out()
˓→ # optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
```

```
sage: A.In()
˓→ # optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the In basis
```

```
sage: A.an_element()
˓→ # optional - sage.combinat sage.modules
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 $P_{B'}$:
allowing for mixed arithmetic:

\[
(1 + \text{Out.from_set}(1)) \ast \text{In.from_set}(2,3)
\]

In our example, there are three realizations:

\[
\text{A.realizations()}
\]

Instead of manually defining the shorthands \( F, \text{In}, \text{Out} \), as above one can just do:

\[
\text{A.inject_shorthands()}
\]

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

```python
sage: P = A['x']; P
# optional - sage.combinat sage.modules
Univariate Polynomial Ring in x over
The subset algebra of {1, 2, 3} over Rational Field
sage: x = P.gen()
# optional - sage.combinat sage.modules
```

However we can create a polynomial with mixed coefficients, and compute with it:

```python
sage: p = P([1, In[{1}], Out[{2}]]); p
# optional - sage.combinat sage.modules
Out[{2}]*x^2 + In[{1}]*x + F[{}
```

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)
# optional - sage.combinat sage.modules
(-4*In[{}] + 2*In[{1}] + 4*In[{2}] + 2*In[{3}] +
 - 4*In[{1, 2}] - 2*In[{1, 3}] - 4*In[{2, 3}] + 2*In[{1, 2, 3}])*x^3
 + (F[{1}] + 3*F[{1}]) + 2*F[{2}] - 2*F[{1, 2}] - 2*F[{2, 3}] + 2*F[{1, 2, 3}])*x^2
 + (2*F[{1}] + 2*F[{1}])*x
 + F[{}
```

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

```python
sage: In[{1}] * x
#
```
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()
˓→# optional - sage.combinat 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()
˓→# optional - sage.combinat sage.modules
True
```

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

```python
sage: A in Algebras(QQ).WithRealizations()
˓→# optional - sage.combinat 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 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()
˓→# optional - sage.combinat sage.modules
True
```

and in the category of algebras (regressiveness):

```python
sage: A in Algebras(QQ)
˓→# optional - sage.combinat 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 regressive construction. That is self. `WithRealizations()` is a subcategory of self by default:

2.2. Categories
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, 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.

additional_structure()

Return whether self defines additional structure.

OUTPUT:

• self if self defines additional structure and None otherwise. This default implementation returns self.

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()  # `+`
Category 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()
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
```

- an enumerated set morphism preserve the distinguished enumeration?

```
sage: EnumeratedSets().additional_structure()
```
• a module with basis morphism preserve the distinguished basis?

\[\textbf{sage}: \text{Modules(QQ).WithBasis().additional_structure()}\]

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 \texttt{structure()}), and to test whether a category is a full subcategory of some other category (see \texttt{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:

\[\textbf{sage}: \text{CoxeterGroups().is_full_subcategory(Groups())} \]

False

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

\textbf{all_super_categories}(\texttt{proper=False})

Returns the list of all super categories of this category.

INPUT:

• \texttt{proper} – a boolean (default: \texttt{False}); whether to exclude this category.

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

\textbf{Note:} Whenever speed matters, the developers are advised to use instead the lazy attributes \texttt{_all_super_categories()}, \texttt{_all_super_categories_proper()} or \texttt{_set_of_super_categories()}, as appropriate. Simply because lazy attributes are much faster than any method.

EXAMPLES:

\[\textbf{sage}: C = \text{Rings()}; C\]
Category of rings
\[\textbf{sage}: C.all_super_categories()\]
[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]  
\[\textbf{sage}: C.all_super_categories(proper = \textbf{True})\]
[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]  
\[\textbf{sage}: \text{Sets().all_super_categories()}\]
[Category of sets, Category of sets with partial maps, Category of objects]
\[\textbf{sage}: \text{Sets().all_super_categories(proper=True)}\]
[Category of sets with partial maps, Category of objects]  

(continues on next page)
### classmethod `an_instance()`

Return an instance of this class.

**EXAMPLES:**

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

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

```
sage: Algebras.an_instance()
Category of algebras over Rational Field
sage: Bimodules.an_instance()
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:**

```
sage: Monoids().axioms()
 frozenset({'Associative', 'Unital'})
sage: (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
```

### `category_graph()`

Returns the graph of all super categories of this category

**EXAMPLES:**

```
```
sage: C = Algebras(QQ)
sage: G = C.category_graph()  # optional - sage.graphs
sage: G.is_directed_acyclic()  # optional - 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()  # optional - 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 github 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  # optional - sage.rings.finite_rings
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]
```

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

Groups().full_super_categories()
# todo: not implemented

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:

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 \texttt{self} is a subcategory of \texttt{other} and whether they have the same structure, as determined by \texttt{structure()} from the result of \texttt{additional_structure()} on the super categories.

\textbf{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 \texttt{full_super_categories()} for a discussion.

\textbf{See also:}

- \texttt{is_subcategory()}
- \texttt{full_super_categories()}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Magmas().Associative().is_full_subcategory(Magmas())
True
sage: Magmas().Unital().is_full_subcategory(Magmas())
False
sage: Rings().is_full_subcategory(Magmas().Unital() & AdditiveMagmas().AdditiveUnital())
True
\end{verbatim}

Here are two typical examples of false negatives:

\begin{verbatim}
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
\end{verbatim}

\textbf{Todo:} The latter is a consequence of \texttt{EuclideanDomains} currently being a structure category. Is this what we want?

\begin{verbatim}
sage: EuclideanDomains().is_full_subcategory(Rings())
False
\end{verbatim}

\texttt{is_subcategory(c)}

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

\textbf{EXAMPLES:}

\begin{verbatim}
sage: AbGrps = CommutativeAdditiveGroups()
sage: Rings().is_subcategory(AbGrps)
True
sage: AbGrps.is_subcategory(Rings())
False
\end{verbatim}
The `is_subcategory` function takes into account the base.

```python
sage: M3 = VectorSpaces(FiniteField(3))  # optional - sage.rings.finite_rings sage.modules
sage: M9 = VectorSpaces(FiniteField(9, 'a'))  # optional - sage.rings.finite_rings sage.modules
sage: M3.is_subcategory(M9)  # optional - sage.rings.finite_rings sage.modules
False
```

Join categories are properly handled:

```python
sage: CatJ = Category.join((CommutativeAdditiveGroups(), Semigroups()))
sage: Rings().is_subcategory(CatJ)
True

sage: V3 = VectorSpaces(FiniteField(3))  # optional - sage.rings.finite_rings
sage: POSet = PartiallyOrderedSets()
sage: PoV3 = Category.join((V3, POSet))  # optional - sage.rings.finite_rings
sage: A3 = AlgebrasWithBasis(FiniteField(3))  # optional - sage.rings.finite_rings
sage: PoA3 = Category.join((A3, POSet))  # optional - sage.rings.finite_rings
sage: PoA3.is_subcategory(PoV3)  # optional - sage.rings.finite_rings
True
sage: PoV3.is_subcategory(PoV3)  # optional - sage.rings.finite_rings
True
sage: PoV3.is_subcategory(PoA3)  # optional - sage.rings.finite_rings
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:**

```python
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]
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, ...]
```

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:

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

The join of a single category is the category itself:

```
sage: Category.join([Monoids()])
Category of monoids
```

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

```
sage: Category.join((Sets(), Rings(), Monoids()))
Category of rings
```

In particular, the unit is the top category `Objects`:

```
sage: Groups() & Objects()
Category of groups
```

If the optional parameter `as_list` is `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)
```

(continues on next page)
[]
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

Note that several of the above examples are actually join categories; they are just nicely displayed:

sage: 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
sage: UniqueFactorizationDomains() & Algebras(QQ)
Join of Category of unique factorization domains
and Category of commutative algebras over Rational Field

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:

sage: Category.meet([[Algebras(ZZ), Algebras(QQ), Groups()]])
Category of monoids

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

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

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()`
• `Category` for details

EXAMPLES:

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

```
sage: Monoids().or_subcategory(Groups())
Category of groups
sage: Monoids().or_subcategory(EnumeratedSets())
Traceback (most recent call last):
  ... ValueError: Subcategory of `Category of monoids` required;
got `Category of enumerated sets`
```

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

```
sage: 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`:

```
sage: 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:

```
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.DynamicMetaclas'>
```

By [github issue #11935](#), some categories 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  # optional - sage.rings.finite_rings
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
```

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
sage: Algebras(QQ).required_methods()
{'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
sage: Objects().structure()
frozenset()

sage: def structure(C):
....:     return Category._sort(C.structure())
(continues on next page)```
In the following example, we only list the smallest structure categories to get a more readable output:

```python
sage: structure(Magmas())
(Category of magmas,)
sage: structure(Rings())
(Category of unital magmas, Category of additive unital additive magmas)
sage: structure(Fields())
(Category of euclidean domains,)
sage: structure(Algebras(QQ))
(Category of unital magmas, 
Category of right modules over Rational Field, 
Category of left modules over Rational Field)
sage: structure(HopfAlgebras(QQ).Graded().WithBasis().Connected())
(Category of hopf algebras over Rational Field, 
Category of graded modules over Rational Field)
```

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

See also:

- [github 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'>
```

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

Note: Since github 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(s=None)

Bases: Category

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

Many categories in Sage are parametrized, like C = 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 VectorSpaces(K) is a super category of C only in that case. In such cases, and as an optimization (see github 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))
# optional - sage.rings.finite_rings
sage: C2 = Algebras(GF(3))
# optional - sage.rings.finite_rings
sage: C3 = Algebras(ZZ)
sage: from sage.categories.category import CategoryWithParameters
sage: isinstance(C1, CategoryWithParameters)  # optional - sage.rings.finite_rings
True
sage: C1.parent_class is C2.parent_class  # optional - sage.rings.finite_rings
True
```
optional - sage.rings.finite_rings

True

sage: C1.parent_class is C3.parent_class
#<optional - sage.rings.finite_rings

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.rings.Rngs.parent_class'>,
<class 'sage.categories.semirings.Semirings.parent_class'>)

Note that, by default, the result is not cached:
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'>
```

Reduction for pickling also assumes the existence of this lazy attribute:

```python
sage: PC._reduction((<built-in function getattr>, (Category of rings, 'parent_class'))) sage: loads(dumps(PC)) is Rings().parent_class True
```

```python
class sage.categories.category.JoinCategory(super_categories, **kwds)
Bases: CategoryWithParameters

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

EXAMPLES:
```
```python
dsage: 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 github 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: G = SymmetricGroup(10) # optional - sage.groups sage.rings.finite_rings sage: A3 = G.algebra(GF(3)) # optional - sage.groups sage.rings.finite_rings sage: A5 = G.algebra(GF(5)) # optional - sage.groups sage.rings.finite_rings sage: type(A3.category()) # optional - sage.groups sage.rings.finite_rings <class 'sage.categories.category.JoinCategory_with_category'> sage: type(A3) is type(A5) # optional - sage.groups sage.rings.finite_rings True
```

```python
Category._repr_object_names()

Return the name of the objects of this category.

EXAMPLES:
```
Category Framework, Release 10.1

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

**Category.** **_repr_**()

Return the print representation of this category.

**EXAMPLES:**

```python
sage: 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:**

```python
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
```

**additional_structure()**

Return None.

Indeed, a join category defines no additional structure.

**See also:**

`Category.additional_structure()`

**EXAMPLES:**

```python
sage: Modules(ZZ).additional_structure()
```

**is_subcategory**(C)

Check whether this join category is subcategory of another category C.

**EXAMPLES:**

```python
sage: Category.join([Rings(),Modules(QQ)]).is_subcategory(Category.join([Rings(),...
```
super_categories()

Returns the immediate super categories, as per `Category.super_categories()`.

EXAMPLES:

```python
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 github issue #15801).

EXAMPLES:

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

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)
# optional - sage.groups
[Category of G-sets for Symmetric group of order 8! as a permutation group, Category of Hecke modules over Rational Field, Category of Lie algebras over Rational Field, Category of additive magmas, ..., Category of fields, ...,]
```
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 $\mathcal{C}()$ that satisfy some predefined axiom $\mathcal{A}$.

The first step is to open the hood and check whether there already exists a class implementing the category $\mathcal{C}() \cdot \mathcal{A}()$. For example, taking $\mathcal{C}=\text{Semigroups}$ and the Finite axiom, there already exists a class for the category of finite semigroups:

```
sage: Semigroups().Finite()
Category of finite semigroups
sage: type(Semigroups().Finite())
<class 'sage.categories.finite_semigroups.FiniteSemigroups_with_category'>
```

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:

```
sage: Semigroups().Infinite()
Category of infinite semigroups
sage: type(Semigroups().Infinite())
<class 'sage.categories.category.JoinCategory_with_category'>
```

This category is indeed just constructed as the intersection of the categories of semigroups and of infinite sets respectively:
In this case, one needs to create a new class to implement the axiom \texttt{Infinite} for this category. This boils down to adding a nested class \texttt{Semigroups.Infinite} inheriting from \texttt{CategoryWithAxiom}.

In the following example, we implement a category \texttt{Cs}, with a subcategory for the objects satisfying the \texttt{Finite} axiom defined in the super category \texttt{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 C's")
```

Now a parent declared in the category \texttt{Cs().Finite()} inherits from all the methods of finite sets and of finite \texttt{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 C's
```

\textbf{Note:}

- This follows the same idiom as for \textit{Covariant Functorial Constructions}.
- From an object oriented point of view, any subcategory \texttt{Cs()} of \texttt{Sets} inherits a \texttt{Finite} method. Usually \texttt{Cs} could complement this method by overriding it with a method \texttt{Cs.Finite} which would make a super call to \texttt{Sets.Finite} and then do extra stuff.

  In the above example, \texttt{Cs} also wants to complement \texttt{Sets.Finite}, though not by doing more stuff, but by providing it with an additional mixin class containing the code for finite \texttt{Cs}. To keep the analogy, this mixin class is to be put in \texttt{Cs.Finite}.

- By defining the axiom \texttt{Finite}, \texttt{Sets} fixes the semantic of \texttt{Cs.Finite()} for all its subcategories \texttt{Cs}: namely "the category of \texttt{Cs} which are finite as sets". Hence, for example, \texttt{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 \texttt{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'
```

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):
    ....:     class ParentMethods:
    ....:         def foo(self):
    ....:             print("I am a method on finite C's")
sage: Cs.Finite = FiniteCs
sage: Cs().Finite()
Category of finite cs
```

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 github 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.

### 2.3. Axioms
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:

```
sage: Semigroups()
Category of semigroups
sage: Semigroups() is Magmas().Associative()
True
```

```
sage: FiniteGroups()
Category of finite groups
sage: FiniteGroups() is Groups().Finite()
True
```

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')
```

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")
```

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:
Hence, for whatever this notation is worth, one can currently do:

```
sage: Algebras.WithBasis(QQ)
Category of algebras with basis over Rational Field
```

We don’t recommend using syntax like `Algebras.WithBasis(QQ)`, as it may eventually be deprecated.

As a second step, Sage tries some obvious heuristics to deduce the link from the name of the category with axiom (see `base_category_class_and_axiom()` for the details). This typically covers the following examples:

```
sage: FiniteCoxeterGroups()
Category of finite coxeter groups
sage: FiniteCoxeterGroups() is CoxeterGroups().Finite()
True
sage: FiniteCoxeterGroups._base_category_class_and_axiom_origin
'deduced by base_category_class_and_axiom'
sage: FiniteDimensionalAlgebrasWithBasis(QQ)
Category of finite dimensional algebras with basis over Rational Field
sage: FiniteDimensionalAlgebrasWithBasis(QQ) is Algebras(QQ).FiniteDimensional().WithBasis()
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'
```

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 github 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
sage: class Cs(Category):
    ....:     def super_categories(self):
    ....:         return [Sets()]
    ....:     class SubcategoryMethods:
    ....:         def Green(self):
    ....:             '<documentation of the axiom Green'>
    ....:             return self._with_axiom("Green")
    ....:     class Green(CategoryWithAxiom):
    ....:         class ParentMethods:
    ....:             def foo(self):
    ....:                 print("I am a method on green C's")
```

With the current implementation, the name of the axiom must also be added to a global container:

```
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: all_axioms += ("Green",)
```

We can now use the axiom as usual:

```
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):

```
sage: Cs().Green
<bound method Cs.SubcategoryMethods.Green of Category of cs>
```

Thanks to this feature (implemented in CategoryWithAxiom.__classget__()), the user is systematically referred to the documentation of this method when doing introspection on Ds().Green:

```
sage: C = Cs()
sage: C.Green # not tested
's<documentation of the axiom Green>
```

It is therefore the natural spot for the documentation of the axiom.

**Note:** The presence of the nested class Green in Cs is currently mandatory even if it is empty.

**Todo:** Specify whether or not one should systematically use @cached_method in the definition of the axiom. And
make sure all the definition of axioms in Sage are consistent in this respect!

**Todo:** We could possibly define an @axiom decorator? This could hide two little implementation details: whether or not to make the method a cached method, and the call to _with_axiom(...) under the hood. It could do possibly do some more magic. The gain is not obvious though.

**Note:** 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 `CategoryWithAxiom._repr_object_names_static()`).

During a Sage session, new axioms should only be added at the end of all_axioms, as above, so as to not break the cache of 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 all_axioms in `sage.categories.category_with_axiom`.

**Design note**

Let us state again that, unlike what the existence of all_axioms might suggest, the definition of an axiom is local to a category and its subcategories. In particular, two independent categories Cs() and 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 Cs and 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: 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: from sage.categories.magas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: Magmas() & AdditiveMagmas() is MagmasAndAdditiveMagmas()
False
```

(continues on next page)
sage: Magmas() & AdditiveMagmas()  # todo: not implemented
Category of magmas and additive magmas

Still, there is a workaround to get the natural notations:

sage: (Magmas() & AdditiveMagmas()).Distributive()
Category of distributive magmas and additive magmas
sage: (Monoids() & CommutativeAdditiveGroups()).Distributive()
Category of rings

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

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 an 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
Magmas().Associative()
Category of semigroups
Magmas().Associative().Unital()
Category of monoids
Magmas().Unital().Associative()
Category of monoids
```

At the level of the classes implementing these categories, the following comes as a general naturalization of the previous section:

```sage
Magmas.Unital
<class 'sage.categories.magmas.Magmas.Unital'>
Magmas.Associative
<class 'sage.categories.semigroups.Semigroups'>
Magmas.Associative.Unital
<class 'sage.categories.monoids.Monoids'>
```

However, the following may look suspicious at first:

```sage
Magmas.Unital.Associative
Traceback (most recent call last):
...
AttributeError: type object 'Magmas.Unital' has no attribute 'Associative'
```

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 $\mathcal{C}_S$ 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 $\mathcal{C}_S$, we want to provide the system with a collection $(D_S)_{S \in \mathcal{S}}$ where each $S$ is a subset of the axioms and the corresponding $D_S$ is a class for the subcategory of the objects of $\mathcal{C}_S$ satisfying the axioms in $S$. For example, if $\mathcal{C}_S$ is the category of magmas, the pairs $(S, D_S)$ would include:

```python
dict({Associative} : Semigroups
     {Associative, Unital} : Monoids
     {Associative, Unital, Inverse} : Groups
     {Associative, Commutative} : Commutative Semigroups
     {Unital, Inverse} : Loops
```

Then, given a subset $T$ of axioms, we want the system to be able to select automatically the relevant classes $(D_S)_{S \in \mathcal{S}, S \subseteq T}$, and build from them a category for the objects of $\mathcal{C}_S$ 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 $D_T$:

2.3. Axioms

77
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, \(D_{S'}\) and \(A\) are respectively the base category class and axiom of the category with axiom \(D_S\) that we have met in the first section.

At this point, we urge the reader to explore the code of \texttt{Magmas} and \texttt{DistributiveMagmasAndAdditiveMagmas} 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 \textit{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 `AdditiveMagmas` with all (or at least sufficiently many) of the following axioms:

```sage
sage: 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:

```sage
sage: Magmas().Commutative().Associative()
Category of commutative semigroups
```

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 github 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().CartesianProducts().extra_super_categories()
[Category of semigroups]
```

Similarly, the fact that the algebra of a commutative magma is commutative is implemented in `Magmas.Commutative.Algebras.extra_super_categories()`:

```python
sage: Magmas().Commutative().Algebras(QQ).extra_super_categories()
[Category of commutative magmas]
```

**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()`.
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`.

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`:

```python
sage: from sage.categories.category_with_axiom import CategoryWithAxiom

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
```

In general, if several categories `C1s(), C2s(), ...` are mapped to the same category when applying some axiom `A` (that is `C1s().A() == C2s().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 \([C2s()]\) and not \([C2s() . A()]\) 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 github 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 \(\text{axiom()}\) 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
```

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()]
    ....:
    ....:     class SubcategoryMethods:
    ....:         pass
    ....:     class B(CategoryWithAxiom):
    ....:         pass
```

(continues on next page)
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:

```
sage: sorted(A1s().B().C().axioms())
['B', 'C', 'E', 'F']
```

Note also that this is a join category:

```
sage: type(A1s().B().C())
<class 'sage.categories.category.JoinCategory_with_category'>
sage: A1s().B().C().super_categories()
[Category of e a1s, 
 Category of f as, 
 Category of b a1s, 
 Category of c as]
```

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
sage: C is A1s().D().E().F()
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()]
    ....:
    ....:     class B(CategoryWithAxiom):
    ....:         class C(CategoryWithAxiom):
```
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()
```

# commutativity

```python
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())
```

Finally, we “accidentally” implement the category of b c a1s, both in A3s.B.C and A3s.E.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,

```
sage: 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 \texttt{JoinCategory}. The comparison of two constructible categories with the \texttt{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
```

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 (\texttt{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*; blame goes 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.

#### Axioms

We say that an axiom \(A\) is *defined* by a category \(Cs()\) if \(Cs()\) defines an appropriate method \texttt{Cs.SubcategoryMethods.A} with the semantic of the axiom specified in the documentation; for any subcategory \(Ds()\), \texttt{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 \texttt{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 \( \text{Cs()} \mid \rightarrow \text{Cs()} \cdot A() \) 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 \( \text{Finite} \):

```
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 \texttt{super_categories}, \texttt{extra_super_categories}, and friends should always return strict supercategories.

For example, to specify that a finite division ring is a finite field, \( \text{DivisionRings().Finite()} \) should not return \( \text{Fields().Finite()} \) ! 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 \texttt{CategoryWithAxiom} should be an implemented category (i.e. not a \texttt{JoinCategory}). This is checked by \texttt{CategoryWithAxiom._test_category_with_axiom()}.
- Arborescent structure: Let \( \text{Cs()} \) be a category, and \( S \) be some set of axioms defined in some super categories of \( \text{Cs()} \) but not satisfied by \( \text{Cs()} \). Suppose we want to provide a category with axiom for the elements of \( \text{Cs()} \) 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 \( \text{Cs}.A_1.A_2.\ldots.A_k \) is an implemented category. Furthermore, every intermediate step \( \text{Cs}.A_1.A_2.\ldots.A_i \) with \( i \leq k \) should be a category with axiom having \( A_i \) as axiom and \( \text{Cs}.A_1.A_2.\ldots.A_{i-1} \) 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 \( \text{Cs()} \) 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 \( \text{Cs} \) can never have a nested class \( \text{Cs}.A.A.A \). Similarly, applying the specification recursively, a category satisfying \( A \) cannot have a nested class \( \text{Cs}.A_1.A_2.A_3.A \) where \( A_1, A_2, A_3 \) 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 \texttt{CategoryWithParameters} should either be a \texttt{CategoryWithParameters} or a \texttt{Category_singleton}.

2.3. Axioms
• A :CategoryWithAxiom having a :Category_singleton as base category should be a :CategoryWithAxiom_singleton. This is handled automatically by :CategoryWithAxiom.__init__() and checked in :CategoryWithAxiom._test_category_with_axiom().

• A :CategoryWithAxiom having a :Category_over_base_ring as base category should be a :Category_over_base_ring. This currently has to be handled by hand, using :CategoryWithAxiom_over_base_ring. This is checked in :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 :Category_singleton as base category should be a :CategoryWithAxiom_singleton.

  Nothing difficult to implement, but this will need to rework the current “no subclass of a concrete class” assertion test of :Category_singleton.__classcall__().

• Similarly, a covariant functorial construction category having a :Category_over_base_ring as base category should be a :Category_over_base_ring.

The following specification might be desirable, or not:

• A join category involving a :Category_over_base_ring should be a :Category_over_base_ring. In the mean time, a :base_ring method is automatically provided for most of those by :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 :Sets.Finite), or in a separate file (typically in a class :FiniteSets in a lazily imported module :sage.categories.finite_sets).

• Single point of truth: a theorem, like Wedderburn’s, should be implemented in a single spot.

• Single entry point: for example, from the entry :Rings, one can explore a whole range of related categories just by applying axioms and constructions:

  sage: Rings().Commutative().Finite().NoZeroDivisors()
  Category of finite integral domains
  sage: 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 name space.

Note that this is not about precluding the existence of multiple natural ways to construct the same category:
• 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:

```
sage: FiniteSets(Sets())
Category of finite sets
sage: 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:

```
sage: Sets()._with_axiom('Finite')
Category of finite sets
sage: FiniteSets()
Category of finite sets
sage: Sets().Finite()
Category of finite sets
```

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.CartesianProducts() & B.CartesianProducts() = (A&B).CartesianProducts()
```

• Once full subcategories are implemented (see github 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 $J \subset C_i$ for all $i$, and $J \subset C.A()$ 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 $\ldots$; 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 $C_2.A()$, $\ldots$ 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 $Cs$ be a category and $A$ an axiom defined for this category. To compute $Cs().A()$, there are two cases.

Adding an axiom $A$ to a category $Cs()$ not implementing it

In this case, $Cs().A()$ returns the join of:

- $Cs()$
- $Bs().A()$ for every direct super category $Bs()$ of $Cs()$
- the categories appearing in $Cs().A_extra_super_categories()$

This is a highly recursive process. In fact, as such, it would run right away into an infinite loop! Indeed, the join of $Cs()$ with $Bs().A()$ would trigger the construction of $Cs().A()$ 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 \( \text{Cs()} \) implementing it

In this case \( \text{Cs()} . \text{A()} \) simply constructs an instance \( D \) of \( \text{Cs.A} \) which models the desired category. The non trivial part is the construction of the super categories of \( D \). Very much like above, this includes:

- \( \text{Cs()} \)
- \( \text{Bs()} . \text{A()} \) for every super category \( \text{Bs()} \) of \( \text{Cs()} \)
- the categories appearing in \( D . \text{extra_super_categories()} \)

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 \( \text{Cs()} . \text{A()} \) ! To avoid this, \texttt{Category.join()} is given an optional argument to specify that the axiom \( A \) should not be applied to \( \text{Cs()} \).

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.

\[
\text{Theorem}
\]

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.

\[
\text{Remark}
\]

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 \texttt{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 \texttt{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.

\[
\text{Lemma}
\]

Let \( C \) be a category implementing an axiom \( A \). Recall that the construction of \( C . \text{A()} \) 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 \( C . \text{A()} \).
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\_)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\_axiom\_as\_tuple('B')} or $D$.\text{B\_extra\_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.

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(s=None):
    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()
```

(continues on next page)
[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]

super_categories()

class sage.categories.category_with_axiom.Blahs(s=None)
    Bases: 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 an 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

2.3. Axioms
extra_super_categories()

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

This is relevant when e.g. Foos(**) does some non trivial transformations:

sage: Modules(QQ) is VectorSpaces(QQ)
True
sage: type(Modules(QQ))
<class 'sage.categories.vector_spaces.VectorSpaces_with_category'>

sage: ModulesWithBasis(QQ) is VectorSpaces(QQ).WithBasis()
True

(continues on next page)
\begin{verbatim}

sage: type(ModulesWithBasis(QQ))
<class 'sage.categories.vector_spaces.VectorSpaces.WithBasis_with_category'>

static __classget__(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 Cs implements an axiom A with a nested class Cs.A, the expression Cs().A evaluates to the method defining the axiom A and not the nested class. See those design notes for the rationale behind this behavior.

EXAMPLES:

\begin{verbatim}
sage: Sets().Infinite()
Category of infinite sets
sage: Sets().Infinite
Cached version of <function ...Infinite at ...>
sage: Sets().Infinite.f == Sets.SubcategoryMethods.Infinite.f
True
\end{verbatim}

We check that this also works when the class is implemented in a separate file, and lazy imported:

\begin{verbatim}
sage: Sets().Finite
Cached version of <function ...Finite at ...>
\end{verbatim}

There is no binding behavior when accessing Finite or Infinite from the class of the category instead of the category itself:

\begin{verbatim}
sage: Sets.Finite
<class 'sage.categories.finite_sets.FiniteSets'>
sage: Sets.Infinite
<class 'sage.categories.sets_cat.Sets.Infinite'>
\end{verbatim}

This method also initializes the attribute _base_category_class_and_axiom if not already set:

\begin{verbatim}
sage: Sets.Infinite._base_category_class_and_axiom
(<class 'sage.categories.sets_cat.Sets'>, 'Infinite')
sage: Sets.Infinite._base_category_class_and_axiom_origin
'set by __classget__'
\end{verbatim}

__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:

\begin{verbatim}
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()
\end{verbatim}
\end{verbatim}

2.3. Axioms
Note: This is implemented by taking \_repr\_object\_names from self._without\_axioms\(\text{named}=\text{True}\), and adding the names of the relevant axioms in appropriate order.

\texttt{static \_repr\_object\_names\_static}(\texttt{category, axioms})

\textbf{INPUT:}

- \texttt{base\_category} – a category
- \texttt{axioms} – a list or iterable of strings

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: from \texttt{sage.categories.category\_with\_axiom import CategoryWithAxiom}
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Semigroups(), ["Flying", "Blue \rightarrow"])
'flying blue semigroups'
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Algebras(QQ), ["Flying", "WithBasis", "Blue"])
'flying blue algebras with basis over Rational Field'
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Algebras(QQ), ["WithBasis"])
'algebras with basis over Rational Field'
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Sets().Finite().\rightarrow Subquotients(), ["Finite"])
'subquotients of finite sets'
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Monoids(), ["Unital"])
'monoids'
\texttt{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'
\end{verbatim}

If the axioms is a set or frozen set, then they are first sorted using \texttt{canonicalize\_axioms}:

\begin{verbatim}
\texttt{sage: CategoryWithAxiom\._repr\_object\_names\_static(Semigroups(), \texttt{set}(["Finite", "Commutative", "Facade"]))
'facade finite commutative semigroups'
\end{verbatim}

\textbf{See also:}

- \texttt{\_repr\_object\_names()}

\textbf{Note:} The logic here is shared between \texttt{\_repr\_object\_names()} and category.JoinCategory.\_repr\_object\_names()

- \texttt{\_test\_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 similarly for Category_over_base_ring.

EXAMPLES:

```
sage: Sets().Finite()._test_category_with_axiom()
sage: 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
```

If named is True, then _without_axioms stops at the first category that has an explicit name of its own:

```
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:

```
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:

```
sage: Sets().Finite().additional_structure()
sage: Monoids().additional_structure()
```

2.3. Axioms
axioms()

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

See also:

category.axioms()

EXAMPLES:

```
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(3)).axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive', 'Finite',
 'FiniteDimensional', 'Unital', 'WithBasis']
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
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
```

base_category()

Return the base category of self.

EXAMPLES:

```
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]
```

Variants:

```python
sage: Sets().Finite().super_categories()
[Category of sets]
sage: 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(s=None)

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

2.3. Axioms
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

super_categories()

class 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

super_categories()
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:

```python
sage: from sage.categories.category_with_axiom import axiom
sage: axiom("Finite")(Semigroups())
Category of finite semigroups
```

Upon assigning the result to a class this becomes a method:

```python
sage: class As:
    ....:     def _with_axiom(self, axiom): return self, axiom
    ....:     Finite = axiom("Finite")
sage: As().Finite()
(<__main__.As ... at ...>, 'Finite')
```

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:

```python
sage: from sage.categories.category_with_axiom import axiom_of_nested_class
sage: axiom_of_nested_class(TestObjects, TestObjects.FiniteDimensional)
'FiniteDimensional'
sage: axiom_of_nested_class(TestObjects.FiniteDimensional, TestObjects.FiniteDimensional.Finite)
'TestObjects.FiniteDimensional.Finite'
sage: axiom_of_nested_class(Sets, FiniteSets)
'Finite'
sage: 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.magmas.Magmas'>, 'Associative')
sage: 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,
    ...             CategoryWithAxiom
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')
```

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):
... AssertionException: 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)
  (<class 'sage.categories.semigroups.Semigroups'>, 'Facade')
```

**Note:** In the following example, we could possibly retrieve Sets from the class name. However this cannot be implemented robustly until github issue #9107 is fixed. Anyway this feature has not been needed so far:

```python
sage: Sets.Infinite
  <class 'sage.categories.sets_cat.Sets.Infinite'>
sage: base_category_class_and_axiom(Sets.Infinite)
Traceback (most recent call last):
... TypeError: Could not retrieve the base category class and axiom
for <class 'sage.categories.sets_cat.Sets.Infinite'>.
...```

```python
sage.categories.category_with_axiom.uncamelcase(s, separator='')
```

EXAMPLES:

```python
sage: sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras")
  'finite dimensional algebras'
sage: sage.categories.category_with_axiom.uncamelcase("JTrivialMonoids")
  'j trivial monoids'
```
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.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:**

```python
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
```

It would be a mistake to call it in opposite order:

```python
sage: 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:

```python
sage: 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))                                        # optional - sage.rings.finite_rings
Finite Field of size 3
```

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:

```
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 is_Functor
sage: is_Functor(F)
True
sage: I = IdentityFunctor(abgrps)
sage: I
The identity functor on Category of commutative additive groups
```
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:

```
sage: from sage.categories.functor import Functor
sage: F = Functor(Rings(), Fields())
sage: F
Functor from Category of rings to Category of fields
sage: F(ZZ)
Rational Field
sage: F(GF(2))
\texttt{#optional - sage.rings.finite_rings}
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.

```
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)
```

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
```

(continues on next page)
Defn: $x \rightarrow a + b$

```
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
```

**codomain()**

The codomain of self

**EXAMPLES:**

```
sage: F = ForgetfulFunctor(FiniteFields(), Fields())
sage: F.codomain()
Category of fields
```

**domain()**

The domain of self

**EXAMPLES:**

```
sage: F = ForgetfulFunctor(FiniteFields(), Fields())
sage: F.domain()
Category of finite enumerated fields
```

```python
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
```
```

```python
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

sage.categories.functor.is_Functor(x)
Test whether the argument is a functor

NOTE:
There is a deprecation warning when using it from top level. Therefore we import it in our doc test.

EXAMPLES:

```python
sage: from sage.categories.functor import is_Functor
sage: F1 = QQ.construction()[0]
sage: F1
FractionField
sage: is_Functor(F1)
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
```

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')
```
and play around with the examples in the documentation of $S$ and of \texttt{FiniteSemigroups}.

Rename the class to \texttt{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 \texttt{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
- Sebastien Besnier (2014-05-5): FormalCompositeMap contains a list of Map instead of only two Map. See github issue #16291.

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
```

(continues on next page)
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 a over Rational Field

then

**Ring morphism:**

**From:** Univariate Polynomial Ring in x over Rational Field

**To:** Univariate Polynomial Ring in x over Rational Field

**Defn:** a |--> 2*x

sage: (f*g)(2*a^2+5)
5*a^2

sage: (g*f)(2*x^2+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:**

```
sage: f = QQ.coerce_map_from(ZZ)
sage: g = MatrixSpace(QQ, 2, 2).coerce_map_from(QQ)
```

```
# optional - sage.modules
sage: list((g * f).domains())
[Integer Ring, Rational Field]
```

**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:**

```
sage: R.\langle x\rangle = QQ[]
sage: S.\langle a\rangle = 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: fg = f * g
sage: fg.first() == g
True
sage: fg == fg.then() * fg.first()
True
```

**isinjective()**

Tell whether self is injective.
It raises `NotImplementedError` if it can’t be determined.

**EXAMPLES:**

```python
sage: V1 = QQ^2
    # optional - sage.modules
sage: V2 = QQ^3
    # optional - sage.modules
sage: phi1 = (QQ^1).hom(Matrix([[1, 1]]), V1)
    # optional - sage.modules
sage: phi2 = V1.hom(Matrix([[1, 2, 3], [4, 5, 6]]), V2)
    # optional - sage.modules

If both constituents are injective, the composition is injective:

```python
sage: from sage.categories.map import FormalCompositeMap
sage: c1 = FormalCompositeMap(Hom(QQ^1, V2, phi1.category_for()),
                          # optional - sage.modules
                          phi1, phi2)
sage: c1.is_injective()  # optional - sage.modules
True
```

If it cannot be determined whether the composition is injective, an error is raised:

```python
sage: psi1 = V2.hom(Matrix([[1, 2], [3, 4], [5, 6]]), V1)
    # optional - sage.modules
sage: c2 = FormalCompositeMap(Hom(V1, V1, phi2.category_for()),
                          # optional - sage.modules
                          psi2, psi1)
sage: c2.is_injective()  # optional - sage.modules
Traceback (most recent call last):
... error: 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:

```python
sage: psi2 = V1.hom([[1], [1]], QQ^1)
    # optional - sage.modules
sage: c3 = FormalCompositeMap(Hom(V2, QQ^1, phi2.category_for()),
                          # optional - sage.modules
                          psi2, psi1)
sage: c3.is_injective()  # optional - sage.modules
False
```

**is_surjective()**

Tell whether `self` is surjective.

It raises `NotImplementedError` if it can’t be determined.

**EXAMPLES:**

```python
sage: from sage.categories.map import FormalCompositeMap
sage: V3 = QQ^3
    # optional - sage.modules
```

(continues on next page)
If both maps are surjective, the composition is surjective:

```python
sage: phi32 = V3.hom(Matrix([[1], [2], [3], [4], [5], [6]]), V2)  # optional - sage.modules
sage: phi21 = V2.hom(Matrix([[1], [1]]), V1)  # optional - sage.modules
sage: c_phi = FormalCompositeMap(Hom(V3, V1, phi32.category_for()), phi32, phi21)
```

```python
sage: c_phi.is_surjective()  # optional - sage.modules
True
```

If the second map is not surjective, the composition is not surjective:

```python
sage: FormalCompositeMap(Hom(V3, V1, phi32.category_for()), phi32, V2.hom(Matrix([[0], [0]]), V1)).is_surjective()  # optional - sage.modules
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()), V2.hom(Matrix([[0], [0]]), V1), V1.hom(Matrix([[1]]), V1)).is_surjective()  # optional - sage.modules
False
```

Otherwise, surjectivity of the composition cannot be determined:

```python
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()), V2.hom(Matrix([[1], [1]], [1, 1]]), V2), V2.hom(Matrix([[1], [1]], V1)).is_surjective()  # optional - sage.modules
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:**

```python
sage: P.<x> = QQ[]
sage: incl = P.coerce_map_from(ZZ)
```

(continues on next page)
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

the following example has been attached to \_integer\_() of sage.rings.polynomial.
polynomial_element.Polynomial before (see comment there):

sage: k = GF(47)
# optional - sage.rings.finite_rings
sage: R.<x> = PolynomialRing(k)
# optional - sage.rings.finite_rings
sage: R.coerce_map_from(ZZ).section()
# optional - sage.rings.finite_rings
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
    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: 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[]

(continues on next page)
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).then() == f
True

sage: f = QQ.coerce_map_from(ZZ)

sage: f = f.extend_domain(ZZ).extend_codomain(QQ)

sage: 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

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:

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^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-1)
-1^2 + 2^x*y + y^2 + 2^x + 2^y - 1

category_for()

Returns the category self is a morphism for.

Note: This is different from the category of maps to which this map belongs as an object.

EXAMPLES:
```python
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)
```

FIXME: find a better name for this method

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:

```python
sage: 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:

```python
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))  # optional - sage.rings.finite_rings
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: 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
```

**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:

```python
sage: Q = QuadraticField(-5)  # optional - sage.rings.number_field
sage: phi = CDF._internal_convert_map_from(Q)  # optional - sage.rings.number_field
sage: print(phi.parent())  # optional - 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:

```python
sage: import gc
sage: del Q  # optional - sage.rings.number_field
sage: _ = gc.collect()

sage: phi.parent()  # optional - sage.rings.number_field
Traceback (most recent call last):
... ValueError: This map is in an invalid state,
the domain has been garbage collected

You can still obtain copies of the maps used by the coercion system with strong references:

```python
sage: Q = QuadraticField(-5)  # optional - sage.rings.number_field
sage: phi = CDF.convert_map_from(Q)  # optional - sage.rings.number_field
sage: print(phi.parent())  # optional - 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
sage: import gc
sage: del Q  # optional - sage.rings.number_field
sage: _ = gc.collect()

sage: phi.parent()  # optional - 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
```

3.1. Base class for maps
INPUT:

- **self** – a Map in some \(\text{Hom}(X, Y, \text{category\_right})\)
- **left** – a Map in some \(\text{Hom}(Y, Z, \text{category\_left})\)

Returns the composition of **self** followed by **right** as a morphism in \(\text{Hom}(X, Z, \text{category})\) where \(\text{category}\) is the meet of \(\text{category\_left}\) and \(\text{category\_right}\).

Caveat: see the current restrictions on \textit{Category.meet()}.

**EXAMPLES:**

```
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_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
```

\textbf{pre\_compose(right)}

INPUT:

- **self** – a Map in some \(\text{Hom}(Y, Z, \text{category\_left})\)
- **left** – a Map in some \(\text{Hom}(X, Y, \text{category\_right})\)

Returns the composition of **right** followed by **self** as a morphism in \(\text{Hom}(X, Z, \text{category})\) where \(\text{category}\) is the meet of \(\text{category\_left}\) and \(\text{category\_right}\).

**EXAMPLES:**

```
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
```

(continues on next page)
To: Integer Ring
then
Generic morphism:
From: Integer Ring
To: Rational Field

```
sage: 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):
  ... 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)
True
```

**sage.categories.map.unpickle_map(_class, parent, _dict, _slots)**

Auxiliary function for unpickling a map.

**3.1. Base class for maps**
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 ref’ed anymore.

EXAMPLES:

In the following, the \texttt{Hom} object is indeed cached:

\begin{verbatim}
sage: K = GF(17)  # optional - sage.rings.finite_rings
sage: H = Hom(ZZ, K)  # optional - sage.rings.finite_rings
sage: H                  # optional - sage.rings.finite_rings
Set of Homomorphisms from Integer Ring to Finite Field of size 17
sage: H is Hom(ZZ, K)  # optional - sage.rings.finite_rings
True
\end{verbatim}

Nonetheless, garbage collection occurs when the original references are overwritten:

\begin{verbatim}
sage: for p in prime_range(200):  # optional - sage.rings.finite_rings
....:  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  # optional - sage.rings.finite_rings
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]  # optional - sage.rings.finite_rings
sage: len(L)  # optional - sage.rings.finite_rings
1
sage: L  # optional - sage.rings.finite_rings
[Finite Field of size 199]
\end{verbatim}

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

\texttt{sage.categories.homset.End(X, category=None)}

Create the set of endomorphisms of \texttt{X} in the category category.

\textbf{INPUT:}
• X – anything
• category – (optional) category in which to coerce X

OUTPUT:
A set of endomorphisms in category

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)          # optional - sage.modules
sage: End(V)                          # optional - 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: G = AlternatingGroup(3)         # optional - sage.groups
sage: S = End(G); S                   # optional - sage.groups
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: from sage.categories.homset import is_Endset
sage: is_Endset(S)                     # optional - sage.groups
True
sage: S.domain()                      # optional - sage.groups
Alternating group of order 3!/2 as a permutation group
```

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:

```
sage: G = GL(3, 3)                     # optional - sage.groups
sage: G.category()                    # optional - sage.groups
Category of finite groups
sage: H = Hom(G, G)                   # optional - sage.groups
sage: H.homset_category()             # optional - sage.groups
Category of finite groups
sage: H.category()                    # optional - sage.groups
Category of endsets of unital magmas
```

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)  # optional - sage.modules
sage: Hom(V, V)  # optional - 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: G = AlternatingGroup(3)  # optional - sage.groups
sage: Hom(G, G)  # optional - sage.groups
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: Hom(ZZ, QQ, Sets())  # optional - sage.groups
Set of Morphisms from Integer Ring to Rational Field in Category of sets
```

```
sage: Hom(FreeModule(ZZ, 1), FreeModule(QQ, 1))  # optional - sage.modules
Set of Morphisms from Ambient free module of rank 1 over the principal ideal domain Integer Ring to Vector space of dimension 1 over Rational Field in Category of commutative additive groups
```

```
sage: Hom(FreeModule(QQ, 1), FreeModule(ZZ, 1))  # optional - sage.modules
Set of Morphisms from Vector space of dimension 1 over Rational Field to Ambient free module of rank 1 over the principal ideal domain Integer Ring in Category of commutative additive groups
```

Here, we test against a memory leak that has been fixed at [github issue #11521](https://github.com/sagemath/sage/issues/11521) by using a weak cache:
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  
# optional - sage.rings.finite_rings  
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]  
# optional - sage.rings.finite_rings  
sage: len(L), L[0]  
# optional - sage.rings.finite_rings
(1, Finite Field of size 997)

To illustrate the choice of the category, we consider the following parents as running examples:

sage: X = ZZ; X
Integer Ring
sage: Y = SymmetricGroup(3); Y  
# optional - 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)  
# optional - 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())  
# optional - 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())  
# optional - 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'>
Do not call this hook directly to create homsets, as it does not handle unique representation:

```python
sage: Hom(QQ, QQ) == QQ._Hom_(QQ, category=QQ.category())
True
sage: Hom(QQ, QQ) is QQ._Hom_(QQ, category=QQ.category())
False
```

**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).

```python
class sage.categories.homset.Homset(X, Y, category=None, base=None, check=True):
    Bases: Set_generic
    The class for collections of morphisms in a category.

    EXAMPLES:
    ```
    sage: H = Hom(QQ^2, QQ^3)
    # optional - sage.modules
    sage: loads(H.dumps()) is H  # optional - sage.modules
    True
    ```

    Homsets of unique parents are unique as well:
    ```
    sage: H = End(AffineSpace(2, names='x,y'))  # optional - sage.modules
    sage: loads(dumps(AffineSpace(2, names='x,y'))) is AffineSpace(2, names='x,y')  # optional - sage.modules
    True
    sage: loads(dumps(H)) is H  # optional - sage.modules
    True
    ```

    Conversely, homsets of non-unique parents are non-unique:
    ```
    sage: P11 = ProductProjectiveSpaces(QQ, [1, 1])  # optional - sage.schemes
    sage: H = End(P11)  # optional - sage.schemes
    sage: loads(dumps(P11)) is ProductProjectiveSpaces(QQ, [1, 1])  # optional - sage.schemes
    False
    sage: loads(dumps(P11)) == ProductProjectiveSpaces(QQ, [1, 1])  # optional - sage.schemes
    True
    sage: loads(dumps(H)) is H  # optional - sage.schemes
    True
    ```
```
```python
False
sage: loads(dumps(H)) == H #␣
˓→optional - sage.schemes
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
```

**domain()**

Return the domain of this homset.

**EXAMPLES:**

```python
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
```

**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.

**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:**

```python
sage: H = Hom(ZZ, ZZ)
sage: H.element_class_set_morphism
<class 'sage.categories.morphism.SetMorphism'>
sage: H.homset_category() ˓→optional - sage.groups
˓→optional - sage.groups
```

3.2. Homsets 125
identity()

The identity map of this homset.

**Note:** Of course, this only exists for sets of endomorphisms.

**EXAMPLES:**

```python
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
```

natural_map()

Return the “natural map” of this homset.

**Note:** By default, a formal coercion morphism is returned.

**EXAMPLES:**

```python
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'])  # optional - sage.rings.finite_rings
sage: H.natural_map()  # optional - sage.rings.finite_rings
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
```

one()

The identity map of this homset.
Note: Of course, this only exists for sets of endomorphisms.

EXAMPLES:

```python
sage: K = GaussianIntegers()  #
˓→optional - sage.rings.number_field
sage: End(K).one()  #
˓→optional - sage.rings.number_field
Identity endomorphism of Gaussian Integers 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:

```python
sage: H = Hom(ZZ^2, ZZ^3); H  #
˓→optional - sage.modules
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 (euclidean
domains and infinite enumerated sets and metric spaces)
sage: type(H)  #
˓→optional - sage.modules
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: H.reversed()  #
˓→optional - sage.modules
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 (euclidean
domains and infinite enumerated sets and metric spaces)
sage: type(H.reversed())  #
˓→optional - sage.modules
<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
```

```python
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
```
sage.categories.homset.hom(X,Y,f)

Return $\text{Hom}(X,Y)(f)$, where $f$ is data that defines an element of $\text{Hom}(X,Y)$.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: phi = hom(R, QQ, [2])
sage: phi(x^2 + 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.parent())
False
sage: g = P.hom([2*t])
sage: is_Endset(g.parent())
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
sage: is_Homset(f.category())
False
sage: is_Homset(f.parent())
True
```

### 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.

```
sage: X.<a, b> = ZZ[]
sage: Y.<c> = ZZ[]
sage: X.hom([(c, c^2)])
Ring morphism:
  From: Multivariate Polynomial Ring in a, b over Integer Ring
  To:   Univariate Polynomial Ring in c over Integer Ring
```

(continues on next page)
Defn:  
\[ a \rightarrow c \]
\[ b \rightarrow c^2 \]

AUTHORS:

- William Stein (2005): initial version
- David Joyner (2005-12-17): added examples

```python
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.

is_injective()
    Return whether this morphism is injective.

is_surjective()
    Return whether this morphism is surjective.
```

3.3. Morphisms
sage: Hom(ZZ, ZZ).identity().is_surjective()
True

section()
Return a section of this morphism.
EXAMPLES:

sage: T = Hom(ZZ, ZZ).identity()
sage: T.section() is T
True

class sage.categories.morphism.Morphism
Bases: Map
category()
Return the category of the parent of this morphism.
EXAMPLES:

sage: R.<t> = ZZ[]
sage: f = R.hom([t^2])
sage: f.category()
Category of endsets of unital magmas and right modules over
(euclidean domains and infinite enumerated sets and metric spaces)
and left modules over (euclidean domains
and infinite enumerated sets and metric spaces)
sage: K = CyclotomicField(12)
# optional - sage.rings.number_field
sage: L = CyclotomicField(132)
# optional - sage.rings.number_field
sage: phi = L._internal_coerce_map_from(K)
# optional - sage.rings.number_field
sage: phi.category()
# optional - sage.rings.number_field
Category of homsets of number fields

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: K = CyclotomicField(12)
# optional - sage.rings.number_field
sage: L = CyclotomicField(132)
# optional - sage.rings.number_field
sage: phi = L._internal_coerce_map_from(K)
# optional - sage.rings.number_field
(continues on next page)
is_identity()

Return True if this morphism is the identity morphism.

Note: Implemented only when the domain has a method gens()

EXAMPLES:

```sage
definition
R.<t> = ZZ()

f = R.hom([t])
f.is_identity()
True

R.<t> = ZZ()

f = R.hom([t+1])
f.is_identity()
False
```

A morphism between two different spaces cannot be the identity:

```sage
R2.<t2> = QQ()

h = R.hom([t2])
h.is_identity()
False
```

pushforward(I)

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()
Y.<y> = ZZ()
x^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()
phi = Hom(X, Z)(z)
phi(x^2+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.
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
```

**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: S = SymmetricGroup(4)  # _
  # optional - sage.groups
sage: phi = Hom(S, ZZ)(lambda x: ZZ(x.sign()))  # _
  # optional - sage.groups
sage: x = S.an_element(); x  # _
  # optional - sage.groups
(2,3,4)
sage: phi(x)  # _
  # optional - sage.groups
1
sage: phi.register_as_conversion()  # _
  # optional - sage.groups
sage: ZZ(x)  # _
  # optional - sage.groups
1
```

**class** `sage.categories.morphism.SetMorphism`

**Bases:** `Morphism`

**INPUT:**

- `parent` — a Homset
- `function` — a Python function that takes elements of the domain as input and returns elements of the domain.

**EXAMPLES:**

```python
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
```

(continues on next page)
sage: f.codomain()
Integer Ring
sage: TestSuite(f).run()

sage.categories.morphism.is_Morphism(x)

## 3.4 Coercion via construction functors

class sage.categories.pushout.AlgebraicClosureFunctor

    Bases: ConstructionFunctor

    Algebraic Closure.

    EXAMPLES:

    sage: F = CDF.construction()[0]
    sage: F(QQ)  # optional - sage.rings.number_field
    Algebraic Field
    sage: F(RR)  # optional - sage.rings.number_field
    Complex Field with 53 bits of precision
    sage: F(F(QQ))  # optional - sage.rings.number_field
    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, precs=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)  # optional - sage.rings.number_field
    sage: F = K.construction()[0]  # optional - sage.rings.number_field
    sage: F(ZZ['t'])  # optional - sage.rings.number_field
    Univariate Quotient Polynomial Ring in a
    over Univariate Polynomial Ring in t over Integer Ring
    with modulus a^3 + a^2 + 1

3.4. Coercion via construction functors
Note that, even if a field is algebraically closed, the algebraic extension will be constructed as the quotient of a univariate polynomial ring:

```sage
sage: F(CC) #optional - 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
sage: F(RR) #optional - sage.rings.number_field
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 `ZZ.extension(...)`:

```sage
sage: F(ZZ) #optional - sage.rings.number_field
Order in Number Field in a with defining polynomial x^3 + x^2 + 1
```

This also holds for non-absolute number fields:

```sage
sage: x = polygen(QQ, 'x')
sage: K.<a,b> = NumberField([x^3 + x^2 + 1, x^2 + x + 1]) #optional - sage.rings.number_field
sage: F = K.construction()[0] #optional - sage.rings.number_field
sage: O = F(ZZ); O #optional - sage.rings.number_field
Relative Order in Number Field in a with defining polynomial x^3 + x^2 + 1 over its base field
sage: O.ambient() is K #optional - sage.rings.number_field
True
```

Special cases are made for cyclotomic fields and residue fields:

```sage
sage: C = CyclotomicField(8) #optional - sage.rings.number_field
sage: F, R = C.construction() #optional - sage.rings.number_field
sage: F #optional - sage.rings.number_field
AlgebraicExtensionFunctor
sage: R #optional - sage.rings.number_field
Rational Field
sage: F(R) #optional - sage.rings.number_field
Cyclotomic Field of order 8 and degree 4
sage: F(ZZ) #optional - sage.rings.number_field
Maximal Order in Cyclotomic Field of order 8 and degree 4
```
sage: K.<z> = CyclotomicField(7)  # optional - sage.rings.number_field
sage: P = K.factor(17)[0][0]  # optional - sage.rings.number_field
sage: k = K.residue_field(P)  # optional - sage.rings.number_field
sage: F, R = k.construction()  # optional - sage.rings.number_field
sage: F  # optional - sage.rings.number_field
AlgebraicExtensionFunctor
sage: R  # optional - sage.rings.number_field
Cyclotomic Field of order 7 and degree 6
sage: F(R) is k  # optional - sage.rings.number_field
True
sage: F(ZZ)  # optional - sage.rings.number_field
Residue field of Integers modulo 17
sage: F(CyclotomicField(49))  # optional - sage.rings.number_field
Residue field in zbar of Fractional ideal (17)

**expand()**

Decompose the functor $F$ into sub-functors, whose product returns $F$.

**EXAMPLES:**

sage: P.<x> = QQ[]

sage: K.<a> = NumberField(x^3 - 5, embedding=0)  # optional - sage.rings.number_field
sage: L.<b> = K.extension(x^2 + a)  # optional - sage.rings.number_field
sage: F, R = L.construction()  # optional - sage.rings.number_field
sage: prod(F.expand())(R) == L  # optional - sage.rings.number_field
True

sage: K = NumberField([x^2 - 2, x^2 - 3], 'a')  # optional - sage.rings.number_field

sage: F, R = K.construction()  # optional - sage.rings.number_field
sage: F  # optional - sage.rings.number_field
AlgebraicExtensionFunctor
sage: L = F.expand(); L  # optional - sage.rings.number_field
[AlgebraicExtensionFunctor, AlgebraicExtensionFunctor]

merge(other)
Merging with another \texttt{AlgebraicExtensionFunctor}.

INPUT:

\texttt{other} – Construction Functor.

OUTPUT:

- If \texttt{self==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, \texttt{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:
The following demonstrate coercions for finite fields using Conway or pseudo-Conway polynomials:

\begin{verbatim}
sage: k = GF(3^2, prefix='z'); a = k.gen()     # optional - sage.rings.finite_rings
sage: l = GF(3^3, prefix='z'); b = l.gen()     # optional - sage.rings.finite_rings
sage: a + b # indirect doctest                      # optional - sage.rings.finite_rings
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
\end{verbatim}

Note that embeddings are compatible in lattices of such finite fields:

\begin{verbatim}
sage: m = GF(3^5, prefix='z'); c = m.gen()     # optional - sage.rings.finite_rings
sage: (a + b) + c == a + (b + c) # indirect doctest     # optional - sage.rings.finite_rings
True
sage: from sage.categories.pushout import pushout
sage: n = pushout(k, l)                      # optional - sage.rings.finite_rings
sage: o = pushout(l, m)                      # optional - sage.rings.finite_rings
sage: q = pushout(n, o)                      # optional - sage.rings.finite_rings
sage: q(o(b)) == q(n(b)) # indirect doctest # optional - sage.rings.finite_rings
True
\end{verbatim}

Coercion is also available for number fields:
\begin{verbatim}
sage: P.<x> = QQ[]
# optional - sage.rings.number_field
sage: L.<b> = NumberField(x^8 - x^4 + 1, embedding=CDF.0)
# optional - sage.rings.number_field
sage: M1.<c1> = NumberField(x^2 + x + 1, embedding=b^4 - 1)
# optional - sage.rings.number_field
sage: M2.<c2> = NumberField(x^2 + 1, embedding=-b^6)
# optional - sage.rings.number_field
sage: M1.coerce_map_from(M2)
# optional - sage.rings.number_field
sage: M2.coerce_map_from(M1)
# optional - sage.rings.number_field
sage: c1 + c2; parent(c1 + c2)
# indirect doctest
# optional - sage.rings.number_field
-b^6 + b^4 - 1
Number Field in b with defining polynomial x^8 - x^4 + 1
with b = -0.2588190451025208? + 0.9659258262890683?*I
sage: pushout(M1['x'], M2['x'])
# optional - sage.rings.number_field
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
\end{verbatim}

In the previous example, the number field \( L \) becomes the pushout of \( M_1 \) and \( M_2 \) 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:

\begin{verbatim}
sage: cbrt2 = CDF(2)^(1/3)
sage: zeta3 = CDF.zeta(3)
sage: K.<a> = NumberField(x^3 - 2, embedding=cbrt2 * zeta3)
# optional - sage.rings.number_field
sage: L.<b> = NumberField(x^6 - 2, embedding=1.1)
# optional - sage.rings.number_field
sage: L.coerce_map_from(K)
# optional - sage.rings.number_field
sage: K.coerce_map_from(L)
# optional - sage.rings.number_field
sage: pushout(K, L)
# optional - sage.rings.number_field
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', Number Field in a with
defining polynomial x^3 - 2 with a = -0.6299605249474365? + 1.091123659717227?*I,
Number Field in b with defining polynomial x^6 - 2 with b = 1.122462048309373?)[/n
\end{verbatim}

\texttt{rank = 3}

\textbf{class} \text{ sage.categories.pushout.BlackBoxConstructionFunctor}(\text{box})

\textbf{Bases:} \text{ConstructionFunctor}

Construction functor obtained from any callable object.

\textbf{EXAMPLES:}
```python
sage: from sage.categories.pushout import BlackBoxConstructionFunctor
sage: FG = BlackBoxConstructionFunctor(gap)  # optional - sage.libs.gap
sage: FS = BlackBoxConstructionFunctor(singular)  # optional - sage.libs.singular
sage: FG
BlackBoxConstructionFunctor
sage: FG(ZZ)  # optional - sage.libs.gap
Integers
sage: FG(ZZ).parent()  # optional - sage.libs.gap
Gap
sage: FS(QQ['t'])  # optional - sage.libs.singular
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 1
// block 1 : ordering lp
// : names t
// block 2 : ordering C
sage: FG == FS  # optional - sage.libs.gap sage.libs.singular
False
sage: FG == loads(dumps(FG))  # optional - sage.libs.gap
True
```

```
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:
```
Completion commutes with fraction fields.

**EXAMPLES:**

```
sage: F1 = Zp(5).construction()[0]
      # optional - sage.rings.padics
sage: F2 = QQ.construction()[0]
sage: F1.commutes(F2)                                # optional - sage.rings.padics
True
```

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`. 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:**

```
sage: R1.<a> = Zp(5, prec=20)[]                      # optional - sage.rings.padics
sage: R2 = Qp(5, prec=40)                            # optional - sage.rings.padics
sage: R2(1) + a                                      # indirect doctest
      # optional - sage.rings.padics
          (1 + O(5^20))*a + 1 + O(5^40)
sage: R3 = RealField(30)                            # optional - sage.rings.padics
sage: R4 = RealField(50)                            # optional - sage.rings.padics
sage: R3(1) + R4(1)                                  # indirect doctest
      # optional - sage.rings.padics
2.000000000
sage: (R3(1) + R4(1)).parent()                       # optional - sage.rings.padics
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
    Poly[y](FractionField(Poly[x](FractionField(...))))
    sage: F == loads(dumps(F))
    True
    sage: F == CompositeConstructionFunctor(*F.all)
    True
    sage: F(GF(2)['t'])
    # optional - sage.rings.finite_rings
    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)

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: 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
    Poly[y](FractionField(Poly[x](FractionField(...))))
    sage: prod(F.expand()) == F
    True

class sage.categories.pushout.ConstructionFunctor

    Bases: Functon

    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 \( \mathbb{Z}[\{'x'\}] \) and \( \mathbb{Q} \) are obtained from \( \mathbb{Z} \) by applying a polynomial ring construction respectively the fraction field construction. Each construction functor has an attribute \texttt{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
QuotientFunctor
sage: Q15.merge(Q35)
QuotientFunctor
sage: Q15.merge(Q35)(ZZ)
Ring of integers modulo 5
```

Functors can not only be applied to objects, but also to morphisms in the respective categories. For example:

```
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]

# optional - sage.rings.finite_rings
sage: f = A.hom([a + b, a - b], A)
# optional - sage.rings.finite_rings
sage: F(A)
# optional - sage.rings.finite_rings
```

(continues on next page)
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

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:

sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.commutes(P)
False
sage: P.commutes(F)
expand()  
Decompose self into a list of construction functors.

**Note:** The default is to return the list only containing `self`.

**EXAMPLES:**

```python
sage: F = QQ.construction()[0]
sage: F.expand()
[FractionField]
sage: Q = ZZ.quo(2).construction()[0]
sage: Q.expand()
[QuotientFunctor]
sage: P = ZZ['t'].construction()[0]
sage: FP = F*P
sage: FP.expand()
[FractionField, Poly['t']]
```

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:**

```python
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.merge(F)
FractionField
sage: F.merge(P)
FractionField
sage: P.merge(F)
Poly['t']
sage: P.merge(P)
Poly['t']
```

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:

```
sage: GSym01 = PermutationGroup([[(0,1),(2,), (3,)]]); GSym01
Permutation Group with generators 
[(0,1)]
sage: GASym01 = PermutationGroup([[(0,1),(2,3)]]); GASym01
Permutation Group with generators 
[(0,1)(2,3)]
sage: from sage.categories.action import Action
sage: from sage.structure.element import Matrix
sage: class TensorIndexAction(Action):
    ...:     def _act_(self, g, x):
    ...:         if isinstance(x, Matrix):
    ...:             if g(0) == 1:
    ...:                 if g(2) == 2:
    ...:                     return x.transpose()
    ...:                 else:
    ...:                     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)
(continues on next page)```
\begin{verbatim}
[1 3]
[2 4]
sage: GAsym01_action.act(GAsym01.0, M)  # optional - sage.groups sage.modules
[-1 -3]
[-2 -4]
sage: Sym01 = M.parent().invariant_module(GSym01, action=GSym01_action); Sym01  # optional - sage.groups sage.modules
(Permutation Group with generators [(0,1)])-invariant submodule of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: list(Sym01.basis())  # optional - sage.groups sage.modules
[B[0], B[1], B[2]]
sage: list(Sym01.basis().map(Sym01.lift))  # optional - sage.groups sage.modules
[[[1 0] [0 1] [0 0]
  [0 0], [1 0], [0 1]]

sage: ASym01 = M.parent().invariant_module(GASym01, action=GASym01_action)  # optional - sage.groups sage.modules
sage: ASym01  # optional - sage.groups sage.modules
(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())  # optional - sage.groups sage.modules
[B[0]]
sage: list(ASym01.basis().map(ASym01.lift))  # optional - sage.groups sage.modules
[[[0 1]
  [-1 0]]

sage: from sage.categories.pushout import pushout
sage: pushout(Sym01, QQ)  # optional - sage.groups sage.modules
(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()  # optional - sage.groups sage.modules
Category of integral domains
sage: F.codomain()  # optional - sage.groups sage.modules

(continues on next page)
\end{verbatim}

3.4. Coercion via construction functors 145
Category Framework, Release 10.1

**Category of fields**

```python
sage: F(GF(5)) is GF(5)  # optional - sage.rings.finite_rings
True
```

```python
sage: F(ZZ['t'])
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
```

```python
sage: P.<x,y> = QQ[]
sage: f = P.hom([x+2*y,3*x-y],P)
sage: F(f)
Ring endomorphism of
  Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
  Defn: x |--> x + 2*y
          y |--> 3*x - y
sage: F(f)(1/x)
1/(x + 2*y)
sage: F == loads(dumps(F))
True
```

```python
rank = 5
```

**class** `sage.categories.pushout.IdentityConstructionFunctor`

- **Bases:** `ConstructionFunctor`

A construction functor that is the identity functor.

```python
rank = -100
```

**class** `sage.categories.pushout.InfinitePolynomialFunctor(gens, order, implementation)`

- **Bases:** `ConstructionFunctor`

A Construction Functor for Infinite Polynomial Rings (see `infinite_polynomial_ring`).

**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 \( v_1, ..., v_n \) 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(InfPoly[['a,b'], "lex", "dense"])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
sage: B.<x,y,a_3,a_1> = PolynomialRing(QQ, order='lex')
sage: B.construction()
```

(continues on next page)
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:

```
sage: R = PolynomialRing(QQ, names=['x','y','a_3','a_1'], order='lex')
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
  ... CoercionException: Overlapping variables ('a', 'b'),['a_1', 'a_3']) are incompatible
```

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:

```
sage: R = PolynomialRing(QQ, names=['x','y','a_1','a_3'], order='lex')
sage: A.construction()[0] * R.construction()[0]
Traceback (most recent call last):
  ... CoercionException: Overlapping variables ('a', 'b'),['a_1', 'a_3']) are incompatible
```

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()[0] * R.construction()[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)
```

$X$ 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
```

(continues on next page)
sage: w^2 + z^3
w_2 + z_3
sage: _.parent() is P
True

**expand()**

Decompose the functor $F$ into sub-functors, whose product returns $F$.

**EXAMPLES:**

```
sage: A = InfinitePolynomialRing(QQ, ['x', 'y'], order='degrevlex')
sage: F = A.construction()[0]; F
InfPoly{[x,y], "degrevlex", "dense"}
sage: F.expand()
[InfPoly{[y], "degrevlex", "dense"}, InfPoly{[x], "degrevlex", "dense"}]
sage: A = InfinitePolynomialRing(QQ, ['x', 'y', 'z'], order='degrevlex')
sage: F = A.construction()[0]; F
InfPoly{[x,y,z], "degrevlex", "dense"}
sage: F.expand()
[InfPoly{[z], "degrevlex", "dense"},
  InfPoly{[y], "degrevlex", "dense"},
  InfPoly{[x], "degrevlex", "dense"}]
sage: prod(F.expand())==F
True
```

**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:**

```
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]
sage: Y.construction()[0].merge(Y.construction()[0])
InfPoly{[x,y], "degrevlex", "dense"}
sage: y^3 + X(x^2)
x_2 + y_3
sage: _.parent().construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
```

**rank = 9.5**

```
class sage.categories.pushout.LaurentPolynomialFunctor(var, multi_variate=False)

Bases: ConstructionFunctor

Construction functor for Laurent polynomial rings.

**EXAMPLES:**
```
sage: L.<t> = LaurentPolynomialRing(ZZ)
sage: F = L.construction()[0]
sage: F
LaurentPolynomialFunctor
sage: F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
sage: K.<x> = LaurentPolynomialRing(ZZ)
sage: F(K)
Univariate Laurent Polynomial Ring in t

over Univariate Laurent Polynomial Ring in x over Integer Ring
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 Ring
      Defn: x |--> x + 2*y
            y |--> 3*x - y
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:

sage: from sage.categories.pushout import LaurentPolynomialFunctor
sage: F1 = LaurentPolynomialFunctor('t')
sage: F2 = LaurentPolynomialFunctor('t', multi_variate=True)
sage: F1.merge(F2)
LaurentPolynomialFunctor

Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2
sage: F1.merge(F1)(LaurentPolynomialRing(GF(2), 'a'))

Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2

rank = 9

class sage.categories.pushout.MatrixFunctor(nrows, ncols, is_sparse=False)

Bases: ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:

sage: MS = MatrixSpace(ZZ, 2, 3)

Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2

(continues on next page)
MatrixFunctor

```
sage: MS = MatrixSpace(ZZ, 2)                      #.
optimal - sage.modules
sage: F = MS.construction()[0]; F                  #.
optimal - sage.modules
MatrixFunctor
sage: P.<x,y> = QQ[]                              #.
optimal - sage.modules
sage: R = F(P); R                                 #.
optimal - sage.modules
Full MatrixSpace of 2 by 2 dense matrices
over Multivariate Polynomial Ring in x, y over Rational Field
sage: f = P.hom([x+y, x-y], P); F(f)               #.
optimal - sage.modules
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
sage: M = R([x, y, x*y, x + y])                   #.
optimal - sage.modules
sage: F(f)(M)                                    #.
optimal - sage.modules
[ 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:**

```
sage: F1 = MatrixSpace(ZZ, 2, 2).construction()[0]  #.
optimal - sage.modules
sage: F2 = MatrixSpace(ZZ, 2, 3).construction()[0]  #.
optimal - sage.modules
sage: F3 = MatrixSpace(ZZ, 2, 2, sparse=True).construction()[0]  #.
optimal - sage.modules
sage: F1.merge(F2)                                  #.
optimal - sage.modules
sage: F1.merge(F3)                                  #.
optimal - sage.modules
MatrixFunctor
sage: F13 = F1.merge(F3)                            #.
optimal - sage.modules
sage: F13.is_sparse                                 #.
optimal - sage.modules
False
sage: F1.is_sparse                                  #.
```
rank = 10

class sage.categories.pushout.MultiPolynomialFunctor(vars, term_order)

Bases: ConstructionFunctor

A constructor for multivariate polynomial rings.

EXAMPLES:

```python
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]
    # optional - sage.rings.finite_rings
sage: F(A)
    # optional - sage.rings.finite_rings
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])
    # optional - sage.rings.finite_rings
sage: F(f)
    # optional - sage.rings.finite_rings
Ring endomorphism of Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Finite Field of size 5
    of size 5
    Defn: a |--> a + b
    b |--> a - b
sage: F(f)(F(A)(x)*a)
(a + b)*x
```

expand()

Decompose self into a list of construction functors.

EXAMPLES:

```python
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]]
```

Now an actual use case:
merge(other)

Merge self with another construction functor, or return None.

EXAMPLES:

```sage
sage: F = sage.categories.pushout.MultiPolynomialFunctor([['x','y']], None)
sage: G = sage.categories.pushout.MultiPolynomialFunctor([['t']], None)
sage: F.merge(G) is None
True
go:
sage: 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)])],  #
    ....:     [1,2]); PF
    PermutationGroupFunctor([(1,2)])

gens()

    EXAMPLES:

    sage: P1 = PermutationGroup([(1,2)])  #
    ....:     optional - sage.groups
    sage: PF, P = P1.construction()  #
    ....:     optional - sage.groups
    sage: PF.gens()  #
    ....:     optional - sage.groups
    ((1,2),)

merge(other)

    Merge self with another construction functor, or return None.

    EXAMPLES:

    sage: P1 = PermutationGroup([(1,2)])  #
    ....:     optional - sage.groups
    sage: PF1, P = P1.construction()  #
    ....:     optional - sage.groups
    sage: P2 = PermutationGroup([(1,3)])  #
    ....:     optional - sage.groups
    sage: PF2, P = P2.construction()  #
    ....:     optional - sage.groups
    sage: PF1.merge(PF2)  #
    ....:     optional - sage.groups
    PermutationGroupFunctor([(1,2), (1,3)])

    rank = 10

class sage.categories.pushout.PolynomialFunctor(var, multi_variate=False, sparse=False, implementation=None)
    Bases: ConstructionFunctor

    Construction functor for univariate polynomial rings.

    EXAMPLES:

    sage: P = ZZ['t'].construction()[0]
    sage: P(GF(3))  #
    ....:     optional - sage.rings.finite_rings
    Univariate Polynomial Ring in t over Finite Field of size 3
    sage: P == loads(dumps(P))  #
    ....:     optional - sage.rings.finite_rings

(continues on next page)
By github issue #9944, the construction functor distinguishes sparse and dense polynomial rings. Before, the following example failed:

```python
sage: R.<x> = PolynomialRing(GF(5), sparse=True)
# ˓→optional - sage.rings.finite_rings
sage: F, B = R.construction()
# ˓→optional - sage.rings.finite_rings
sage: F(B) is R
# ˓→optional - sage.rings.finite_rings
True
sage: S.<x> = PolynomialRing(ZZ)
sage: R.has_coerce_map_from(S)
# ˓→optional - sage.rings.finite_rings
False
sage: S.has_coerce_map_from(R)
# ˓→optional - sage.rings.finite_rings
False
sage: S.0 + R.0
# ˓→optional - sage.rings.finite_rings
2*x
sage: (S.0 + R.0).parent()
# ˓→optional - sage.rings.finite_rings
Univariate Polynomial Ring in x over Finite Field of size 5
sage: (S.0 + R.0).parent().is_sparse()
# ˓→optional - sage.rings.finite_rings
False
```

merge(\texttt{other})

Merge \texttt{self} with another construction functor, or return \texttt{None}.

\textbf{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 \texttt{None} unless the to-be-merged functors coincide.

\textbf{EXAMPLES:}

```python
sage: P = ZZ['x'].construction()[0]
sage: Q = ZZ['y', 'x'].construction()[0]
sage: P.merge(Q)
sage: P.merge(P) \textbf{is} P
True
```

\textbf{rank = 9}
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:

```
sage: P.<x,y> = ZZ[]
sage: Q = P.quo([x^2 + y^2] * P)
    # optional - sage.rings.finite_rings
sage: F = Q.construction()[0]
    # optional - sage.rings.finite_rings
sage: F(QQ['x','y'])
    # optional - sage.rings.finite_rings
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'])
    # optional - sage.rings.finite_rings
True
```

```
sage: F(QQ['x','y','z'])
    # optional - sage.rings.finite_rings
Traceback (most recent call last):
... CoercionException: Cannot apply this quotient functor to
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
sage: F(QQ['y','z'])
    # optional - sage.rings.finite_rings
Traceback (most recent call last):
... TypeError: Could not find a mapping of the passed element to this ring.
```

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:

```
sage: P.<x> = QQ[]
sage: Q1 = P.quo([(x^2+1)^2*(x^2-3)])
    # optional - sage.rings.finite_rings
sage: Q2 = P.quo([(x^2+1)^2*(x^5+3)])
    # optional - sage.rings.finite_rings
sage: from sage.categories.pushout import pushout
sage: pushout(Q1,Q2)
    # indirect doctest
Univariate Quotient Polynomial Ring in xbar over Rational Field
with modulus x^4 + 2*x^2 + 1
```
The following was fixed in github issue #8800:

```python
sage: pushout(GF(5), Integers(5))
#optional - sage.rings.finite_rings
```

Finite Field of size 5

**rank = 4.5**

**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:**

```python
sage: M = ZZ^3
#optional - sage.modules
sage: S = M.submodule([(1,2,3), (4,5,6)]); S
#optional - sage.modules
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]
#optional - sage.modules
sage: F(GF(2)^3)
#optional - sage.rings.finite_rings sage.modules
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

**coercion_reversed = True**

**merge**(other)

Two Subspace Functors are merged into a construction functor of the sum of two subspaces.

**EXAMPLES:**

```python
sage: M = GF(5)^3
#optional - sage.modules sage.rings.finite_rings
sage: S1 = M.submodule([(1,2,3),(4,5,6)]); S1
#optional - sage.modules sage.rings.finite_rings
sage: S2 = M.submodule([(2,2,3)])
#optional - sage.modules sage.rings.finite_rings
sage: F1 = S1.construction()[0]
#optional - sage.modules sage.rings.finite_rings
sage: F2 = S2.construction()[0]
#optional - sage.modules sage.rings.finite_rings
sage: F1.merge(F2)
#optional - sage.modules sage.rings.finite_rings
```

(continues on next page)
SubspaceFunctor

```python
sage: F1.merge(F2)(GF(5)^3) == S1 + S2  #
→optional - sage.modules sage.rings.finite_rings
True
sage: F1.merge(F2)(GF(5)[t]^3)  #
→optional - sage.modules sage.rings.finite_rings
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
rank = 11
```

### class sage.categories.pushout.VectorFunctor

```
Bases: ConstructionFunctor

A construction functor for free modules over commutative rings.
```

```python
sage: F = (ZZ^3).construction()[0]  #
→optional - sage.modules
sage: F  #
→optional - sage.modules
VectorFunctor
sage: F(GF(2)[t])  #
→optional - sage.rings.finite_rings sage.modules
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

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.

```python
EXAMPLES:

```python
sage: M1 = QQ^3
→optional - sage.modules
sage: P.<t> = ZZ[]
sage: M2 = FreeModule(P, 3)
→optional - sage.modules
sage: M1([1,1/2,1/3]) + M2([t,t^2+t,3])  # indirect doctest
→optional - 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:
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 *explicitly* requested for $M_4$. It is therefore not possible to coerce with a different inner product:

```
sage: M4 = FreeModule(QQ, 3, inner_product_matrix=Matrix(3, 3, 1))  # optional - sage.modules
sage: M4 == M1  # optional - sage.modules
True
sage: M4.inner_product_matrix() == M1.inner_product_matrix()  # optional - sage.modules
True
sage: M4([1,1/2,1/3]) + M3([t,t^2+t,3])  # indirect doctest  # optional - sage.modules
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]'
```

Names are removed when they conflict:

```
sage: from sage.categories.pushout import VectorFunctor, pushout
sage: M_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='M_ZZx')  # optional - sage.modules
sage: N_ZZx = FreeModule(ZZ['x'], 4, with_basis=None, name='N_ZZx')  # optional - sage.modules
sage: pushout(M_ZZx, QQ)  # optional - sage.modules
Rank-4 free module M_ZZx_base_ext
over the Univariate Polynomial Ring in x over Rational Field
sage: pushout(M_ZZx, N_ZZx)  # optional - sage.modules
(continues on next page)
```
optional - sage.modules

Rank-4 free module
over the Univariate Polynomial Ring in x over Integer Ring

sage: pushout(pushout(M_ZZx, N_ZZx), QQ)  

optional - sage.modules

Rank-4 free module
over the Univariate Polynomial Ring in x over Rational Field

\[ \text{rank} = 10 \]

sage.categories.pushout.construction_tower\( (R) \)

An auxiliary function that is used in \texttt{pushout()} and \texttt{pushout_lattice()}.  

\textbf{INPUT:}

An object

\textbf{OUTPUT:}

A constructive description of the object from scratch, by a list of pairs of a construction functor and an object to which the construction functor is to be applied. The first pair is formed by \texttt{None} and the given object.

\textbf{EXAMPLES:}

sage: from sage.categories.pushout import construction_tower
sage: construction_tower(MatrixSpace(FractionField(QQ['t']), 2))

\[(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 \texttt{pushout()}.  

\textbf{INPUT:}

A construction tower as returned by \texttt{construction_tower()}.  

\textbf{OUTPUT:}

A new construction tower with all the construction functors expanded.  

\textbf{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[x,y,z], 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[t], Rational Field), (FractionField, Integer Ring),
(FractionField, Integer Ring), (FractionField, Integer Ring)]

3.4. Coercion via construction functors
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 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 coerces 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 $\text{Frac}(\text{Poly}_x(\text{Complete}_7(\text{Frac}(\mathbb{Z}))))$:

```python
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7), Frac(ZZ['x']))  # optional - 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

```python
sage: pushout(Zp(7), Frac(QQ['x']))  # optional - 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']))  # optional - 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.

```sage
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)
```

```sage
sage: pushout(ZZ['x,y,z'], QQ['w,x,z,t'])
Multivariate Polynomial Ring in w, x, y, z, t over Rational Field
```

Some other examples:

```sage
sage: pushout(Zp(7)['y'], Frac(QQ['t'])['x,y,z'])  # optional - 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
```

```sage
sage: pushout(ZZ['x,y,z'], Frac(ZZ['x'])['y'])
Multivariate Polynomial Ring in y, z
over Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

```sage
sage: pushout(MatrixSpace(RDF, 2, 2), Frac(ZZ['x']))  # optional - sage.modules
Full MatrixSpace of 2 by 2 dense matrices
over Power Series Ring in x over Real Double Field
```

```sage
sage: pushout(ZZ, MatrixSpace(ZZ['x'], 3, 3))  # optional - sage.modules
Full MatrixSpace of 3 by 3 dense matrices
over Power Series Ring in x over Integer Ring
```

A construction with `coercion_reversed=True` (currently only the `SubspaceFunctor` construction) is only applied if it leads to a valid coercion:

```sage
sage: A = ZZ^2  # optional - sage.modules
sage: V = span([[1, 2]], QQ)  # optional - sage.modules
sage: P = sage.categories.pushout.pushout(A, V)  # optional - sage.modules
sage: P
Vector space of dimension 2 over Rational Field
```

```sage
sage: P.has_coerce_map_from(A)  # optional - sage.modules
True
```

```sage
sage: V = (QQ^3).span([[1, 2, 3/4]])  # optional - sage.modules
sage: A = ZZ^3  # (continues on next page)
Vector space of dimension 3 over Rational Field

```python
sage: B = A.span([[0, 0, 2/3]])
```

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)
    ....: def _apply Functor(self):
```

```python
sage: class EvenPolynomialFunctor(ConstructionFunctor):
    ....: rank = 10
    ....: def __init__(self):
    ....:     super().__init__(Rings(), Rings())
    ....: def _apply Functor(self, R):
    ....:     return EvenPolynomialRing(R.base(), R.variable_name())
```

Even Power Univariate Polynomial Ring in x over Rational Field

Even Power Univariate Polynomial Ring in x over Rational Field

Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of precision

Even Power Univariate Polynomial Ring in x over Rational Field

Even Power Univariate Polynomial Ring in x over Rational Field

Univariate Polynomial Ring in x over Rational Field

Univariate Polynomial Ring in x over Rational Field

Univariate Polynomial Ring in x over Real Field with 53 bits of precision

Even Power Univariate Polynomial Ring in x over Rational Field

Even Power Univariate Polynomial Ring in x over Rational Field

Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of precision
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

\[
\sum_{i=0}^{I} c_i x^{e_i}
\]

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)
    ....:    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)

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) 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', '...')

sage: GP_QQ = GPolynomialFunctor('X', QQ)
sage: pushout(GP_ZZ(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
sage: 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

sage: pushout(GP_ZZt(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZt(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Integer Ring
sage: pushout(GP_ZZt(QQ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['c,d']))

(continues on next page)
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ([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

Some tests with Cartesian products:

```
sage: from sage.sets.cartesian_product import CartesianProduct
sage: A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z'])),
   ....:   Sets().CartesianProducts()
  ....: sage: B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z'])),
   ....:   Sets().CartesianProducts()
sage: 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))
sage: 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)
sage: 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):
   ....:     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(
```

3.4. Coercion via construction functors 165
other.cartesian_factors() +
tuple(f for f in self.cartesian_factors()
    if f.variable_name() not in o_vars))

if C is None:
    return
elif isinstance(C[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)

sage: pushout(CartesianProductPoly((ZZ['x'],)), ZZ['y'])
The Cartesian product of
(Univariate Polynomial Ring in x over Integer Ring,
 Univariate Polynomial Ring in y over Integer Ring)
sage: pushout(QQ['b,c']['y'], CartesianProductPoly((ZZ['a,b']['x'],)))
The Cartesian product of
(Univariate Polynomial Ring in x over
 Multivariate Polynomial Ring in a, b over Integer Ring,
 Univariate Polynomial Ring in y over
 Multivariate Polynomial Ring in b, c over Rational Field)

sage: pushout(CartesianProductPoly((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
• Peter Bruin
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']))
    #→ optional - sage.rings.padics
sage: A.codomain()    #→ optional - 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() #→ optional - sage.rings.padics
True
sage: A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], 3, 3))    #→ optional - sage.modules
sage: B    #→ optional - sage.modules
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 `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 $G \times S \rightarrow S$ is a functor from $G$ to Sets.

**Warning:** An Action object only keeps a weak reference to the underlying set which is acted upon. This decision was made in github issue #715 in order to allow garbage collection within the coercion framework (this is where actions are mainly used) and avoid memory leaks.

To avoid garbage collection of the underlying set, it is sufficient to create a strong reference to it before the action is created.

**AUTHOR:**
- Robert Bradshaw: initial version

```python
sage: from sage.categories.action import Action
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>
```

```python
sage: _ = gc.collect()
sage: from sage.categories.action import Action
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 ...>
```

**class** `sage.categories.action.Action`

Bases: `Functor`

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

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:

```python
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*x
sage: A.act(int(5), x)
5*x
sage: A = IntegerMulAction(ZZ, R, False)  # Right action
sage: A.act(5, x)
5*x
sage: A.act(int(5), x)
5*x
```

actor()
codomain()
domain()
is_left()
left_domain()
operation()
right_domain()

class sage.categories.action.ActionEndomorphism

Bases: Morphism

The endomorphism defined by the action of one element.

EXAMPLES:

```python
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
    # optional - sage.modules
sage: v = V((1, 2, 3))
    # optional - sage.modules
sage: cm = get_coercion_model()

sage: a = cm.get_action(V, QQ, operator.mul)
    # optional - sage.modules
sage: a
    # optional - sage.modules
Right scalar multiplication by Rational Field
on Vector space of dimension 3 over Rational Field

sage: ~a
    # optional - sage.modules
Right inverse action by Rational Field
on Vector space of dimension 3 over Rational Field

sage: (~a)(v, 1/3)
    # optional - sage.modules
(3, 6, 9)

sage: b = cm.get_action(QQ, V, operator.mul)
    # optional - sage.modules
sage: b
    # optional - sage.modules
Left scalar multiplication by Rational Field
on Vector space of dimension 3 over Rational Field

sage: ~b
    # optional - sage.modules
Left inverse action by Rational Field
on Vector space of dimension 3 over Rational Field

sage: (~b)(1/3, v)
    # optional - sage.modules
(3, 6, 9)

sage: c = cm.get_action(ZZ, list, operator.mul)

sage: ~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 github issue #14711 did not become a problem:

```python
sage: E = ModularSymbols(11).2
# optional - sage.modular
sage: s = E.modular_symbol_rep()
# optional - sage.modular
sage: del E, s
# optional - sage.modular
sage: import gc
# optional - sage.modular
sage: _ = gc.collect()
# optional - sage.modular
sage: E = ModularSymbols(11).2
# optional - sage.modular
sage: v = E.manin_symbol_rep()
# optional - sage.modular
sage: c,x = v[0]
# optional - sage.modular
sage: y = x.modular_symbol_rep()
# optional - sage.modular
sage: coercion_model.get_action(QQ, parent(y), op=operator.mul)
# optional - sage.modular
```

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

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

AdditiveCommutative
   alias of CommutativeAdditiveGroups
class Algebras(category, *args)
   Bases: AlgebrasCategory
class ParentMethods
   Bases: object
group()
Return the underlying group of the group algebra.
EXAmPLES:

sage: GroupAlgebras(QQ).example(GL(3, GF(11))).group()
˓→# optional - sage.groups sage.modules sage.rings.finite_rings
General Linear Group of degree 3 over Finite Field of size 11
sage: SymmetricGroup(10).algebra(QQ).group()
˓→# optional - sage.groups sage.modules
Symmetric group of order 10! as a permutation group

class Finite(base_category)
   Bases: CategoryWithAxiom_singleton
class Algebras(category, *args)
   Bases: AlgebrasCategory
class ParentMethods
   Bases: object
   extra_super_categories()
   Implement Maschke’s theorem.

4.2. Additive groups 173
In characteristic 0 all finite group algebras are semisimple.

EXAMPLES:

```python
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
```

### 4.3 Additive magmas

```python
class sage.categories.additive_magmas.AdditiveMagmas(s=None)

    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]
```

The following axioms are defined by this category:

```python
sage: AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
```
(continues on next page)
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:
        
        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]

class CartesianProducts(category, *args)
    Bases: CartesianProductsCategory
    extra_super_categories()
        Implement the fact that a Cartesian product of commutative additive magmas is a commutative 
        additive magma.
        EXAMPLES:
        
        sage: C = AdditiveMagmas().AdditiveCommutative().CartesianProducts()
        sage: C.extra_super_categories()
        [Category of commutative additive monoids
        sage: C.additive_inverse()
        Category of commutative additive groups

AdditiveCommutative
    alias of AdditiveSemigroups

AdditiveInverse
    alias of AdditiveMonoids

AdditiveAssociative
    alias of AdditiveSemigroups

AdditiveCommutative
    class AdditiveAssociative(base_category)
    Bases: CategoryWithAxiom_singleton

AdditiveUnital
    class AdditiveUnital(base_category)
    Bases: CategoryWithAxiom_singleton

AdditiveInverse
    class AdditiveInverse(base_category)
    Bases: CategoryWithAxiom_singleton

4.3. Additive magmas
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().
       \rightarrow CartesianProducts()
    sage: C.extra_super_categories()
    [Category of additive inverse additive unital additive magmas]
    sage: 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: 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)
    #optional - sage.modules
    sage: A.one_basis()  #optional - sage.modules
    0
    sage: A.one()  #optional - sage.modules
    B[0]
    sage: A(3)  #optional - sage.modules
    3*B[0]

extra_super_categories()
    EXAMPLES:

    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]
class CartesianProducts(*category, *args)
    Bases: CartesianProductsCategory

class ParentMethods
    Bases: object
    zero()
        Returns the zero of this group

    EXAMPLES:
    sage: GF(8, 'x').cartesian_product(GF(5)).zero()  # optional - 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:
    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
    zero()

    EXAMPLES:
    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

extra_super_categories()
    Implement the fact that a homset between two unital additive magmas is a unital additive magma.

    EXAMPLES:
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:

\begin{verbatim}
sage: A = AdditiveAbelianGroup([3, 3])  # optional - sage.groups
sage: A in AdditiveMagmas()  # optional - sage.groups
True
sage: A.is_empty()  # optional - sage.groups
False
sage: B = CommutativeAdditiveMonoids().example()
sage: B.is_empty()
False
\end{verbatim}

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:

\begin{verbatim}
sage: S = CommutativeAdditiveMonoids().example()
sage: S.zero()
0
\end{verbatim}

class SubcategoryMethods
Bases: object

AdditiveInverse()
Return the full subcategory of the additive inverse objects of self.

An inverse \textit{additive magma} is a \textit{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 \textit{additive loop}.

See also:
- Wikipedia article Inverse_element, Wikipedia article Quasigroup
EXAMPLES:

```python
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
```

```python
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().

    EXAMPLES:

```python
sage: A = Sets().WithRealizations().example(); A
# ˓→optional - sage.combinat
The subset algebra of {1, 2, 3} over Rational Field
sage: A.zero.__module__
# ˓→optional - sage.combinat
'sage.categories.additive_magmas'
sage: A.zero()
# ˓→optional - sage.combinat
0
```

```python
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
class Algebras(category, *args)
    Bases: AlgebrasCategory

class ParentMethods
    Bases: object

    algebra_generators()
    The generators of this algebra, as per MagmaticAlgebras.ParentMethods.algebra_generators().

    They correspond to the generators of the additive semigroup.
```

### 4.3. Additive magmas

179
EXAMPLES:

```python
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)         #...
˓→optional - sage.modules
sage: A.algebra_generators()    #...
˓→optional - 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**(gl, 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:

```python
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)         #...
˓→optional - sage.modules
sage: a, b, c, d = A.algebra_generators()    #...
˓→optional - sage.modules
sage: a * d * b                   #...
˓→optional - sage.modules
B[a + 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!

**extra_super_categories**( )

EXAMPLES:

```python
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]
```

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:

```python
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()
```

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]
```

class ParentMethods

Bases: object

addition_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()          #<optional - sage.modules>
+  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
```

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: R = IntegerModRing(11)
sage: P = R.addition_table(names='elements'); P          #<optional - sage.modules>
+  0  1  2  3  4  5  6  7  8  9  10
--------------------------
0| 0  1  2  3  4  5  6  7  8  9  10
1| 1  2  3  4  5  6  7  8  9  10  0
2| 3  4  5  6  7  8  9  10  0  1  2
3| 4  5  6  7  8  9  10  0  1  2  3
5| 6  7  8  9  10  0  1  2  3  4  5
6| 7  8  9  10  0  1  2  3  4  5  6
8| 9  10  0  1  2  3  4  5  6  7  8
9| 10  0  1  2  3  4  5  6  7  8  9
```

```python
sage: T = R.addition_table(names='digits'); T          #<optional - sage.modules>
+  00  01  02  03  04  05  06  07  08  09  10
--------------------------
00| 00  01  02  03  04  05  06  07  08  09  10
01| 01  02  03  04  05  06  07  08  09  10  00
02| 02  03  04  05  06  07  08  09  10  00  01
03| 03  04  05  06  07  08  09  10  00  01  02
04| 04  05  06  07  08  09  10  00  01  02  03
05| 05  06  07  08  09  10  00  01  02  03  04
06| 06  07  08  09  10  00  01  02  03  04  05
07| 07  08  09  10  00  01  02  03  04  05  06
08| 08  09  10  00  01  02  03  04  05  06  07
09| 09  10  00  01  02  03  04  05  06  07  08
10| 10  00  01  02  03  04  05  06  07  08  09
```
Specifying the elements in an alternative order can provide more insight into how the operation behaves.

```
sage: S = IntegerModRing(7)
sage: elts = [0, 3, 6, 2, 5, 1, 4]
sage: S.addition_table(elements=elts) #optional - sage.modules
+ a b c d e f g
+-------------
a| a b c d e f g
b| b c d e f g a
c| c d e f g a b
d| d e f g a b c
e| e f g a b c d
f| f g a b c d e
g| g a b c d e f
```

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: T = IntegerModRing(12)
sage: elts = [0, 3, 6, 9]
sage: T.addition_table(names='elements', elements=elts) #optional - 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: R = IntegerModRing(3)
sage: T = R.addition_table() #optional - sage.modules
sage: T.column_keys() #optional - sage.modules
(0, 1, 2)
sage: sorted(T.translation().items()) #optional - sage.modules
[('a', 0), ('b', 1), ('c', 2)]
sage: T.change_names(['x', 'y', 'z']) #optional - sage.modules
sage: sorted(T.translation().items()) #optional - sage.modules
[('x', 0), ('y', 1), ('z', 2)]
sage: T #optional - sage.modules
+ x y z
+-----
```

(continues on next page)
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
sage: S = CommutativeAdditiveSemigroups().example()
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: 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 $S.summation(x, y)$, and reciprocally.

As a bonus effect, $S.summation$ by itself models the binary function from $S$ to $S$:

```sage
sage: bin = S.summation
sage: 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.

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:

```sage
sage: S = CommutativeAdditiveSemigroups().example()
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: 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 $S.summation(x, y)$, and reciprocally.

As a bonus effect, $S.summation$ by itself models the binary function from $S$ to $S$: 
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.

```python
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:
```sage``
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``
AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
AdditiveMagmas().AdditiveAssociative().AdditiveUnital()
C = AdditiveMagmas().AdditiveAssociative().AdditiveUnital()
C.AdditiveCommutative()
Category of commutative additive monoids
```

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 $x \in M$,

    $$0 + x = x + 0 = x$$

4.3. Additive magmas
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()
Category of additive monoids
sage: CommutativeAdditiveMonoids().AdditiveUnital()
Category of commutative additive monoids
```

```
super_categories()
EXAMPLES:

```sage
sage: AdditiveMagmas().super_categories()
[Category of sets]
```

4.4 Additive monoids

class sage.categories.additive_monoids.AdditiveMonoids(base_category)

Bases: CategoryWithAxiom_singleton

The category of additive monoids.

An additive monoid is an unital additive semigroup, that is a set endowed with a binary operation $+$ which is associative and admits a zero (see Wikipedia article Monoid).

EXAMPLES:

```sage
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:

```
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:
```
```
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
sage: S.sum().parent() == S
True
```

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:

```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: C = AdditiveSemigroups(); C
Category of additive semigroups
sage: C.super_categories()
[Category of additive magmas]
sage: C.all_super_categories()
```

(continues on next page)
[Category of additive semigroups, 
Category of additive magmas, 
Category of sets, 
Category of sets with partial maps, 
Category of objects]

sage: C.axioms()
frozenset({'AdditiveAssociative'})
sage: C is AdditiveMagmas().AdditiveAssociative()
True

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)
    #optional - sage.modules
sage: A.algebra_generators()
    #optional - 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')
sage: A = S.algebra(QQ)
    #optional - sage.modules
sage: a, b, c, d = A.algebra_generators()
    #optional - sage.modules

extra_super_categories()

EXAMPLES:

```python
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: AdditiveSemigroups().Algebras(QQ).extra_super_categories()
[Category of semigroups]
sage: CommutativeAdditiveSemigroups().Algebras(QQ).super_categories()
[Category of additive semigroup algebras over Rational Field,
 Category of additive commutative magma algebras over Rational Field]
```

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: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: C = AdditiveSemigroups().CartesianProducts()
sage: C.extra_super_categories()
[Category of additive semigroups]
sage: C.axioms()
frozenset({'AdditiveAssociative'})
```

class Homsets(category, *args)

Bases: HomsetsCategory

extra_super_categories()

Implement the fact that a homset between two semigroups is a semigroup.

EXAMPLES:

```python
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]
```

class ParentMethods

Bases: object
4.6 Affine Weyl groups

```python
class sage.categories.affine_weyl_groups.AffineWeylGroups(s=None):
    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]
```sage
c.example()
NotImplemented
```sage
d = WeylGroup(['A', 4, 1]); d
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
d.category()
```sage
category_of_irreducible_affine_weyl_groups()
```sage
c.example()
NotImplemented

```python
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
d = WeylGroup(['A', 2, 1])  #
optional - sage.combinat sage.groups
d = WeylGroup(['A', 2, 1])  #
optional - sage.combinat sage.groups
```sage
d = WeylGroup(['A', 2, 1])  #
optional - sage.combinat sage.groups
```
def affine_grassmannian_to_partition(self):
    """Bijection between affine Grassmannian elements of type $A_k^{(1)}$ and $k$-bounded partitions.
    """
    INPUT:
    • self is affine Grassmannian element of the affine Weyl group of type $A_k^{(1)}$ (i.e. all reduced words end in 0)
    OUTPUT:
    • $k$-bounded partition
    See also:
    affine_grassmannian_to_core()
    EXAMPLES:

sage: k = 2
sage: W = WeylGroup(['A', k, 1])
    #optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([0,2,1])
    #optional - sage.combinat sage.groups
sage: la = w.affine_grassmannian_to_partition(); la
    #optional - sage.combinat sage.groups
[2, 2]

sage: la.to_grassmannian() == w
    #optional - sage.combinat sage.groups
True

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:
class ParentMethods

Bases: object

affine_grassmannian_elements_of_given_length(k)

Return the affine Grassmannian elements of length \( k \).

This is returned as a finite enumerated set.

EXAMPLES:

```
sage: W = WeylGroup(['A', 3, 1])  #-
..... optional - sage.combinat sage.groups
sage: [x.reduced_word()  #-
..... optional - sage.combinat sage.groups
..... for x in W.affine_grassmannian_elements_of_given_length(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:

```
sage: W = WeylGroup(['A', 3, 1])  #-
..... optional - sage.combinat sage.groups
sage: W.special_node()  #-
..... optional - 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:

```
sage: AffineWeylGroups().additional_structure()
```

**super_categories()**

EXAMPLES:

```
sage: 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:**

```
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:**

```
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:**

```
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:

    sage: AlgebraModules(QQ['a'])
    Category of algebra modules over Univariate Polynomial Ring in a over Rational Field
    sage: AlgebraModules(QQ['a']).super_categories()
    [Category of 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:

    sage: AlgebraModules(QQ['x']).algebra()
    Univariate Polynomial Ring in x over Rational Field

    classmethod an_instance()

    Returns an instance of this class

    EXAMPLES:

    sage: AlgebraModules.an_instance()
    Category of algebra modules over Univariate Polynomial Ring in x over Rational Field

    super_categories()

    EXAMPLES:

    sage: AlgebraModules(QQ['x']).super_categories()
    [Category of 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)
Bases: CategoryWithAxiom_over_base_ring

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:** $\textit{Algebras}$ will be eventually be replaced by $\textit{magmatic_algebras.MagmaticAlgebras}$ for consistency with e.g. Wikipedia article $\textit{Algebras}$ which assumes neither associativity nor the existence of a unit (see github issue #15043).

**Todo:** Should $R$ be a commutative ring?

**EXAMPLES:**

```sage
sage: Algebras(ZZ)
Category of algebras over Integer Ring
sage: sorted(Algebras(ZZ).super_categories(), key=str)
[Category of associative algebras over Integer Ring,
 Category of rings,
 Category of unital algebras over Integer Ring]
```

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:**

```sage
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 distributive magmas and additive magmas,
 Category of Cartesian products of monoids,
 Category of Cartesian products of vector spaces 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:

```
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]
```

**Warning:** This is only correct in certain cases (finite dimension, ...). See github issue #15647.
class ElementMethods
Bases: object
Filtered
alias of FilteredAlgebras
Graded
alias of GradedAlgebras
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: 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}
```

**Todo:** this could possibly remove the elements that retract to zero.
.. sectionheader:: 4.9. Algebras

.. class:: Super

    .. class:: super

    alias of SuperAlgebras

.. class:: SuperTensorProducts

    .. class:: element

    Bases: object

    .. class:: parent

    Bases: object

    .. method:: extra_super_categories()

    EXAMPLES:
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
sage: 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: 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:
word: 
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]

(continues on next page)
An example of an algebra with basis:
the free algebra on the generators ('a', 'b', 'c') over Rational Field

\begin{verbatim}
sage: A.product(a * b, b)  #optional - sage.combinat
\end{verbatim}

Please see the source code of \texttt{A} (with \texttt{A??}) for how to implement other algebras with basis.

\begin{verbatim}
class CartesianProducts(category, *args)
\end{verbatim}
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.

class ParentMethods

    Bases: object

    one()

    one_from_cartesian_product_of_one_basis()

    Returns 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: 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()

    sage: B = cartesian_product((A, A, A))
    sage: B.one_from_cartesian_product_of_one_basis()

    sage: B[(0, word: )] + B[(1, word: )] + B[(2, word: )]

    sage: B.one()

    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:

    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, ...
     Field]
class ElementMethods
Bases: object

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
sage: R.<x> = QQ[]
sage: A = algebras.DifferentialWeyl(R)  # optional - sage.combinat
sage: H = A.hochschild_complex(A)  # optional - sage.combinat
sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.combinat
sage: T = SGA.trivial_representation()  # optional - sage.combinat
sage: H = SGA.hochschild_complex(T)  # optional - sage.combinat
```

one()
Return the multiplicative unit element.

EXAMPLES:

```sage
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: A.one_basis()  # optional - sage.combinat sage.modules
word:
sage: A.one()  # optional - 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

4.10. 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: 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()  # optional - sage.combinat
    word:
    sage: B = tensor((A, A, A))  # optional - sage.combinat
    sage: B.one_basis()  # optional - sage.combinat
    (word: , word: , word: )
    sage: B.one()  # optional - sage.combinat

    product_on_basis(t1, t2)
    The product of the algebra on the basis, as per AlgebrasWithBasis.ParentMethods.
    product_on_basis.

    EXAMPLES:

    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()  # optional - sage.combinat
    sage: x = tensor((a, b, c)); x    # optional - sage.combinat
    sage: y = tensor((c, b, a)); y    # optional - sage.combinat
    sage: x * y    # optional - sage.combinat

(continues on next page)

\[
sage: x = tensor(((a + 2*b), c)); x
\]

\[
\rightarrow \text{optional - sage.combinat}
\]


\[
sage: y = tensor((c, a)) + 1; y
\]

\[
\rightarrow \text{optional - sage.combinat}
\]

B[ ] # B[c] + B[a]

\[
sage: x * y
\]

\[
\rightarrow \text{optional - sage.combinat}
\]


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]

**example**(*alphabet=('a', 'b', 'c'))

Return an example of algebra with basis.

**EXAMPLES:**

\[
sage: AlgebrasWithBasis(QQ).example()
\]

\[
\rightarrow \text{optional - sage.combinat}
\]

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:

\[
sage: AlgebrasWithBasis(QQ).example((1,2,3))
\]

\[
\rightarrow \text{optional - sage.combinat}
\]

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

**extra_super_categories**()

Implement the fact that an aperiodic semigroup is \(H\)-trivial.

**EXAMPLES:**

### 4.11. Aperiodic semigroups
**4.12 Associative algebras**

```python
sage: Semigroups().Aperiodic().extra_super_categories()
[Category of h trivial semigroups]
```

A category 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 github issue #15043 is implemented, \texttt{Algebras} is the category of associative unital algebras; thus, unlike the name suggests, \texttt{AssociativeAlgebras} is not a subcategory of \texttt{Algebras} but of \texttt{MagmaticAlgebras}.

**EXAMPLES:**

```python
sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: C = AssociativeAlgebras(ZZ); C
Category of associative algebras over Integer Ring
```

## 4.13 Bialgebras

```python
sage: Bialgebras(ZZ)
Category of bialgebras over Integer Ring
```

**EXAMPLES:**

```python
sage: Bialgebras(ZZ).super_categories()
[Category of algebras over Integer Ring, Category of coalgebras over Integer Ring]
```

```python
class ElementMethods
    Bases: object
    is_grouplike()
        Return whether self is a grouplike element.
```

**EXAMPLES:**
sage: s = SymmetricFunctions(QQ).schur()
sage: s([5]).is_grouplike()
False
sage: s([]).is_grouplike()
True

is_primitive()
Return whether self is a primitive element.

EXAMPLES:

sage: s = SymmetricFunctions(QQ).schur()
sage: s([5]).is_primitive()
False
sage: p = SymmetricFunctions(QQ).powersum()
sage: p([5]).is_primitive()
True

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()

super_categories()
EXAMPLES:

sage: 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 with basis over Rational Field,
     Category of bialgebras over Rational Field,
     Category of coalgebras with basis over Rational Field]

class ElementMethods
    Bases: object

    adams_operator(n)

    Compute the \( n \)-th convolution power of the identity morphism \( \text{Id} \) on \( \text{self} \).

    INPUT:
    • \( n \) – a nonnegative integer

    OUTPUT:
    • the image of \( \text{self} \) under the convolution power \( \text{Id}^n \)

    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:

    sage: h = SymmetricFunctions(QQ).h()
    # optional - sage.combinat
    sage: h[5].adams_operator(2)  # optional - sage.combinat
    sage: h[5].plethysm(2*h[1])  # optional - sage.combinat
    sage: h([]).adams_operator(0)  # optional - sage.combinat
    h[]
    sage: h([]).adams_operator(1)  # optional - sage.combinat
    h[]
    sage: h[3,2].adams_operator(0)  # optional - sage.combinat
    (continues on next page)
convolution_product(*maps)

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

$$(f_1 * f_2 * \cdots * f_n) := \mu^{(n-1)} \circ (f_1 \otimes f_2 \otimes \cdots \otimes f_n) \circ \Delta^{(n-1)},$$

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.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: Id = lambda x: x
sage: Antipode = lambda x: x.antipode()
sage: s = SymmetricFunctions(QQ).schur() #@
    optional - sage.combinat
sage: s[3].convolution_product(Id, Id) #@
    optional - sage.combinat
2*s[2, 1] + 4*s[3]
```

(continues on next page)
The method accepts multiple arguments, or a single argument consisting of a list of maps:

```
sage: s[3,2].convolution_product(Id, Id)  # optional - sage.combinat
```

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:

```
sage: (s[3,2].convolution_product(Antipode, Id))  # optional - sage.combinat
....: == s[3,2].convolution_product(Id, Antipode))
True
```

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()  # optional - sage.combinat
sage: Psi[2,1].convolution_product(Id, Id, Id)  # optional - sage.combinat
3*Psi[1, 2] + 6*Psi[2, 1]
```

```
sage: (Psi[5,1] - Psi[1,5]).convolution_product(Id, Id, Id)  # optional - sage.combinat
-3*Psi[1, 5] + 3*Psi[5, 1]
```

```
sage: G = SymmetricGroup(3)  # optional - sage.combinat
sage: QG = GroupAlgebra(G, QQ)  # optional - sage.combinat
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)  # optional - sage.combinat
5*[1, 2, 3] + 2*[2, 3, 1] + 2*[3, 1, 2]
```

```
sage: x.convolution_product(Id, Id, Id)  # optional - sage.combinat
4*[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + 3*[3, 2, 1]
```

```
sage: x.convolution_product([Id] * 6)  # optional - sage.combinat
9*[1, 2, 3]
```

```python
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.

**EXAMPLES:**

We construct some maps: the identity, the antipode and projection onto the homogeneous component of degree 2:

```python
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])
```

Compute the convolution product of the identity with itself and with the projection Proj2 on the Hopf algebra of non-commutative symmetric functions:

```python
sage: R = NonCommutativeSymmetricFunctions(QQ).ribbon()  #optional - sage.combinat
sage: T = R.convolution_product([Id, Id])  #optional - sage.combinat
sage: [T(R(comp)) for comp in Compositions(3)]
```

(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:

```
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()  # optional - sage.combinat
sage: T = m.convolution_product()  # optional - sage.combinat
sage: [T(m(lam))  # optional - sage.combinat
     ....:     for lam in SetPartitions(0).list() + SetPartitions(2).list()]
```

```
[m{}, 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:

```
sage: T = m.convolution_product(Proj2, Id)  # optional - sage.combinat
sage: [T(m(lam))  # optional - sage.combinat
     for lam in SetPartitions(3)]
```

```
[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, 3}, {2}}]
```

Compute the convolution product of the antipode with itself and the identity map on group algebra of the symmetric group:

```
sage: G = SymmetricGroup(3)  # optional - sage.groups sage.combinat
sage: QG = GroupAlgebra(G, QQ)  # optional - sage.groups sage.combinat
sage: x = QG.sum_of_terms(  # optional - sage.groups sage.combinat
     ....:     [(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)  # optional - sage.groups sage.combinat
sage: T(x)  # optional - sage.groups sage.combinat
2*[1, 3, 2] + [2, 1, 3] + 2*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2, 1]
4.15 Bimodules

```python
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:
    \[ r * (x * s) = (r * x) * s. \]

    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 Integer Ring]
```

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()
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
```

right_base_ring()

Return the right base ring over which elements of this category are defined.

EXAMPLES:

```python
sage: Bimodules(QQ, ZZ).right_base_ring()
Integer Ring
```

super_categories()

EXAMPLES:

```python
sage: Bimodules(QQ, ZZ).super_categories()
[Category of left modules over Rational Field, Category of right modules over Integer Ring]
```

## 4.16 Classical Crystals

### class sage.categories.classical_crystals.ClassicalCrystals(s=None)

Bases: `Category_singleton`

The category of classical crystals, that is crystals of finite Cartan type.

EXAMPLES:

```python
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
```

### 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:

```python
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]]
```

(continues on next page)
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]]]

sage: B = crystals.Tableaux(['D',3],shape=[1])
sage: [[b,b.lusztig_involution()] for b in B]

[[[1]], [[-1]], [[2]], [[-2]], [[3]], [[3]],
[[2]], [[-2]], [[2]], [[-1]], [[1]]]

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

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)
One may specify an alternate \texttt{WeylCharacterRing}:

\begin{verbatim}
 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
\end{verbatim}

It should have the same Cartan type and use the same realization of the weight lattice as \texttt{self}:

\begin{verbatim}
 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
\end{verbatim}

demazure_character\((w,f=None)\)
Return the Demazure character associated to \(w\).

\textbf{INPUT}:

\begin{itemize}
  \item \(w\) – an element of the ambient weight lattice realization of the crystal, or a reduced word, or an
    element in the associated Weyl group
\end{itemize}

\textbf{OPTIONAL}:

\begin{itemize}
  \item \(f\) – a function from the crystal to a module
\end{itemize}

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 \texttt{sage.categories.regular_crystals.RegularCrystals.ParentMethods.demazure_operator()}) to the highest weight element of the classical crystal. The simple Demazure operators \(D_i\) (see \texttt{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.

\textbf{EXAMPLES}:

\begin{verbatim}
 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 + x1^2*x3 + x1*x3^2

 sage: T = crystals.Tableaux(['A',3],shape=[2,1])
sage: T.demazure_character([1,2,3])
x1^2*x2 + x1^2*x2 + x1^2*x3 + x1^2*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 + x1^2*x2 + x1^2*x3 + x1^2*x3 + x2^2*x3
\end{verbatim}
sage: T = crystals.Tableaux(['B',2], shape = [2])
sage: e = T.weight_lattice_realization().basis()
sage: weight = -2*e[1]
sage: T.demazure_character(weight)
\[x1^2 + x1 \cdot x2 + x2^2 + x1 + x2 + x1/x2 + 1/x2 + 1/x2^2 + 1\]

sage: T = crystals.Tableaux("B2",shape=[1/2,1/2])
sage: b2=WeylCharacterRing("B2",base_ring=QQ).ambient()
sage: T.demazure_character([1,2],f=\lambda x:b2(x.weight()))
b2(-1/2,1/2) + b2(1/2,-1/2) + b2(1/2,1/2)

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:

sage: ClassicalCrystals().super_categories()
[Category of regular crystals,
 Category of finite crystals,
 Category of highest weight crystals]
4.17 Coalgebras

```python
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]

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]

    Warning: This is only correct in certain cases (finite dimension, ...). See github issue #15647.

class ElementMethods:
    Bases: object
    coproduct()
    Return the coproduct of self.
    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,b] = A.algebra_generators()
sage: a, a.coproduct()
    (B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
```

(continues on next page)
sage: b, b.coproduct()
(B[(1,3)], B[(1,3)] # B[(1,3)])

counit()
Return the counit of self.

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, b] = A.algebra_generators()
sage: a, a.counit()
(B[(1,2,3)], 1)
sage: b, b.counit()
(B[(1,3)], 1)

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: 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)])

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:
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, 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: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: f = m[2,1]
sage: f.coproduct.__module__
    'sage.categories.coalgebras'
sage: m.coproduct_on_basis
    NotImplemented
    sage: m.coproduct == m.coproduct_by_coercion
    True
    sage: f.coproduct()
    ```

    ```python
    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()
    ```

count_by_coercion(x)

    Return the counit of `x` if `counit_by_basis` is not implemented.

    EXAMPLES:

    ```python
    sage: sp = SymmetricFunctions(QQ).sp()
sage: sp.an_element()
    ```
\[ \text{sage: } sp.counit(sp.an_element()) \]
\[ 2 \]
\[ \text{sage: } o = \text{SymmetricFunctions}(\mathbb{Q}).o() \]
\[ \text{sage: } o.an_element() \]
\[ 2^*o[0] + 2^*o[1] + 3^*o[2] \]
\[ \text{sage: } o.counit(o.an_element()) \]
\[ -1 \]

\section*{4.17. Coalgebras}

\begin{verbatim}
class SubcategoryMethods
    Bases: object

    Cocommutative()
        Return the full subcategory of the cocommutative objects of self.
        A coalgebra \( C \) is said to be \textit{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:
        \[ \text{sage: } C1 = \text{Coalgebras}(\mathbb{Z}).\text{Cocommutative()}.\text{WithBasis()}; C1 \]
        Category of cocommutative coalgebras with basis over Integer Ring
        \[ \text{sage: } C2 = \text{Coalgebras}(\mathbb{Z}).\text{WithBasis()}.\text{Cocommutative()} \]
        \[ \text{sage: } C1 \text{ is } C2 \]
        True
        \[ \text{sage: } \text{BialgebrasWithBasis}(\mathbb{Q}).\text{Cocommutative()} \]
        Category of cocommutative bialgebras with basis over Rational Field
\end{verbatim}

\begin{verbatim}
class Super(base_category)
    Bases: SuperModulesCategory

    class SubcategoryMethods
        Bases: object

        Supercocommutative()
            Return the full subcategory of the supercocommutative objects of self.

            EXAMPLES:
            \[ \text{sage: } \text{Coalgebras}(\mathbb{Z}).\text{WithBasis()}.\text{Super()}.\text{Supercocommutative()} \]
            Category of supercocommutative super coalgebras with basis over Integer Ring
            \[ \text{sage: } \text{BialgebrasWithBasis}(\mathbb{Q}).\text{Super()}.\text{Supercocommutative()} \]
            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
\end{verbatim}

\begin{verbatim}
class Supercocommutative(base_category)
    Bases: CategoryWithAxiom_over_base_ring

\end{verbatim}
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]
```

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]
```

The category of bialgebras does not occur in these results, since super bialgebras are not bialgebras.

class TensorProducts(category, *args)

Bases: TensorProductsCategory

class ElementMethods

Bases: object

class ParentMethods

Bases: object

eextra_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

WithBasis

alias of CoalgebrasWithBasis

class WithRealizations(category, *args)

Bases: WithRealizationsCategory

class ParentMethods

Bases: object

coproduct(x)

    Return the coproduct of x.

    EXAMPLES:
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: S = N.complete()
sage: N.coproduct.__module__
'sage.categories.coalgebras'
sage: N.coproduct(S[2])

\textit{counit}(x)

Return the counit of \(x\).

EXAMPLES:

sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: f = s[2,1]
sage: f.counit.__module__
'sage.categories.coalgebras'
sage: f.counit()
0
sage: N = NonCommutativeSymmetricFunctions(QQ)
sage: N.counit.__module__
'sage.categories.coalgebras'
sage: N.counit(N.one())
1
sage: x = N.an_element(); x
sage: N.counit(x)
2

\textit{super_categories}()

EXAMPLES:

sage: Coalgebras(QQ).super_categories()
[Category of vector spaces over Rational Field]

4.18 Coalgebras with basis

class \texttt{sage.categories.coalgebras\_with\_basis.CoalgebrasWithBasis(base\_category)}

Bases: \texttt{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]
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()
Psi[2,2].coproduct_iterated(0)
Psi[2,2].coproduct_iterated(2)
```

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
A = HopfAlgebrasWithBasis(QQ).example(); A
a, b = A.algebra_generators()
a, A.coproduct(a)
b, A.coproduct(b)
```

coproduct_on_basis()
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:

```python
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._group.gens()
```

`counit()`

If `counit_on_basis()` is available, construct the counit morphism from `self` to `self ⊗ self` by extending it by linearity.

EXAMPLES:

```python
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()
```

`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:

```python
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._group.gens()
```
class Super(base_category)
    Bases: SuperModulesCategory
    extra_super_categories()
    EXAMPLES:
    sage: C = Coalgebras(ZZ).WithBasis().Super()
    sage: sorted(C.super_categories(), key=str)  # indirect doctest
    [Category of graded coalgebras with basis over Integer Ring,
     Category of super coalgebras over Integer Ring,
     Category of super modules with basis over Integer Ring]

4.19 Commutative additive groups

class sage.categories.commutative_additive_groups.CommutativeAdditiveGroups(base_category)
    Bases: CategoryWithAxiom_singleton, AbelianCategory
    The category of abelian groups, i.e. additive abelian monoids where each element has an inverse.
    EXAMPLES:
    sage: C = CommutativeAdditiveGroups(); C
    Category of commutative additive groups
    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:
4.20 Commutative additive monoids

class sage.categories.commutative_additive_monoids.CombinativeAdditiveMonoids(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().
    AdditiveUnital()
True

Note: This category is currently empty and only serves as a place holder to make C.example() work.
4.21 Commutative additive semigroups

class sage.categories.commutative_additive_semigroups.CommutativeAdditiveSemigroups(base_category):
    Bases: CategoryWithAxiom_singleton

    The category of additive abelian semigroups, i.e. sets with an associative and abelian operation +.

    EXAMPLES:

    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: sorted(C.axioms())
    ['AdditiveAssociative', 'AdditiveCommutative']
    sage: C is AdditiveMagmas().AdditiveAssociative().AdditiveCommutative()
    True

    Note: This category is currently empty and only serves as a place holder 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
    algebra()
    EXAMPLES:

    sage: 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]
4.23 Commutative algebras

```python
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]

    This is just a shortcut for:
    sage: Algebras(QQ).Commutative()
    Category of commutative algebras over Rational Field

Todo:
    • product (= Cartesian product)
    • coproduct (= tensor product over base ring)
```

```python
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]
```

4.24 Commutative ring ideals

```python
class sage.categories.commutative_ring_ideals.CommutativeRingIdeals(R):
    Bases: Category_ideal
    The category of ideals in a fixed commutative ring.

    EXAMPLES:
```
4.25 Commutative rings

class sage.categories.commutative_rings.CommutativeRings(base_category)
    Bases: CategoryWithAxiom_singleton
    The category of commutative rings
    commutative rings with unity, i.e. rings with commutative \( \ast \) and a multiplicative identity
    EXAMPLES:

    sage: C = CommutativeRings(); C
    Category of commutative rings
    sage: C.super_categories()
    [Category of rings, Category of commutative monoids]

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), #
        # optional - sage.rings.finite_rings....
        QQ, GF(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:
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 $z^s$ 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 lists 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:**

```python
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:

```python
sage: K = GF(25, 'z')  #
    optional - sage.rings.finite_rings
sage: a = K.multiplicative_generator()  #
    optional - sage.rings.finite_rings
sage: K.cyclotomic_cosets(a**2, cosets=[1])  #
    optional - sage.rings.finite_rings
[[1, 2, 3, 4, z + 1, z + 3, 2*z + 1, 2*z + 2, 3*z + 3],
  2*z + 1, 2*z + 2, 3*z + 3],
```

(continues on next page)
\[3z + 4, 4z + 2, 4z + 4\]

We compute some examples of minimal polynomials:

\[
\text{sage: } K = GF(27, 'z')
\]

\[\text{optional - sage.rings.finite_rings}\]

\[
\text{sage: a = K.multiplicative_generator()}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[
\text{sage: R.<X> = PolynomialRing(K, 'X')}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[
\text{sage: a.minimal_polynomial('X')}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[X^3 + 2X + 1\]

\[
\text{sage: cyc3 = Zmod(26).cyclotomic_cosets(3, cosets=[1]); cyc3}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[\text{[[1, 3, 9]]}\]

\[
\text{sage: prod(X - a**i for i in cyc3[0])}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[X^3 + 2X + 1\]

\[
\text{sage: (a**7).minimal_polynomial('X')}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[X^3 + X^2 + 2X + 1\]

\[
\text{sage: cyc7 = Zmod(26).cyclotomic_cosets(3, cosets=[7]); cyc7}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[\text{[[7, 11, 21]]}\]

\[
\text{sage: prod(X - a**i for i in cyc7[0])}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[X^3 + X^2 + 2X + 1\]

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:

\[
\text{sage: } K = GF(5)
\]

\[\text{optional - sage.rings.finite_rings}\]

\[
\text{sage: a = K.multiplicative_generator()}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[
\text{sage: H = K.cyclotomic_cosets(a**2, cosets=[1, 2]); H}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[\text{[[1, 4], [2, 3]]}\]

\[
\text{sage: sorted(x - y for D in H for x in D for y in D if x != y)}
\]

\[\text{optional - sage.rings.finite_rings}\]

\[\text{[1, 2, 3, 4]}\]

\[
\text{sage: K = GF(37)}
\]

\[\text{optional - sage.rings.finite_rings}\]
The method `cyclotomic_cosets` works on any finite commutative ring:

```python
sage: R = cartesian_product([GF(7), Zmod(14)])
# optional - sage.rings.finite_rings
sage: a = R((3,5))
# optional - sage.rings.finite_rings
sage: R.cyclotomic_cosets((3,5), [(1,1)])
# optional - sage.rings.finite_rings
[[[1, 1), (2, 11), (3, 5), (4, 9), (5, 3), (6, 13)]]
```

```python
class ParentMethods
Bases: object

over(base=None, gen=None, gens=None, name=None, names=None)
Return this ring, considered as an extension of base.

INPUT:
• base – a commutative ring or a morphism or None (default: None); the base of this extension or its defining morphism
• gen – a generator of this extension (over its base) or None (default: 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: F = GF(5^2)
# optional - sage.rings.finite_rings
sage: k = GF(5^4)
# optional - sage.rings.finite_rings
sage: z4 = k.gen()
# optional - sage.rings.finite_rings
sage: K = k.over(F)
# optional - sage.rings.finite_rings
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()
# optional - sage.rings.finite_rings
```
However, it is possible to specify another generator and/or another name. For example:

```python
sage: Ka = k.over(F, name='a')  # optional - sage.rings.finite_rings
Field in a with defining polynomial
x^2 + (4*z2 + 3)*x + z2 over its base
sage: Ka.gen()  # optional - sage.rings.finite_rings
a
sage: Ka(z4)  # optional - sage.rings.finite_rings
a

sage: Kb = k.over(F, gen=-z4+1, name='b')  # optional - sage.rings.finite_rings
Field in b with defining polynomial
x^2 + z2*x + 4 over its base
sage: Kb.gen()  # optional - sage.rings.finite_rings
b
sage: Kb(-z4+1)  # optional - sage.rings.finite_rings
b
```

Note that the shortcut `K.<a>` is also available:

```python
sage: KKa.<a> = k.over(F)  # optional - sage.rings.finite_rings
sage: KKa is Ka  # optional - sage.rings.finite_rings
True
```

Building an extension on top of another extension is allowed:

```python
sage: L = GF(5^12).over(K)  # optional - sage.rings.finite_rings
Field in z12 with defining polynomial
x^3 + (1 + (4*z2 + 2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base
sage: L.base_ring()  # optional - sage.rings.finite_rings
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()                                    # optional - sage.rings.finite_rings
[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()                               # optional - sage.modules
sage: E                                             # optional - sage.modules
Univariate Polynomial Ring in x over Rational Field over its base
sage: E.base_ring()                                # optional - sage.modules
Rational Field
```

Here is an example where `base` is a defining morphism:

```python
sage: k.<a> = QQ.extension(x^2 - 2)                 # optional - sage.rings.number_field
sage: l.<b> = QQ.extension(x^4 - 2)                 # optional - sage.rings.number_field
sage: f = k.hom([b^2])                              # optional - sage.rings.number_field
sage: L = l.over(f)                                 # optional - sage.rings.number_field
sage: L                                             # optional - sage.rings.number_field
Field in b with defining polynomial x^2 - a over its base
sage: L.base_ring()                                # optional - sage.rings.number_field
Number Field in a with defining polynomial x^2 - 2
```

Similarly, one can create a tower of extensions:

```python
sage: K = k.over()                                 # optional - sage.rings.number_field
sage: L = l.over(Hom(K, l)(f))                     # optional - sage.rings.number_field
sage: L                                             # optional - sage.rings.number_field
Field in b with defining polynomial x^2 - a over its base
sage: L.base_ring()                                # optional - sage.rings.number_field
Field in a with defining polynomial x^2 - 2 over its base
sage: L.bases()                                    # optional - sage.rings.number_field
```

(continues on next page)
4.26 Complete Discrete Valuation Rings (CDVR) and Fields (CDVF)

class sage.categories.complete_discrete_valuation.CompleteDiscreteValuationFields(s=None)
    Bases: Category_singleton
    The category of complete discrete valuation fields
    EXAMPLES:

    sage: Zp(7) in CompleteDiscreteValuationFields()  # optional - sage.rings.padics
    False
    sage: QQ in CompleteDiscreteValuationFields()     # optional - sage.rings.padics
    False
    sage: LaurentSeriesRing(QQ, 'u') in CompleteDiscreteValuationFields()  # optional - sage.rings.padics
    True
    sage: Qp(7) in CompleteDiscreteValuationFields()   # optional - sage.rings.padics
    True
    sage: TestSuite(CompleteDiscreteValuationFields()).run()

class ElementMethods
    Bases: object
    
    denominator()
    
    Return the denominator of this element normalized as a power of the uniformizer
    
    EXAMPLES:

    sage: K = Qp(7)               # optional - sage.rings.padics
    sage: x = K(1/21)            # optional - sage.rings.padics
    sage: x.denominator()       # optional - sage.rings.padics
    7 + O(7^21)
    sage: x = K(7)              # optional - sage.rings.padics
    sage: x.denominator()       # optional - sage.rings.padics
    1 + O(7^20)

    Note that the denominator lives in the ring of integers:

    sage: x.denominator().parent() # optional - 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 $p^{(n)} - n$ otherwise:

```
sage: x = K(0, 5); x  #← optional - sage.rings.padics
0(7^5)
sage: x.denominator()  #←
1 + 0(7^20)
sage: x = K(0, -5); x  #←
0(7^5)
sage: x.denominator()  #←
7^5 + 0(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: K = Qp(7, 5)  #←
← optional - sage.rings.padics
sage: x = K(1/21)  #←
← optional - sage.rings.padics
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()  #←
← optional - sage.rings.padics
7-adic Ring with capped relative precision 5
```

**valuation()**

Return the valuation of this element.

**EXAMPLES:**

```
sage: K = Qp(7)  #←
← optional - sage.rings.padics
sage: x = K(7); x  #←
← optional - sage.rings.padics
sage: x.valuation()  #←
1
```

4.26. Complete Discrete Valuation Rings (CDVR) and Fields (CDVF) 235
super_categories()

EXAMPLES:

sage: CompleteDiscreteValuationFields().super_categories()
[Category of discrete valuation fields]

class sage.categories.complete_discrete_valuation.CompleteDiscreteValuationRings(s=None)

Bases: Category_singleton

The category of complete discrete valuation rings

EXAMPLES:

sage: Zp(7) in CompleteDiscreteValuationRings()    #optional - sage.rings.padics
True
sage: QQ in CompleteDiscreteValuationRings()       #optional - sage.rings.padics
False
sage: QQ[['u']] in CompleteDiscreteValuationRings() #optional - sage.rings.padics
True
sage: Qp(7) in CompleteDiscreteValuationRings()     #optional - 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: K = Qp(7) #optional - sage.rings.padics
sage: x = K(1/21) #optional - sage.rings.padics
sage: x.denominator() #optional - sage.rings.padics
7 + O(7^21)

sage: x = K(7) #optional - sage.rings.padics
sage: x.denominator() #optional - sage.rings.padics
1 + O(7^20)

Note that the denominator lives in the ring of integers:

sage: x.denominator().parent() #optional - 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:
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: R = ZpCA(17)  # optional - sage.rings.padics
sage: R(-1, 2).lift_to_precision(10)  # optional - sage.rings.padics
16 + 16*17 + O(17^10)  # optional - sage.rings.padics
sage: R(1, 15).lift_to_precision(30)  # optional - sage.rings.padics
Traceback (most recent call last):
... PrecisionError: precision higher than allowed by the precision cap
sage: (R(-1, 2).lift_to_precision().precision_absolute())  # optional - sage.rings.padics
...:  == R.precision_cap()
True

sage: R = Zp(5); c = R(17, 3); c.lift_to_precision(8)  # optional - sage.rings.padics
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()  # optional - sage.rings.padics
True
Return the numerator of this element, normalized in such a way that \( x = \frac{x.numerator}{x.denominator} \) always holds true.

**EXAMPLES:**

```python
sage: K = Qp(7, 5)  # optional - sage.rings.padics
sage: x = K(1/21)  # optional - sage.rings.padics
sage: x.numerator()  # optional - sage.rings.padics
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + O(7^5)

sage: x == x.numerator() / x.denominator()  # optional - sage.rings.padics
True
```

Note that the numerator lives in the ring of integers:

```python
sage: x.numerator().parent()  # optional - sage.rings.padics
7-adic Ring with capped relative precision 5
```

**valuation()**

Return the valuation of this element.

**EXAMPLES:**

```python
sage: R = Zp(7)  # optional - sage.rings.padics
sage: x = R(7); x  # optional - sage.rings.padics
7 + O(7^21)

sage: x.valuation()  # optional - sage.rings.padics
1
```

**super_categories()**

**EXAMPLES:**

```python
sage: CompleteDiscreteValuationRings().super_categories()
[Category of discrete valuation rings]
```

## 4.27 Complex reflection groups

class sage.categories.complex_reflection_groups.ComplexReflectionGroups(s=\texttt{None})

**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 an 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:**

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups()
Category of complex reflection groups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]
```

An example of a reflection group:

```python
sage: W = ComplexReflectionGroups().example(); W
5-colored permutations of size 3

W is in the category of complex reflection groups:

```python
sage: W in ComplexReflectionGroups()       #optional - sage.combinat sage.groups
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 faithful reflection representation of `self`.

**EXAMPLES:**

```python
4.27. Complex reflection groups
```
The symmetric group on \{0, ..., 3\}

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().example()
```

Return an example of a complex reflection group.

EXAMPLES:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().example() #optional - sage.combinat sage.groups
5-colored permutations of size 3
```

Return the super categories of `self`.

EXAMPLES:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]
```

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 declara-
tions.

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:

- \texttt{ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods.index\_set()}
- \texttt{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:

- \texttt{ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply\_simple\_reflection()}
- \texttt{ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply\_simple\_reflection\_left()}
- \texttt{ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply\_simple\_reflection\_right()}
- \texttt{ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods._mul_()}

It’s recommended to implement either \_mul\_ or both apply\_simple\_reflection\_left and
apply\_simple\_reflection\_right.

See also:

- \texttt{complex\_reflection\_groups.ComplexReflectionGroups}
- \texttt{generalized\_coxeter\_groups.GeneralizedCoxeterGroups}

EXAMPLES:

```python
sage: from sage.categories.complex_reflection_or_generalized_coxeter_groups import...
sage: C = ComplexReflectionOrGeneralizedCoxeterGroups(); C
Category of complex reflection or generalized coxeter groups
sage: C.super\_categories()
[Category of finitely generated enumerated groups]

sage: C.required\_methods()
{'element': {'optional': ['reflection\_length'],
'required': []},
```
'parent': {'optional': ['distinguished_reflection', 'hyperplane_index_set', 'irreducible_components', 'reflection', 'reflection_index_set'],
            'required': ['__contains__', 'index_set']}

class ElementMethods
    Bases: object

    apply_conjugation_by_simple_reflection(i)
    Conjugate self by the i-th simple reflection.

    EXAMPLES:

    sage: W = WeylGroup(['A',3])
    sage: w = W.from_reduced_word([3,1,2,1])
    sage: w.apply_conjugation_by_simple_reflection(1).reduced_word()
    [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 – (optional, default: 'all'): either 'simple', 'distinguished', or 'all'

    EXAMPLES:

    sage: W = ReflectionGroup((1,1,3))  # optional - gap3
    sage: W.one().apply_reflections([1])  # optional - gap3
    (1,4)(2,3)(5,6)
    sage: W.one().apply_reflections([2])  # optional - gap3
    (1,3)(2,5)(4,6)
    sage: W.one().apply_reflections([2,1])  # optional - gap3
    (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: W = WeylGroup("A3", prefix='s')
    sage: w = W.an_element(); w
    s_1^*s_2^*s_3
    sage: AS = W.domain()
    sage: r1 = AS.roots()[4]
    sage: r1
(0, 1, 0, -1)
sage: r2 = AS.roots()[5]
sage: r2
(0, 0, 1, -1)
sage: w.apply_reflections([r1, r2], word_type='all')
s1

sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: W.one().apply_reflections([1], word_type='distinguished')  #
#optional - gap3
(1,4)(2,3)(5,6)
sage: W.one().apply_reflections([2], word_type='distinguished')  #
#optional - gap3
(1,3)(2,5)(4,6)
sage: W.one().apply_reflections([3], word_type='distinguished')  #
#optional - gap3
(1,5)(2,4)(3,6)
sage: W.one().apply_reflections([2,1], word_type='distinguished')  #
#optional - gap3
(1,2,6)(3,4,5)

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')

(continues on next page)
apply_simple_reflection\(_{\text{left}}\)(\(i\))

Return 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)
```

EXAMPLES:

```python
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)

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_right(1)
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection_right(2)
[[1, 0, 0], [2, 1, 3]]
sage: w.apply_simple_reflection_right(3)
[[2, 0, 0], [3, 1, 2]]

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:
### is_reflection()

Return whether `self` is a reflection.

**EXAMPLES:**

```python
sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: [t.is_reflection() for t in W.reflections()]  # optional - gap3
[True, True, True, True, True]

sage: len([t for t in W.reflections() if t.is_reflection()])  # optional - gap3
6
```

```python
sage: W = ReflectionGroup((2,1,3))  # optional - gap3
sage: [t.is_reflection() for t in W.reflections()]  # optional - gap3
[True, True, True, True, True, True, True, True, True]

sage: len([t for t in W.reflections() if t.is_reflection()])  # optional - gap3
9
```

### reflection_length()

Return the reflection length of `self`.

This is the minimal length of a factorization of `self` into reflections.

**EXAMPLES:**

```python
sage: W = ReflectionGroup((1,1,2))  # optional - gap3
sage: sorted([t.reflection_length() for t in W])  # optional - gap3
[0, 1]

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]

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, 2, 2, 2, 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]
```

### class Irreducible(base_category)

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()
[4-colored permutations of size 3]

class ParentMethods
    Bases: object

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' →ḡ,'f')) # optional - gap3
    sage: for i in W.hyperplane_index_set(): # optional - gap3
        ....: 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:
    
    sage: W = ReflectionGroup((1,1,3)) # optional - gap3
    sage: distinguished_reflections = W.distinguished_reflections() # optional - gap3
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 – (optional, default: 'simple'): either 'simple', 'distinguished', or 'all'

If word is \( [i_1, i_2, \ldots, i_k] \), then this returns the corresponding product of simple reflections \( s_{i_1} s_{i_2} \cdots s_{i_k} \).
If \texttt{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:**

- \texttt{index_set()}
- \texttt{reflection_index_set()}
- \texttt{hyperplane_index_set()}
- \texttt{apply_simple_reflections()}
- \texttt{reduced_word()}
- \texttt{_test_reduced_word()}

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: 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 \texttt{word_type} for the colored symmetric group:

```python
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: W = WeylGroup("A3", prefix='s')
sage: AS = W.domain()
sage: r1 = AS.roots()[4]
sage: r1
(0, 1, 0, -1)
sage: r2 = AS.roots()[5]
sage: r2
(0, 0, 1, -1)
sage: W.from_reduced_word([r1, r2], word_type='all')
s3*s2
sage: W = WeylGroup("G2", prefix='s')
sage: W.from_reduced_word(W.domain().positive_roots(), word_type='all')
s1*s2
```

```python
sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: W.from_reduced_word([1,2,3], word_type='all').reduced_word()  #_.
```

(continues on next page)
optional - gap3
[1, 2, 3]

sage: W.from_reduced_word([1,2,3], word_type='all').reduced_word_in_reflections()  # optional - gap3
[1, 2, 3]

sage: W.from_reduced_word([1,2,3]).reduced_word_in_reflections()  #
optional - gap3
[1, 2, 3]

**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:**

```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)
sage: W.group_generators()
Finite family {1: [[0, 0],
  [2, 1]],
  2: [[0, 1],
  [1, 2]]}
```

The simple reflections are also semigroup generators, even for an infinite group:

```python
sage: W = WeylGroup(['A',2,1])
sage: W.semigroup_generators()
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: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: W.hyperplane_index_set()      # optional - gap3
(1, 2, 3, 4, 5, 6)
```

```python
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=[1,3,'asdf',7,9,11])  # optional - gap3
sage: W.hyperplane_index_set()                                                # optional - gap3
(1, 3, 'asdf', 7, 9, 11)
```

```python
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=('a','b','c','d','e','f'))  # optional - gap3
sage: W.hyperplane_index_set()                                                # optional - gap3
('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:**

```python
sage: W = CoxeterGroups().Finite().example(); W
The 5-th dihedral group of order 10
sage: W.index_set()
(1, 2)
```

```python
sage: W = ColoredPermutations(1, 4)
(1, 2, 3)
```

```python
sage: W = ReflectionGroup((1,1,4), index_set=('a','b','c','d','e','f'))  # optional - gap3
sage: W.index_set()                                                      # optional - gap3
('a', 'b', 'c', 'd', 'e', 'f')
```

**irreducible_component_index_sets()**

Return a list containing the index sets of the irreducible components of `self` as finite reflection groups.

**EXAMPLES:**

```python
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
```

(continues on next page)
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.

**is_irreducible()**
Return True if \(self\) is irreducible.

**is_reducible()**
Return True if \(self\) is not irreducible.

**number_of_irreducible_components()**
Return the number of irreducible components of \(self\).
sage: SymmetricGroup(3).number_of_irreducible_components()
1
sage: ColoredPermutations(1,3).number_of_irreducible_components()
1
sage: ReflectionGroup((1,1,3),(2,1,3)).number_of_irreducible_components()
# optional - gap3
2

number_of_simple_reflections()

Return the number of simple reflections of self.

EXAMPLES:

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, \ldots, \(N\), this gives the \(i\)-th reflection of self.

See also:

• reflections_index_set()
• reflections()

EXAMPLES:

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: for i in W.reflection_index_set():
    print(\('%s\ %s\'(i, W.reflection(i)))
....
1 (1,7)(2,4)(5,6)(8,10)(11,12)
2 (1,4)(2,8)(3,5)(7,10)(9,11)
3 (2,5)(3,9)(4,6)(8,11)(10,12)
4 (1,8)(2,7)(3,6)(4,10)(9,12)
5 (1,6)(2,9)(3,8)(5,11)(7,12)
6 (1,11)(3,10)(4,9)(5,7)(6,12)

reflection_index_set()

Return the index set of the reflections of self.

See also:

• reflection()
• reflections()
EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: W.reflection_index_set()  # optional - gap3
(1, 2, 3, 4, 5, 6)
sage: W = ReflectionGroup((1,1,4), reflection_index_set=[1,3,'asdf',7,9,
˓→11])  # optional - gap3
sage: W.reflection_index_set()  # optional - gap3
(1, 3, 'asdf', 7, 9, 11)
sage: W = ReflectionGroup((1,1,4), reflection_index_set=('a','b','c','d','e
˓→','f'))  # optional - gap3
sage: 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: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):  # optional - gap3
    ....:     print('%s %s' % (index, 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),reflection_index_set=['a','b','c'])  #˓→optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):  # optional - gap3
    ....:     print('%s %s' % (index, 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))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):  # optional - gap3
    ....:     print('%s %s' % (index, reflections[index]))
1 (1,2,3)
2 (1,3,2)

sage: W = ReflectionGroup((1,1,3), (3,1,2))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):  # optional - gap3
    ....:     print('%s %s' % (index, 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)
```

(continues on next page)
semigroupGenerators()

Return the simple reflections of self, as distinguished group generators.

See also:

• simple_reflections()
• Groups.ParentMethods.group_generators()
• Semigroups.ParentMethods.semigroup_generators()

EXAMPLES:

```
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)
sage: W.group_generators()
Finite family {1: [[0, 0],
    [2, 1]],
  2: [[0, 1],
    [1, 2]]}
```

The simple reflections are also semigroup generators, even for an infinite group:

```
sage: W = WeylGroup(['A',2,1])
sage: W.semigroup_generators()
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]}
```

simple_reflection(i)

Return the i-th simple reflection si of self.

INPUT:

• i – an element from the index set

See also:
• `index_set()`
  • `simple_reflections()`

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: 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)
```

```python
sage: W = ReflectionGroup((1,1,4), index_set=[1,3,'asdf'])  # optional - gap3
sage: for i in W.index_set():
    print('%s %s' % (i, W.simple_reflection(i)))  # optional - gap3
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:

```python
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, 2]
sage: SymmetricGroup(5).simple_reflection_orders()
[2, 2, 2, 2]
sage: C = ColoredPermutations(4, 3)
sage: C.simple_reflection_orders()
[2, 2, 4]
```

`simple_reflections()`

Return the simple reflections \((s_i)_{i \in I}\) of \(\text{self}\) as a family indexed by `index_set()`.

See also:

• `simple_reflection()`
  • `index_set()`

EXAMPLES:

For the symmetric group, we recognize the simple transpositions:

```python
sage: W = SymmetricGroup(4); W
Symmetric group of order 4! as a permutation group
sage: s = W.simple_reflections()
sage: s
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}
```
Here are the simple reflections for a colored symmetric group and a reflection group:

```
sage: W = ColoredPermutations(1,3)
sage: W.simple_reflections()
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()`.

### some_elements()

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:**

```
sage: W = WeylGroup(['A',3])
sage: W.some_elements()
[[0 1 0 0] [1 0 0 0] [1 0 0 0] [1 0 0 0] [0 0 0 1]
 [1 0 0 0] [0 0 1 0] [0 1 0 0] [0 1 0 0] [1 0 0 0]
 [0 0 1 0] [0 1 0 0] [0 0 0 1] [0 0 1 0] [0 1 0 0]
 [0 0 0 1], [0 0 0 1], [0 0 1 0], [0 0 0 1], [0 0 1 0]]
sage: W = ColoredPermutations(1,4)
sage: W.some_elements()
[[[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]]]
```

### 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

super_categories()

Return the super categories of self.

EXAMPLES:

sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]

4.29 Coxeter Group Algebras

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 \( H_{q_1, q_2}(W) \) 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:

    sage: W = WeylGroup(['B',2])
sage: W.element_class._repr_ = lambda x: str(x.reduced_word())
sage: K = QQ['q1,q2'].fraction_field()
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: E = KW.demazure_lusztig_eigenvectors(q1,q2)
sage: E.keys()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
sage: w = W.an_element()
sage: E[w]
(q2/(-q1+q2))*2121 + ((-q2)/(-q1+q2))*121 - 212 + 12

\textit{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
- \( q1, q2 \) – two elements of the ground ring
- \( \text{bar} \) – a boolean (default \( False \))

See \textit{demazure_lusztig_operators()} for details.

**EXAMPLES:**

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: w = W.an_element()
sage: KW.demazure_lusztig_operator_on_basis(w, 0, q1, q2)
(-q2)*323123 + (q1+q2)*123
sage: KW.demazure_lusztig_operator_on_basis(w, 1, q1, q2)
q1*1231
sage: KW.demazure_lusztig_operator_on_basis(w, 2, q1, q2)
q1*1232
sage: KW.demazure_lusztig_operator_on_basis(w, 3, q1, q2)
(q1+q2)*123 + (-q2)*12

At \( q_1 = 1 \) and \( q_2 = 0 \) we recover the action of the isobaric divided differences \( \pi_i \):

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 \( s_i \):

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(q1, q2, side='right', affine=True)

Return the Demazure Lusztig operators acting on self.

**INPUT:**
- q1, q2 – two elements of the ground ring $K$
- side – "left" or "right" (default: "right"); which side to act upon
- 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 `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 `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.

**EXAMPLES:**

```python
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()
```

**Note:** For a finite Weyl group $W$, the level 0 action of the affine Weyl group $\tilde{W}$ 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(s=None)

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)}
```

Here are some further examples:

```python
sage: FiniteCoxeterGroups().example()  # optional - sage.combinat sage.groups
The 5-th dihedral group of order 10

sage: FiniteWeylGroups().example()  # optional - sage.combinat sage.groups
The symmetric group on {0, ..., 3}

sage: WeylGroup(["B", 3])  # optional - sage.combinat sage.groups
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)

sage: S4 = SymmetricGroup(4); S4  # optional - sage.groups
Symmetric group of order 4! as a permutation group

sage: S4 in CoxeterGroups().Finite()  # optional - sage.groups
True
```

Those will eventually be also in this category:

```python
sage: DihedralGroup(5)  # optional - sage.groups
Dihedral group of order 10 as a permutation group
```

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:

```
sage: CoxeterGroups().is_full_subcategory(Groups())
False
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: CoxeterGroups().is_full_subcategory(GeneralizedCoxeterGroups())
True
```

Algebras
 alias of CoxeterGroupAlgebras

class ElementMethods
 Bases: object

absolute_covers()

Refurn the list of covers of self in absolute order.

See also:
absolute_length()

EXAMPLES:

```
sage: W = WeylGroup("A", 3)
# optional - sage.combinat sage.groups
sage: s = W.simple_reflections()
# optional - sage.combinat sage.groups
sage: w0 = s[1]
# optional - sage.combinat sage.groups
sage: w1 = s[1]*s[2]*s[3]
# optional - sage.combinat sage.groups
sage: w0.absolute_covers()
# optional - sage.combinat sage.groups
[ [0 0 1] [0 1 0] [0 1 0] [0 0 0 1] [0 1 0 0] [0 1 0],
 [1 0 0] [1 0 0] [0 0 1 0] [1 0 0 0] [0 0 0 1] [0 0 1 0]
 [0 1 0] [0 0 0 1] [1 0 0 0] [0 0 1 0] [0 0 0 1] [0 0 1 0]
 [0 0 0 1], [0 0 1 0], [0 0 0 1], [0 0 0 1], [0 1 0], [1 0 0 0]
]```

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:

```sage```
absolute_length()
```sage```

EXAMPLES:

```sage```
sage: W = WeylGroup("A", 3)  #
    # optional - sage.combinat sage.groups
sage: s = W.simple_reflections()  #
    # optional - sage.combinat sage.groups
sage: w0 = s[1]  #
    # optional - sage.combinat sage.groups
sage: w1 = s[1]*s[2]*s[3]  #
    # optional - sage.combinat sage.groups
sage: w0.absolute_le(w1)  #
    # optional - sage.combinat sage.groups
True
sage: w1.absolute_le(w0)  #
    # optional - sage.combinat sage.groups
False
sage: w1.absolute_le(w1)  #
    # optional - sage.combinat sage.groups
True
```sage```

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 permutations in the symmetric groups, the absolute length is the size minus the number of its disjoint cycles.

See also:

```sage```
absolute_le()
```sage```

EXAMPLES:

```sage```
sage: W = WeylGroup("A", 3)  #
    # optional - sage.combinat sage.groups
sage: s = W.simple_reflections()  #
    # optional - sage.combinat sage.groups
sage: (s[1]*s[2]*s[3]).absolute_length()  #
    # optional - sage.combinat sage.groups
3
sage: W = SymmetricGroup(4)  #
    # optional - sage.combinat sage.groups
sage: s = W.simple_reflections()  #
    # optional - sage.combinat sage.groups
sage: (s[3]*s[2]*s[1]).absolute_length()  #
    # optional - sage.combinat sage.groups
3
```sage```

apply_demazure_product(element, side='right', length_increasing=True)

Return the Demazure or 0-Hecke product of self with another Coxeter group element.

INPUT:

- **element** – either an element of the same Coxeter group as `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 `self` on which the element should be applied. If `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: W = WeylGroup(['C', 4], prefix="s")
#optional - sage.combinat sage.groups
sage: v = W.from_reduced_word([1,2,3,4,3,1])
#optional - sage.combinat sage.groups
sage: v.apply_demazure_product([1,3,4,3,3])
#optional - sage.combinat sage.groups
s3*s1*s2*s3*s4*s3*s1
sage: v.apply_demazure_product([1,3,4,3], side='left')
#optional - sage.combinat sage.groups
s3*s4*s1*s2*s3*s4*s2*s3*s1
sage: v.apply_demazure_product((1,3,4,3), side='left')
#optional - sage.combinat sage.groups
s3*s4*s1*s2*s3*s4*s2*s3*s1
sage: v.apply_demazure_product(v)
#optional - sage.combinat sage.groups
s2*s3*s4*s1*s2*s3*s4*s2*s3*s2*s1
```

**apply_simple_projection**(i, side='right', length_increasing=True)

Return the result of the application of the simple projection \( \pi_i \) (resp. \( \pi_i \)) on `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 `CoxeterGroups.ParentMethods.simple_projections()` for the definition of the simple projections.

EXAMPLES:

```python
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: W = WeylGroup(['C', 4], prefix="s")
#optional - sage.combinat sage.groups
sage: v = W.from_reduced_word([1,2,3,4,3,1])
#optional - sage.combinat sage.groups
sage: v
#optional - sage.combinat sage.groups
(continues on next page)```
optional - sage.combinat sage.groups
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(2) #...
optional - sage.combinat sage.groups
s1*s2*s3*s4*s3*s1*s2
sage: v.apply_simple_projection(2, side='left') #...
optional - sage.combinat sage.groups
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(1, length_increasing=False) #...
optional - sage.combinat sage.groups
s1*s2*s3*s4*s3

binary_factorizations(predicate=The constant function (...) -> True)

Return the set of all the factorizations \( self = uv \) such that \( l(self) = l(u) + l(v) \).

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 \( self \).

One can pass as optional argument a predicate \( p \) such that \( p(u) \implies p(u') \) for any \( u \) left factor of \( self \) and \( u' \) left factor of \( u \). Then this returns only the factorizations \( self = uv \) such \( p(u) \) holds.

EXAMPLES:

We construct the set of all factorizations of the maximal element of the group:

```
sage: W = WeylGroup(['A', 3])
optional - sage.combinat sage.groups
sage: s = W.simple_reflections()
optional - sage.combinat sage.groups
sage: w0 = W.from_reduced_word([1,2,3,1,2,1])
optional - sage.combinat sage.groups
sage: w0.binary_factorizations().cardinality() #...
24
```

The same number of factorizations, by bounded length:

```
sage: [w0.binary_factorizations(
optional - sage.combinat sage.groups
..... lambda u: u.length() <= l
..... ).cardinality() #...
..... for l in [-1,0,1,2,3,4,5,6]]
[0, 1, 4, 9, 15, 20, 23, 24]
```

The number of factorizations of the elements just below the maximal element:

```
sage: [(s[i]*w0).binary_factorizations().cardinality() #...
optional - sage.combinat sage.groups
..... for i in [1,2,3]]
[12, 12, 12]
sage: w0.binary_factorizations(lambda u: False).cardinality() #...
optional - sage.combinat sage.groups
0
```

4.30. Coxeter Groups 265
bruhat_le(other)

Return whether self <= other in the Bruhat order.

INPUT:
• other – an element of the same Coxeter group

OUTPUT: a boolean

EXAMPLES:

```sage
W = WeylGroup(['A', 3])
#optional - sage.combinat sage.groups
sage: u = W.from_reduced_word([1,2,1])
#optional - sage.combinat sage.groups
sage: v = W.from_reduced_word([1,2,3,2,1])
#optional - sage.combinat sage.groups
sage: u.bruhat_le(u)  #optional - sage.combinat sage.groups
True
sage: u.bruhat_le(v)  #optional - sage.combinat sage.groups
True
sage: v.bruhat_le(u)  #optional - sage.combinat sage.groups
False
sage: v.bruhat_le(v)  #optional - sage.combinat sage.groups
True
sage: s = W.simple_reflections()
#optional - sage.combinat sage.groups
sage: s[1].bruhat_le(W.one())  #optional - sage.combinat sage.groups
False
```

The implementation uses the equivalent condition that any reduced word for other contains a reduced word for 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: \( O(l + c) \), where \( l \) is the minimum of the lengths of \( u \) and \( v \), and \( c \) is the cost of the low level methods \( \text{first_descent()}, \text{has_descent()}, \text{apply_simple_reflection()} \), etc. Those are typically \( O(n) \), where \( n \) is the rank of the Coxeter group.

bruhat_lower_covers()

Return all elements that self covers in (strong) Bruhat order.

If \( w = self \) has a descent at \( i \), then the elements that \( w \) covers are exactly \( \{ws_i, u_1s_i, u_2s_i, ..., u_j s_i \} \), 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])
#optional - sage.combinat sage.groups
w = W.from_reduced_word([3,2,3])
#optional - sage.combinat sage.groups
print([v.reduced_word() for v in w.bruhat_lower_covers()])
```

(continues on next page)
We now show how to construct the Bruhat poset:

```
sage: W = WeylGroup("A3")
    #...
   ....:   for v in W.simple_reflection(1).bruhat_lower_covers()}
[[]]
sage: print([v.reduced_word()   for v in W.one().bruhat_lower_covers()])
[]
sage: W = WeylGroup("B", 4, 1)
    #...
sage: w = W.from_reduced_word([0,2])
    #...
   ....:   for v in w.bruhat_lower_covers()}
[[2], [0]]
sage: W = WeylGroup("A3", prefix="s", implementation="permutation")
    #...
   ....:   for v in W.simple_reflection(1).bruhat_lower_covers()}
[[]]
```

```
sage: W = WeylGroup("A3")
    #...
sage: covers = tuple([u, v]
    #...
   ....:   for v in W for u in v.bruhat_lower_covers())
sage: P = Poset((W, covers), cover_relations = True)
    #...
sage: P.show()
```

Alternatively, one can just use:

```
sage: P = W.bruhat_poset()
    #...
```

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 = v r.

**ALGORITHM:**

See **bruhat_lower_covers()**

**EXAMPLES:**
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 bruhat_lower_covers(). Namely, it runs through all \(i\) in the index set. Let \(w\) equal self. If \(w\) has no right descent \(i\), then \(w s_i\) is a cover; if \(w\) has a decent at \(i\), then \(u_j s_i\) is a cover of \(w\) where \(u_j\) is a cover of \(w s_i\).

EXAMPLES:

sage: W = WeylGroup(['A', 3, 1], prefix="s") # optional - sage.combinat sage.groups
sage: w = W.long_element() # optional - sage.combinat sage.groups
sage: w.bruhat_upper_covers() # optional - sage.combinat sage.groups
[]

sage: W = WeylGroup(['A', 3]) # optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([1,2,1]) # optional - sage.combinat sage.groups
sage: S = [v for v in W if w in v.bruhat_lower_covers()] # optional - sage.combinat sage.groups
sage: C = w.bruhat_upper_covers() # optional - sage.combinat sage.groups
sage: set(S) == set(C) # optional - sage.combinat sage.groups
True

bruhat_upper_covers_reflections()

Return all 2-tuples of covers and reflections \((v, r)\) where \(v\) covers self and \(r\) is the reflection such that self = v r.

ALGORITHM:
See bruhat_upper_covers()

EXAMPLES:
canonical_matrix()

Return the matrix of self in the canonical faithful representation.

This is an \(n\)-dimensional 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])
# optional - sage.combinat sage.groups
sage: s = W.simple_reflections()
# optional - sage.combinat sage.groups
sage: (s[1]*s[2]*s[3]).canonical_matrix()
# optional - 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]
```

(continues on next page)
cover_reflections($side='right'$)

Return the set of reflections $t$ such that $self \cdot t$ covers $self$.

If $side$ is 'left', $t \cdot self$ covers $self$.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 4], prefix="s")
# optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([3,1,2,1])
# optional - sage.combinat sage.groups
sage: w.cover_reflections()  # optional - sage.combinat sage.groups
[s3, s2*s3*s2, s4, s1*s2*s3*s4*s3*s2*s1]
sage: w.cover_reflections(side='left')  # optional - sage.combinat sage.groups
[s4, s2, s1*s2*s1, s3*s4*s3]
```

coxeter_sorting_word($c$)

Return the $c$-sorting word of $self$.

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 $self$ as a list of integers.

EXAMPLES:

```python
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]
```

deodhar_factor_element($w$, $index_set$)

Return Deodhar’s Bruhat order factoring element.

INPUT:

- $w$ is an element of the same Coxeter group $\tilde{W}$ as $self$
- $index_set$ is a subset of Dynkin nodes defining a parabolic subgroup $W'$ of $\tilde{W}$
It is assumed that \( v = \text{self} \) and \( w \) are minimum length coset representatives for \( \mathbb{W}/\mathbb{W}' \) such that \( v \leq w \) in Bruhat order.

**OUTPUT:**

Deodhar’s element \( f(v, w) \) is the unique element of \( \mathbb{W}' \) such that, for all \( v' \) and \( w' \) in \( \mathbb{W}' \), \( vv' \leq ww' \) in \( \mathbb{W} \) if and only if \( v' \leq f(v, w) * w' \) in \( \mathbb{W}' \) where \( * \) is the Demazure product.

**EXAMPLES:**

```
sage: \( W = \text{WeylGroup(['A', 5], prefix='s')} \)
  # optional - sage.combinat sage.groups
sage: \( v = W.\text{from_reduced_word([5])} \)
  # optional - sage.combinat sage.groups
sage: \( w = W.\text{from_reduced_word([4,5,2,3,1,2])} \)
  # optional - sage.combinat sage.groups
sage: \( v.\text{deodhar_factor_element}(w, [1,3,4]) \)
  # optional - sage.combinat sage.groups
s3*e1
sage: \( W = \text{WeylGroup(['C', 2])} \)
  # optional - sage.combinat sage.groups
sage: \( w = W.\text{from_reduced_word([2,1])} \)
  # optional - sage.combinat sage.groups
sage: \( w.\text{deodhar_factor_element}(W.\text{from_reduced_word([2]),[1])} \)
  # optional - sage.combinat sage.groups
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\((w, index\_set)\)**

Letting \( v = \text{self} \), given a Bruhat relation \( v \mathbb{W}' \geq w \mathbb{W}' \) among cosets with respect to the subgroup \( \mathbb{W}' \) given by the Dynkin node subset \( index\_set \), returns the Bruhat-maximum lift \( x \) of \( w\mathbb{W}' \) such that \( v \geq x \).

**INPUT:**

• \( w \) is an element of the same Coxeter group \( \mathbb{W} \) as \( \text{self} \).
• \( index\_set \) is a subset of Dynkin nodes defining a parabolic subgroup \( \mathbb{W}' \).

**OUTPUT:**

The unique Bruhat-maximum element \( x \) in \( \mathbb{W} \) such that \( x \mathbb{W}' = w \mathbb{W}' \) and \( v \geq x \).

See also:

```
sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_up()
```

**EXAMPLES:**

```
sage: \( W = \text{WeylGroup(['A', 3], prefix='s')} \)
  # optional - sage.combinat sage.groups
sage: \( v = W.\text{from_reduced_word([1,2,3,2])} \)
  # optional - sage.combinat sage.groups
sage: \( w = W.\text{from_reduced_word([3,2])} \)
  # optional - sage.combinat sage.groups
```

(continues on next page)
deodhar_lift_up(w, index_set)

Letting \( v = \text{self} \), given a Bruhat relation \( v \, \bar{W} \leq w \, \bar{W}' \) among cosets with respect to the subgroup \( \bar{W}' \) given by the Dynkin node subset \( \text{index_set} \), returns the Bruhat-minimum lift \( x \) of \( w \bar{W}' \) such that \( v \leq x \).

**INPUT:**
- \( w \) is an element of the same Coxeter group \( \bar{W} \) as \( \text{self} \).
- \( \text{index_set} \) is a subset of Dynkin nodes defining a parabolic subgroup \( \bar{W}' \).

**OUTPUT:**
The unique Bruhat-minimum element \( x \) in \( \bar{W} \) such that \( x \, \bar{W}' = w \, \bar{W}' \) and \( v \leq x \).

See also:
sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_down()

**EXAMPLES:**

```python
sage: W = WeylGroup(['A', 3], prefix="s")
  #opt - sage.combinat sage.groups
sage: v = W.from_reduced_word([1,2,3])
  #opt - sage.combinat sage.groups
sage: w = W.from_reduced_word([1,3,2])
  #opt - sage.combinat sage.groups
sage: v.deodhar_lift_up(w, [3])
  #opt - sage.combinat sage.groups
s1*s2*s3*s2
```

descents(side='right', index_set=None, positive=False)

Return the descents of \( \text{self} \), as a list of elements of the index_set.

**INPUT:**
- \( \text{index_set} \) - a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
- \( \text{side} \) - 'left' or 'right' (default: 'right')
- \( \text{positive} \) - a boolean (default: False)

The \( \text{index_set} \) option can be used to restrict to the parabolic subgroup indexed by \( \text{index_set} \).

If \( \text{positive} \) is True, then returns the non-descents instead

**Todo:** find a better name for \( \text{positive} \): 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:**

```python
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]
```
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 descents() for a description of the options.
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]
0
sage: w.first_descent()
1

has_descent(i, side='right', positive=False)
Return whether i is a (left/right) descent of self.
See descents() for a description of the options.
EXAMPLES:

sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0] * s[1] * s[2]
sage: w.has_descent(2)
True
sage: [ w.has_descent(i) for i in [0,1,2] ]
[False, False, True]
sage: [ w.has_descent(i, side='left') for i in [0,1,2] ]
[True, False, False]
sage: [ w.has_descent(i, positive=True) for i in [0,1,2] ]
[True, True, False]

This default implementation delegates the work to has_left_descent() and has_right_descent().

has_full_support()
Return whether self has full support.
An element is said to have full support if its support contains all simple reflections.
EXAMPLES:
```python
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
```

**has_left_descent**(*i*)

Return whether *i* is a left descent of self.

This default implementation uses that a left descent of *w* is a right descent of *w*⁻¹.

**EXAMPLES:**

```python
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
```

**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, ..., 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")  # optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([3,1,2,1])  # optional - sage.combinat sage.groups
sage: w.inversions_as_reflections()  # optional - sage.combinat sage.groups
[s1, s1*s2*s1, s2, s1*s2*s3*s2*s1]
```
**is_coxeter_sortable**(*c, sorting_word= 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.

**INPUT:**
- *c* – a Coxeter element.
- *sorting_word* – sorting word (default: None) used to not recompute the c-sorting word if already computed.

**OUTPUT:**
- *is self c-sortable*

**EXAMPLES:**

```python
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])  # optional - sage.combinat sage.groups
sage: c = W.from_reduced_word([1, 2, 3])  # optional - sage.combinat sage.groups

Number of c-sortable elements in A₃ (Catalan number):

```python
sage: len([w for w in W if w.is_coxeter_sortable(c)])  #optional - sage.combinat sage.groups
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 ∈ S for which m(s, t) > 2, no reduced word of w contains the ‘braid’ word stst... of length m(s, t) as a contiguous subword. See [Ste1996].

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A', 3])
sage: len([w for w in W if w.is_fully_commutative()])
14
sage: W = CoxeterGroup(['B', 3])
sage: len([w 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:

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].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()
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

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 $C'_{s}C'_{w_i}$ and $C'_{u_j+1}$ appears in $C'_{t}C'_{u_j}$ 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:
The cell computation uses the optional package `coxeter3` in the background if available to speed up the computation, even in the different implementations implementations:

```sage
W = CoxeterGroup('A3', implementation='coxeter3')  # optional - coxeter3
s1, s2, s3 = W.simple_reflections()  # optional - coxeter3
s1.kazhdan_lusztig_cell()  # optional - coxeter3
{(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)}
```

Next, we compute a right cell and a two-sided cell in $A_3$:

```sage
W = CoxeterGroup('A3', implementation='coxeter3')  # optional - coxeter3
s1, s2, s3 = W.simple_reflections()  # optional - coxeter3
w = s1 * s3  # optional - coxeter3
w.kazhdan_lusztig_cell(side='right')  # optional - coxeter3
{{1, 3}, [1, 3, 2], \ldots}
w.kazhdan_lusztig_cell(side='two-sided')  # optional - coxeter3
{{1, 3}, [1, 3, 2], [2, 1, 3], \ldots}
```

Some slightly longer computations in `$B_4$`:

```sage
W = CoxeterGroup('B4', implementation='coxeter3')  # optional - coxeter3
s1, s2, s3, s4 = W.simple_reflections()  # optional - coxeter3
s1.kazhdan_lusztig_cell(side='right')  # long time, (4 seconds)  # optional - coxeter3
{{1}, [1, 2], [1, 2, 3], \ldots}
```
left_inversions_as_reflections()

Return the set of reflections \( r \) such that \( r \ self < self \).

EXAMPLES:

```
sage: W = WeylGroup(['A', 3], prefix="s")
    #optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([3,1,2,1])
    #optional - sage.combinat sage.groups
sage: w.left_inversions_as_reflections()
    #optional - 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:

```
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
sage: W = CoxeterGroups().example()
sage: R.<x> = ZZ[]
sage: s = sum(x^w.length() for w in W)
sage: p = prod(sum(x^i for i in range(j)) for j in range(1, 5))
```
(continues on next page)
\begin{verbatim}
  sage: s - p
 0
\end{verbatim}

See also:
\texttt{reduced_word()}

Todo: Should use \texttt{reduced_word_iterator} (or reverse_iterator)

\texttt{lower_cover_reflections}(side='right')

Return the reflections $t$ such that $\texttt{self}$ covers $\texttt{self} \cdot t$.

If \texttt{side} is 'left', \texttt{self} covers $t \cdot \texttt{self}$.

EXAMPLES:

\begin{verbatim}
  sage: W = WeylGroup(['A', 3],prefix="s")
  #\_optional - sage.combinat sage.groups
  sage: w = W.from_reduced_word([3,1,2,1])
  #\_optional - sage.combinat sage.groups
  sage: w.lower_cover_reflections()
  #\_optional - sage.combinat sage.groups
  [s1*s2*s3*s2*s1, s2, s1]
  sage: w.lower_cover_reflections(side='left')
  #\_optional - sage.combinat sage.groups
  [s2*s3*s2, s3, s1]
\end{verbatim}

\texttt{lower_covers}(side='right', index_set=None)

Return all elements that $\texttt{self}$ covers in weak order.

INPUT:
\begin{itemize}
  \item \texttt{side} – 'left' or 'right' (default: 'right')
  \item \texttt{index_set} – a list of indices or None
\end{itemize}

OUTPUT: a list

EXAMPLES:

\begin{verbatim}
  sage: W = WeylGroup(['A', 3])
  #\_optional - sage.combinat sage.groups
  sage: w = W.from_reduced_word([3,2,1])
  #\_optional - sage.combinat sage.groups
  sage: [x.reduced_word() for x in w.lower_covers()]
  #\_optional - sage.combinat sage.groups
  [[3, 2]]
  sage: [x.reduced_word() for x in w.lower_covers(side='left')]
  #\_optional - sage.combinat sage.groups
  [[2, 1]]
  sage: w = W.from_reduced_word([3,2,3,1])
  #\_optional - sage.combinat sage.groups
  sage: [x.reduced_word() for x in w.lower_covers()]
  #\_optional - sage.combinat sage.groups
\end{verbatim}

To obtain covers for left weak order, set the option \texttt{side} to 'left':

\begin{verbatim}
  sage: [x.reduced_word() for x in w.lower_covers(side='left')]
  #\_optional - sage.combinat sage.groups
  [[2, 1]]
\end{verbatim}

(continues on next page)
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])]
```

```python
sage: [x.reduced_word() for x in w.lower_covers(side='left')]
```

### 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:**

```python
sage: W = WeylGroup(['A', 4], prefix="s")
```

```python
sage: v = W.from_reduced_word([2,3,4,1,2])
```

```python
sage: u = W.from_reduced_word([2,3,2,1])
```

```python
sage: v.min_demazure_product_greater(u)
```

### 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:**

```python
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[2]
sage: w.reduced_word()
```

```
[0, 1, 2]
```

See also:
• `reduced_words()`, `reduced_word_reverse_iterator()`,
• `length()`, `reduced_word_graph()`

`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 ≥ 2`).

This graph is always connected (a theorem due to Tits) and has no multiple edges.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3], prefix='s')
# optional - sage.combinat sage.groups
sage: w0 = W.long_element()
# optional - sage.combinat sage.groups
sage: G = w0.reduced_word_graph()
# optional - sage.combinat sage.groups sage.graphs
sage: G.num_verts()  # optional - sage.combinat sage.groups sage.graphs
16
sage: len(w0.reduced_words())  # optional - sage.combinat sage.groups sage.graphs
16
sage: G.num_edges()  # optional - sage.combinat sage.groups sage.graphs
18
sage: len([e for e in G.edges(sort=False) if e[2] == 2])  # optional - sage.combinat sage.groups sage.graphs
10
sage: len([e for e in G.edges(sort=False) if e[2] == 3])  # optional - sage.combinat sage.groups sage.graphs
8
```

See also:

`reduced_words()`, `reduced_word_reverse_iterator()`, `length()`, `reduced_word()`

`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
```
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())  # optional - sage.combinat
[[0, 2], [2, 0]]
```

```python
sage: W = WeylGroup(['E', 6])  # optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([2, 3, 4, 2])  # optional - sage.combinat sage.groups
sage: sorted(w.reduced_words())  # optional - sage.combinat sage.groups
[[2, 3, 4, 2], [3, 2, 4, 2], [3, 4, 2, 4]]
```

```python
sage: W = ReflectionGroup(['A', 3], index_set=['AA', 'BB', '5'])  # optional - gap3
sage: w = W.long_element()  # optional - gap3
sage: w.reduced_words()  # optional - gap3
[['BB', '5', 'AA', 'BB', '5', 'AA'],
 ['5', 'BB', '5', 'AA', 'BB', '5'],
 ['BB', 'AA', 'BB', '5', 'BB', 'AA'],
 ['AA', 'BB', 'AA', '5', 'BB', 'AA'],
 ['AA', 'BB', 'BB', '5', 'BB', 'BB'],
 ['BB', 'AA', 'BB', 'AA', '5'],
 ['BB', 'BB', 'BB', 'AA', 'BB'],
 ['AA', 'BB', 'BB', 'BB', 'AA'],
 ['BB', 'BB', 'BB', 'BB', 'BB'],
 ['BB', '5', 'AA', 'BB', '5'],
 ['AA', '5', 'BB', '5', 'AA', 'BB'],
 ['5', 'BB', 'AA', 'BB', 'BB'],
 ['5', 'BB', 'BB', 'BB', 'BB'],
 ['5', 'AA', 'BB', '5', 'AA', 'BB'],
 ['BB', '5', 'BB', 'AA', 'BB'],
 ['BB', 'AA', 'BB', 'BB', '5', 'AA']]
```
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())
\[0, 2\], [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:

```
sage: W = WeylGroup(['A', 3])
˓→optional - sage.combinat sage.groups
sage: s = W.simple_reflections()
˓→optional - sage.combinat sage.groups
sage: (s[1]*s[2]*s[3]).reflection_length()
˓→optional - sage.combinat sage.groups
3
```

```
sage: W = SymmetricGroup(4)
˓→optional - sage.combinat sage.groups
sage: s = W.simple_reflections()
˓→optional - sage.combinat sage.groups
sage: (s[3]*s[2]*s[3]).reflection_length()
˓→optional - sage.combinat sage.groups
1
```

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:

```
sage: W = CoxeterGroups().example()
sage: w = W.from_reduced_word([1,2,1])
sage: w.support()
{1, 2}
```

`upper_covers(side='right', index_set=None)`

Return all elements that cover `self` in weak order.

INPUT:

- `side` – 'left' or 'right' (default: 'right')
- `index_set` – a list of indices or None

OUTPUT: a list

EXAMPLES:

```
sage: W = WeylGroup(['A', 3])  # optional - sage.combinat sage.groups
sage: w = W.from_reduced_word([2,3])  # optional - sage.combinat sage.groups
sage: [x.reduced_word() for x in w.upper_covers()]
[[3, 2, 1], [2, 3, 2]]  # optional - sage.combinat sage.groups
```

To obtain covers for left weak order, set the option `side` to 'left':

```
sage: [x.reduced_word() for x in w.upper_covers(side='left')]
[[1, 2, 3], [2, 3, 2]]  # optional - sage.combinat sage.groups
```

Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:

```
sage: [x.reduced_word() for x in w.upper_covers(index_set=[1])]
[[3, 2, 1]]  # optional - sage.combinat sage.groups

sage: [x.reduced_word()]
[[1, 2, 3]]  # optional - sage.combinat sage.groups
```

`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:
Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:

```
sage: [x.reduced_word() for x in w.weak_covers(index_set=[1, 2])]
```

```
[[2, 3, 2]]
```

**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:**

```
sage: W = WeylGroup(['A', 3])
```

```
optional - sage.combinat sage.groups
```

```
sage: u = W.from_reduced_word([1, 2])
```

```
optional - sage.combinat sage.groups
```

```
sage: v = W.from_reduced_word([1, 2, 3, 2])
```

```
optional - sage.combinat sage.groups
```

```
sage: u.weak_le(v)
```

```
optional - sage.combinat sage.groups
```

(continues on next page)
Comparison for left weak order is achieved with the option side:

```
sage: u.weak_le(v, side='left')  # optional - sage.combinat sage.groups
False
```

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: \( O(l \cdot c) \), where \( l \) is the minimum of the lengths of \( u \) and \( v \), 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:

```
sage: W = WeylGroup(['A', 3])  # optional - sage.combinat sage.groups
sage: P4 = Permutations(4)  # optional - sage.combinat sage.groups
sage: def P4toW(w):
    return W.from_reduced_word(w.reduced_word())  # optional - sage.combinat sage.groups

sage: for u in P4:  # long time (5s on sage.math, 2011)  # optional - sage.combinat sage.groups
    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:

```
sage: W = CoxeterGroup(['A', 2])  # optional - sage.combinat sage.groups
sage: W.braid_group_as_finitely_presented_group()  # optional - sage.combinat sage.groups
```
Finitely presented group $< S_1, S_2 | S_1 S_2 S_1 S_2^{-1} S_1^{-1} S_2^{-1} >$

\[\text{sage: } W = \text{WeylGroup(['B', 2])}\]  # optional - sage.combinat sage.groups
\[\text{sage: } W.braid_group_as_finitely_presented_group()\]  # optional - sage.combinat sage.groups

Finitely presented group $< S_1, S_2 | (S_1 S_2)^2 (S_1^{-1} S_2^{-1})^2 >$

\[\text{sage: } W = \text{ReflectionGroup(['B', 3], index_set=['AA', 'BB', 'S'])} \]  # optional - sage.combinat sage.groups
\[\text{sage: } W.braid_group_as_finitely_presented_group()\]  # optional - sage.combinat sage.groups

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:**

\[\text{sage: } W = \text{CoxeterGroups().example()}\]
\[\text{sage: } s = W.simple_reflections()\]
\[\text{sage: } w = s[0] \ast s[1] \ast s[2] \ast s[1]\]
\[\text{sage: } \text{word} = w.\text{reduced_word()}; \text{word}\]
\[[0, 1, 2, 1]\]

\[\text{sage: } \text{sorted}(W.braid_orbit(\text{word}))\]  # optional - sage.combinat sage.groups
\[[[0, 1, 2, 1], [0, 2, 1, 2], [2, 0, 1, 2]]\]

\[\text{sage: } \text{sorted}(W.braid_orbit([2, 1, 1, 2, 1]))\]  # optional - sage.combinat sage.groups
\[[[1, 2, 1, 1, 2], [2, 1, 1, 2, 1], [2, 1, 2, 1, 2], [2, 2, 1, 2, 2]]\]

\[\text{sage: } W = \text{ReflectionGroup(['A', 3], index_set=['AA', 'BB', 'S'])} \]  # optional - gap3
\[\text{sage: } w = W.\text{long_element()}\]  # optional - gap3
\[\text{sage: } W.braid_orbit(w.\text{reduced_word()})\]  # optional - gap3
\[[['BB', '5', 'AA', 'BB', '5', 'AA'], ['5', 'BB', '5', 'AA', 'BB', '5'], ['BB', 'AA', 'BB', '5', 'BB', 'AA']]\]
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:

```
sage: W = CoxeterGroups().example()
sage: sorted(W.braid_orbit_iter([0, 1, 2, 1]))
# optional - sage.combinat sage.groups
[[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:

```
sage: W = WeylGroup("A", 2)
# optional - sage.combinat sage.groups
sage: W.braid_relations()
# optional - sage.combinat sage.groups
[[[1, 2, 1]], [2, 1, 2]]
```

```
sage: W = WeylGroup("B", 3)
# optional - sage.combinat sage.groups
sage: W.braid_relations()
# optional - sage.combinat sage.groups
```

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:

sage: W = CoxeterGroup(['H', 3])
    #...
    #optional - sage.combinat sage.groups sage.graphs
sage: G = W.bruhat_graph(); G
    #...
    #optional - sage.combinat sage.groups sage.graphs
    Digraph on 120 vertices

sage: W = CoxeterGroup(['A', 2, 1])
    #...
    #optional - sage.combinat sage.groups sage.graphs
sage: s1, s2, s3 = W.simple_reflections()
    #...
    #optional - sage.combinat sage.groups sage.graphs
sage: W.bruhat_graph(s1, s3*s2*s3)
    #...
    #optional - sage.combinat sage.groups sage.graphs
    Digraph on 6 vertices

sage: W.bruhat_graph(s1, s3*s2*s3)
    #...
    #optional - sage.combinat sage.groups sage.graphs
    Digraph on 0 vertices

sage: W = WeylGroup("A3", prefix="s")
    #...
    #optional - sage.combinat sage.groups sage.graphs
sage: s1, s2, s3 = W.simple_reflections()
    #...
    #optional - sage.combinat sage.groups sage.graphs
sage: G = W.bruhat_graph(s1*s3, s1*s2*s3*s1); G
    #...
    #optional - sage.combinat sage.groups sage.graphs
    Digraph on 10 vertices

Check that the graph has the correct number of edges (see github issue #17744):

sage: len(G.edges(sort=False))
    #...
    #optional - sage.combinat sage.groups sage.graphs
16

bruhat_interval(x, y)

Return the list of \( t \) such that \( x <= t <= y \).

EXAMPLES:
\textbf{bruhat_interval}\textunderscore\textit{poset}(\textit{x}, \textit{y}, \textit{facade=False})

Return the poset of the Bruhat interval between \textit{x} and \textit{y} in Bruhat order.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = WeylGroup("A3", prefix="s")
sage: s1, s2, s3 = W.simple_reflections()
sage: W.bruhat_interval_poset(s2, s1*s3*s2*s1*s3)  # optional -- sage.combinat sage.groups
Finite poset containing 16 elements

sage: W = WeylGroup(['A', 2, 1], prefix="s")
sage: s0, s1, s2 = W.simple_reflections()
sage: W.bruhat_interval_poset(1, s0*s1*s2)  # optional -- sage.combinat sage.groups
Finite poset containing 8 elements
\end{verbatim}

\textbf{canonical\textunderscore representation()}

Return the canonical faithful representation of \textit{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: W = WeylGroup("A3")
sage: W.canonical_representation()  # optional -- sage.combinat sage.groups
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
\end{verbatim}

\textbf{coxeter\textunderscore diagram()}


Return the Coxeter diagram of self.

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['H', 3], implementation="reflection")
  #˓→optional - sage.combinat sage.groups
sage: G = W.coxeter_diagram(); G
  #˓→optional - sage.combinat sage.groups sage.graphs
Graph on 3 vertices
sage: G.edges(sort=True)
  #˓→optional - sage.combinat sage.groups sage.graphs
[(1, 2, 3), (2, 3, 5)]
sage: CoxeterGroup(G) is W
  #˓→optional - sage.combinat sage.groups sage.graphs
True
sage: G = Graph([(0, 1, 3), (1, 2, oo)])
  #˓→optional - sage.combinat sage.groups sage.graphs
sage: W = CoxeterGroup(G)
  #˓→optional - sage.combinat sage.groups sage.graphs
sage: W.coxeter_diagram() == G
  #˓→optional - sage.combinat sage.groups sage.graphs
True
sage: CoxeterGroup(W.coxeter_diagram()) is W
  #˓→optional - sage.combinat sage.groups sage.graphs
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 self).

**EXAMPLES:**

```python
sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()
  #˓→optional - sage.combinat sage.groups
[1, 2, 3, 4]
sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()
  #˓→optional - sage.combinat sage.groups
[1, 2, 3, 4]
sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()
  #˓→optional - sage.combinat sage.groups
[1, 2, 4, 3]
sage: CoxeterGroup(['F', 4]).coxeter_element().reduced_word()
  #˓→optional - sage.combinat sage.groups
[1, 2, 3, 4]
sage: CoxeterGroup(['E', 8]).coxeter_element().reduced_word()
  #˓→optional - sage.combinat sage.groups
```

(continues on next page)
optional - sage.combinat sage.groups
[1, 3, 2, 4, 5, 6, 7, 8]
sage: CoxeterGroup(['H', 3]).coxeter_element().reduced_word()  #
optional - sage.combinat sage.groups
[1, 2, 3]

This method is also used for well generated finite complex reflection groups:

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3]
sage: W = ReflectionGroup((2,1,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3, 4]
sage: W = ReflectionGroup((4,1,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3, 4]
sage: W = ReflectionGroup((4,4,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3, 4]

coxeter_matrix()
Return the Coxeter matrix associated to self.

EXAMPLES:

sage: G = WeylGroup(['A', 3])  #
optional - sage.combinat sage.groups
sage: G.coxeter_matrix()  #
optional - sage.combinat sage.groups
[1 3 2]
[3 1 3]
[2 3 1]

coxeter_type()
Return the Coxeter type of self.

EXAMPLES:

sage: W = CoxeterGroup(['H', 3])  #
optional - sage.combinat sage.groups
sage: W.coxeter_type()  #
optional - sage.combinat sage.groups
Coxeter type of ['H', 3]

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.

See \texttt{CoxeterGroups.ParentMethods.simple_projections()}.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A', 2])
# optional - sage.combinat sage.groups
sage: w = W.demazure_product([2,2,1])
# optional - sage.combinat sage.groups
sage: w.reduced_word()  # optional - sage.combinat sage.groups
[2, 1]

sage: W = WeylGroup(['B', 2])
# optional - sage.combinat sage.groups
sage: w = W.demazure_product([2,1,2,1,2])
# optional - sage.combinat sage.groups
sage: w.reduced_word()  # optional - sage.combinat sage.groups
[2, 1, 2, 1]
```

**elements_of_length**\((n)\)

Return all elements of length \(n\).

**EXAMPLES:**

```python
sage: A = AffinePermutationGroup(['A', 2, 1])
# optional - sage.combinat sage.groups
sage: [len(list(A.elements_of_length(i))) for i in [0..5]]
# optional - sage.combinat sage.groups
[1, 3, 6, 9, 12, 15]

sage: W = CoxeterGroup(['H', 3])
# optional - sage.combinat sage.groups
sage: [len(list(W.elements_of_length(i))) for i in range(4)]
# optional - sage.combinat sage.groups
[1, 3, 5, 7]

sage: W = CoxeterGroup(['A', 2])
# optional - sage.combinat sage.groups
sage: [len(list(W.elements_of_length(i))) for i in range(6)]
# optional - sage.combinat sage.groups
[1, 2, 2, 1, 0, 0]
```

**fully_commutative_elements**

Return the set of fully commutative elements in this Coxeter group.
See also:

FullyCommutativeElements

EXAMPLES:

```python
sage: CoxeterGroup(['A', 3]).fully_commutative_elements()
#...
˓→optional - sage.combinat sage.groups

Fully commutative elements of
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

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, 0, 2), (2, 0, 1, 3), (1, 2, 3, 0), (2, 3, 0, 1)]
sage: sorted(tuple(w.descents()) for w in G)
[(), (0,), (0,), (0,), (1,), (1,), (1,), (1,), (2,), (2,), (2,)]
sage: G = S.grassmannian_elements(side = "left")
sage: G.cardinality()
12
sage: sorted(tuple(w.descents(side = "left")) for w in G)
[(), (0,), (0,), (0,), (1,), (1,), (1,), (1,), (2,), (2,), (2,)]
```

index_set()

Return the index set of self.

EXAMPLES:

```python
sage: W = CoxeterGroup([[1,3],[3,1]])
˓→optional - sage.combinat sage.groups
sage: W.index_set()
˓→optional - sage.combinat sage.groups
(1, 2)
```

(continues on next page)
The cells are computed by using `kazhdan_lusztig_cell()`.

As detailed there, installation of the optional package `coxeter3` is recommended (though not required) before using this function as it speeds up the computation.

**INPUT:**

- `side` – (default: 'left') either 'left', 'right', or 'two-sided'

**EXAMPLES:**

We compute the right cells in the Coxeter group of type $A_2$ 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:

```python
sage: W = CoxeterGroup('A2')  # optional - sage.combinat sage.groups
sage: KL_cells = W.kazhdan_lusztig_cells(side='right')  # optional - sage.combinat sage.groups
sage: set(tuple(sorted(C, key=lambda w: w.reduced_word())) for C in KL_cells)
{([-1 1] [ 0 -1]
  [ 0 1], [ 1 -1]),
  ([ 0 -1]
  [-1 0]),
  ([1 0]
  [0 1]),
  ([ 1 0] [-1 1]
  [ 1 -1], [-1 0])}
```

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:

```
sage: W = CoxeterGroup('A3', implementation='coxeter3')  # optional -coxeter3
sage: KL_cells = W.kazhdan_lusztig_cells()  # optional -coxeter3
sage: set([tuple(sorted(C)) for C in KL_cells])  # optional -coxeter3
\{(\[]\),
  ([1], [1, 2, 3, 2, 1], [2, 1, 3, 2, 1], [3, 2, 1]),
  ([1, 2], [2, 3, 2], [3, 2, 1]),
  ([1, 2], [2, 3, 2], [3, 2, 1]),
  ([1, 2, 1, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([1, 2, 3], [2, 3, 2], [3, 2, 1]),
  ([1, 2, 3], [2, 3, 2], [3, 2, 1]),
  ([1, 2, 3], [2, 3, 2], [3, 2, 1]),
  ([1, 2, 3], [2, 3, 2], [3, 2, 1]),
  ([2, 1, 2, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1])\}
sage: len(KL_cells)  # optional -coxeter3
10

sage: W = CoxeterGroup('A3', implementation='permutation')  # optional -coxeter3
sage: len(W.kazhdan_lusztig_cells())  # optional -coxeter3
10
```

Computing the two sided cells in $B_3$:

```
sage: W = CoxeterGroup('B3', implementation='coxeter3')  # optional -coxeter3
sage: b3_cells = W.kazhdan_lusztig_cells('two-sided')  # optional -coxeter3
sage: len(b3_cells)  # optional -coxeter3
6
sage: set([tuple(sorted(C)) for C in W.kazhdan_lusztig_cells()])  # optional -coxeter3
\{(\[]\),
  ([1], [1, 2, 3, 2, 1], [2, 1, 3, 2, 1], [3, 2, 1]),
  ([1, 2], [1, 2, 3, 2, 1], [2, 3, 2, 1], [3, 2, 1]),
  ([1, 2, 1, 3], [1, 2, 3, 2, 1], [2, 3, 2, 1], [3, 2, 1]),
  ([1, 2, 3, 2], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 1, 2, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 1, 2, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 1, 2, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 1, 2, 3], [2, 3, 2, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1]),
  ([2, 3, 1], [3, 2, 1])\}
```
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:

```
sage: A = AffinePermutationGroup(['A', 7, 1])
#optional - sage.combinat sage.groups
sage: p = A.random_element_of_length(10)
#optional - sage.combinat sage.groups
sage: p in A
#optional - sage.combinat sage.groups
True
sage: p.length() == 10
#optional - sage.combinat sage.groups
True

sage: W = CoxeterGroup(['A', 4])
#optional - sage.combinat sage.groups
sage: p = W.random_element_of_length(5)
#optional - sage.combinat sage.groups
sage: p in W
#optional - sage.combinat sage.groups
True
sage: p.length() == 5
#optional - sage.combinat sage.groups
True
```

sign_representation($base\_ring$=None, $side$='twosided')

Return the sign representation of `self` over $base\_ring$.

INPUT:

- `base\_ring` – (optional) the base ring; the default is $\mathbb{Z}$
- `side` – ignored

EXAMPLES:
simple_projection\((i, \text{side}='right', \text{length\_increasing}=\text{True})\)

Return the simple projection \(\pi_i\) (or \(\bar{\pi}_i\) if \text{length\_increasing} is False).

INPUT:
• \(i\) - an element of the index set of \text{self}

See simple_projections() for the options and for the definition of the simple projections.

EXAMPLES:

\begin{verbatim}
sage: W = CoxeterGroups().example()
sage: W
The symmetric group on {0, ..., 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)
\end{verbatim}

simple_projections\((\text{side}='right', \text{length\_increasing}=\text{True})\)

Return the family of simple projections, also known as 0-Hecke or Demazure operators.

INPUT:
• \text{self} – a Coxeter group \(W\)
• \text{side} – ‘left’ or ‘right’ (default: ‘right’)
• \text{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:
\[\pi_i(w) = ws_i \text{ if } i \text{ is not a descent of } w, \pi_i(w) = w \text{ otherwise.}\]
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 \( \pi^2_i = \pi \) not involutions \( s^2_i = 1 \). As such, the simple projections generate the 0-Hecke monoid.

By symmetry, one can also define the projections \( (\bar{\pi}_i)_{i \in I} \) (when the option length_increasing is False):
\[
\bar{\pi}_i(w) = ws_i \text{ if } i \text{ is a descent of } w \\pi_i(w) = w \text{ otherwise},
\]
as well as the analogues acting on the left (when the option side is ‘left’).

**EXAMPLES:**

```
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)
```

**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:**

```
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)]
```

**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:**

```
sage: D6 = FiniteCoxeterGroups().example(5)
sage: I = D6.weak_order_ideal(predicate=lambda w: w.length() <= 3)
sage: I.cardinality()
```

(continues on next page)
We now consider an infinite Coxeter group:

```
sage: W = WeylGroup(["A",1,1])
sage: I = W.weak_order_ideal(predicate = lambda w: w.length() <= 2)
sage: list(iter(I))
[(1 0), (-1 2), [ 1 0], [ 3 -2], [-1 2]
 [0 1], [ θ 1], [ 2 -1], [ 2 -1], [-2 3]
]
```

Even when the result is finite, some features of `FiniteEnumeratedSets` are not available:

```
sage: I.cardinality() # todo: not implemented
5
sage: list(I) # todo: not implemented
```

unless this finiteness is explicitly specified:

```
sage: I = W.weak_order_ideal(predicate = lambda w: w.length() <= 2,
....:                      category = FiniteEnumeratedSets())
sage: I.cardinality()
5
sage: list(I)
[(1 0), (-1 2), [ 1 0], [ 3 -2], [-1 2]
 [0 1], [ θ 1], [ 2 -1], [ 2 -1], [-2 3]
]
```

**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 $u = u_i s_i$, 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:**

```
sage: CoxeterGroups().additional_structure()
```
super_categories()

EXAMPLES:

```
sage: CoxeterGroups().super_categories()
[Category of generalized 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 $U_q(g)$ $I$-crystal morphism $\Psi : B \to C$ is a map $\Psi : B \cup \{0\} \to C \cup \{0\}$ such that:

- $\Psi(0) = 0$.
- If $b \in B$ and $\Psi(b) \in C$, then $\text{wt}(\Psi(b)) = \text{wt}(b)$, $\varepsilon_i(\Psi(b)) = \varepsilon_i(b)$, 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 $j \in \sigma_i$:
  \[
  \varepsilon_i(b) = \frac{1}{\gamma_j} \varepsilon_{\sigma_j}(\Psi(b)), \quad \varphi_i(b) = \frac{1}{\gamma_j} \varphi_{\sigma_j}(\Psi(b)), \quad \text{wt}(\Psi(b)) = \sum_i \varepsilon_i \sum_{j \in \sigma_i} \gamma_j \lambda_j,
  \]
  where $\text{wt}(b) = \sum_i c_i \lambda_i$.
- 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:
  \[
  \Psi(b') = e_i \Psi(b) = \prod_{j \in \sigma_i} \varepsilon^i_{\sigma_j}(\Psi(b)), \quad \Psi(b') = f_i \Psi(b) = \prod_{j \in \sigma_i} \varphi^i_{\sigma_j}(\Psi(b)).
  \]

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 $A_1$: 
We now look at the decomposition of $B(\Lambda_1) \otimes B(\Lambda_1)$ into $B(2\Lambda_1) \oplus B(0)$:

```
sage: B0 = crystals.Tableaux(['A',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 Family  
  (The crystal of tableaux of type ['A', 1] and shape(s) [[2]],  
  The crystal of tableaux of type ['A', 1] and shape(s) [[]])  
  Defn: [[1], [1]] |--> [[1, 1]]  
  [[1], [2]] |--> []  
sage: psi.is_isomorphism()  
True
```

We can always construct the trivial morphism which sends everything to 0:

```
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:

```
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]]
sage: psi(b.f_string([1,2,4]))
[[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(f"{b} |--> {psi(b)}")
[[1]] |--> [[1, 1]]
[[2]] |--> [[2, 2]]
[[3]] |--> [[3, 3]]
[[0]] |--> [[3, -3]]
[[-3]] |--> [[-3, -3]]
[[-2]] |--> [[-2, -2]]
[[-1]] |--> [[-1, -1]]
```

`Element`  
alias of `CrystalMorphismByGenerators`  

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(['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_surjective()
False
sage: im_gens = [None, B.module_generators[0]]
sage: psi = C.crystal_morphism(im_gens, codomain=B)
sage: psi.is_surjective()
True
```

**scaling_factors()**

Return the scaling factors $\gamma_i$.

**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.scaling_factors()
Finite family {1: 2, 2: 2, 3: 1}
```
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)}
```

class sage.categories.crystals.CrystalMorphismByGenerators(parent, on_gens, cartan_type=None, virtualization=None, scaling_factors=None, gens=None, check=True)

Bases: CrystalMorphism

A crystal morphism defined by a set of generators which create a virtual crystal inside the codomain.

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 finite 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

See also:

sage.categories.crystals.Crystals.ParentMethods.crystal_morphism()

im_gens()

Return the image of the generators of `self` as a tuple.

EXAMPLES:

```python
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, None), generators=T.highest_weight_vectors())
sage: psi.im_gens()
(Non, [[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:**

```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.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])
```

### class `sage.categories.crystals.Crystals(s=None)`

**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
```

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:

```python
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']}
```

class ElementMethods

    Bases: object

    Epsilon()

    EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(0).Epsilon()
(0, 0, 0, 0, 0)
sage: C(1).Epsilon()
(0, 0, 0, 0, 0)
sage: C(2).Epsilon()
(1, 0, 0, 0, 0)
```

Phi()

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(0).Phi()
(0, 0, 0, 0, 0)
sage: C(1).Phi()
(1, 0, 0, 0, 0)
sage: C(2).Phi()
(1, 1, 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:
    • \index_set – (optional) a subset of the index set of \self

    EXAMPLES:

```python
sage: B = crystals.infinity.Tableaux("A2")
sage: b0 = B.highest_weight_vector()
sage: b = b0.f_string([1, 2, 1, 2])
sage: L = b.all_paths_to_highest_weight()
sage: list(L)
[[2, 1, 2, 1], [2, 2, 1, 1]]

sage: Y = crystals.infinity.GeneralizedYoungWalls(3)
sage: y0 = Y.highest_weight_vector()
sage: y = y0.f_string([0, 1, 2, 3, 2, 1, 0])
```

(continues on next page)
```
sage: list(y.all_paths_to_highest_weight())
[[0, 1, 2, 3, 2, 1, 0],
 [0, 1, 3, 2, 2, 1, 0],
 [0, 3, 1, 2, 2, 1, 0],
 [0, 3, 2, 1, 1, 0, 2],
 [0, 3, 2, 1, 1, 2, 0]]

sage: B = crystals.Tableaux("A3", shape=[4,2,1])
sage: b0 = B.highest_weight_vector()
sage: b = b0.f_string([1, 1, 2, 3])
sage: list(b.all_paths_to_highest_weight())
[[1, 3, 2, 1], [3, 1, 2, 1], [3, 2, 1, 1]]
```

**cartan_type()**

Returns the Cartan type associated to self

**EXAMPLES:**

```
sage: C = crystals.Letters(['A', 5])
sage: C(1).cartan_type()
['A', 5]
```

**e(i)**

Return $e_i$ of self if it exists or $\text{None}$ otherwise.

This method should be implemented by the element class of the crystal.

**EXAMPLES:**

```
sage: C = Crystals().example(5)
sage: x = C[2]; x
3
sage: x.e(1), x.e(2), x.e(3)
(None, 2, None)
```

**e_string(list)**

Applies $e_{i_1} \cdots e_{i_r}$ to self for list as $[i_1, \ldots, i_r]$

**EXAMPLES:**

```
sage: C = crystals.Letters(['A',3])
sage: b = C(3)
sage: b.e_string([2,1])
1
sage: b.e_string([1,2])
```

**epsilon(i)**

**EXAMPLES:**

```
sage: C = crystals.Letters(['A',5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```
\textbf{\texttt{f(i)}}

Return \(f_i\) of \texttt{self} if it exists or \texttt{None} otherwise.

This method should be implemented by the element class of the crystal.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = Crystals().example(5)
sage: x = C[1]; x
2
sage: x.f(1), x.f(2), x.f(3)
(None, 3, None)
\end{verbatim}

\textbf{\texttt{f_string(list)}}

Applies \(f_{i_1} \cdots f_{i_r}\) to self for \texttt{list} as \([i_1, ..., i_r]\)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A',3])
sage: b = C(1)
sage: b.f_string([1,2])
3
sage: b.f_string([2,1])
\end{verbatim}

\textbf{\texttt{index_set()}}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A',5])
sage: C(1).index_set()
(1, 2, 3, 4, 5)
\end{verbatim}

\textbf{\texttt{is_highest_weight(index_set=None)}}

Return \texttt{True} if \texttt{self} is a highest weight.

Specifying the option \texttt{index_set} to be a subset \(I\) of the index set of the underlying crystal, finds all highest weight vectors for arrows in \(I\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A',5])
sage: C(1).is_highest_weight()
True
sage: C(2).is_highest_weight()
False
sage: C(2).is_highest_weight(index_set = [2,3,4,5])
True
\end{verbatim}

\textbf{\texttt{is_lowest_weight(index_set=None)}}

Returns \texttt{True} if \texttt{self} is a lowest weight. Specifying the option \texttt{index_set} to be a subset \(I\) of the index set of the underlying crystal, finds all lowest weight vectors for arrows in \(I\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A',5])
sage: C(1).is_lowest_weight()
False
\end{verbatim}

(continues on next page)
sage: C(6).is_lowest_weight()
True
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
sage: C(2).phi(1)
0
```

**phi_minus_epsilon**(i)

Return $\varphi_i - \varepsilon_i$ 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
```

**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 = C(rows=[[1,2],[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]]
```

**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]], [[2, 4]], [[1, 3]], [[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]]]

\( \text{tensor}(\*) \)

Return the tensor product of self with the crystal elements els.

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
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

\( \text{to_highest_weight}(index\_set=None) \)

Return the highest weight element \( u \) and a list \( [i_1, ..., i_k] \) such that \( \text{self} = f_{i_1}...f_{i_k}u \), where \( i_1, ..., i_k \) are elements in index_set.

By default the index set is assumed to be the full index set of self.

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

**to_lowest_weight**(index_set=None)

Return the lowest weight element $u$ and a list $[i_1, ..., i_k]$ such that $self = e_{i_1}...e_{i_k}u$, where $i_1, ..., i_k$ are elements in index_set.

By default the index set is assumed to be the full index set of self.

EXAMPLES:

```
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]], []]
sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
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
```

**weight()**

Return the weight of this crystal element.

This method should be implemented by the element class of the crystal.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C(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:

```
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
```

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)}
```
an_element()

Returns an element of self

```
sage: C = crystals.Letters(['A', 5])
sage: C.an_element()
1
```

cartan_type()

Returns the Cartan type of the crystal

```
sage: C = crystals.Letters(['A',1])
sage: C.cartan_type()
['A', 1]
```

connected_components()

Return the connected components of self as subcrystals.

```
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]]]
```

connected_components_generators()

Return a tuple of generators for each of the connected components of self.

```
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_generators()
([[[1, 1], [2]], 1], [[1, 2], [2]], 1), [[1, 2], [3]], 1])
```

crystal_morphism(on_gens, codomain=None, cartan_type=None, index_set=None, generators=None, automorphism=None, virtualization=None, scaling_factors=None, category=None, check=True)

Construct a crystal morphism from self to another crystal codomain.

INPUT:

- `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 self under the crystal morphism
- `codomain` – (default: self) the codomain of the morphism
- `cartan_type` – (optional) the Cartan type of the morphism; the default is the Cartan type of self
- `index_set` – (optional) the index set of the morphism; the default is the index set of the Cartan type
• **generators** – (optional) the generators to define the morphism; the default is the generators of `self`  
• **automorphism** – (optional) the automorphism to perform the twisting  
• **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; the default is the identity dictionary  
• **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 $\varphi$; the default are all scaling factors to be one  
• **category** – (optional) the category for the crystal morphism; the default is the category of `Crystals`.  
• **check** – (default: `True`) check if the crystal morphism is valid

**See also:**

For more examples, see `sage.categories.crystals.CrystalHomset`.  

**EXAMPLES:**

We construct the natural embedding of a crystal using tableaux into the tensor product of single boxes via the reading word:

```python
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: mg = T.highest_weight_vectors()[2]; mg
[[[1]], [[2]], [[1]]]
sage: 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]]]
sage: b = B.module_generators[0]
sage: b.pp()  
1 1
2
sage: psi(b)
[[[1]], [[2]], [[1]]]
sage: psi(b.f(2))
[[[1]], [[3]], [[1]]]
sage: psi(b.f_string([2,1,1]))
[[[2]], [[3]], [[2]]]
sage: lw = b.to_lowest_weight()[0]
sage: lw.pp()  
2 3
3
sage: psi(lw)
[[[3]], [[3]], [[2]]]
sage: psi(lw) == mg.to_lowest_weight()[0]
True
```

We now take the other isomorphic highest weight component in the tensor product:
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]]
```

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(extended=True).fundamental_weights()
sage: B0 = crystals.GeneralizedYoungWalls(3, La[0])
sage: B1 = crystals.GeneralizedYoungWalls(3, La[1])
sage: phi = B0.crystal_morphism(B1.module_generators, automorphism={0:1, 1:2, 2:3, 3:0})
sage: phi
```

(continues on next page)
['A', 3, 1] Twisted Crystal morphism:
   From: Highest weight crystal of generalized Young walls of Cartan type ['A ˓→', 3, 1] and highest weight Lambda[0]
   To: Highest weight crystal of generalized Young walls of Cartan type ['A ˓→', 3, 1] and highest weight Lambda[1]
   Defn: [] |--> []

sage: x = B0.module_generators[0].f_string([0, 1, 2, 3]); x
[[0, 3], [1], [2]]
sage: phi(x)
[[], [1, 0], [2], [3]]

We construct a virtual crystal morphism from type $G_2$ into type $D_4$:

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
   1 |--> 2

   1
   2 |--> 3

   2
   3 |--> -3

   3
   0 |--> -3

   3
   -3 |--> -2

   -3
   -2 |--> -1

   -2
   -1 |--> -1

\textbf{digraph}(\texttt{subset=None, index\_set=None})

Return the \texttt{DiGraph} associated to \texttt{self}.

INPUT:

\begin{itemize}
\item \texttt{subset} – (optional) a subset of vertices for which the digraph should be constructed
\item \texttt{index\_set} – (optional) the index set to draw arrows
\end{itemize}

EXAMPLES:

sage: C = Crystals().example(5)
sage: 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: C = Crystals().example(3)
sage: G = C.digraph()
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external window)
```

One may also overwrite the colors:

```python
sage: C = Crystals().example(3)
sage: G = C.digraph()
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 = Crystals().example(4)
sage: G = C.digraph()
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
definitions...
sage: G.vertices(sort=True, key=str)
[(-Lambda[0] + 2*Lambda[1] - delta,),
  Lambda[1] - 1/2*delta),
  Lambda[1] - 1/2*delta),
```

Here is a way to construct a picture of a Demazure crystal using the subset option:

```python
definitions...
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)
```

We can also choose to display particular arrows using the index_set option:
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 = "\n ->" ];\n N_1 [ label = " ", texlbl = "$1$" ];\n N_2 [ label = " ", texlbl = "$2$" ];\n N_0 -> N_1 [ label = " ", texlbl = "1" ];\n N_1 -> 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_()
'\texttt{\ldots\texttt{tikzpicture\ldots}}'
sage: view(T) # optional - dot2tex graphviz, not tested (opens external window)

One can for example also color the edges using the following options:

sage: T = crystals.Tableaux(\['A',2\],shape=[1])
sage: T._latex_(color_by_label={0:"black", 1:"red", 2:"blue"})
'\texttt{\ldots\texttt{tikzpicture\ldots}}'

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:

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)

Use C.metapost("filename.mp",[options]), where options can be:

- thicklines = True (for thicker edges)
- labels = False (to suppress labeling of the vertices)
- scaling_factor=value, where value is a floating point number, 1.0 by default. Increasing or decreasing the scaling factor changes the size of the image. tallness=1.0. Increasing makes the image taller without increasing the width.

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

\[ a_2 \ a_3 \ a_4 \ a_1 \]

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 \( A_2 \), the pattern has the form
\[
a_3 \ a_2 \ a_1
\]
where applying \( e(1) \) \( a_1 \) times, \( e(2) \) \( a_2 \) times then \( e(3) \) \( a_1 \) times returns to the highest weight. These data determine the vertex and may be translated into a Gelfand-Tsetlin pattern or tableau.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 2])
sage: C.metapost(tmp_filename())
```

```python
sage: C = crystals.Letters(['A', 5])
sage: C.metapost(tmp_filename())
Traceback (most recent call last):
  ... 
NotImplementedError
```

### number_of_connected_components()

Return the number of connected components of \( \text{self} \).

**EXAMPLES:**

```python
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
```

### plot(**options)

Return the plot of \( \text{self} \) as a directed graph.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: print(C.plot())
Graphics object consisting of 17 graphics primitives
```

### plot3d(**options)

Return the 3-dimensional plot of \( \text{self} \) as a directed graph.

**EXAMPLES:**

```python
sage: C = crystals.KirillovReshetikhin(['A', 3, 1], 2, 1)
sage: print(C.plot3d())
Graphics3d Object
```

### subcrystal

Construct the subcrystal from \( \text{generators} \) using \( e_i \) and/or \( f_i \) for all \( i \) in \( \text{index_set} \).

**INPUT:**

- \( \text{index_set} \) – (default: None) the index set; if None then use the index set of the crystal
- \( \text{generators} \) – (default: None) the list of generators; if None then use the module generators of the crystal
- \( \text{max_depth} \) – (default: infinity) the maximum depth to build
category Framework, Release 10.1

- **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:**

```python
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]], [[1, 3]], [[2, 4]], [[1, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], max_depth=1))
[[[1, 4]], [[1, 3]], [[2, 4]], [[1, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], direction='upper'))
[[[1, 4]], [[1, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], direction='lower'))
[[[1, 4]], [[2, 4]]]
sage: G = C.subcrystal(index_set=[1,2,3]).digraph()
sage: GA = crystals.Tableaux('A3', shape=[2]).digraph()
sage: G.is_isomorphic(GA, edge_labels=True)
True
```

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]
[[[1, 3, 6], [2, 4, 5]]]
sage: S.module_generators[0].e(3).f(3)
[[[1, 2, 6], [3, 4, 5]]]
```
An example of a type $B_2$ virtual crystal inside of a type $A_3$ ambient crystal:

```python
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
```

tensor(*crystals, **options)

Return the tensor product of self with the crystals B.

EXAMPLES:

```python
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],
The infinity crystal of tableaux of type ['A', 3])
sage: tensor([C, C, B]) is T
True

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])
sage: 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]
```
TensorProducts()

Return the full subcategory of objects of self constructed as tensor products.

See also:

• tensor.TensorProductsCategory
• RegressiveCovariantFunctorialConstruction.

EXAMPLES:

```
sage: 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: Crystals().TensorProducts().extra_super_categories()
[Category of crystals]
```

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: 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.
```

super_categories()

EXAMPLES:

```
sage: Crystals().super_categories()
[Category of enumerated sets]
```

4.32 CW Complexes

class sage.categories.cw_complexes.CWComplexes(s=None)

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:

    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

class Connected(base_category)

    Bases: CategoryWithAxiom

    The category of connected CW complexes.

class ElementMethods

    Bases: object

    dimension()

        Return the dimension of self.

        EXAMPLES:

            sage: from sage.categories cw_complexes import CWComplexes
            sage: X = CWComplexes().example()
            sage: 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
```

**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
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
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.dimension()
2
```

```python
class SubcategoryMethods
    Bases: object
```

326 Chapter 4. Individual Categories
Connected()

Return the full subcategory of the connected objects of \texttt{self}.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().Connected()
Category of connected CW complexes
```

FiniteDimensional()

Return the full subcategory of the finite dimensional objects of \texttt{self}.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes().FiniteDimensional(); C
Category of finite dimensional CW complexes
```

super_categories()

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().super_categories()
[Category of topological spaces]
```

4.33 Discrete Valuation Rings (DVR) and Fields (DVF)

class \texttt{sage.categories.discrete_valuation.DiscreteValuationFields}(s=None)

Bases: \texttt{Category\_singleton}

The category of discrete valuation fields

EXAMPLES:

```
sage: Qp(7) in DiscreteValuationFields()  # optional - sage.rings.padics
True
sage: TestSuite(DiscreteValuationFields()).run()
```

class ElementMethods

Bases: \texttt{object}

valuation()

Return the valuation of this element.

EXAMPLES:

```
sage: x = Qp(5)(50)  # optional - sage.rings.padics
sage: x.valuation()  # optional - sage.rings.padics
2
```
class ParentMethods
    Bases: object

    residue_field()
    Return the residue field of the ring of integers of this discrete valuation field.

    EXAMPLES:
    sage: Qp(5).residue_field()  # optional - sage.rings.padics
    Finite Field of size 5
    sage: K.<u> = LaurentSeriesRing(QQ)
    sage: K.residue_field()
    Rational Field

    uniformizer()
    Return a uniformizer of this ring.

    EXAMPLES:
    sage: Qp(5).uniformizer()  #optional - sage.rings.padics
    5 + O(5^21)

    super_categories()
    EXAMPLES:
    sage: DiscreteValuationFields().super_categories()
    [Category of fields]

class sage.categories.discrete_valuation.DiscreteValuationRings(s=None)
    Bases: Category_singleton
    The category of discrete valuation rings

    EXAMPLES:
    sage: GF(7)['x'] in DiscreteValuationRings()  #optional - sage.rings.finite_rings
    True
    sage: TestSuite(DiscreteValuationRings()).run()

class ElementMethods
    Bases: object

    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:
sage: x = Zp(5)(50)  # optional - sage.rings.padics
sage: x.is_unit()  # optional - sage.rings.padics
False
sage: x = Zp(7)(50)  # optional - sage.rings.padics
sage: x.is_unit()  # optional - sage.rings.padics
True

\textbf{lcm}(\texttt{other})

Return the least common multiple of \texttt{self} and \texttt{other}, normalized so that it is a power of the distinguished uniformizer.

\textbf{quo_rem}(\texttt{other})

Return the quotient and remainder for Euclidean division of \texttt{self} by \texttt{other}.

\textbf{valuation}()

Return the valuation of this element.

\textbf{EXAMPLES:}

sage: x = Zp(5)(50)  # optional - sage.rings.padics
sage: x.valuation()  # optional - sage.rings.padics
2

\textbf{class ParentMethods}

\textbf{Bases: object}

\textbf{residue_field}()

Return the residue field of this ring.

\textbf{EXAMPLES:}

sage: Zp(5).residue_field()  # optional - sage.rings.padics
Finite Field of size 5
sage: K.<u> = QQ[[[]]

\textbf{uniformizer}()

Return a uniformizer of this ring.

\textbf{EXAMPLES:}

sage: Zp(5).uniformizer()  # optional - sage.rings.padics
5 + O(5^21)
sage: K.<u> = QQ[[[]]

sage: K.uniformizer()

u

super_categories()
EXAMPIES:

sage: DiscreteValuationRings().super_categories()
[Category of euclidean domains]

4.34 Distributive Magmas and Additive Magmas

class sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas(base_category):
   Bases: CategoryWithAxiom_singleton

   The category of sets $(S, +, *)$ with $*$ distributing on $+$.
   This is similar to a ring, but $+$ and $*$ are only required to be (additive) magmas.
   
   EXAMPLES:

   sage: from sage.categories.distributive_magmas_and_additive_magmas import *
   sage: C = DistributiveMagmasAndAdditiveMagmas(); C
   Category of distributive magmas and additive magmas
   sage: C.super_categories()
   [Category of magmas and additive magmas]

   class AdditiveAssociative(base_category):
      Bases: CategoryWithAxiom_singleton

   class AdditiveCommutative(base_category):
      Bases: CategoryWithAxiom_singleton

   class AdditiveUnital(base_category):
      Bases: CategoryWithAxiom_singleton

   class Associative(base_category):
      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:
```python
sage: C = (Magmas() & AdditiveMagmas()).Distributive().CartesianProducts()
sage: C.extra_super_categories()
[Category of distributive magmas and additive magmas]
sage: 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:

```python
sage: DivisionRings()
Category of division rings
sage: DivisionRings().super_categories()
[Category of domains]
```

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:

```python
sage: 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:

```python
sage: DivisionRings().Finite_extra_super_categories()
(Category of commutative magmas,)
sage: 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.
Category Framework, Release 10.1

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

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]
    sage: C is Rings().NoZeroDivisors()
    True

    Commutative
        alias of IntegralDomains

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

    super_categories()
    EXAMPLES:
    
    sage: 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 len on a list except that the return value is specified to be a Sage Integer or infinity, instead of a Python int.
• $iter(S)$: an iterator for the elements of the set;
• $S.list()$: a fresh list of the elements of the set, when possible; raises a 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 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:

• FiniteEnumeratedSets().example()
• InfiniteEnumeratedSets().example()

EXAMPLES:

```sage
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)

        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

    Finite
        alias of FiniteEnumeratedSets

    Infinite
        alias of InfiniteEnumeratedSets

    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

**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: P = Partitions()
    #~    optional - sage.combinat
sage: list(P.iterator_range(stop=5))
    #~    optional - sage.combinat
[[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(0, 5))
    #~    optional - sage.combinat
[[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(3, 5))
    #~    optional - sage.combinat
[[1, 1], [3]]
sage: list(P.iterator_range(3, 10))
    #~    optional - sage.combinat
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
sage: list(P.iterator_range(3, 10, 2))
    #~    optional - sage.combinat
[[1, 1], [2, 1], [4], [2, 2]]
sage: it = P.iterator_range(3)
    #~    optional - sage.combinat
sage: [next(it) for x in range(10)]
    #~    optional - sage.combinat
[[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)
    #~    optional - sage.combinat
sage: [next(it) for x in range(5)]
    #~    optional - sage.combinat
[[1, 1], [2, 1], [4], [2, 2], [1, 1, 1, 1]]
sage: next(P.iterator_range(stop=-3))
    #~    optional - sage.combinat
Traceback (most recent call last):
... Not Implemented: cannot list an infinite set
sage: next(P.iterator_range(start=-3))
    #~
```

(continues on next page)
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:

```python
sage: (GF(3)^2).list()  # optional - sage.libs.pari
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
```

```python
sage: R = Integers(11)
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
sage: l = R.list(); l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
sage: l.remove(0); l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.list()
[1, 2, 3]
```

map(f, name, is_injective=None)

Return the image \( \{f(x)|x \in \text{self}\} \) of this enumerated set by \( f \), as an enumerated set.

INPUT:

- \texttt{is\_injective} – boolean (default: True) whether to assume that \( f \) is injective.

EXAMPLES:

```python
sage: R = Compositions(4).map(attrcall('partial_sums')); R  # optional - sage.combinat
Image of Compositions of 4 by The map *.partial_sums() from Compositions of 4

sage: R.cardinality()  # optional - sage.combinat
8

sage: R.list()  # optional - sage.combinat
[[1, 2, 3, 4], [1, 2, 4], [1, 3, 4], [1, 4], [2, 3, 4], [2, 4], [3, 4], [4]]
```

```python
sage: [r for r in R]  # optional - sage.combinat
[[1, 2, 3, 4], [1, 2, 4], [1, 3, 4], [1, 4], [2, 3, 4], [2, 4], [3, 4], [4]]
```

```python
sage: R.category()  # optional - sage.combinat
```

(continues on next page)
Warning: If the function is not injective, then there may be repeated elements:

```
sage: P = Compositions(4)  # optional - sage.combinat
sage: P.list()              # optional - sage.combinat
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: P.map(attrcall('major_index')).list()  # optional - sage.combinat
[6, 3, 4, 1, 5, 2, 3, 0]
```

Pass `is_injective=False` to get a correct result in this case:

```
sage: P.map(attrcall('major_index'), is_injective=False).list()  # optional - sage.combinat
[6, 3, 4, 1, 5, 2, 0]
```

### 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 `obj`.

EXAMPLES:

```
sage: C = InfiniteEnumeratedSets().example()
sage: C._next_from_iterator(10) # indirect doctest
11
```

TODO: specify the behavior when `obj` is not in `self`.

### random_element() 

Return a random element in `self`.

Unless otherwise stated, and for finite enumerated sets, the probability is uniform.

This is a generic implementation from the category `EnumeratedSets()`. It raise a `NotImplementedError` since one does not know whether the set is finite.

EXAMPLES:

```
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
```
**rank**\( (x) \)

The rank of an element of \( self \)

\( self.\text{rank}(x) \) returns the rank of \( x \), that is its position in the enumeration of \( self \). This is an integer between 0 and \( n-1 \) where \( n \) is the cardinality of \( self \), or None if \( x \) is not in \( self \).

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 \( \text{obj} \). For infinite enumerated sets, this won’t terminate when \( x \) is not in \( self \).

**EXAMPL**ES:

```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 \texttt{EnumeratedSets()} which can be used when the method \texttt{__iter__} is provided. It returns an iterator for up to the first 100 elements of \( self \).

**EXAMPL**ES:

```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 \( x.\text{tuple()} \). Each following call of \( x.\text{tuple()} \) returns the same tuple.

For looping, it may be better to do \texttt{for e in x:, not for e in x.\text{tuple()}:\.}

If \( x \) is not known to be finite, then an exception is raised.

**EXAMPL**ES:

```python
sage: (GF(3)^2).tuple()  # optional - sage.libs.pari
((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
```

**unrank**\( (r) \)

The \( r \)-th element of \( self \)

\( self.\text{unrank}(r) \) returns the \( r \)-th element of \( self \), where \( r \) is an integer between 0 and \( n-1 \) where \( n \) is the cardinality of \( self \).
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}.

**EXAMPLES:**

```python
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
```

\texttt{\texttt{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}.

**See also:**

\texttt{unrank()} , \texttt{iterator\_range()} 

**EXAMPLES:**

```python
sage: P = Partitions()  # optional - sage.combinat
sage: P.unrank_range(stop=5)  # optional - sage.combinat
[[], [1], [2], [1, 1], [3]]
sage: P.unrank_range(0, 5)  # optional - sage.combinat
[[], [1], [2], [1, 1], [3]]
sage: P.unrank_range(3, 5)  # optional - sage.combinat
[[1, 1], [3]]
sage: P.unrank_range(3, 10)  # optional - sage.combinat
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
```

(continues on next page)
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:

sage: EnumeratedSets().super_categories()

[Category of sets]

4.38 Euclidean domains

AUTHORS:

• Teresa Gomez-Diaz (2008): initial version
• Julian Rueth (2013-09-13): added euclidean degree, quotient remainder, and their tests

class sage.categories.euclidean_domains.EuclideanDomains(s=None)
    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:

sage: EuclideanDomains()
Category of euclidean domains
sage: EuclideanDomains().super_categories()
[Category of principal ideal domains]

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) ≤ 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:

```sage
sage: R.<x> = QQ[]
sage: x.euclidean_degree()
1
sage: 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:

```sage
sage: R.<x> = PolynomialRing(QQ, sparse=True)
sage: EuclideanDomains().element_class.gcd(x,x+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:

```sage
sage: R.<x> = QQ[]
sage: x.quo_rem(x)
(1, 0)
```

**class** ParentMethods

**Bases:** object

**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].

EXEMPLARY:

```
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 is an object of the category of Euclidean domains.

EXEMPLARY:

```
sage: Parent(QQ,category=EuclideanDomains()).is_euclidean_domain()
True
```

`super_categories()`
EXEMPLARY:

```
sage: 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.

EXEMPLARY:

```
sage: K = Fields()
sage: K
Category of fields
sage: Fields().super_categories()
[Category of euclidean domains, Category of division 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
```

(continues on next page)
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:

        sage: QQ.one().euclidean_degree()
        0

    factor()

    Return a factorization of self.

    Since self is either a unit or zero, this function is trivial.

    EXAMPLES:

        sage: x = GF(7)(5)
        # optional - sage.rings.finite_rings
        sage: x.factor()
        # optional - sage.rings.finite_rings
        5
        sage: RR(0).factor()
        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:

        sage: K = GF(5)
        # optional - sage.rings.finite_rings
        sage: K(2).gcd(K(1))
        # optional - sage.rings.finite_rings
        1
        sage: K(0).gcd(K(0))
        # optional - sage.rings.finite_rings
        0
        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:

```python
sage: gcd(15.0, 12.0)
3.00000000000000
```

But for other floating point numbers, the gcd is just 0.0 or 1.0:

```python
sage: gcd(3.2, 2.18)
1.00000000000000
sage: gcd(0.0, 0.0)
0.000000000000000
```

AUTHOR:
- Simon King (2011-02) – github issue #10771
- Vincent Delecroix (2015) – github issue #17671

**inverse_of_unit()**

Return the inverse of this element.

**EXAMPLES:**

```python
sage: x = polygen(ZZ, 'x')
sage: NumberField(x^7 + 2, 'a')(2).inverse_of_unit() # optional - sage.rings.number_field
1/2
```

Trying to invert the zero element typically raises a `ZeroDivisionError`:

```python
sage: 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()
+infinity
```

**is_unit()**

Returns True if self has a multiplicative inverse.

**EXAMPLES:**

```python
sage: QQ(2).is_unit()
True
```

(continues on next page)
**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
```

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)
60.0000000000000
```

But for others floating point numbers, it is just 0.0 or 1.0:

```python
sage: lcm(3.2, 2.18)
1.00000000000000
sage: lcm(0.0, 0.0)
0.000000000000000
```

**AUTHOR:**

- Simon King (2011-02) – github issue #10771
- Vincent Delecroix (2015) – github issue #17671

**quo_rem**(other)

Return the quotient with remainder of the division of this element by other.

**INPUT:**

- other – an element of the field

**EXAMPLES:**

```python
sage: f, g = QQ(1), QQ(2)
sage: f.quo_rem(g)
(1/2, 0)
```

**xgcd**(other)

Compute the extended gcd of self and other.

**INPUT:**

- other – an element with the same parent as self
OUTPUT:

A tuple \((r, s, t)\) of elements in the parent of \(\text{self}\) such that \(r = s \times \text{self} + t \times \text{other}\). Since the computations are done over a field, \(r\) is zero if \(\text{self}\) and \(\text{other}\) are zero, and one otherwise.

AUTHORS:


EXAMPLES:

```python
sage: K = GF(5)  # optional - sage.rings.finite_rings
sage: K(2).xgcd(K(1))  # optional - sage.rings.finite_rings
(1, 3, 0)

sage: K(0).xgcd(K(4))  # optional - sage.rings.finite_rings
(1, 0, 4)

sage: K(1).xgcd(K(1))  # optional - sage.rings.finite_rings
(1, 1, 0)

sage: GF(5)(0).xgcd(GF(5)(0))  # optional - sage.rings.finite_rings
(0, 0, 0)
```

The \(\text{xgcd}\) 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 \(\text{gcd}\) on \(\mathbb{Z}\):

```python
sage: xgcd(12.0, 8.0)
(4.00000000000000, 1.00000000000000, -1.00000000000000)

sage: xgcd(3.1, 2.98714)
(1.00000000000000, 0.322580645161290, 0.000000000000000)

sage: xgcd(0.0, 1.1)
(1.00000000000000, 0.000000000000000, 0.909090909090909)
```

**Finite**

alias of `FiniteFields`

```python
class ParentMethods
    Bases: object

    fraction_field()
        Returns the fraction field of \(\text{self}\), which is \(\text{self}\).

    EXAMPLES:

    sage: QQ.fraction_field() is QQ
    True

    is_field(proof=True)
        Returns True as \(\text{self}\) is a field.

    EXAMPLES:
```

346 Chapter 4. Individual Categories
sage: QQ.is_field()  
True  
sage: 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:

sage: QQ.is_integrally_closed()  
True  
sage: QQbar.is_integrally_closed()  
...optional - sage.rings.number_field  
True  
sage: Z5 = GF(5); Z5  
...optional - sage.rings.finite_rings  
Finite Field of size 5  
sage: Z5.is_integrally_closed()  
...optional - sage.rings.finite_rings  
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:

sage: QQ.is_perfect()  
True  
sage: GF(2).is_perfect()  
...optional - sage.rings.finite_rings  
True  
sage: FunctionField(GF(2), 'x').is_perfect()  
...optional - sage.rings.finite_rings  
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: K.<a> = Qq(125)  
...optional - sage.rings.padics  
(continues on next page)
4.40 Filtered Algebras

class sage.categories.filtered_algebras.FilteredAlgebras(base_category)

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]

REFERENCES:

• Wikipedia article Filtered_algebra

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:

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

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
sage: C = AlgebrasWithBasis(ZZ).Filtered(); C
Category of filtered algebras with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of algebras with basis over Integer Ring,  
Category of filtered algebras over Integer Ring,  
Category of filtered modules with basis over Integer Ring]
```

```python
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 = \"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:

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
```

---

4.41. Filtered Algebras With Basis 349
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 gr A. The method to_graded_conversion() returns the canonical R-module isomorphism A → 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 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.)

EXEMPLARY:

```
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 RR^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 : self → other.

Let A and B be two filtered algebras with basis, and let (F_i)_{i∈I} and (G_i)_{i∈I} be their filtrations. Let f : A → B be a linear map which preserves the filtration (i.e., satisfies f(F_i) ⊆ G_i for all i ∈ I). Then, there is a canonically defined graded linear map gr f : gr A → gr B which satisfies

\[(gr f)(p_i(a)) = p_i(f(a)) \quad \text{for all } i ∈ I \text{ and } a ∈ F_i,\]

where the p_i on the left hand side is the canonical projection from F_i onto the i-th graded component of 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 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 \( \text{gr} f \).

**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 \( \mathbb{R}^3 \) 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)
sage: f(x)
U['y'] + 1
sage: f(x*y*z)
U['y']^2*U['z'] + U['y']*U['z']
sage: f(x*x*y*z)
U['y']^3*U['z'] + 2*U['x']*U['y']*U['z']
sage: f(A.one())
1
sage: 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 \( \text{gr} f \)

```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'])
```

(continues on next page)
Example 2.

We shall now construct $gr f$ 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: 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**y*y**z)
2*h[1, 1, 1, 1, 1] + h[2, 1, 1]
sage: f(A.one())
h[]
```

The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:
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: 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
    (continues on next page)
```
algebra with basis: the universal enveloping algebra of Lie algebra of \( \mathbb{R}^3 \) with cross product over Rational Field

```python
sage: grf(h[1])
bar(U['x']) + bar(U['y'])
sage: grf(h[2])
bar(U['x'])^2
sage: grf(h[1, 1])
bar(U['x'])^2 + bar(U['y'])^2
sage: grf(h[2, 2])
bar(U['x'])^4
sage: grf(h[3, 2, 1])
bar(U['x'])^6
sage: grf(h.one())
2*bar(1)
```

**Example 4.**

The construct \( \text{gr} f \) also makes sense when \( f \) is a filtration-preserving map between graded algebras.

```python
sage: def map_on_basis(lam):
....:     return h[lam] + h[len(lam)]
sage: f = h.module_morphism(on_basis=map_on_basis,
....:                        codomain=h)  # redefining f
sage: f(h[1])
2*h[1]
sage: f(h[2])
h[1] + h[2]
sage: f(h[1, 1])
h[1, 1] + h[2]
sage: f(h[2, 1])
h[2] + h[2, 1]
sage: f(h.one())
2*h[]
```

**Example 5.**

For another example, let us compute \( \text{gr} f \) for a map \( f \) between two Clifford algebras:

```python
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
```

(continues on next page)
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

projection(i)
Return the $i$-th projection $p_i : F_i \to G_i$ (in the notations of the class documentation AssociatedGradedAlgebra, 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 $A \to gr A$ with the canonical projection of the graded $R$-module $gr A$ onto its $i$-th graded component $G_i$. The codomain of this map is $gr A$, 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:

```
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 $A \to gr A$ induced by the basis of $A$ (where $A = \"self\$).

This is an isomorphism of $R$-modules, not of algebras. See the class documentation AssociatedGradedAlgebra.

See also:

from_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); 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
```

## 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 \texttt{gr} as a functor.

class \texttt{sage.categories.filtered_modules.FilteredModules}(\texttt{base\_category})

Bases: \texttt{FilteredModulesCategory}

The category of filtered modules over a given ring \( R \).

A \textit{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 \textit{filtration} of the given module \( M \).

EXAMPLES:

```python
sage: Modules(ZZ).Filtered()
Category of filtered modules over Integer Ring
sage: Modules(ZZ).Filtered().super_categories()
[Category of modules over Integer Ring]
```

REFERENCES:

- Wikipedia article Filtration (mathematics)

class \texttt{Connected}(\texttt{base\_category})

Bases: \texttt{CategoryWithAxiom\_over\_base\_ring}

class \texttt{SubcategoryMethods}

Bases: \texttt{object}

\texttt{Connected()}

Return the full subcategory of the connected objects of \texttt{self}.

A filtered \( R \)-module \( M \) with filtration \( (F_0, F_1, F_2, \ldots) \) (indexed by \( \mathbb{N} \)) is said to be \textit{connected} if \( F_0 \) is isomorphic to \( R \).

EXAMPLES:

```python
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
```

\texttt{extra\_super\_categories()}

Add \texttt{VectorSpaces} to the super categories of \texttt{self} if the base ring is a field.

EXAMPLES:

```python
sage: Modules(QQ).Filtered().is_subcategory(VectorSpaces(QQ))
True
sage: Modules(ZZ).Filtered().extra_super_categories()
[]
```

4.42. Filtered Modules
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'>
```

**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.

```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 $(b_j)_{j \in J}$ 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.
```

358 Chapter 4. Individual Categories
as homogeneouselements 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
sage: C = ModulesWithBasis(ZZ).Filtered(); C
Category of filtered modules with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of filtered modules over Integer Ring,
 Category of modules with basis over Integer Ring]
sage: C is ModulesWithBasis(ZZ).Filtered()
True
```

```python
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:**

```
An example in a graded algebra:

\begin{verbatim}
sage: S = NonCommutativeSymmetricFunctions(QQ).S() #
    # optional - sage.combinat sage.modules
sage: (x, y) = (S[2], S[3]) #
    # optional - sage.combinat sage.modules
sage: x.homogeneous_degree() #
    # optional - sage.combinat sage.modules
2
sage: (x^3 + 4*y^2).homogeneous_degree() #
    # optional - sage.combinat sage.modules
6
sage: ((1 + x)^3).homogeneous_degree() #
    # optional - sage.combinat sage.modules
Traceback (most recent call last):
... ValueError: element is not homogeneous
\end{verbatim}

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis):

\begin{verbatim}
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
\end{verbatim}

**degree_on_basis**(*m*)

Return the degree of the basis element indexed by *m* in *self*.

**EXAMPLES:**

\begin{verbatim}
sage: A = GradedModulesWithBasis(QQ).example() #
    # optional - sage.combinat sage.modules
sage: A.degree_on_basis(Partition((2,1))) #
    # optional - sage.combinat sage.modules
3
\end{verbatim}
homogeneous_component(n)

Return the homogeneous component of degree \( n \) of the element \( \text{self} \).

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:

```plaintext
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: 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)
```

(continues on next page)
sage: g.homogeneous_component(7)
U['x']^2*U['y']^2*U['z']^3
sage: g.homogeneous_component(8)
\@

**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:**

```
sage: A = ModulesWithBasis(ZZ).Filtered().example()
#optional - sage.combinat sage.modules
sage: x = A(Partition((3,2,1)))
#optional - sage.combinat sage.modules
sage: y = A(Partition((4,4,1)))
#optional - sage.combinat sage.modules
sage: z = A(Partition((2,2,2)))
#optional - sage.combinat sage.modules
sage: x.degree()
#optional - sage.combinat sage.modules
6
sage: (x + 2*z).degree()
#optional - sage.combinat sage.modules
6
sage: (y - x).degree()
#optional - sage.combinat sage.modules
Traceback (most recent call last):
...  
ValueError: element is not homogeneous
```

An example in a graded algebra:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
#optional - sage.combinat sage.modules
sage: (x, y) = (S[2], S[3])
#optional - sage.combinat sage.modules
sage: x.homogeneous_degree()
#optional - sage.combinat sage.modules
2
sage: (x^3 + 4*y^2).homogeneous_degree()
#optional - sage.combinat sage.modules
6
sage: ((1 + x)^3).homogeneous_degree()
#optional - sage.combinat sage.modules
Traceback (most recent call last):
...
Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis):

```sage
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
```

**is_homogeneous()**

Return whether the element `self` is homogeneous.

**EXAMPLES:**

```sage
sage: A = ModulesWithBasis(ZZ).Filtered().example()  # optional - sage.combinat
sage: x = A(Partition((3,2,1)))  # optional - sage.combinat
sage: y = A(Partition((4,4,1)))  # optional - sage.combinat
sage: z = A(Partition((2,2,2)))  # optional - sage.combinat
sage: (3*x).is_homogeneous()  # optional - sage.combinat
True
sage: (x - y).is_homogeneous()  # optional - sage.combinat
False
sage: (x+2*z).is_homogeneous()  # optional - sage.combinat
True
```

Here is an example with a graded algebra:

```sage
sage: S = NonCommutativeSymmetricFunctions(QQ).S()  # optional - sage.combinat
sage: (x, y) = (S[2], S[3])  # optional - sage.combinat
sage: (3*x).is_homogeneous()  # optional - sage.combinat
True
sage: (x^3 - y^2).is_homogeneous()  # optional - sage.combinat
True
sage: ((x + y)^2).is_homogeneous()  # optional - sage.combinat
True
```

(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()` method

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: A = ModulesWithBasis(ZZ).Filtered().example()  # optional - sage.combinat
-sage: x = A(Partition((3,2,1)))  # optional - sage.combinat
-sage: y = A(Partition((4,4,1)))  # optional - sage.combinat
-sage: z = A(Partition((2,2,2)))  # optional - sage.combinat
-sage: x.maximal_degree()  # optional - sage.combinat
6
-sage: (x + 2*z).maximal_degree()  # optional - sage.combinat
6
-sage: (y - x).maximal_degree()  # optional - sage.combinat
9
-sage: (3*z).maximal_degree()  # optional - sage.combinat
6
```

Now, we test this on a graded algebra:
Let us now test a filtered algebra:

\begin{verbatim}
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).maximal_degree() 2
sage: (y*x).maximal_degree() 2
sage: A.one().maximal_degree() 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() 1
\end{verbatim}

**truncate**\((n)\)

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:**

\begin{verbatim}
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: A = ModulesWithBasis(ZZ).Graded().example()
sage: x = A.an_element(); x
sage: x.truncate(0)
\end{verbatim}
sage: x.truncate(1)
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

class ParentMethods
Bases: object

basis(d=None)

Return the basis for (the d-th homogeneous component of) self.

INPUT:
• d – (optional, 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(1))_{i in Partitions of the integer 4}

Without arguments, the full basis is returned:
Checking this method on a filtered algebra. Note that this will typically raise a
NotImplementedError when this feature is not implemented.

From the documentation:

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
```

Without arguments, the full basis is returned:

```python
sage: A.basis()
```

An example with a graded algebra:

```python
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.basis()
```

See also:

to_graded_conversion()
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 $A \to \text{gr} A$ 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 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, 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).

**EXAMPLES:**

```python
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
```

homogeneous_component($d$)

Return the $d$-th homogeneous component of self.

**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
```

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}
sage: cat = GradedModulesWithBasis(ZZ)
sage: C = CombinatorialFreeModule(ZZ, ['a', 'b'], category=cat)
sage: C.degree_on_basis = lambda x: 1 if x == 'a' else 2
sage: C.homogeneous_component_basis(1)
Finite family {'a': B['a']}
sage: C.homogeneous_component_basis(2)
Finite family {'b': B['b']}
```

\textbf{induced\_graded\_map}(other, f)

Return the graded linear map between the associated graded modules of \texttt{self} and \texttt{other} canonically induced by the filtration-preserving map \(f: \texttt{self} \to \texttt{other}\).

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 \(grf: \text{gr} A \to \text{gr} B\) which satisfies

\[
(grf)(p_i(a)) = p_i(f(a)) \quad \text{for all } i \in I \text{ and } a \in F_i,
\]

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 \(\text{gr} B\).

\textbf{INPUT:}

- \texttt{other} – a filtered algebra with basis
- \texttt{f} – a filtration-preserving linear map from \texttt{self} to \texttt{other} (can be given as a morphism or as a function)

\textbf{OUTPUT:}

The graded linear map \(grf\).

\textbf{EXAMPLES:}

\textbf{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
An example of a filtered module with basis: the free module
on partitions over Rational Field
sage: M = A.indices(); M
Partitions
sage: p1, p2, p21, p321 = [A.basis()[Partition(i)] for i in [[1], [2], [2,1], [3,2,1]]]
```

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
```
We now compute \( \text{gr} f \)

```python
sage: grA = A.graded_algebra(); grA
#optional - sage.combinat
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)
#optional - sage.combinat
.... for i in [p1, p2, p21, p321]]
sage: pp2 + 4 * pp21
#optional - sage.combinat
Bbar[[2]] + 4*Bbar[[2, 1]]
sage: grf = A.induced_graded_map(A, f); grf
#optional - sage.combinat
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)
#optional - sage.combinat
Bbar[[1]]
sage: grf(pp2 + 4 * pp21)
#optional - sage.combinat
Bbar[[1, 1]] + 4*Bbar[[2, 1]]
```

Example 2.

We shall now construct \( \text{gr} f \) 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:

```python
sage: h = SymmetricFunctions(QQ).h()  # optional - sage.combinat
sage: def map_on_basis(lam):  # redefining map_on_basis  # optional - sage.combinat
    ...:     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,  # optional - sage.combinat
    ...:     codomain=h)  # redefining f
sage: f(p1)  # optional - sage.combinat
h[] + h[1]
sage: f(p2)  # optional - sage.combinat
h[] + h[1] + h[1, 1]
sage: f(A.zero())  # optional - sage.combinat
0
sage: f(p2 - 3*p1)  # optional - sage.combinat
-2*h[] - 2*h[1] + h[1, 1]
```

The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:

```python
sage: grh = h.graded_algebra(); grh is h  # optional - sage.combinat
True
sage: grf = A.induced_graded_map(h, f); grf  # optional - sage.combinat
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)  # optional - sage.combinat
h[1]
sage: grf(pp2)  # optional - sage.combinat
h[1, 1]
sage: grf(pp321)  # optional - sage.combinat
h[3, 2, 1]
sage: grf(pp2 - 3*pp1)  # optional - sage.combinat
-3*h[1] + h[1, 1]
sage: grf(pp21)  # optional - sage.combinat
```

(continues on next page)
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 $A$:

```python
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)])

sage: f = h.module_morphism(on_basis=map_on_basis, codomain=A) # redefining f

sage: f(h[1])
P[] + P[1]
sage: f(h[2])
P[] + 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: f(h.one())
P[]
```

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

```python
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
```

(continues on next page)
Example 4.

The construct $gr f$ also makes sense when $f$ is a filtration-preserving map between graded modules.

```python
sage: def map_on_basis(lam):  # redefining map_on_basis
    ....:     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 = h.module_morphism(on_basis=map_on_basis, codomain=h)  # redefining f
    ....:
    sage: f(h[1])  # optional - sage.combinat
    h[1] + h[1]
    sage: f(h[2])  # optional - sage.combinat
    h[1] + h[1] + h[1, 1]
    sage: f(h[1, 1])  # optional - sage.combinat
    h[1] + h[2]
    sage: f(h[2, 1])  # optional - sage.combinat
    sage: f(h.one())  # optional - sage.combinat
    h[]
    sage: grf = h.induced_graded_map(h, f); grf  # optional - sage.combinat
    Generic endomorphism of
    Symmetric Functions over Rational Field in the homogeneous basis
    sage: grf(h[1])  # optional - sage.combinat
    h[1]
    sage: grf(h[2])  # optional - sage.combinat
    h[1, 1]
```

(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 AssociatedGradedAlgebra, where \( A = \langle 'self' \rangle \)).

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:

```sage
sage: A = Modules(ZZ).WithBasis().Filtered().example() #←
    ...optional - sage.combinat
sage: p = -2 * A.an_element(); p #←
    ...optional - sage.combinat
sage: q = A.projection(2)(p); q #←
    ...optional - sage.combinat
-6*Bbar[[2]]
sage: q.parent() is A.graded_algebra() #←
    ...optional - sage.combinat
True
sage: A.projection(3)(p) #←
    ...optional - sage.combinat
0
```

to_graded_conversion()

Return the canonical \( R \)-module isomorphism \( A \to \text{gr} A \) induced by the basis of \( A \) (where \( A = \langle 'self' \rangle \)).

This is an isomorphism of \( R \)-modules. See the class documentation AssociatedGradedAlgebra.

See also:

\textit{from_graded_conversion()}

EXAMPLES:

```sage
sage: A = Modules(QQ).WithBasis().Filtered().example() #←
    ...optional - sage.combinat
sage: p = -2 * A.an_element(); p #←
    ...optional - sage.combinat
sage: q = A.to_graded_conversion()(p); q #←
    ...optional - sage.combinat
```

class Subobjects(category, *args)
    Bases: SubobjectsCategory

class ElementMethods
    Bases: object
    degree()
    Return the degree of self.

    EXAMPLES:

    sage: E.<x,y,z> = ExteriorAlgebra(QQ)  # optional - sage.combinat sage.modules
    sage: S = E.submodule([x + y, x*y - y*z, y])  # optional - sage.combinat sage.modules
    sage: B = S.basis()  # optional - sage.combinat sage.modules
    [x, y, x*y - y*z]
    sage: B[0].degree()  # optional - sage.combinat sage.modules
    1
    sage: B[1].degree()  # optional - sage.combinat sage.modules
    1
    sage: (B[0] + 3*B[1]).degree()  # optional - sage.combinat sage.modules
    1

    The degree of inhomogeneous elements is not defined (following the behavior of the exterior algebra):

    sage: (B[0] + B[2]).degree()  # optional - sage.combinat sage.modules
    Traceback (most recent call last):
    ...  
    ValueError: element is not homogeneous

    We can still get the maximal degree:

    sage: (B[0] + B[2]).maximal_degree()  # optional - sage.combinat sage.modules

    maximal_degree()
    The maximum of the degrees of the homogeneous components of self.

4.43. Filtered Modules With Basis
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
glob: E.<x,y,z> = ExteriorAlgebra(QQ)  #@
  @optional - sage.combinat sage.modules
glob: F = E.submodule([x + 1, x*y - 1])  #@
  @optional - sage.combinat sage.modules
glob: B = F.basis()  #@
  @optional - sage.combinat sage.modules
glob: [B[0].lift(), B[1].lift()]  #@
  @optional - sage.combinat sage.modules
[-x*y + 1, x*y + x]
glob: B[0].maximal_degree()  #@
  @optional - sage.combinat sage.modules
2
glob: B[1].maximal_degree()  #@
  @optional - sage.combinat sage.modules
2
```

```

class ParentMethods

Bases: object

degree_on_basis(m)

Return the degree of the basis element indexed by m in self.

EXAMPLES:

```python
glob: E.<x,y,z> = ExteriorAlgebra(QQ)  #@
  @optional - sage.combinat sage.modules
glob: S = E.submodule([x + y, x*y - y*z, y])  #@
  @optional - sage.combinat sage.modules
glob: B = S.basis()  #@
  @optional - sage.combinat sage.modules
  @optional - sage.combinat sage.modules
[x, y, x*y - y*z]
glob: S.degree_on_basis(0)  #@
  @optional - sage.combinat sage.modules
1
glob: S.degree_on_basis(1)  #@
  @optional - sage.combinat sage.modules
1
glob: S.degree_on_basis(2)  #@
  @optional - sage.combinat sage.modules
2
```
4.44 Finite Complex Reflection Groups

class sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups(base_category)
    Bases: CategoryWithAxiom

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()
- ComplexReflectionGroups.Finite.ElementMethods.to_matrix()

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  # optional - gap3
Reducible real reflection group of rank 4 and type A2 x B2
sage: W.reflections()  # optional - gap3
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:
sage: W = ColoredPermutations(1,3); W  # optional - sage.combinat sage.groups
1-colored permutations of size 3
sage: [t.character_value() for t in W]  # optional - 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:

sage: W = ReflectionGroup((1,1,3)); W  # optional - gap3
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  # optional - sage.combinat sage.groups
2-colored permutations of size 2
sage: [t.character_value() for t in W]  # optional - sage.combinat sage.groups
[2, 0, 0, -2, 0, 0, 0, 0]

sage: W = ColoredPermutations(3,1); W  # optional - sage.combinat sage.groups
3-colored permutations of size 1
sage: [t.character_value() for t in W]  # optional - sage.combinat sage.groups
[1, zeta3, -zeta3 - 1]

\texttt{reflection\_length(in\_unitary\_group=False)}

Return the reflection length of \texttt{self}.

This is the minimal numbers of reflections needed to obtain \texttt{self}.

INPUT:

* \texttt{in\_unitary\_group} – (default: \texttt{False}) if True, the reflection length is computed in the unitary group which is the dimension of the move space of \texttt{self}

EXAMPLES:

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((3,1,2))  # optional - gap3
sage: 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]
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], [ 0  1], [ 1  0], [-1 -1], [-1  0]]

sage: W = ColoredPermutations(1,3)  # optional - sage.combinat sage.groups
sage: [t.to_matrix() for t in W]     # optional - sage.combinat sage.groups
[[ 1  0  0]
 [ 0  1  0], [ 0  1  0], [ 1  0  0], [ 0  0  1], [ 0  0  1]]

A different representation is given by the colored permutations:

```python
sage: W = ColoredPermutations(3, 1)  # optional - sage.combinat sage.groups
sage: [t.to_matrix() for t in W]     # optional - sage.combinat sage.groups
[[1], [zeta3], [-zeta3 - 1]]
```

class Irreducible(base_category)

Bases: CategoryWithAxiom

class ParentMethods

 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

\[ \ell_R(w) = \dim\im(w - 1). \]

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 \( \text{gens} \) 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: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: sorted(W.reduced_word())     # optional - gap3
....:   for w in W.absolute_order_ideal()  
[[], [1], [1, 2], [1, 2, 1], [2]]

sage: sorted(W.reduced_word())     # optional - gap3
....:   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())     # optional - gap3
....:   for w in W.absolute_order_ideal(W.from_reduced_word([2])))  
[[], [2]]

sage: W = CoxeterGroup(['A', 3])  # optional - sage.combinat sage.groups
sage: len(list(W.absolute_order_ideal()))  # optional - sage.combinat sage.groups
14

sage: W = CoxeterGroup(['A', 2])  # optional - sage.combinat sage.groups
sage: for (w, l) in W.absolute_order_ideal(return_lengths=True):  # optional - 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 \texttt{self} as a finite lattice.

**INPUT:**

• **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 \( \dim(\text{Fix}(\sigma)) + \dim(\text{Fix}(\sigma^{-1}\tau)) = \dim(\text{Fix}(\tau)) \)

**See also:**

*noncrossing_partition_lattice()*

**EXAMPLES:**

```python
sage: P = ReflectionGroup((1,1,3)).absolute_poset(); P  # optional - gap3
```
Finite poset containing 6 elements

```
sage: sorted(w.reduced_word() for w in P) # optional - gap3
[[], [1], [1, 2], [1, 2, 1], [2], [2, 1]]
```

```
sage: W = ReflectionGroup(4); W # optional - gap3
Irreducible complex reflection group of rank 2 and type ST4
```

```
sage: W.absolute_poset() # optional - gap3
Finite poset containing 24 elements
```

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:

```
sage: W = ReflectionGroup(31) # optional - gap3
sage: W.coxeter_number() # optional - gap3
30
```

generalized_noncrossing_partitions($m$, $c=None$, positive=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
- positive – (default: False) if True, only those generalized noncrossing partitions of full support are returned

EXAMPLES:

```
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] # optional - gap3
....: for chain in chains)
[[[], [], [1, 2]],
[[], [1], [2]],
[[], [1, 2], []],
[[], [1, 2, 1], [1]],
[[], [2], [1, 2, 1]],
[[1], [], [2]]]
```

(continues on next page)
sage: chains = W.generalized_noncrossing_partitions(2,  # optional - gap3
positive=True)

sage: sorted([w.reduced_word() for w in chain]  # optional - gap3
for chain in chains)

[[[], [1, 2], []],
 [[]], [1, 2, 1], [1]],
 [[], [1]],
 [[1, 2], [], []],
 [[1], [2], []],
 [[1, 2, 1], [], [1]],
 [[1, 2, 1], [1], []],
 [[2], [], [1, 2, 1]],
 [[2], [1, 2, 1], []]]

noncrossing_partition_lattice(c=None, L=None, in_unitary_group=True)

Return the interval $[1, c]$ in the absolute order of self as a finite lattice.

See also:

absolute_order_ideal()

INPUT:

- c – (default: None) if an element $c$ in self is given, it is used as the maximal element in the interval

- L – (default: None) if a subset $L$ (must be hashable!) of self is given, it is used as the underlying set (only cover relations are checked).

- 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:

sage: W = SymmetricGroup(4)  # optional - sage.combinat sage.groups
sage: W.noncrossing_partition_lattice()  # optional - sage.combinat sage.groups
Finite lattice containing 14 elements

sage: W = WeylGroup(['G', 2])  # optional - sage.combinat sage.groups
sage: W.noncrossing_partition_lattice()  # optional - sage.combinat sage.groups
Finite lattice containing 8 elements
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

base_change_matrix()

Return the base change from the standard basis of the vector space of self to the basis given by the independent roots of 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]
sage: W = ReflectionGroup(23)  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[1 0 0]
[0 1 0]
[0 0 1]
sage: W = ReflectionGroup((3,1,2))  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[1 0]
```

(continues on next page)
sage: W = ReflectionGroup((4,2,2))  # optional - gap3
sage: W.base_change_matrix()  # optional - gap3
[ 1 0]
[E(4) 1]

cardinality()

Return the cardinality of self.

It is given by the product of the degrees of self.

EXAMPLES:

sage: W = ColoredPermutations(1,3)  # optional - sage.combinat sage.groups
sage: W.cardinality()  # optional - sage.combinat sage.groups
6
sage: W = ColoredPermutations(2,3)  # optional - sage.combinat sage.groups
sage: W.cardinality()  # optional - sage.combinat sage.groups
48
sage: W = ColoredPermutations(4,3)  # optional - sage.combinat sage.groups
sage: W.cardinality()  # optional - sage.combinat sage.groups
384
sage: W = ReflectionGroup((4,2,3))  # optional - gap3  # optional - sage.combinat sage.groups
sage: W.cardinality()  # optional - gap3  # optional - sage.combinat sage.groups
192

codegrees()

Return the codegrees of self.

OUTPUT: a tuple of Sage integers

EXAMPLES:

sage: W = ColoredPermutations(1,4)  # optional - sage.combinat sage.groups
sage: W.codegrees()  # optional - sage.combinat sage.groups
(2, 1, 0)
sage: W = ColoredPermutations(3,3)  # optional - sage.combinat sage.groups
sage: W.codegrees()  # optional - sage.combinat sage.groups
(6, 3, 0)
sage: W = ReflectionGroup(31)  # optional - gap3
sage: W.codegrees()  # optional - gap3
(28, 16, 12, 0)

**degrees()**

Return the degrees of `self`.

OUTPUT: a tuple of Sage integers

**EXAMPLES:**

sage: W = ColoredPermutations(1,4)  # optional - sage.combinat sage.groups
sage: W.degrees()  # optional - sage.combinat sage.groups
(2, 3, 4)

sage: W = ColoredPermutations(3,3)  # optional - sage.combinat sage.groups
sage: W.degrees()  # optional - sage.combinat sage.groups
(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: W = ColoredPermutations(1,3)  # optional - sage.combinat sage.groups
sage: W.is_real()  # optional - sage.combinat sage.groups
True

sage: W = ColoredPermutations(4,3)  # optional - sage.combinat sage.groups
sage: W.is_real()  # optional - sage.combinat sage.groups
False

**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)  # optional - sage.combinat sage.groups
sage: W.is_well_generated()          # optional - sage.combinat sage.groups
    True

sage: W = ColoredPermutations(4,3)  # optional - sage.combinat sage.groups
sage: W.is_well_generated()          # optional - sage.combinat sage.groups
    True

sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.is_well_generated()          # optional - gap3
    False

sage: W = ReflectionGroup((4,4,3))  # optional - gap3
sage: W.is_well_generated()          # optional - gap3
    True
```

**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:**
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))]
[0, 0, 1, 3, 6, 10, 15, 21]
```
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: W = ColoredPermutations(1,3) # optional - sage.combinat sage.groups
sage: W.rank() # optional - sage.combinat sage.groups
2
sage: W = ColoredPermutations(2,3) # optional - sage.combinat sage.groups
sage: W.rank() # optional - sage.combinat sage.groups
3
sage: W = ColoredPermutations(4,3) # optional - sage.combinat sage.groups
sage: W.rank() # optional - sage.combinat sage.groups
3
sage: W = ReflectionGroup((4,2,3)) # optional - gap3
sage: W.rank() # optional - gap3
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:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: 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
SymmetricGroup(3) in C
˓→optional - sage.groups
True
```

Note: The category of well generated finite complex reflection groups is currently implemented as an axiom. See discussion on github issue #11187. This may be a bit of overkill. Still it’s nice to have a full subcategory.

```python
class WellGenerated(base_category):
    Bases: CategoryWithAxiom

class Irreducible(base_category):
    Bases: CategoryWithAxiom
```

The category of finite irreducible well-generated finite complex reflection groups.

```python
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() for n in [3,4,5]] # optional - sage.combinat sage.groups
(continues on next page)```
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 self, and is equal to the highest degree of 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)  # optional - sage.combinat sage.groups
sage: W.coxeter_number()  # optional - sage.combinat sage.groups
3

fuss_catalan_number\((m, positive=False, polynomial=False)\)

Return the \(m\)-th Fuss-Catalan number associated to self.

This is defined by

\[
\prod_{i=1}^{n} \frac{d_i + mh}{d_i^2},
\]

where \(d_1, \ldots, d_n\) are the degrees and \(h\) is the Coxeter number.

INPUT:
* positive – optional boolean (default False) if True, return instead the positive Fuss-Catalan number
• 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 $\frac{1}{m+1} \binom{m+1}{n}$.
• The Fuss-Catalan numbers for $G(r,1,n)$ all coincide for $r > 1$.

EXAMPLES:

```
sage: W = ColoredPermutations(1,3)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[5, 12, 22]

sage: W = ColoredPermutations(1,4)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[14, 55, 140]

sage: W = ColoredPermutations(1,5)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[42, 273, 969]

sage: W = ColoredPermutations(2,2)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[6, 15, 28]

sage: W = ColoredPermutations(2,3)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[20, 84, 220]

sage: W = ColoredPermutations(2,4)
#optional - sage.combinat sage.groups
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[70, 495, 1820]
```

```
number_of_reflections_of_full_support()
Return the number of reflections with full support.

EXAMPLES:
```

sage: W = Permutations(4)
#optional - sage.combinat sage.groups
(continues on next page)
rational_catalan_number\( (p, \text{polynomial}=False) \)

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)  # optional - sage.combinat sage.groups
sage: [W.rational_catalan_number(p) for p in [5,7,8]]  # optional - sage.combinat sage.groups
[7, 12, 15]

example()

Return an example of an irreducible well-generated complex reflection group.

EXAMPLES:
```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated().Irreducible()
sage: C.example()  # optional - sage.combinat sage.groups
4-colored permutations of size 3
```

```python
class ParentMethods
    Bases: object

    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
easier 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:**

    ```python
    sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 2, 3, 4]
    sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 2, 3, 4]
    sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 2, 4, 3]
    sage: CoxeterGroup(['F', 4]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 2, 3, 4]
    sage: CoxeterGroup(['E', 8]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 3, 2, 4, 5, 6, 7, 8]
    sage: CoxeterGroup(['H', 3]).coxeter_element().reduced_word()  # optional - sage.combinat sage.groups
    [1, 2, 3]
    ```

    This method is also used for well generated finite complex reflection groups:

    ```python
    sage: W = ReflectionGroup((1,1,4))  # optional - gap3
    sage: W.coxeter_element().reduced_word()  # optional - gap3
    [1, 2, 3]
    sage: W = ReflectionGroup((2,1,4))  # optional - gap3
    sage: W.coxeter_element().reduced_word()  # optional - gap3
    [1, 2, 3, 4]
    ```
```
sage: W = ReflectionGroup((4,1,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3, 4]

sage: W = ReflectionGroup((4,4,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
[1, 2, 3, 4]

**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:**

```python
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: sorted(c.reduced_word() for c in W.coxeter_elements())
[[1, 2], [2, 1]]

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: sorted(c.reduced_word() 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:**

```python
sage: W = ReflectionGroup((3,1,2))  # optional - gap3
sage: W.is_well_generated()  # optional - gap3
True
```

**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:**
example()  
Return an example of a well-generated complex reflection group.

EXAMPLES:

```python
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 G(3,1,2)
```

example()  
Return an example of a complex reflection group.

EXAMPLES:

```python
sage: 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
```

### 4.45 Finite Coxeter Groups

```python
class sage.categories.finite_coxeter_groups.FiniteCoxeterGroups(base_category)
```

Bases: `CategoryWithAxiom`

The category of finite Coxeter groups.

EXAMPLES:

```python
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:

```python
sage: WeylGroups().Finite().example()
The symmetric group on \{0, \ldots, 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
```

```python
class ElementMethods
    Bases: object

    bruhat_upper_covers()

        Returns all the elements that cover self in Bruhat order.

        EXAMPLES:

        sage: W = WeylGroup(['A',4])
        sage: w = W.from_reduced_word([3,2])
        sage: print([v.reduced_word() for v in w.bruhat_upper_covers()])
        [[4, 3, 2], [3, 4, 2], [2, 3, 2], [3, 1, 2], [3, 2, 1]]

        sage: W = WeylGroup(['B',6])
        sage: w = W.from_reduced_word([1,2,1,4,5])
        sage: C = w.bruhat_upper_covers()
        sage: len(C)
        9
        sage: print([v.reduced_word() for v in C])
        [[6, 4, 5, 1, 2, 1], [4, 5, 6, 1, 2, 1], [3, 4, 5, 1, 2, 1], [2, 3, 4, 5, 1, 2, 1], [1, 2, 3, 4, 5, 1, 2, 1], [4, 5, 4, 1, 2, 1], [4, 5, 3, 1, 2, 1], [4, 5, 2, 3, 1, 2, 1], [4, 5, 1, 2, 3, 1, 2, 1]]

        sage: ww = W.from_reduced_word([5,6,5])
        sage: CC = ww.bruhat_upper_covers()
        sage: print([v.reduced_word() for v in CC])
        [[6, 5, 6, 5], [4, 5, 6, 5], [5, 6, 4, 5], [5, 6, 5, 4], [5, 6, 5, 3], [5, 6, 5, 2], [5, 6, 5, 1]]
```

Recursive algorithm: write \( w \) for \( self \). If \( i \) is a non-descent of \( w \), then the covers of \( w \) are exactly \( \{ ws_i, u_1 s_i, u_2 s_i, \ldots, u_j s_i \} \), where the \( u_k \) are those covers of \( ws_i \) 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:

```
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
 sage: len(W.long_element().covered_reflections_subgroup())
 24
 sage: s = W.simple_reflection(1)
 sage: Gs = s.covered_reflections_subgroup()
 sage: len(Gs)
 2
```

(continues on next page)
sage: s in [u.lift() for u in Gs]
    True
sage: len(W.one().covered_reflections_subgroup())
    1

coxeter_knuth_graph()

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:**

sage: W = WeylGroup(['A',4], prefix='s')
sage: w = W.from_reduced_word([1,2,1,3,2])
sage: D = w.coxeter_knuth_graph()
sage: 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)]
sage: 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)]
sage: w = W.from_reduced_word([1,3])
sage: D = w.coxeter_knuth_graph()
sage: D.vertices(sort=True)
[(1, 3), (3, 1)]
sage: D.edges(sort=False)
[]

coxeter_knuth_neighbor($w$)

Return the Coxeter-Knuth (oriented) neighbors of the reduced word $w$ of self.

**INPUT:**

- $w$ -- reduced word of 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:**

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)}
sage: word = [1,2,1,3,2,4,3]
sage: w = W.from_reduced_word(word)
\texttt{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)}

\textbf{is\_coxeter\_element()}

Return whether this is a Coxeter element.

This is, whether \texttt{self} has an eigenvalue $e^{2\pi i / h}$ where $h$ is the Coxeter number.

\textbf{See also:}
coxeter\_elements()

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{class ParentMethods}

\textbf{Bases: object}

Ambiguity resolution: the implementation of some\_elements is preferable to that of \texttt{FiniteGroups}. The same holds for \_\_iter\_, although a breadth first search would be more natural; at least this maintains backward compatibility after \texttt{github issue #13589}.

\textbf{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 succintly defined as follows.

Let $u$ and $v$ be two elements of the Coxeter group $W$. Let $S(u)$ be the support of $u$. Then $u \leq v$ if and only if $v_{S(u)} = u$ (here $v = v^t v_I$ denotes the usual parabolic decomposition with respect to the standard parabolic subgroup $W_I$).

\textbf{See also:}
bruhat\_poset(), shard\_poset(), weak\_poset()

\textbf{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()
\end{verbatim}
103

sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: len(P.maximal_elements())
13

bruhat_poset(facade=False)

Return the Bruhat poset of self.

See also:

bhz_poset(), shard_poset(), weak_poset()

EXAMPLES:

sage: W = WeylGroup(['A', 2])
sage: P = W.bruhat_poset()
sage: P
Finite poset containing 6 elements
sage: P.show()

Here are some typical operations on this poset:

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
sage: P.is_join_semilattice()
False

By default, the elements of $P$ are aware that they belong to $P$:

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: P = W.bruhat_poset(facade = True)
sage: P.an_element().parent()
Weyl Group of type ['A', 3] (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.
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 (optional, 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
```

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)
```

degrees()

Return the degrees of the Coxeter group.

The output is an increasing list of integers.

EXAMPLES:

```sage
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)

```python
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 \([w_0, w_1, ..., w_k]\), the output is \([w_0, w_0w_1w_0, ..., w_0w_1 \cdots w_kw_1 \cdots w_0]\).

**INPUT:**
- `word` – a word in the indices of the simple generators of `self`.

**EXAMPLES:**

```python
sage: CoxeterGroup(['A', 2]).inversion_sequence([1,2,1])
[[-1, 1], [0, -1], [1, 0]]
sage: 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]]
```

### is_real()

Return `True` since `self` is a real reflection group.

**EXAMPLES:**

```python
sage: CoxeterGroup(['F', 4]).is_real()
True
sage: CoxeterGroup(['H', 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: D10 = FiniteCoxeterGroups().example(10)
sage: D10.long_element()
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
sage: D10.long_element([1])
(1,)
sage: D10.long_element([2])
(2,)
sage: D10.long_element([])
()

sage: D7 = FiniteCoxeterGroups().example(7)
sage: D7.long_element()
(1, 2, 1, 2, 1, 2, 1)

One can require instead a reduced word for w0:

sage: A3 = CoxeterGroup(['A', 3])
sage: A3.long_element(as_word=True)
[1, 2, 1, 3, 2, 1]

m_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 (optional, default 1)
- on_roots (optional, 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: CoxeterGroup('A[3]').m_cambrian_lattice((1,2))
Finite lattice containing 5 elements
sage: CoxeterGroup('A[3]').m_cambrian_lattice((1,2),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,...))
- 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
sage: 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.
```

4.45. Finite Coxeter Groups
EXAMPLES:

```
sage: WeylGroup(['A',2]).reflections_from_w0()
[[0 1 0] [0 0 1] [1 0 0]
 [1 0 0] [0 1 0] [0 0 1]
 [0 0 1], [1 0 0], [0 1 0]]

sage: WeylGroup(['A',3]).reflections_from_w0()
[[0 1 0 0] [0 0 1 0] [1 0 0 0] [0 0 0 1] [1 0 0 0] [1 0 0 0]
 [1 0 0 0] [0 1 0 0] [0 0 1 0] [0 1 0 0] [0 0 0 1] [0 1 0 0]
 [0 0 1 0] [1 0 0 0] [0 1 0 0] [0 0 1 0] [0 0 0 1] [0 1 0 0]
 [0 0 0 1], [0 0 0 1], [0 0 0 1], [1 0 0 0], [0 1 0 0], [0 1 0]]
```

```
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 groups is available as `shard_poset()`.

See also:

`bhz_poset()`, `bruhat_poset()`, `weak_poset()`

EXAMPLES:

```
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()
q^3 - 11*q^2 + 23*q - 13
sage: 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:

```
sage: D8 = FiniteCoxeterGroups().example(8)
sage: D8.w0
(1, 2, 1, 2, 1, 2, 1, 2)
sage: D3 = FiniteCoxeterGroups().example(3)
```

(continues on next page)
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:

```
sage: W = WeylGroup(['A', 2])
sage: P = W.weak_poset()
sage: P
Finite lattice containing 6 elements
sage: 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
```

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
```

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
```

By default, the elements of \( P \) are aware of that they belong to \( P \):

```
sage: 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:
\begin{verbatim}
    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)
\end{verbatim}

See also:
\begin{itemize}
    \item Poset() for more on posets and facade posets.
\end{itemize}

Todo:
\begin{itemize}
    \item Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
    \item The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.
\end{itemize}

\textbf{weak_poset}(\texttt{side='right', facade=True})

\textbf{INPUT:}
\begin{itemize}
    \item \texttt{side} – “left”, “right”, or “twosided” (default: “right”)
    \item \texttt{facade} – a boolean (default: False)
\end{itemize}

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:
\begin{itemize}
    \item \texttt{bz_poset()}, \texttt{bruhat_poset()}, \texttt{shard_poset()}
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
    sage: W = WeylGroup(['A', 2])
    sage: P = W.weak_poset()
    sage: P
    Finite lattice containing 6 elements
    sage: 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

    so this method has an alias \texttt{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

    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:
\end{verbatim}
By default, the elements of $P$ are aware of that they belong to $P$:

```python
sage: P.is_lattice()
False
```

If instead one wants the elements to be plain elements of the Coxeter group, one can use the `facade` option:

```python
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.

```python
def extra_super_categories():
    EXAMPLES:
        sage: CoxeterGroups().Finite().super_categories()
        [Category of finite generalized coxeter groups,
        Category of coxeter groups]
```

### 4.46 Finite Crystals

```python
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
```

```python
class TensorProducts(category, *args):
    Bases: TensorProductsCategory

    The category of finite crystals constructed by tensor product of finite crystals.
```

4.46. Finite Crystals 407
extra_super_categories()

EXAMPLES:

```
sage: 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
```

extra_super_categories()

EXAMPLES:

```
sage: 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(base_category)

Bases: CategoryWithAxiom_over_base_ring

The category of finite dimensional algebras with a distinguished basis.

EXAMPLES:

```
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]
sage: C.example()
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
```

class Cellular(base_category)

Bases: CategoryWithAxiom_over_base_ring

Cellular algebras.
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

$$C : \prod_{\mu \in \Lambda} T(\mu) \times T(\mu) \longrightarrow A; (\mu, s, t) \mapsto c^\mu_{st}$$

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:

$$a c^\mu_{st} = \sum_{u \in T(\mu)} r_a(s, u)c^\mu_{ut} \pmod{A^{>\mu}},$$

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 $\iota : A \longrightarrow A; c^\mu_{st} \mapsto c^\mu_{ts}$ is an algebra anti-isomorphism.

A cellular basis for $A$ is any basis of the form $\{c^\mu_{st} \mid \mu \in \Lambda, s, t \in T(\mu)\}$.

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:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)
optional - sage.groups sage.modules
sage: elt = S([3,1,2,4])
optional - sage.groups sage.modules
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()
optional - sage.groups sage.modules
[3, 1, 2, 4]
```

class ParentMethods

Bases: object

4.47. Finite dimensional algebras with basis 409
cell_module(mu, **kwds)

Return the cell module indexed by mu.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: S.cell_module(Partition([2,1]))  # optional - 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
```

cell_module_indices(mu)

Return the indices of the cell module of self indexed by mu.

This is the finite set $M(\lambda)$.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: S.cell_module_indices([2,1])  # optional - sage.combinat sage.groups sage.modules
Standard tableaux of shape [2, 1]
```

cell_poset()

Return the cell poset of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)  # optional - sage.groups sage.modules
sage: S.cell_poset()  # optional - sage.combinat sage.groups sage.modules
Finite poset containing 5 elements
```

cells()

Return the cells of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: dict(S.cells())  # optional - sage.combinat 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)  # optional - sage.groups sage.modules
```
sage: S.cellular_basis()  # optional - sage.groups sage.modules
Cellular basis of Symmetric group algebra of order 3 over Rational Field

**cellular_involution(x)**

Return the cellular involution of x in self.

**EXAMPLES:**

```python
sage: S = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: for b in S.basis(): b, S.cellular_involution(b)
```

```
([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])
```

**simple_module_parameterization()**

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: S = SymmetricGroupAlgebra(QQ, 4)  # optional - sage.groups sage.modules
sage: S.simple_module_parameterization()
```

```
([4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1])
```

**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:**
Category Framework, Release 10.1

```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)  # optional - sage.groups sage.modules
sage: S3 = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: T = S2.tensor(S3)  # optional - sage.groups sage.modules
sage: T.cell_module_indices(((1,1), (2,1)))  # optional - sage.groups sage.modules
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:
```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)  # optional - sage.groups sage.modules
sage: S3 = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: T = S2.tensor(S3)  # optional - sage.groups sage.modules
sage: T.cell_poset()  # optional - sage.combinat sage.graphs sage.groups sage.modules
Finite poset containing 6 elements

cellular_involution()
Return the image of the cellular involution of the basis element indexed by i.

EXAMPLES:
```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)  # optional - sage.groups sage.modules
sage: S3 = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
sage: T = S2.tensor(S3)  # optional - sage.groups sage.modules
sage: for b in T.basis(): b, T.cellular_involution(b)  # optional - sage.groups sage.modules
(([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], [1, 2] # [2, 1, 3]),
([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]))
```
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: QS3 = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
   sage: a = QS3([2,1,3])  # optional - sage.groups sage.modules
   sage: a.to_matrix(side='left')  # optional - sage.groups sage.modules
   [[0 0 1 0 0 0]
   [0 0 0 1 0 0]
   [1 0 0 0 0 0]
   [0 0 0 0 1 0]
   [0 1 0 0 0 0]
   [0 0 0 1 0 0]]

   sage: a.to_matrix(side='right')  # optional - sage.groups sage.modules
   [[0 0 1 0 0 0]
   [0 0 0 1 0 0]
   [1 0 0 0 0 0]]

extra_super_categories()

   Tensor products of cellular algebras are cellular.

   EXAMPLES:
to_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: QS3 = SymmetricGroupAlgebra(QQ, 3)  #(optional - sage.groups sage.modules
sage: a = QS3([2,1,3])  #optional - sage.groups sage.modules
sage: a.to_matrix(side='left')  #optional - sage.groups sage.modules
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 0 0 1 0]
[0 0 1 0 0 0]
sage: a.to_matrix(side='right')  #optional - sage.groups sage.modules
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 1 0 0 0 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 0 0 1 0]
sage: a.to_matrix(base_ring=RDF, side='left')  #optional - sage.groups sage.modules
[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, ...
\begin{verbatim}
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.
\end{verbatim}

**ALGORITHM:**

The Cartan invariant matrix of $A$ is computed from the dimension of the summands of its Peirce decomposition.

**See also:**

- \texttt{peirce_decomposition()}
- \texttt{isotypic_projective_modules()}

**EXAMPLES:**

For a semisimple algebra, in particular for group algebras in characteristic zero, the Cartan invariants matrix is the identity:

\begin{verbatim}
sage: A3 = SymmetricGroup(3).algebra(QQ)
   # optional - sage.groups sage.modules
sage: A3.cartan_invariants_matrix()
   #optional - sage.groups sage.modules
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]
\end{verbatim}

For the path algebra of a quiver, the Cartan invariants matrix counts the number of paths between two vertices:
In the commutative case, the Cartan invariant matrix is diagonal:

```
sage: Z12 = Monoids().Finite().example(); Z12
An example of a finite multiplicative monoid: the integers modulo 12
sage: A = Z12.algebra(QQ)  # optional - sage.modules
sage: A.cartan_invariants_matrix()  # optional - sage.modules sage.rings.number_fields
[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:

```
sage: from sage.monoids.hecke_monoid import HeckeMonoid  # optional - sage.groups sage.modules
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)  # optional - sage.groups sage.modules
sage: A.cartan_invariants_matrix()  # optional - sage.groups sage.modules sage.rings.number_field
[1 0 0 0 0 0 0 0]
[0 2 1 0 1 1 0 0]
[0 1 1 0 1 0 0 0]
[0 0 0 1 0 1 1 0]
[0 1 1 0 1 0 0 0]
[0 1 0 1 0 2 1 0]
[0 0 0 1 0 1 1 0]
[0 0 0 0 0 0 0 1]
```

center()

Return the center of self.

See also:

center_basis()

EXAMPLES:

```
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
```
(continues on next page)
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]}
True
sage: [c.lift() for c in center.basis()]
[x + y]

The center of a semisimple algebra is semisimple:

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: 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.center_basis()
(x + y,)

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 $\overline{e}$ be an idempotent of $\overline{A}$, and $x$ any lift thereof in $A$. This returns an idempotent $e$ of $A$ such that $\pi(e) = \pi(x)$ and $e$ is a polynomial in $x$.

INPUT:
• $x$ – an element of $A$ that projects on an idempotent $π$ of the semisimple quotient of $A$. Alternatively one may give as input the idempotent $π$, 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:

```
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
Todo: Add some non trivial example
```

is_commutative()
Return whether self is a commutative algebra.

EXAMPLES:

```
sage: S4 = SymmetricGroupAlgebra(QQ, 4)  # optional - sage.groups sage.modules
sage: S4.is_commutative()  # optional - sage.groups sage.modules
False
sage: S2 = SymmetricGroupAlgebra(QQ, 2)  # optional - sage.groups sage.modules
sage: S2.is_commutative()  # optional - sage.groups sage.modules
True
```

is_identity_decomposition_into_orthogonal_idempotents()
Return whether 1 is a decomposition of the identity into orthogonal idempotents.

INPUT:
• 1 – a list or iterable of elements of self

EXAMPLES:

```
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: 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])
```

(continues on next page)
Here the idempotents do not sum up to 1:

```python
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x])
False
```

Here \(1 + x\) and \(-x\) are neither idempotent nor orthogonal:

```python
sage: A.is_identity_decomposition_into_orthogonal_idempotents([1 + x, -x])
False
```

With the algebra of the \(0\)-Hecke monoid:

```python
sage: from sage.monoids.hecke_monoid import HeckeMonoid
# optional - sage.groups
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)  # optional - sage.groups sage.modules
sage: idempotents = A.orthogonal_idempotents_central_mod_radical()  # optional - sage.groups sage.modules sage.rings.number_field
sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)  # optional - sage.groups sage.modules sage.rings.number_field
True
```

Here are some more counterexamples:

1. Some orthogonal elements summing to 1 but not being idempotent:

   ```python
   sage: 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()  
   ...:     # optional - sage.libs.pari
   sage: a, b = y, 1 - y  
   ...:     # optional - sage.libs.pari
   sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
```

(continues on next page)
For comparison:

```
sage: A = PQAlgebra(QQ, x**2 - x); y = A.x()
  # optional - sage.libs.pari
sage: a, b = y, 1-y
  # optional - sage.libs.pari
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, A.zero(), b))
# optional - sage.libs.pari
True
sage: A = PQAlgebra(QQ, x**3 - x**2 + x - 1); y = A.x()
  # optional - sage.libs.pari
sage: a = (y**2 + 1) / 2
  # optional - sage.libs.pari
sage: b = 1 - a
  # optional - sage.libs.pari
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
# optional - sage.libs.pari
True
```

2. Some idempotents summing to 1 but not orthogonal:

```
sage: R.<x> = PolynomialRing(GF(2))
  # optional - sage.rings.finite_rings
sage: A = PQAlgebra(GF(2), x)
  # optional - sage.rings.finite_rings
sage: a = A.one()
  # optional - sage.rings.finite_rings
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a,))
# optional - sage.rings.finite_rings
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, a, a))
# optional - sage.rings.finite_rings
False
```

3. Some orthogonal idempotents not summing to the identity:

```
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a,a))
# optional - sage.rings.finite_rings
False
sage: A.is_identity_decomposition_into_orthogonal_idempotents(())
# optional - sage.rings.finite_rings
False
```

\texttt{isotypic\_projective\_modules}(side='left')

Return the isotypic projective side self-modules.

Let $P_i$ be representatives of the indecomposable projective side-modules of this finite dimensional algebra $A$, and $S_i$ be the associated simple modules.
The regular side representation of $A$ can be decomposed as a direct sum $A = \bigoplus_i Q_i$ where each $Q_i$ is an isotypic projective module; namely $Q_i$ is the direct sum of $\dim S_i$ copies of the indecomposable projective module $P_i$. This decomposition is not unique.

The isotypic projective modules are constructed as $Q_i = e_i A$, where the $(e_i)_i$ is the decomposition of the identity into orthogonal idempotents obtained by lifting the central orthogonal idempotents of the semisimple quotient of $A$.

**INPUT:**

- `side` – ‘left’ or ‘right’ (default: ‘left’)

**OUTPUT:** a list of subspaces of `self`.

**EXAMPLES:**

```python
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: 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
sage: [[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`:

```python
sage: sum([Qi.dimension() for Qi in Q]) == A.dimension()  # optional - 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 $(\overline{f_i})_i$ 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 $\overline{f_k}$ and $f = \sum_{i<k} f_i$.

See [CR1962] for correctness and termination proofs.

See also:
Category Framework, Release 10.1

  central_orthogonal_idempotents()
- idempotent_lift()

EXAMPLES:

```python
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.orthogonal_idempotents_central_mod_radical()  # optional - sage.rings.number_field
1/2*B[9] - 1/2*B[3],
B[0])
sage: sum(idempotents) == 1  # optional - sage.rings.number_field
True
sage: all(e*e == e for e in idempotents)  # optional - sage.rings.number_field
True
sage: all(e*f == 0 and f*e == 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)  # optional - sage.rings.number_field
True
```

We construct orthogonal idempotents for the algebra of the 0-Hecke monoid:

```python
sage: from sage.monoids.hecke_monoid import HeckeMonoid  # optional - sage.groups
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)  # optional - sage.groups
```

(continues on next page)
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 \((e_i)_{i=0,...,n}\) of the algebra that sum to 1 (default: the idempotents returned by \(\text{orthogonal_idempotents_central_mod_radical()}\))
- **check** – (default: True) whether to check that the idempotents are indeed orthogonal and idempotent and sum to 1

OUTPUT:

A list of lists \(l\) such that \(l[i][j]\) is the subspace \(e_i A e_j\).

See also:

- \(\text{orthogonal_idempotents_central_mod_radical()}\)
- \(\text{cartan_invariants_matrix()}\)

EXAMPLES:

\[
\text{sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A}
\]
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
\[
\text{sage: A.orthogonal_idempotents_central_mod_radical()}
\]
# optional - sage.groups
\[
\text{sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)}
\]
# optional - sage.groups
\[
\text{True}
\]

\[
\text{peirce_decomposition(idempotents=None, check=True)}\]

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 \((e_i)_{i=0,...,n}\) of the algebra that sum to 1 (default: the idempotents returned by \(\text{orthogonal_idempotents_central_mod_radical()}\))
- **check** – (default: True) whether to check that the idempotents are indeed orthogonal and idempotent and sum to 1

OUTPUT:

A list of lists \(l\) such that \(l[i][j]\) is the subspace \(e_i A e_j\).

See also:

- \(\text{orthogonal_idempotents_central_mod_radical()}\)
- \(\text{cartan_invariants_matrix()}\)

EXAMPLES:

\[
\text{sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A}
\]
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
\[
\text{sage: A.orthogonal_idempotents_central_mod_radical()}
\]
# optional - sage.groups
\[
\text{x, y}
\]
\[
\text{sage: decomposition = A.peirce_decomposition(); decomposition}
\]
# optional - sage.groups
\[
\text{[[Free module generated by \{0\} over Rational Field,}
\]
\text{Free module generated by \{0, 1\} over Rational Field],}
\][Free module generated by \{\} over Rational Field,]
\text{Free module generated by \{\} over Rational Field]]
\[
\text{sage: [[x.lift() for x in decomposition[i][j].basis()]} for i in range(2)]
\]
# optional - sage.groups
\[
\text{[[[x], [a, b]],}
\][[], [y]]]
\]

We recover that the group algebra of the symmetric group \(S_4\) is a block matrix algebra:
sage: A = SymmetricGroup(4).algebra(QQ)  # optional - sage.groups sage.modules

sage: decomposition = A.peirce_decomposition()  # long time  # optional - sage.groups sage.modules sage.rings.number_field

sage: [[decomposition[i][j].dimension()  # long time (4s)  # optional - sage.groups sage.modules sage.rings.number_field
   ....: for j in range(len(decomposition))]
   ....: for i in range(len(decomposition))]

[[9, 0, 0, 0, 0],
 [0, 9, 0, 0, 0],
 [0, 0, 4, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]

The dimension of each block is $d^2$, where $d$ is the dimension of the corresponding simple module of $S_4$. The latter are given by:

sage: [p.standard_tableaux().cardinality() for p in Partitions(4)]  # optional - sage.combinat

[1, 3, 2, 3, 1]

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()
sage: idemp = A.orthogonal_idempotents_central_mod_radical()  # optional - sage.rings.number_field
sage: A.peirce_summand(idemp[0], idemp[1])  # optional - sage.rings.number_field

Free module generated by {0, 1} over Rational Field
sage: A.peirce_summand(idemp[1], idemp[0])  # optional - sage.rings.number_field

Free module generated by {} over Rational Field

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 $S_4$:

sage: A4 = SymmetricGroup(4).algebra(QQ)  # optional - sage.groups
sage: e = A4.central_orthogonal_idempotents()[2]  # optional - sage.groups sage.rings.number_field
sage: A4.peirce_summand(e, e)  # optional - sage.rings.number_field

(continues on next page)
optional - sage.groups sage.rings.number_field
Free module generated by \{0, 1, 2, 3\} over Rational Field

\textbf{principal\_ideal}(a, side='left')

Construct the side principal ideal generated by \(a\).

**EXAMPLES:**

In order to highlight the difference between left and right principal ideals, our first example deals with a non commutative algebra:

\begin{verbatim}
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: x, y, a, b = A.basis()

In this algebra, multiplication on the right by \(x\) annihilates all basis elements but \(x\):
\end{verbatim}

\begin{verbatim}
sage: x^x, y^x, a^x, b^x
(x, 0, 0, 0)
\end{verbatim}

so the left ideal generated by \(x\) is one-dimensional:

\begin{verbatim}
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]
\end{verbatim}

Multiplication on the left by \(x\) annihilates only \(x\) and fixes the other basis elements:

\begin{verbatim}
sage: x^x, x^y, x^a, x^b
(x, 0, a, b)
\end{verbatim}

so the right ideal generated by \(x\) is 3-dimensional:

\begin{verbatim}
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]
\end{verbatim}

See also:

- \textit{peirce\_summand}

\textbf{radical}()

Return the Jacobson radical of \textit{self}.

This uses \textit{radical\_basis}, whose default implementation handles algebras over fields of characteristic zero or fields of characteristic \(p\) in which we can compute \(x^{1/p}\).

See also:

- \textit{radical\_basis}, \textit{semisimple\_quotient}

**EXAMPLES:**
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

The radical is an ideal of \( A \), and thus a finite dimensional non unital associative algebra:

sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: radical in AssociativeAlgebras(QQ).WithBasis().FiniteDimensional()
True
sage: radical in Algebras(QQ)
False
sage: radical.dimension()
2
sage: radical.basis()
Finite family {0: B[0], 1: B[1]}

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: 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)
We construct the group algebra of the Klein Four-Group over the rationals:

```sage
sage: A = KleinFourGroup().algebra(QQ)  # optional - sage.groups sage.modules
```

This algebra belongs to the category of finite dimensional algebras over the rationals:

```sage
sage: A in Algebras(QQ).FiniteDimensional().WithBasis()  # optional - sage.groups sage.modules
True
```

Since the field has characteristic 0, Maschke’s Theorem tells us that the group algebra is semisimple. So its radical is the zero ideal:

```sage
sage: A in Algebras(QQ).Semisimple()  # optional - sage.groups sage.modules
True
sage: A.radical_basis()  # optional - sage.groups sage.modules
() 
```

Let’s work instead over a field of characteristic 2:

```sage
sage: A = KleinFourGroup().algebra(GF(2))  # optional - sage.groups sage.rings.finite_rings sage.modules
sage: A in Algebras(GF(2)).Semisimple()  # optional - sage.groups sage.rings.finite_rings sage.modules
False
sage: A.radical_basis()  # optional - sage.groups sage.rings.finite_rings 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`:

```sage
sage: class AnAlgebra(CombinatorialFreeModule):  # optional - sage.modules
    ....:     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())]
    ....:         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))
    ....:     def radical_basis(self):  # optional - sage.rings.finite_rings sage.modules
    ....:         CombinatorialFreeModule.__init__(self, F, basis_keys,
    ....:         category=Algebras(F).FiniteDimensional().WithBasis())
    ....:         return self.basis()[self.base_ring().one()]
    ....:         def product_on_basis(self, w1, w2):
    ....:             return self.from_vector(vector(w1*w2))
```

(continues on next page)
\begin{itemize}
\item \texttt{AnAlgebra(GF(49, 'a')).radical_basis()}  # ...
\item \texttt{(B[1] + B[xbar],)}
\item \texttt{sage: AnAlgebra(GF(49, 'a')).radical_basis()}  # ...
\end{itemize}

\textbf{semisimple_quotient()}

Return the semisimple quotient of \texttt{self}.

This is the quotient of \texttt{self} by its radical.

\textbf{See also:}
\texttt{radical()}

\textbf{EXAMPLES:}

\begin{itemize}
\item \texttt{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
\item \texttt{sage: a, b, x, y = sorted(A.basis())}
\item \texttt{sage: S = A.semisimple_quotient(); S}
Semisimple quotient of \texttt{A}.
\item \texttt{(xs + ys) * xs}

\texttt{B['x']}
\end{itemize}

Sanity check: the semisimple quotient of the $n$-th descent algebra of the symmetric group is of dimension the number of partitions of $n$:

\begin{itemize}
\item \texttt{sage: [ DescentAlgebra(QQ,n).B().semisimple_quotient().dimension() for n in range(6) ]}
\item \texttt{[1, 1, 2, 3, 5, 7]}
\item \texttt{sage: [Partitions(n).cardinality() for n in range(10)]}
\item \texttt{[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]}
\end{itemize}

\textbf{Todo:}
\begin{itemize}
\item Pickling by construction, as \texttt{A.semisimple_quotient()}?
\item Lazy evaluation of \texttt{_repr_}
\end{itemize}

\textbf{class SubcategoryMethods}

\textbf{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
```

### 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]
sage: C is Bialgebras(QQ).WithBasis().FiniteDimensional()
True
```

### 4.49 Finite dimensional coalgebras with basis

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]
sage: 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(base_category):
    Bases: CategoryWithAxiom_over_base_ring

    Category of finite dimensional graded Lie algebras with a basis.

    A grading of a Lie algebra \( g \) is a direct sum decomposition \( g = \bigoplus_i V_i \) 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 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: L = C.FiniteDimensional().Stratified().Nilpotent()  # optional - sage.combinat sage.modules
            ....: nilpotent=True, category=C)
            sage: L.homogeneous_component_as_submodule(2)  # optional - sage.combinat sage.modules
            Sparse vector space of degree 3 and dimension 1 over Rational Field
            Basis matrix:
            [0 0 1]

    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:

        sage: C = LieAlgebras(QQ).WithBasis().Graded().Stratified().FiniteDimensional()  # optional - sage.combinat sage.modules
        sage: C
        Category of finite dimensional stratified lie algebras with basis over Rational Field
A finite-dimensional stratified Lie algebra is nilpotent:

```python
sage: 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.
```

```python
EXAMPLES:

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)  # optional - sage.combinat sage.modules
sage: L.degree_on_basis(X.leading_support())  # optional - sage.combinat sage.modules
1
sage: X.degree()  # optional - sage.combinat sage.modules
1
sage: Y.degree()  # optional - sage.combinat sage.modules
1
sage: L[X, Y]  # optional - sage.combinat sage.modules
Z
sage: Z.degree()  # optional - sage.combinat sage.modules
2
```

### 4.51 Finite dimensional Hopf algebras with basis

```python
class sage.categories.finite_dimensional_hopf_algebras_with_basis.FiniteDimensionalHopfAlgebrasWithBasis(base_category):
    Bases: CategoryWithAxiom_over_base_ring
    The category of finite dimensional Hopf algebras with a distinguished basis.

EXAMPLES:

sage: FiniteDimensionalHopfAlgebrasWithBasis(QQ)  # fixme: Hopf should be capitalized
Category of finite dimensional hopf algebras with basis over Rational Field
sage: FiniteDimensionalHopfAlgebrasWithBasis(QQ).super_categories()
```

(continues on next page)
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(_base_category):
    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 github issue #16820.

```python
class ElementMethods
    Bases: object

    adjoint_matrix(sparse=False)

    Return the matrix of the adjoint action of self.
```

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: L.an_element().adjoint_matrix()  # optional - sage.combinat sage.modules
[0 0 0]
[0 0 0]
[0 0 0]

sage: L.an_element().adjoint_matrix(sparse=True).is_sparse()  # optional - sage.combinat sage.modules
True
```

```python
sage: L.<x,y> = LieAlgebra(QQ, {'x','y'}: {'x':1})

sage: x.adjoint_matrix()  # optional - sage.combinat sage.modules
[0 1]
[0 0]

sage: y.adjoint_matrix()  # optional - sage.combinat sage.modules
[-1 0]
[ 0 0]
```

[Category of hopf algebras with basis over Rational Field, Category of finite dimensional algebras with basis over Rational Field]
We verify that this forms a representation:

```
sage: sl3 = lie_algebras.sl(QQ, 3)  # optional - sage.combinat sage.modules
sage: e1, e2 = sl3.e(1), sl3.e(2)    # optional - sage.combinat sage.modules
sage: e12 = e1.bracket(e2)         # optional - sage.combinat sage.modules
sage: E1, E2 = e1.adjoint_matrix(), e2.adjoint_matrix()  # optional - sage.combinat sage.modules
sage: E1 * E2 - E2 * E1 == e12.adjoint_matrix()  # optional - sage.combinat sage.modules
True
```

**to_vector** *(order=None, sparse=False)*

Return the vector in \( g.\text{module}() \) corresponding to the element \( \text{self} \) of \( g \) (where \( g \) is the parent of \( \text{self} \)).

Implement this if you implement \( g.\text{module}() \). See \sage.categories.lie_algebras.LieAlgebras.module() \ for how this is to be done.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: L.an_element().to_vector()                                      # optional - sage.combinat sage.modules
(0, 0, 0)
sage: L.an_element().to_vector(sparse=True)                          # optional - sage.combinat sage.modules
(0, 0, 0)
sage: D = DescentAlgebra(QQ, 4).D()                                 # optional - sage.combinat sage.modules
sage: L = LieAlgebra(associative=D)                                 # optional - sage.combinat sage.modules
sage: L.an_element().to_vector()                                     # optional - sage.combinat sage.modules
(1, 1, 1, 1, 1, 1, 1, 1)
```

**Nilpotent**

alias of **FiniteDimensionalNilpotentLieAlgebrasWithBasis**

**class ParentMethods**

**Bases:** object

**as_finite_dimensional_algebra()**

Return \( \text{self} \) as a **FiniteDimensionalAlgebra**.

**EXAMPLES:**

```
sage: L = lie_algebras.cross_product(QQ)  # optional - sage.combinat sage.modules
sage: x, y, z = L.basis()                  # optional - sage.combinat sage.modules
```

(continues on next page)
optional - sage.combinat sage.modules

sage: F = L.as_finite_dimensional_algebra()

#>

optional - sage.combinat sage.modules

sage: X, Y, Z = F.basis()

#>

optional - sage.combinat sage.modules

sage: x.bracket(y)

#>

optional - sage.combinat sage.modules

Z

sage: X * Y

#>

optional - sage.combinat sage.modules

casimir_element(order=2, UEA=None, force_generic=False)

Return the Casimir element in the universal enveloping algebra of self.

A Casimir element of order $k$ is a distinguished basis element for the center of $U(g)$ 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 $U(g)$). When $g$ is a simple Lie algebra, then this spans $Z(U(g))_k$.

INPUT:

- order – (default: 2) the order of the Casimir element
- UEA – (optional) the universal enveloping algebra 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

ALGORITHM:

For the quadratic order (i.e., order=2), then this uses $K_{ij}$, 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 $g$. Otherwise this solves the system of equations

$$f^b_{aj} K^{c \cdots d} + f^c_{aj} K^{e \cdots d} \cdots + f^d_{aj} K^{b \cdots j}$$

for the symmetric tensor $K_{i_1 \cdots i_k}$, where $k$ is the order. This system comes from $[X_i, C^{(k)}] = 0$ with

$$C^{(k)} = \sum_{i_1, \ldots, i_k} K_{i_1 \cdots i_k} X_{i_1} \cdots X_{i_k}.$$
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:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example() # optional - sage.modules
sage: Z = L.center(); Z # optional - sage.modules
An example of a finite dimensional Lie algebra with basis: the 3-dimensional abelian Lie algebra over Rational Field
sage: Z.basis_matrix() # optional - sage.modules
[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()
An example of a finite dimensional Lie algebra with basis: the 3-dimensional abelian Lie algebra over Rational Field

```python
sage: S.basis_matrix()
```

```plaintext
[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: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.modules
sage: a, b, c = L.lie_algebra_generators()  # optional - sage.modules
sage: L.centralizer_basis([a + b, 2*a + c])  # optional - sage.modules

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

```python
sage: H = lie_algebras.Heisenberg(QQ, 2)  # optional - sage.combinat sage.modules
sage: H.centralizer_basis(H)  # optional - sage.combinat sage.modules

[z]
```

```python
sage: D = DescentAlgebra(QQ, 4).D()  # optional - sage.combinat sage.modules
sage: L = LieAlgebra(associative=D)  # optional - sage.combinat sage.modules
sage: L.centralizer_basis(L)  # optional - sage.combinat sage.modules

[\{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()  # optional - sage.combinat sage.modules

[\{D\},
D\{1\} + D\{1, 2\} + D\{2, 3\} + D\{3\},
D\{1, 2, 3\} + D\{1, 3\} + D\{2\}]
```
chevalley_eilenberg_complex(M=None, dual=False, sparse=True, ncpus=None)

Return the Chevalley-Eilenberg complex of self.

Let $\mathfrak{g}$ be a Lie algebra and $M$ be a right $\mathfrak{g}$-module. The Chevalley-Eilenberg complex is the chain complex on

$$C_*(\mathfrak{g}, M) = M \otimes \bigwedge^* \mathfrak{g},$$

where the differential is given by

$$d(m \otimes g_1 \wedge \cdots \wedge g_p) = \sum_{i=1}^{p} (-1)^{i+1} (mg_i) \otimes g_1 \wedge \cdots \wedge \hat{g_i} \wedge \cdots \wedge g_p + \sum_{1 \leq i < j \leq p} (-1)^{i+j} m \otimes [g_i, g_j] \wedge g_1 \wedge \cdots \wedge \hat{g_i} \wedge \cdots \wedge \hat{g_j} \wedge \cdots \wedge g_p.$$

**INPUT:**
- $M$ – (default: the trivial 1-dimensional module) the module $M$
- $dual$ – (default: False) if True, causes the dual of the complex to be computed
- $sparse$ – (default: True) whether to use sparse or dense matrices
- $ncpus$ – (optional) how many cpus to use

**EXAMPLES:**

```sage
sage: L = lie_algebras.sl(ZZ, 2)
# optional - sage.combinat sage.modules
sage: C = L.chevalley_eilenberg_complex(); C
# optional - sage.combinat sage.modules
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
# optional - sage.combinat sage.modules
0 <-- C_0 <------ C_1 <---- C_2 <-- 0
```

**REFERENCES:**
- Wikipedia article Lie_algebra_cohomology#Chevalley-Eilenberg_complex
- [Wei1994] Chapter 7

4.52. Finite Dimensional Lie Algebras With Basis
Todo: Currently this is only implemented for coefficients given by the trivial module $R$, where $R$ is the base ring and $gR = 0$ for all $g \in g$. Allow generic coefficient modules $M$.

cohomology ($\text{deg}=\text{None}, M=\text{None}, \text{sparse}=\text{True}, ncpus=\text{None}$)

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 $M$.

INPUT:
• $\text{deg}$ – the degree of the homology (optional)
• $M$ – (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:

```
sage: L = lie_algebras.so(QQ, 4)
```

```
sage: 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 0 over Rational Field,
 4: Vector space of dimension 0 over Rational Field,
 5: Vector space of dimension 1 over Rational Field}
```

```
sage: L = lie_algebras.Heisenberg(QQ, 2)
```

```
sage: 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 0 over Rational Field,
 4: Vector space of dimension 0 over Rational Field,
 5: Vector space of dimension 1 over Rational Field}
```

```
sage: d = {('x', 'y'): { 'y': 2}}
```

```
sage: L.<x,y> = LieAlgebra(ZZ, d)
```

```
sage: L.cohomology()
```

```
(continues on next page)
optional - sage.combinat sage.modules
{0: Z, 1: Z, 2: C2}

See also:
chevalley_eilenberg_complex()

REFERENCES:
• Wikipedia article Lie_algebra_cohomology

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:

```python
sage: H = lie_algebras.Heisenberg(QQ, 1)  # optional - sage.combinat sage.modules
sage: H.derivations_basis()  # optional - sage.combinat sage.modules

([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 the derivations of $\mathfrak{sl}_2$:

```python
sage: sl2 = lie_algebras.sl(QQ, 2)  # optional - sage.combinat sage.modules
sage: sl2.derivations_basis()  # optional - sage.combinat sage.modules

([ 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:

```python
sage: D = [sl2.module_morphism(matrix=M, codomain=sl2)  # optional - sage.combinat sage.modules
 ...... for M in sl2.derivations_basis()]

sage: all(d(a.bracket(b)) == d(a).bracket(b) + a.bracket(d(b))  # optional - sage.combinat sage.modules
 ...... for a in sl2.basis() for b in sl2.basis() for d in D)
True
```

REFERENCES:
Wikipedia article Derivation_(differential_algebra)

derived_series()
Return the derived series \((g^{(i)})\) of 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

\[
g \supseteq [g, g] \supseteq [[g, g], [g, g]] \supseteq [[[g, g], [g, g]], [[g, g], [g, g]]] \supseteq \cdots.
\]

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #...
...optional - sage.combinat sage.modules
sage: L.derived_series()  #...
...optional - sage.combinat sage.modules
(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: [])
```

```python
sage: L.<x,y> = LieAlgebra(QQ, {'x', 'y'}: {'x':1})  #...
...optional - sage.combinat sage.modules
sage: L.derived_series()  # todo: not implemented - #17416
...optional - sage.combinat sage.modules
(Lie algebra on 2 generators (x, y) over Rational Field,
Subalgebra generated of
Lie algebra on 2 generators (x, y) over Rational Field
with basis: (x),
Subalgebra generated of
Lie algebra on 2 generators (x, y) over Rational Field
with basis: ()
```

derived_subalgebra()
Return the derived subalgebra of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #...
...optional - sage.combinat sage.modules
sage: L.derived_subalgebra()  #...
...optional - sage.combinat sage.modules
(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:
sage: sl3 = LieAlgebra(QQ, cartan_type=[['A', 2]])  # optional - sage.combinat sage.modules
sage: sl3.derived_subalgebra()  # optional - sage.combinat sage.modules
Lie algebra of ['A', 2] in the Chevalley basis
sage: sl3 is sl3.derived_subalgebra()  # optional - sage.combinat sage.modules
True

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()  # optional - sage.modules
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u  # optional - sage.modules
(1, 0, 0)
sage: parent(u) is L  # optional - sage.modules
True

homology(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:
- deg – the degree of the homology (optional)
- M – (default: the trivial module) a right module of self
- sparse – (default: True) whether to use sparse matrices for the Chevalley-Eilenberg chain complex
- ncpus – (optional) how many cpus to use when computing the Chevalley-Eilenberg chain complex

EXAMPLES:

sage: L = lie_algebras.cross_product(QQ)  # optional - sage.combinat sage.modules
sage: L.homology()  # optional - 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)  # optional - sage.combinat sage.libs.pari sage.modules
sage: L.homology()  # optional - sage.combinat sage.libs.pari 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 1 over Finite Field of size 5,
  3: 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

\[
\text{sage: } d = \{(\text{'x'}, \text{'y'}): \{\text{'y'}: 2}\}\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } L.<x,y> = LieAlgebra(ZZ, d)\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } L.homology()\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\{0: \text{Z}, 1: \text{Z x C2}, 2: \emptyset}\}
\]

See also:

\text{chevalley_eilenberg_complex()}

\text{ideal(*gens, **kwds)}

Return the ideal of \text{self} generated by \text{gens}.

INPUT:

- \text{gens} – a list of generators of the ideal
- \text{category} – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

EXAMPLES:

\[
\text{sage: } H = \text{lie_algebras.Heisenberg(QQ, 2)}\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } p1,p2,q1,q2,z = H.basis()\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } I = H.ideal([p1-p2, q1-q2])\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } I.basis().list()\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
[\text{-p1 + p2, -q1 + q2, z}]\]
\[
\text{sage: } I.reduce(p1 + p2 + q1 + q2 + z)\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
2*p1 + 2*q1\]

Passing an extra category to an ideal:

\[
\text{sage: } L.<x,y,z> = \text{LieAlgebra(QQ, abelian=True)}\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } C = \text{LieAlgebras(QQ).FiniteDimensional().WithBasis()}\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } C = C.Subobjects().Graded().Stratified()\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } I = L.ideal(x, y, category=C)\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
\text{sage: } I.homogeneous_component_basis(1).list()\quad \#\text{-optional - sage.combinat sage.modules}
\]
\[
[x, y]\]

\text{inner_derivations_basis()}

Return a basis for the Lie algebra of inner derivations of \text{self} as matrices.

EXAMPLES:
is_abelian()

Return if self is an abelian Lie algebra.

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: L.is_abelian()  # optional - sage.combinat sage.modules
True
```

```sage
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  # optional - sage.combinat sage.modules
sage: L.is_abelian()  # optional - sage.combinat sage.modules
False
```

is_ideal(A)

Return if self is an ideal of A.

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: a, b, c = L.lie_algebra_generators()  # optional - sage.combinat sage.modules
sage: I = L.ideal([2*a - c, b + c])  # optional - sage.combinat sage.modules
sage: I.is_ideal(L)  # optional - sage.combinat sage.modules
True
```

```sage
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  # optional - sage.combinat sage.modules
sage: I.is_ideal(L)  # optional - sage.combinat sage.modules
True
```

```sage
sage: F = LieAlgebra(QQ, 'F', representation='polynomial')  # optional - sage.combinat sage.modules
sage: L.is_ideal(F)  # optional - sage.combinat sage.modules
Traceback (most recent call last):
```
... NotImplementedError: A must be a finite dimensional Lie algebra with basis

\textbf{is\_nilpotent()}  
Return if self is a nilpotent Lie algebra.  
A Lie algebra is nilpotent if the lower central series eventually becomes 0.  
EXAMPLES:

\begin{verbatim}
  sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
  sage: L.is_nilpotent()  # optional - sage.combinat sage.modules
  True
\end{verbatim}

\textbf{is\_semisimple()}  
Return if self is 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:

\begin{verbatim}
  sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
  sage: L.is_semisimple()  # optional - sage.combinat sage.modules
  False
\end{verbatim}

\textbf{is\_solvable()}  
Return if self is a solvable Lie algebra.  
A Lie algebra is solvable if the derived series eventually becomes 0.  
EXAMPLES:

\begin{verbatim}
  sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
  sage: L.is_solvable()  # optional - sage.combinat sage.modules
  True
\end{verbatim}

\textbf{killing\_form}(x, y)  
Return the Killing form on x and y, where x and y are two elements of self.  
The Killing form is defined as

\[ \langle x \mid y \rangle = \text{tr}(ad_x \circ ad_y) \]
EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: a, b, c = L.lie_algebra_generators()  # optional - sage.combinat sage.modules
sage: L.killing_form(a, b)  # optional - sage.combinat sage.modules
```

`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:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.modules
sage: L.killing_form_matrix()  # optional - sage.modules
[0 0 0]
[0 0 0]
[0 0 0]
```

`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 \(ad_x \circ ad_y\) in the basis of `self`.

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.modules
sage: a, b, c = L.lie_algebra_generators()  # optional - sage.modules
sage: L.killing_matrix(a, b)  # optional - sage.modules
[0 0 0]
[0 0 0]
[0 0 0]
```

(continues on next page)
sage: L.killing_matrix(y, x)  # optional - sage.combinat sage.modules
\begin{bmatrix}
0 & -1 \\
0 & 0
\end{bmatrix}

lower_central_series(submodule=False)

Return the lower central series \((g_i)_i\) of \texttt{self} where the rightmost \(g_k = g_{k+1} = \cdots\).

INPUT:

• submodule – (default: False) if True, then the result is given as submodules of \texttt{self}

We define the lower central series of a Lie algebra \(g\) recursively by

\[ g_0 := g \quad \text{and} \quad g_{k+1} = \left[ g, g_k \right] \]

and recall that \(g_k \supseteq g_{k+1}\). Alternatively we can express this as

\[ g \supseteq \left[ g, g \right] \supseteq \left[ \left[ g, g \right], g \right] \supseteq \cdots. \]

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
#optional - sage.combinat sage.modules
sage: L.derived_series()  #optional - sage.combinat sage.modules
(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:

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  #optional - sage.combinat sage.modules
sage: L.lower_central_series(submodule=True)  #optional - sage.combinat sage.modules
(Sparse vector space of dimension 2 over Rational Field,
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0])

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  #optional - sage.combinat sage.modules
sage: L.lower_central_series()  # todo: not implemented - #17416 #
#optional - sage.combinat sage.modules
(Lie algebra on 2 generators (x, y) over Rational Field,
Subalgebra generated of
Lie algebra on 2 generators (x, y) over Rational Field
with basis: (x,))

module(R=None)

Return a dense free module associated to \texttt{self} over \(R\).

EXAMPLES:
**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: L.<X,Y,Z,W> = LieAlgebra(QQ, {('X','Y'): {'Z': 1}},
                        ...)  #optional - sage.combinat sage.modules
....:
    ('X','Z'): {'W': 1}})
sage: K.<A,B> = LieAlgebra(QQ, abelian=True)
                        ...)  #optional - sage.combinat sage.modules
sage: L.morphism({X: A, Y: B})  #optional - sage.combinat sage.modules
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
```

The reverse map \(A \mapsto X, B \mapsto Y\) does not define a Lie algebra morphism, since \([A, B] = 0\), but \([X, Y] \neq 0\):

```python
sage: K.morphism({A:X, B: Y})  #optional - sage.combinat sage.modules
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 isn’t linear):
sage: R.<x> = ZZ[]  # optional - sage.combinat sage.modules
sage: K.<i> = NumberField(x^2 + 1)  # optional - sage.combinat sage.modules sage.rings.number_fields
sage: cc = K.hom([-i])  # optional - sage.combinat sage.modules sage.rings.number_fields
sage: L.<X,Y,Z,W> = LieAlgebra(K, {('X','Y'): {'Z': 1}, ...: ('X','Z'): {'W': 1}})  # optional - sage.combinat sage.modules sage.rings.number_fields
sage: M.<A,B> = LieAlgebra(K, abelian=True)  # optional - sage.combinat sage.modules sage.rings.number_fields
sage: phi = L.morphism({X: A, Y: B}, base_map=cc)  # optional - sage.combinat sage.modules sage.rings.number_fields
sage: phi(X)  # optional - sage.combinat sage.modules sage.rings.number_fields
A
sage: phi(i*X)  # optional - sage.combinat sage.modules sage.rings.number_fields
-i*A

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:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat sage.modules
sage: a,b,c = L.lie_algebra_generators()  # optional - sage.combinat sage.modules
sage: X = L.subalgebra([a, b+c])  # optional - sage.combinat sage.modules
sage: L.product_space(X)  # optional - sage.combinat sage.modules
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])  # optional - sage.combinat sage.modules
sage: X.product_space(Y)  # optional - sage.combinat sage.modules
An example of a finite dimensional Lie algebra with basis: the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix: []
sage: H = lie_algebras.Heisenberg(ZZ, 4)  # optional - sage.combinat sage.modules
sage: Hp = H.product_space(H, submodule=True).basis()  # optional - sage.combinat sage.modules
sage: [H.from_vector(v) for v in Hp]  # optional - sage.combinat sage.modules
(continues on next page)
Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: (x,)

sage: Lp = L.product_space(L)  # todo: not implemented - #17416
˓→optional - sage.combinat sage.modules

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: (x,)

sage: Lp.product_space(L)  # todo: not implemented - #17416
˓→optional - sage.combinat sage.modules

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: (x,)

sage: Lp.product_space(Lp)  # todo: not implemented - #17416
˓→optional - sage.combinat sage.modules

Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis: ()

\[ \text{quotient}(I, \text{names}=\text{None}, \text{category}=\text{None}) \]

Return the quotient of \text{self} by the ideal \( I \).

A quotient Lie algebra.

INPUT:

• \( I \) – 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:

sage: L.<X,Y,Z,W,U> = LieAlgebra(QQ, 2, step=3)
˓→optional - sage.combinat sage.modules

sage: E = L.quotient(U); E
˓→optional - sage.combinat sage.modules

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()
˓→optional - sage.combinat sage.modules

[\text{X, Y, Z, W}]

sage: E(X).bracket(E(Y))
˓→optional - sage.combinat sage.modules
Quotients when the base ring is not a field are not implemented:

```
sage: L = lie_algebras.Heisenberg(ZZ, 1)
#...
˓→ optional - sage.combinat sage.modules
sage: L.quotient(L.an_element())  #...
˓→ optional - sage.combinat sage.modules
Traceback (most recent call last):
...
NotImplementedError: quotients over non-fields not implemented
```

**structure_coefficients**(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:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #...
˓→ optional - sage.combinat sage.modules
sage: L.structure_coefficients()  #...
˓→ optional - sage.combinat sage.modules
Finite family {}
˓→ optional - sage.combinat sage.modules
sage: L.structure_coefficients(True)  #...
˓→ optional - sage.combinat sage.modules
Finite family {((0, 1): (0, 0, 0), (0, 2): (0, 0, 0), (1, 2): (0, 0, 0)}
```

```
sage: G = SymmetricGroup(3)  #...
˓→ optional - sage.combinat sage.modules
sage: S = GroupAlgebra(G, QQ)  #...
˓→ optional - sage.combinant sage.modules
sage: L = LieAlgebra(associative=S)  #...
˓→ optional - sage.combinat sage.modules
sage: L.structure_coefficients()  #...
˓→ optional - sage.combinat sage.modules
Finite family {((2,3), (1,2)): (1,2,3) - (1,3,2),
(t,2,3), (1,3)): -(1,2,3) + (1,3,2),
(1,2,3), (2,3)): -(1,2) + (1,3),
(continues on next page)
\[(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) \}

\textbf{subalgebra}(*gens, **kwds*)

Return the subalgebra of \textit{self} generated by \textit{gens}.

\textbf{INPUT:}

- \textit{gens} – a list of generators of the subalgebra
- \textit{category} – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

\textbf{EXAMPLES:}

\begin{verbatim}
sage: H = lie_algebras.Heisenberg(QQ, 2) #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: p1, p2, q1, q2, z = H.basis() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: S = H.subalgebra([p1, q1]) #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: S.basis().list() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
[p1, q1, z]
sage: S.basis_matrix() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 1]
\end{verbatim}

Passing an extra category to a subalgebra:

\begin{verbatim}
sage: L = LieAlgebra(QQ, 3, step=2) #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: x, y, z = L.homogeneous_component_basis(1) #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: C = C.Subobjects().Graded().Stratified() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: S = L.subalgebra([x, y], category=C) #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
sage: S.homogeneous_component_basis(2).list() #
   \hspace{1em} \textit{optional} - \textit{sage.combinat sage.modules}
[X_12]
\end{verbatim}

\textbf{universal_commutative_algebra}()

Return the universal commutative algebra associated to \textit{self}.

Let \( I \) be the index set of the basis of \textit{self}. Let \( \mathcal{P} = \{ P_{a_{ij}} \}_{a,i,j \in I} \) denote the universal polynomials of a Lie algebra \( L \). The \textit{universal commutative algebra} associated to \( L \) is the quotient ring \( R[ X_{ij} ]_{i,j \in I} / (\mathcal{P}) \).

\textbf{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

$$P_{aij} = \sum_{u \in I} \tau_{ij}^u X_{au} - \sum_{s,t \in I} \tau^a_{st} X_{si} X_{tj},$$

where $a, i, j \in I$.

REFERENCES:
• [AM2020]

EXAMPLES:

\begin{Verbatim}
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  
    #_  
    --optional - sage.combinat sage.modules  
sage: A = L.universal_commutative_algebra()  
    #_  
    --optional - sage.combinat sage.modules  
sage: a, b, c, d = A.gens()  
    #_  
    --optional - sage.combinat sage.modules  
sage: a, b, c, d
(X00bar, X01bar, @, X11bar)

sage: a*d - a
0

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

$$P_{aij} = \sum_{u \in I} \tau_{ij}^u X_{au} - \sum_{s,t \in I} \tau^a_{st} X_{si} X_{tj},$$

where $a, i, j \in I$.

REFERENCES:
• [AM2020]

EXAMPLES:

\begin{Verbatim}
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})  
    #_  
    --optional - sage.combinat sage.modules  
sage: L.universal_polynomials()  
    #_  
    --optional - sage.combinat sage.modules  
Finite family {('x', 'x', 'y'): X01*X10 - X00*X11 + X00,  
    ('y', 'x', 'y'): X10}

sage: L = LieAlgebra(QQ, cartan_type=['A',1])  
    #_  
    --optional - sage.combinat sage.modules  
sage: list(L.universal_polynomials())  
    #_  
    --optional - sage.combinat sage.modules  
[-2*X01*X10 + 2*X00*X11 - 2*X00,  
  -2*X02*X10 + 2*X00*X12 + X01,  
  -2*X02*X11 + 2*X01*X12 - 2*X02,  
  X01*X20 - X00*X21 - 2*X10,  
  X02*X20 - X00*X22 + X11,  
  X02*X21 - X01*X22 - 2*X12,  
  -2*X11*X20 + 2*X10*X21 - 2*X20,  
  -2*X12*X20 + 2*X10*X22 + X21,  
  -2*X12*X21 + 2*X11*X22 - 2*X22]

sage: L = LieAlgebra(QQ, cartan_type=['B', 2])  
    #_  
    --optional - sage.combinat sage.modules  
sage: al = RootSystem(['B', 2]).root_lattice().simple_roots()  
    #_  
    --optional - sage.combinat sage.modules  
sage: k = list(L.basis().keys())[0]
\end{Verbatim}
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:

            sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
            sage: L = C.example()                      # long time
            sage: a, b, c = L.lie_algebra_generators()  # long time
            sage: S = L.subalgebra([2*a + b, b + c])    # long time
            sage: S.ambient() == L                      # long time
            True

        basis_matrix()
            Return the basis matrix of self.

            EXAMPLES:

            sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
            sage: L = C.example()                      # long time
            sage: a, b, c = L.lie_algebra_generators()  # long time
            sage: S = L.subalgebra([2*a + b, b + c])    # long time
            sage: S.basis_matrix()                     # long time
            [ 1 0 -1/2]
            [ 0 1 1]

        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()  # optional - 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)  # optional - 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: v = vector([0, -1, -3])  # optional - sage.modules
sage: v.dense_coefficient_list()  # optional - sage.modules
[0, -1, -3]
sage: v.dense_coefficient_list([2,1,0])  # optional - sage.modules
[-3, -1, 0]
sage: sorted(v.coefficients())  # optional - sage.modules
(continues on next page)
```
class MorphismMethods

    Bases: object

    image()

    Return the image of self as a submodule of the codomain.

    EXAMPLES:

    sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
doctest::
      :option: +optional

    sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # optional - sage.groups sage.modules
    sage: f.image()  # optional - 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)  # optional - sage.groups sage.modules
    sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # optional - sage.groups sage.modules
    sage: f.image_basis()  # optional - sage.groups sage.modules
    ([1, 2, 3], [2, 3, 1], [3, 1, 2])

    kernel()

    Return the kernel of self as a submodule of the domain.

    EXAMPLES:

    sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # optional - sage.groups sage.modules
    sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # optional - sage.groups sage.modules
    sage: K = f.kernel()  # optional - sage.groups sage.modules
    sage: K  # optional - sage.groups sage.modules
    Free module generated by {0, 1, 2} over Rational Field
    sage: K.ambient()  # optional - sage.groups sage.modules
    Symmetric group algebra of order 3 over Rational Field

    kernel_basis()

    Return a basis of the kernel of self in echelon form.

    EXAMPLES:
sage: SGA = SymmetricGroupAlgebra(QQ, 3)  # requires sage.groups sage.modules
sage: f = SGA.module_morphism(lambda x: SGA(x**2), codomain=SGA)  # requires sage.groups sage.modules
sage: f.kernel_basis()  # requires sage.groups sage.modules
([[1, 2, 3] - [3, 2, 1], [1, 3, 2] - [3, 2, 1], [2, 1, 3] - [3, 2, 1])

matrix(base_ring=None, side='left')

Return the matrix of this morphism in the distinguished bases of the domain and codomain.

INPUT:
- base_ring – a ring (default: 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: X = CombinatorialFreeModule(ZZ, [1,2]); x = X.basis()  # requires sage.modules
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); y = Y.basis()  # requires sage.modules

sage: phi.matrix()  # requires sage.modules
[[1 2]
 [3 5]]
sage: phi.matrix(side="right")  # requires sage.modules
[[1 3]
 [2 5]]

sage: phi.matrix().parent()  # requires sage.modules
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: phi.matrix(QQ).parent()  # requires sage.modules
Full MatrixSpace of 2 by 2 dense matrices over Rational Field

The resulting matrix is immutable:

sage: phi.matrix().is_mutable()  # requires sage.modules
False

The zero morphism has a zero matrix:
Todo: Add support for morphisms where the codomain has a different base ring than the domain:

```
sage: Y = CombinatorialFreeModule(QQ, [3,4]); y = Y.basis()  #
...:      codomain=Y).parent()  # todo: not implemented  #
```

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 \( \mathbb{Z} \) and over \( \mathbb{Q} \)):

```
sage: phi.parent().homset_category()  #
...:  Category of commutative additive semigroups
sage: phi.parent().homset_category()  # todo: not implemented  #
...:  Category of finite dimensional modules with basis over Integer Ring
```

```python
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 \( s \in S \). 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.
```
EXEMPLARY:

\begin{verbatim}
sage: F = FiniteDimensionalAlgebrasWithBasis(QQ).example(); F
#optional - sage.modules
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()
#optional - sage.modules
sage: A = F.annihilator([a + 3*b + 2*y]); A
#optional - sage.modules
Free module generated by {0} over Rational Field
sage: [b.lift() for b in A.basis()]
#optional - sage.modules
[-1/2*a - 3/2*b + x]
\end{verbatim}

The category can be used to specify other properties of this subspace, like that this is a subalgebra:

\begin{verbatim}
sage: center = F.annihilator(F.basis(), F.bracket,
....:   category=Algebras(QQ).Subobjects())
sage: (e,) = center.basis()
#optional - sage.modules
e.lift()
#optional - sage.modules
x + y
e * e == e
#optional - sage.modules
True
\end{verbatim}

Taking annihilator is order reversing for inclusion:

\begin{verbatim}
sage: A = F.annihilator([]); A .rename("A")
#optional - sage.modules
sage: Ax = F.annihilator([x]); Ax .rename("Ax")
#optional - sage.modules
sage: Ay = F.annihilator([y]); Ay .rename("Ay")
#optional - sage.modules
sage: Axy = F.annihilator([x,y]); Axy.rename("Axy")
#optional - sage.modules
sage: P = Poset(([A, Ax, Ay, Axy], attrcall("is_submodule")))
#optional - sage.combinat sage.graphs sage.modules
sage: sorted(P.cover_relations(), key=str)
#optional - sage.combinat sage.graphs sage.modules
[[Ax, A], [Axy, Ax], [Axy, Ay], [Ay, A]]
\end{verbatim}

\texttt{annihilator\_basis}(S, action=<built-in function \texttt{mul}>, side='right')

Return a basis of the annihilator of a finite set of elements.

\textbf{INPUT:}

- \texttt{S} – a finite set of objects
- \texttt{action} – a function (default: \texttt{operator.mul})
- \texttt{side} – ‘left’ or ‘right’ (default: ‘right’): on which side of \texttt{self} the elements of \texttt{S} acts.

See \texttt{annihilator()} for the assumptions and definition of the annihilator.

\textbf{EXAMPLES:}
By default, the action is the standard $*$ operation. So our first example is about an algebra:

```
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 $x$:

```
sage: x*x, y*x, a*x, b*x
(x, 0, 0, 0)
```

So the annihilator is the subspace spanned by $y$, $a$, and $b$:

```
sage: F.annihilator_basis([x])
y, a, b
```

The same holds for $a$ and $b$:

```
sage: x*a, y*a, a*a, b*a
(a, 0, 0, 0)
sage: F.annihilator_basis([a])
y, a, b
```

On the other hand, $y$ annihilates only $x$:

```
sage: F.annihilator_basis([y])
x,
```

Here is a non trivial annihilator:

```
sage: F.annihilator_basis([a + 3*b + 2*y])
(-1/2*a - 3/2*b + x,)
```

Let’s check it:

```
sage: (-1/2*a - 3/2*b + x) * (a + 3*b + 2*y)
0
```

Doing the same calculations on the left exchanges the roles of $x$ and $y$:

```
sage: F.annihilator_basis([y], side="left")
x, a, b
sage: F.annihilator_basis([a], side="left")
x, a, b
sage: F.annihilator_basis([b], side="left")
x, a, b
sage: F.annihilator_basis([x], side="left")
y,
sage: F.annihilator_basis([a+3*b+2*x], side="left")
(-1/2*a - 3/2*b + y,)
```

By specifying an inner product, this method can be used to compute the orthogonal of a subspace.

4.53. Finite dimensional modules with basis
sage: x,y,a,b = F.basis()
sage: def scalar(u,v):
....:     return vector([sum(u[i]*v[i] for i in F.basis().keys())])
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 $F$:

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:

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: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
#...
.sage.modules
sage: x = X.basis()
#...
.sage.modules
sage: V = X.echelon_form([x[0]-x[1], x[0]-x[2], x[1]-x[2]]); V
#...
.sage.modules
[x[0] - x[2], x[1] - x[2]]
sage: matrix(list(map(vector, V)))
#...
.sage.modules
[ 1 0 -1]
[ 0 1 -1]
```

```python
sage: F = CombinatorialFreeModule(ZZ, [1,2,3,4])
#...
.sage.modules
sage: B = F.basis()
#...
.sage.modules
#...
.sage.modules
sage: F.echelon_form(elements)
#...
.sage.modules
```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
#optional - sage.modules
sage: a, b, c = F.basis()
#optional - sage.modules
sage: F.echelon_form([8*a+b+10*c, -3*a+b-c, a-b-c])
#optional - sage.modules
[B['a'] + B['c'], B['b'] + 2*B['c']]

sage: R.<x,y> = QQ[]

sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
#optional - sage.modules
sage: x = C.basis()
#optional - sage.modules
sage: C.echelon_form([x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]])
#optional - sage.modules
[x[0] - x[2], x[1] - x[2]]

sage: M = MatrixSpace(QQ, 3, 3)
#optional - sage.modules
sage: A = M([[0, 0, 2], [0, 0, 0], [0, 0, 0]])
#optional - sage.modules
sage: M.echelon_form([A, A])
#optional - sage.modules
[[0 0 1]
 [0 0 0]
 [0 0 0]]

from_vector(vector, order=None, coerce=True)

Build an element of self from a vector.

EXAMPLES:

sage: p_mult = matrix([[0, 0, 0], [0, 0, -1], [0, 0, 0]])
#optional - sage.modules
sage: q_mult = matrix([[0, 0, 1], [0, 0, 0], [0, 0, 0]])
#optional - sage.modules
sage: A = algebras.FiniteDimensional(QQ, [p_mult, q_mult, matrix(QQ, 3, 3)], 'p,q,z')
#optional - sage.combinat sage.modules
sage: A.from_vector(vector([1,0,2]))
#optional - sage.combinat sage.modules
p + 2*z

gens()

Return the generators of self.

OUTPUT:

A tuple containing the basis elements of self.

EXAMPLES:
Category Framework, Release 10.1

```python
sage: F = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
    # optional - sage.modules
sage: F.gens()
    # optional - sage.modules
(B['a'], B['b'], B['c'])
```

The `invariant_module` method is defined as follows:

```python
def 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
    - action -- a function (default: operator.mul)
    - side -- 'left' or 'right' (default: 'right'); which side of self the elements of S acts
    - action_on_basis -- (optional) define the action of S on the basis of self

    OUTPUT:
    - FiniteDimensionalInvariantModule

EXAMPLES:

We build the invariant module of the permutation representation of the symmetric group:

```python
sage: G = SymmetricGroup(3); G.rename('S3')
    # optional - sage.groups sage.modules
sage: M = FreeModule(ZZ, [1,2,3], prefix='M'); M.rename('M')
    # optional - sage.groups sage.modules
sage: action = lambda g, x: M.term(g(x))
    # optional - sage.groups sage.modules
sage: I = M.invariant_module(G, action_on_basis=action); I
    # optional - sage.groups sage.modules
(S3)-invariant submodule of M
sage: I.basis()
    # optional - sage.groups sage.modules
Finite family {0: B[0]}
sage: [I.lift(b) for b in I.basis()]
    # optional - sage.groups sage.modules
```

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 \( \mathbb{Q}[G] \):

```python
sage: G = groups.permutation.Dihedral(4)
    # optional - sage.groups
sage: H = G.subgroup(G.gen(0))
    # optional - sage.groups
sage: H
    # optional - sage.groups
Subgroup generated by [(1,2,3,4)] of (Dihedral group of order 8 as a permutation group)
```

(continues on next page)
sage: H.cardinality()  # optional - sage.groups
4

sage: A = G.algebra(QQ)  # optional - sage.groups sage.modules
sage: I = A.invariant_module(H)  # optional - sage.groups sage.modules
sage: [I.lift(b) for b in I.basis()]  # optional - sage.groups sage.modules
[(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)]
sage: all(h * I.lift(b) == I.lift(b)  # optional - sage.groups sage.modules
....:     for b in I.basis() for h in H)
True

twisted_invariant_module(G, chi, action=<built-in function mul>, action_on_basis=None, side='left', **kwargs)

Create the isotypic 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:

sage: M = CombinatorialFreeModule(QQ, [1,2,3])  # optional - sage.groups sage.modules
sage: G = SymmetricGroup(3)  # optional - sage.groups
sage: def action(g,x): return (M.term(g(x)))  # permute coordinates
sage: T = M.twisted_invariant_module(G, [2,0,-1],  # optional - sage.groups sage.modules
....:     action_on_basis=action)
sage: import __main__; __main__.action = action
sage: TestSuite(T).run()  # optional - sage.groups sage.modules

class TensorProducts(category, *args)

Bases: TensorProductsCategory

type: TensorProducts

extra_super_categories()

Implement the fact that a (finite) tensor product of finite dimensional modules is a finite dimensional module.

EXAMPLES:
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(base_category):
    # category
    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:

sage: L = LieAlgebra(QQ, {'x': 'y', 'y': 'z': 1}, nilpotent=True)
    # optional - sage.combinat sage.modules
    sage: L.is_nilpotent()
    # optional - 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)
    # optional - sage.combinat sage.modules
    sage: G = L.lie_group('G'); G
    # optional - 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: p, q, z = L.basis()
    # optional - sage.combinat sage.modules sage.symbolic
    sage: g = G.exp(p); g
    # optional - sage.combinat sage.modules sage.symbolic
    exp(p1)

(continues on next page)
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
Vector field X on the Lie group G of Heisenberg algebra of rank 1 over Rational Field
sage: X.at(G.one()).display()
X = 2 \partial/\partial x_0 + 3 \partial/\partial x_1 + (3/2*x_0 - x_1) \partial/\partial x_2
```

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
Vector field X on the Lie group G of Heisenberg algebra of rank 1 over Rational Field
sage: X.at(G.one()).display()
X = 2 \partial/\partial x_0 + 3 \partial/\partial x_1 + (3/2*x_0 - x_1) \partial/\partial x_2
```

See also:

NilpotentLieGroup

**step()**

Return the nilpotency step of self.

**EXAMPLES:**

```
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

The category of finite dimensional semisimple algebras with a distinguished basis

**EXAMPLES:**

```
sage: from sage.categories.finite_dimensional_semisimple_algebras_with_basis import FiniteDimensionalSemisimpleAlgebrasWithBasis
sage: C = FiniteDimensionalSemisimpleAlgebrasWithBasis(QQ); C
Category of finite dimensional semisimple algebras with basis over Rational Field
```
This category is best constructed as:

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

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:**

```python
sage: A4 = SymmetricGroup(4).algebra(QQ)  # optional - sage.groups sage.modules
sage: Z4 = A4.center()  # optional - sage.groups sage.modules
sage: idempotents = Z4.central_orthogonal_idempotents()  # optional - sage.groups sage.modules
sage: idempotents

1/6*B[0] + 1/6*B[2] - 1/12*B[3],
```

Lifting those idempotents from the center, we recognize among them the sum and alternating sum of all permutations:

```python
sage: [e.lift() for e in idempotents]  # optional - 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*(1,2) + 1/24*(1,2)(3,4) + 1/24*(1,2,3),
1/24*(1,2,3,4) + 1/24*(1,2,4,3) + 1/24*(1,2,4) + 1/24*(1,3,2) + 1/24*(1,3,4) + 1/24*(1,3)(2,4) + 1/24*(1,3,2,4) + 1/24*(1,4,3,2) + 1/24*(1,4,2) + 1/24*(1,4,3) + 1/24*(1,4) + 1/24*(1,4,2,3) + 1/24*(1,4)(2,3),
..., 1/24*(,) - 1/24*(3,4) - 1/24*(2,3) + 1/24*(2,3,4) + 1/24*(2,4,3) - 1/24*(1,2) - 1/24*(1,2)(3,4) - 1/24*(1,2,3),
- 1/24*(1,2,3,4) - 1/24*(1,2,4,3) - 1/24*(1,2,4) - 1/24*(1,3,2) - 1/24*(1,3,4) - 1/24*(1,3)(2,4) - 1/24*(1,3,2,4) - 1/24*(1,4,3,2) - 1/24*(1,4,2) - 1/24*(1,4,3) - 1/24*(1,4) - 1/24*(1,4,2,3) + 1/24*(1,4)(2,3)]
```
We check that they indeed form a decomposition of the identity of \(Z_4\) into orthogonal idempotents:

```python
sage: Z4.is_identity_decomposition_into_orthogonal_idempotents(idempotents)  # optional - sage.groups sage.modules
True
```

```python
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:

    ```python
    sage: R = CyclotomicField(3)  # optional - sage.rings.number_field
    sage: A3 = AlternatingGroup(3).algebra(R)  # optional - sage.rings.number_field sage.groups
    sage: idempotents = A3.central_orthogonal_idempotents()  # optional - sage.rings.number_field sage.groups
    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)  # optional - sage.rings.number_field sage.groups
    True
    ```
```
```
```python
For the semisimple quotient of a quiver algebra, we recover the vertices of the quiver:

```python
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A  # optional - sage.combinat
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()  # optional - sage.combinat
sage: Aquo.central_orthogonal_idempotents()  # optional - sage.combinat
(B['x'], B['y'])
```
4.56 Finite Enumerated Sets

class *sage.categories.finite_enumerated_sets.FiniteEnumeratedSets*(base_category)

Bases: *CategoryWithAxiom_singleton*

The category of finite enumerated sets

EXAMPLES:

```sage
sage: FiniteEnumeratedSets()
Category of finite enumerated sets
sage: FiniteEnumeratedSets().super_categories()
[Category of enumerated sets, Category of finite sets]
sage: FiniteEnumeratedSets().all_super_categories()
[Category of finite enumerated sets,
 Category of enumerated sets,
 Category of finite sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

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
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E, SymmetricGroup(4)])
# optional - sage.groups sage.modules
sage: C.cardinality()
# optional - sage.groups sage.modules
72
sage: E = FiniteEnumeratedSet([])
sage: C = cartesian_product([E, ZZ, QQ])
sage: C.cardinality()
```

(continues on next page)
```python
sage: C = cartesian_product([ZZ, QQ])
sage: C.cardinality()
+Infinity
```

```python
sage: cartesian_product([GF(5), Permutations(10)]).cardinality()  # optional - sage.rings.finite_rings sage.combinat
18144000
```

```python
sage: cartesian_product([GF(71)]*20).cardinality() == 71**20  # optional - sage.rings.finite_rings
True
```

### last()

Return the last element

**EXAMPLES:**

```python
sage: C = cartesian_product([Zmod(42), Partitions(10), IntegerRange(5)])
sage: C.last()  # optional - 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)  # optional - sage.combinat
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])
```

```python
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: C.random_element(4,7)
(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 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:**
- `EnumeratedSets.ParentMethods.rank()`
- `unrank()`

**EXAMPLES:**

```python
sage: C = cartesian_product([GF(2), GF(11), GF(7)])
# optional - sage.rings.finite_rings
sage: C.rank(C((1,2,5)))
# optional - sage.rings.finite_rings
96
sage: C.rank(C((0,0,0)))
# optional - sage.rings.finite_rings
0

sage: for c in C: print(C.rank(c))
# optional - sage.rings.finite_rings
0
1
2
3
4
5
...
150
151
152
153

sage: F1 = FiniteEnumeratedSet('abcdefgh')
sage: F2 = IntegerRange(250)
sage: F3 = Partitions(20)
# optional - sage.combinat
sage: C = cartesian_product([F1, F2, F3])
# optional - sage.combinat
sage: c = C(('a', 86, [7,5,4,4]))
# optional - sage.combinat
sage: C.rank(c)
# optional - sage.combinat
54213
sage: C.unrank(54213)
# optional - sage.combinat
('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()`
• \texttt{rank()}

EXAMPLES:

\begin{verbatim}
\texttt{sage: C = cartesian_product([GF(3), GF(11), GF(7), GF(5)])}  
\texttt{#optional - sage.rings.finite_rings}
\texttt{sage: c = C.unrank(123); c}  
\texttt{#optional - sage.rings.finite_rings}  
(0, 3, 3, 3)
\texttt{sage: C.rank(c)}  
\texttt{#optional - sage.rings.finite_rings}  
123
\texttt{sage: c = C.unrank(857); c}  
\texttt{#optional - sage.rings.finite_rings}  
(2, 2, 3, 2)
\texttt{sage: C.rank(c)}  
\texttt{#optional - sage.rings.finite_rings}  
857
\texttt{sage: C.unrank(2500)}  
\texttt{#optional - sage.rings.finite_rings}  
Traceback (most recent call last):
...  
IndexError: index i (=2) is greater than the cardinality
\end{verbatim}
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:

```
sage: C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
sage: 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]
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]
sage: list(F.iterator_range(stop=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.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.last()
3
sage: C._last_from_iterator()
3
```

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:

```python
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:

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.tuple()
(1, 2, 3)
sage: C.tuple() is C.tuple()
True
```

unrank_range(**kwds)**

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]
```

(continues on next page)
4.57 Finite fields

```python
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([1,2,3,4])
sage: F.unrank_range(stop=10)
[1, 2, 3, 4]
```

class `sage.categories.finite_fields.FiniteFields`(base_category)

    Bases: `CategoryWithAxiom_singleton`

    The category of finite fields.

    EXAMPLES:

```python
sage: 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:

```python
sage: K.super_categories()
[Category of fields,
 Category of finite commutative rings,
 Category of finite enumerated sets]
```

Some examples of membership testing and coercion:

```python
sage: FiniteField(17) in K
# optional - sage.rings.finite_rings
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
```

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:

```python
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
```

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:

```python
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
```

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:

```python
sage: G = groups.misc.SemimonomialTransformation(GF(5), 3); G
Semimonomial transformation group over Finite Field of size 5 of degree 3
sage: G.cardinality.__module__
'sage.categories.finite_groups'
sage: G.cardinality()
384
```

cayley_graph_disabled(connected_set=None)

AUTHORS:

- Bobby Moretti (2007-08-10)

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 just raises a `NotImplementedError`, until we include a non-GAP way of listing the conjugacy classes representatives.

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
```

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()
[(1,), (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:
semigroup_generators()

Returns semigroup generators for self.

For finite groups, the group generators are also semigroup generators. Hence, this default implementation calls group_generators().

EXAMPLES:

```python
sage: A = AlternatingGroup(4)
sage: A.semigroup_generators()
Family ((2,3,4), (1,2,3))
```

some_elements()

Return some elements of self.

EXAMPLES:

```python
sage: A = AlternatingGroup(4)
sage: A.some_elements()
Family ((2,3,4), (1,2,3))
```

eexample()

Return an example of finite group, as per Category.example().

EXAMPLES:

```python
sage: G = FiniteGroups().example(); G
General Linear Group of degree 2 over Finite Field of size 3
```

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:

```python
sage: FiniteLatticePosets()
Category of finite lattice posets
sage: FiniteLatticePosets().super_categories()
[Category of lattice posets, Category of finite posets]
sage: 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:

is_lattice_morphism(f, codomain)

Return whether f is a morphism of posets from self to codomain.

A map f : P → Q is a poset morphism if

    x ≤ y ⇒ f(x) ≤ f(y)

for all x, y ∈ P.

INPUT:
    • f – a function from self to codomain
    • 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 ↦ 5 \prod_{x ∈ b} x is a morphism of lattices:

We construct the boolean lattice B₂:
And the same lattice with new top and bottom elements numbered respectively $-1$ and $3$:

```
sage: G = DiGraph({-1:[0], 0:[1,2], 1:[3], 2:[3], 3:[4]})
    #
    ...
    ...
    #...
    ...
    #
    sage: L = LatticePoset(G)
    #
    ...
    ...
    #
    sage: L.cover_relations()
    ...
    ...
    
[[[-1, 0], [0, 1], [0, 2], [1, 3], [2, 3], [3, 4]]
	sage: f = {B(0): L(-1), B(1): L(1), B(2): L(2), B(3): L(3)}.__getitem__
    #
    ...
    ...
    #
    sage: B.is_lattice_morphism(f, L)
    True
    sage: f = {B(0): L(-1), B(1): L(1), B(2): L(2), B(3): L(3)}.__getitem__
    ...
    ...
    #
    sage: B.is_lattice_morphism(f, L)
    False
    sage: f = {B(0): L(0), B(1): L(1), B(2): L(2), B(3): L(4)}.__getitem__
    ...
    ...
    #
    sage: B.is_lattice_morphism(f, L)
    False
```

See also:

- `is_poset_morphism()`
- `join_irreducibles()`

`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 can not be written as the join of two elements different from $x$.

**EXAMPLES:**

```
sage: L = LatticePoset({0:[1,2],1:[3],2:[3,4],3:[5],4:[5]})
    #
    ...
    ...
    #
    sage: 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
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
# optional - sage.combinat sage.graphs
sage: L.join_irreducibles_poset() # optional - sage.combinat sage.graphs
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
sage: L = LatticePoset({0:[1,2],1:[3],2:[3,4],3:[5],4:[5]})
# optional - sage.combinat sage.graphs
sage: L.meet_irreducibles() # optional - sage.combinat sage.graphs
[1, 3, 4]
```

See also:

- Dual function: `join_irreducibles_poset()`
- Other: `join_irreducibles()`

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
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
# optional - sage.combinat sage.graphs
sage: L.join_irreducibles_poset() # optional - sage.combinat sage.graphs
Finite poset containing 3 elements
```

See also:

- Dual function: `join_irreducibles_poset()`
- Other: `meet_irreducibles()`
4.60 Finite monoids

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]

class ElementMethods

Bases: object

pseudo_order()

Return the pair \([k, j]\) with \(k\) minimal and \(0 \leq j < k\) such that \(\text{self}^k = \text{self}^j\).

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()
[4, 2]

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.

    OUTPUT:
    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:

    $a_1 * a_2 * \cdots * a_k$

    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

    $(a_1 * a_2) * \cdots * a_k$

    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.

    sage: Sigma2 = groups.permutation.Cyclic(2)  # optional - sage.groups
    sage: BSigma2 = Sigma2.nerve()  # optional - sage.groups
    sage: BSigma2.cohomology(4, base_ring=GF(2))  # optional - sage.groups sage.modules sage.rings.finite_rings
    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 $\emptyset$ (the identity element) and $(1,2)$ in Sage. So the 1-cells and
    2-cells in $B\Sigma_2$ are:

    sage: BSigma2.n_cells(1)  # optional - sage.groups
    [(1,2)]
    sage: BSigma2.n_cells(2)  # optional - sage.groups
    [(1,2) * (1,2)]

    Another construction of the group, with different names for its elements:

    sage: C2 = groups.misc.MultiplicativeAbelian([2])  # optional - sage.groups
    sage: BC2 = C2.nerve()  # optional - sage.groups
    sage: BC2.n_cells(0)  # optional - sage.groups
    (continues on next page)
With mod $p$ coefficients, $B\Sigma_p$ should have its first nonvanishing homology group in dimension $p$:

```
sage: Sigma3 = groups.permutation.Symmetric(3)          # →
sage: BSigma3 = Sigma3.nerve()                         # →
sage: BSigma3.homology(range(4), base_ring=GF(3))     # →
{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: Sigma2 = groups.permutation.Symmetric(2)          # →
sage: BSigma2 = Sigma2.nerve()                         # →
sage: BSigma2.n_skeleton(14)                          # →
Simplicial set with 15 non-degenerate simplices
```

```
sage: Sigma3 = groups.permutation.Symmetric(3)          # →
sage: BSigma3 = Sigma3.nerve()                         # →
sage: BSigma3.n_skeleton(3)                           # →
Simplicial set with 156 non-degenerate simplices
```

```
sage: BSigma3 = Sigma3.nerve()                         # →
sage: BSigma3.n_skeleton(4)                           # →
Simplicial set with 781 non-degenerate simplices
```

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:

```
sage: C3 = groups.misc.MultiplicativeAbelian([3])     # →
sage: list(C3)                                        # →
[1, f, f^2]
sage: BC3 = C3.nerve()                                # →
sage: BC3.n_cells(1)                                  # →
[f, f^2]
```

(continues on next page)
[\[ f \ast f, f \ast f^2, f^2 \ast f, f^2 \ast f^2 \]\n\n```
sage: len(BSigma3.n_cells(2))  # optional - sage.groups
25
sage: len(BC3.n_cells(3))  # optional - sage.groups
8
sage: len(BSigma3.n_cells(3))  # optional - sage.groups
125
```

```
sage: BC3.homology(range(4), base_ring=GF(3))  # optional - sage.groups sage.rings.finite_rings
\{0: Vector space of dimension 0 over Finite Field of size 3,
1: Vector space of dimension 1 over Finite Field of size 3,
2: Vector space of dimension 1 over Finite Field of size 3,
3: Vector space of dimension 1 over Finite Field of size 3\}
```

```
sage: BC5 = groups.permutation.Cyclic(5).nerve()  # optional - sage.groups
```

```
sage: BC5.homology(range(4), base_ring=GF(5))  # optional - sage.groups sage.rings.finite_rings
\{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: 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:**

```
sage: M = Monoids().Finite().example()
sage: M.rhodes_radical_congruence()  # optional - sage.groups sage.modules
\[(0, 6), (2, 8), (4, 10)\]
```

```
sage: from sage.monoids.hecke_monoid import HeckeMonoid  # optional - sage.groups sage.modules
```

```
sage: H3 = HeckeMonoid(SymmetricGroup(3))  # optional - sage.groups sage.modules
```

```
sage: H3.repr_element_method(style="reduced")  # optional - sage.groups sage.modules
```

```
sage: H3.rhodes_radical_congruence()  # optional - sage.groups sage.modules
\([(\[1, 2\], \[2, 1\]), (\[1, 2\], \[1, 2, 1\]), (\[2, 1\], \[1, 2, 1\])\]
```
By Maschke’s theorem, every group algebra over $\mathbb{Q}$ is semisimple hence the Rhodes radical of a group must be trivial:

```python
sage: SymmetricGroup(3).rhodes_radical_congruence()  # optional - sage.groups sage.modules
[]
sage: DihedralGroup(10).rhodes_radical_congruence()  # optional - sage.groups sage.modules
[]
```

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:

```
sage: C = PermutationGroups().Finite(); C
Category of finite enumerated permutation groups
sage: C.super_categories()
[Category of permutation groups,
  Category of finite groups,
  Category of finite finitely generated semigroups]
sage: C.example()
Dihedral group of order 6 as a permutation group
```

class ElementMethods
    Bases: object

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:
        $$ P = \frac{1}{|G|} \sum_{g \in G} p_{\text{cycle type}(g)} $$
```

4.61. Finite Permutation Groups 485
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:

```
sage: S4 = SymmetricGroup(4)
sage: P = S4.cycle_index()
sage: 24 * P
```

If $l = (l_1, \ldots, l_k)$ is a partition, $|G| P[l]$ is the number of elements of $G$ with cycles of length $(p_1, \ldots, p_k)$:

```
sage: 24 * P[ Partition([3,1]) ]
```

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:

```
sage: P.parent()
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 )
h[4]
```

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()
```

```
sage: 60 * AlternatingGroup(5).cycle_index()
```

```
sage: for G in TransitiveGroups(5): # long time
....:     G.cardinality() * G.cycle_index()
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]
```

Permutation groups with arbitrary domains are supported (see github issue #22765):
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:

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]]
sage: P.parent() is F
True

This parent should be a module with basis indexed by partitions:

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 \( \text{self} \).

Optional argument using_polya allows to change the default method.

INPUT:
  • \( n \) – a nonnegative integer
  • 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 \( \text{self} \)
    on the subsets of its domain (i.e. the value of the profile of \( \text{self} \) in \( n \))

See also:
  • profile_series()

EXAMPLES:

sage: C6 = CyclicPermutationGroup(6)
sage: C6.profile(2)
3
sage: C6.profile(3)
4
sage: D8 = DihedralGroup(8)
sage: D8.profile(4, using_polya=False)
8

profile_polynomial(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:**
- 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 $\mathbb{Z}$. 

**See also:**
- `profile()`

**EXAMPLES:**

```python
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')  # optional - sage.symbolic
sage: D8.profile_series(u).parent()  # optional - sage.symbolic
Symbolic Ring
```

`profile_series(variable=\text{'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:**
- 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 $\mathbb{Z}$. 

**See also:**
- `profile()`

**EXAMPLES:**

```python
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')
# optional - sage.symbolic

sage: D8.profile_series(u).parent()
# optional - sage.symbolic
Symbolic Ring

example()  
Returns an example of finite permutation group, as per Category.example().

EXAMPLES:

sage: G = FinitePermutationGroups().example(); G
Dihedral group of order 6 as a permutation group

extra_super_categories()  
Any permutation group is assumed to be endowed with a finite set of generators.

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: 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:
Birational free labelling

Now, birational rowmotion relates to classical constructions such as promotion of rectangular semistandard Young tableaux. The definitions of birational toggles and birational rowmotion extend to the case of a field 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 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 promotion) and rowmotion on order ideals of the poset ([EP2013]).

The 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 $K$ 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 $(K, d, u, v)$, where $K$ 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 facade option set to 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:**
- `linear_extension` – (default: the default linear extension of `self`) a linear extension of `self` (as a linear extension or as a list), or more generally a list of all elements of all elements of `self` each occurring exactly once
- `prefix` – (default: `'x'`) the prefix to name the indeterminates corresponding to the elements of `self` in the labelling (so, setting it to `'frog'` will result in these indeterminates being called `frog1`, `frog2`, ..., `frogn` rather than `x1`, `x2`, ..., `xn`).
- `base_field` – (default: `QQ`) the base field to be used instead of `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!
- `reduced` – (default: `False`) if set to `True`, the result will be the 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
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
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, {...},
```

(continues on next page)
Illustrating labelling with a function:

```python
sage: P = posets.ChainPoset(2).product(posets.ChainPoset(2))
sage: l = P.birational_free_labelling(labels=lambda e : '{x_}' + str(e[0]) + str(e[1]))
sage: sorted(l[1].items())
[((0, 0), x_00), ((0, 1), x_01), ((1, 0), x_10), ((1, 1), x_11)]
sage: l[2]
a
The same, but with min_label and max_label provided:

```
Illustrating labelling with a comma separated list of labels:

```python
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)]
```

Another poset:

```python
sage: P = posets.SSTPoset([2,1])
sage: l = P.birational_free_labelling(linear_extension=lext, addvars="ohai")
sage: l
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, x5, x6, x7, x8, b, ohai over Rational Field,
 {...},
a,
b)
```

Illustrating the warning about facade:

```python
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,
 {...},
a,
b)
```

4.62. Finite posets
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 `birational_free_labelling()`

**OUTPUT:**
The image of the \(K\)-labelling \(f\) under birational rowmotion.

**EXAMPLES:**

```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); t
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, b˓→ over Rational Field,
{...},
a,
b)
sage: sorted(t[1].items())
[(1, x1), (2, x2), (3, x3), (4, x4)]
sage: t = P.birational_rowmotion(t); t
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, b˓→ over Rational Field,
{...},
a,
b)
sage: sorted(t[1].items())
[(1, a*b/x4), (2, (x1*x2*b + x1*x3*b)/(x2*x4)),
(3, (x1*x2*b + x1*x3*b)/(x3*x4)), (4, (x2*b + x3*b)/x4)]
```

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
    for i in range(k):
        t = P.birational_rowmotion(t)
    return t == t0
sage: test_rectangle_periodicity(2, 2, 4)
True
sage: test_rectangle_periodicity(2, 2, 2)
False
```

(continues on next page)
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:

```python
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):
        ...
    return t == t0
sage: test_rectangle_periodicity_tropical(7, 6, 13)
True
```

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:

```python
sage: P = posets.IntegerPartitions(5)

sage: TT = TropicalSemiring(ZZ)

sage: def indicator_labelling(I):
    ...
    dct = {v: TT(v in I) for v in P}
    ...
    return (TT, dct, TT(1), TT(0))

sage: all(indicator_labelling(P.rowmotion(I)) == P.birational_rowmotion(indicator_labelling(I))
    for I in P.order_ideals_lattice(facade=True))
True
```

**birational_toggle($v$, labelling)**

Return the result of applying the birational $v$-toggle $T_v$ to the $K$-labelling labelling of the poset self.

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 self (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 $T_v f$ of self, where $f$ is labelling.

**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
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)]
```

The image of $s$ under the 1-toggle $T_1$ is:

```sage
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)]
```

Now let us apply the 2-toggle $T_2$ (to the old $s$):

```sage
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)]
```

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:
However, labellings have to be sufficiently generic, lest denominators vanish:

We don’t get into zero-division issues in the tropical semiring (unless the zero of the tropical semiring appears in the labelling):

We turn to more interesting posets. Here is the 6-element poset arising from the weak order on $S_3$:

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 $v$:
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
```

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 $T_{v_n}T_{v_{n-1}}\ldots T_{v_1}f$ of self, where $f$ is labelling and $(v_1, v_2, \ldots, v_n)$ is vs (written as list).

EXAMPLES:

```python
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(2), '321': 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,
continues on next page)
Facade set to `False` works:

```
sage: t1 = P.birational_free_labelling(linear_extension=lex)
sage: sorted(t1[1].items())
[(1, (x^2 + x)/(x^2 + x + 1)), (2, (x^4)/(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())
[(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:**

```
sage: P = Poset({'x': ['y', 'w'], 'y': ['z'], 'w': ['z']}, facade=False)
sage: lex = ['x', 'y', 'w', 'z']
sage: t = P.birational_free_labelling(linear_extension=lex)
sage: sorted(P.birational_toggles([P('x'), P('y')], t)[1].items())
[(x, a*x^2*x^3/(x1*x^2 + x1*x3)), (y, a*x^2*x^3/(x1*x^2 + x1*x3)), (w, x3), (z, x^2 + x + 1)/x^3]
```

**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:**

```
sage: P = Poset(((divisors(12), attrcall("divides")), facade=True)
sage: A = P.directed_subsets('up')
sage: sorted(list(A))
[[], [1, 2, 4, 3, 6, 12], [2, 4, 3, 6, 12], [2, 4, 6, 12], [3, 6, 12], [4, 6, 3, 6, 12], [4, 6, 12], [6, 12], [12]]
```

```
sage: P.is_lattice()
True
```

```
sage: P = Poset([1, 2, 3], [4, 5, 6], [6, 7, 7, 7, 7, []])
sage: P.is_lattice()
False
```

```
sage: P = Poset([1, 2, 3], [2, 3, 3], [4, 3, 1])
sage: P.is_lattice()
False
```

```
sage: P = Poset([0: [2, 3], 1: [2, 3]])
sage: P.is_lattice()
False
```

(continues on next page)

4.62. Finite posets
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()
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 `self` to `codomain`.

**INPUT:**

- $f$ – a function from `self` to `codomain`
- `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 $f: B \to D$, $b \mapsto \prod_{x \in b} x$:

```
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
```

On the other hand, $f$ is not an isomorphism to the chain of divisors of 30, ordered by usual comparison:

```
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:

```
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:

```
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.
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:

```python
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
```

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:

```python
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
```

**FIXME:** should this be `is_order_preserving_morphism`?

See also:
- `is_poset_isomorphism()`

is_self_dual()

Return whether the poset is self-dual.

A poset is self-dual if it is isomorphic to its dual poset.

**EXAMPLES:**

```python
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
```
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]]
```

See also:

**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:**

```
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()

**Warning:** This is a brute force implementation, building the order ideal generated by the antichain, and searching for order filter generators of its complement

**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:

 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]]

Then, we retrieve the generators of this ideal:

 sage: gen = P.order_ideal_generators(I)
sage: 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:

 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_ideal_generators(I, **direction='up'**)
sage: sorted(sorted(p) for p in gen)
[[1], [2, 3]]

Complexity: $O(n + m)$ where $n$ is the cardinality of $I$, and $m$ the number of upper covers of elements of $I$. 

4.62. Finite posets 503
**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: 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], [0, 1, 2, 3, 4], [0, 2], [0, 2, 3]]
```

As a lattice on antichains:

```python
sage: J2 = P.order_ideals_lattice(False); J2
Finite lattice containing 8 elements
sage: sorted(J2)
[(0,), (0, 1), (1,), (1, 2), (1, 3), (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:**
\begin{verbatim}
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()
\end{verbatim}

**Warning:** This is a brute force implementation, building the order ideal generated by the antichain, and searching for order filter generators of its complement

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 $I[0] ==$ antichain and each $i$ satisfies self.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:**
\begin{verbatim}
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()]
\end{verbatim}

(continues on next page)
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.orderIdealComplementGenerators(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: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: orb = P.panyushev_orbits()
sage: sorted(sorted(o) for o in orb)
[[set(), {1, 2}, {3}], [{2}, {1}]]
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)
```
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 ($\text{panyushev_complement}()$).

**INPUT:**
- $\text{order_ideal}$ – an order ideal of self, as a set

**OUTPUT:**
- the image of $\text{order_ideal}$ 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}
```

**rowmotion_orbit_iter**(oideal, element_constructor=<class 'set'>, stop=True, check=True)

Iterate over the rowmotion orbit of an order ideal $\text{oidreal}$ of self.

The rowmotion orbit of an order ideal is its orbit under rowmotion (see `rowmotion()`).

**INPUT:**
- $\text{oidreal}$ – an order ideal of self, given as an iterable.
- $\text{element_constructor}$ (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.
- $\text{stop}$ – 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`).
- $\text{check}$ – a Boolean (default: `True`) determining whether $\text{oidreal}$ should be checked for being an order ideal.

**OUTPUT:**
- an iterator over the orbit of the order ideal $\text{oidreal}$ under rowmotion. This iterator $I$ has the property that $I[0] == \text{oidreal}$ and that every $i$ satisfies $\text{self}.\text{rowmotion}(I[i]) == I[i+1]$, 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( {} )
sage: I = Set({})
sage: P.rowmotion_orbit_iter(I)
{
```
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:

sage: P = Poset( {1: [2, 3], 2: [4], 3: [4], 4: []})
sage: Piter = P.rowmotion_orbit_iter([1, 2], 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], [1, 2, 3, 5], [1, 2, 4], [3]]

rowmotion_orbits(element_constructor=None)

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:

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: orb = P.rowmotion_orbits(element_constructor=tuple)
sage: sorted(sorted(e) for e in orb)
[[]]

rowmotion_orbits_plots()

Return plots of the rowmotion orbits of order ideals in self.

The rowmotion orbit of an order ideal is its orbit under rowmotion (see rowmotion()).

EXAMPLES:
sage: P = Poset( {1: [2, 3], 2: [], 3: [], 4: [2]} )
sage: P.rowmotion_orbits_plots()
Graphics Array of size 2 x 5
sage: P = Poset({})
sage: P.rowmotion_orbits_plots()
Graphics Array of size 1 x 1

toggling_orbit_iter(vs, oideal, element_constructor=<class 'set'>, stop=True, check=True)

Iterate over the orbit of an order ideal oideal of self under the operation of toggling the vertices 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, T2T1I, T2T1I, T2T1I, ...), where I denotes oideal and T_i means toggling at i) rather than (I, T1I, T2T1I, T1T2T1I, ...).

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:
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], set(), {1, 2, 3}
sage: list(P.toggling_orbit_iter([3, 2, 1], set([1, 2])))
[1, 2], {1, 2, 3}, set()
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))
[frozenset({1, 2}), frozenset({1, 2, 3}), frozenset()]
sage: list(P.toggling_orbit_iter([3, 2, 1], set([1, 2]), element_˓
˓constructor=tuple))
[(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)
{1}
sage: next(Piter)
set()
sage: next(Piter)
{1, 2, 3}
sage: next(Piter)
{1}

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], ...

in this order.

See order_ideal_toggle() for a definition of toggling.

Warning: The orbits are those under the composition of toggles, 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, ...) (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, ...).
• a partition of the order ideals of \texttt{self}, as a list of sets \texttt{L} such that for each \texttt{L} and \texttt{i}, cyclically:

\texttt{self.order_ideal_toggles(L[i], vs) == L[i+1]}.

**EXAMPLES:**

```
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]
```

**toggling_orbits_plots(vs)**

Return plots of the orbits of order ideals in \texttt{self} under the operation of toggling the vertices \texttt{vs[0]}, \texttt{vs[1]}, \ldots in this order.

See **toggling_orbits()** for more information.

**EXAMPLES:**

```
sage: P = Poset( {1: [2, 3], 2: [], 3: [], 4: [2]} )
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
```

### 4.63 Finite semigroups

**class** \texttt{sage.categories.finite_semigroups.FiniteSemigroups(base_category)}

Bases: \texttt{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 \texttt{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 \texttt{semigroup\_generators()} should now explicitly declare themselves in the category of finitely generated semigroups:

```
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
```

This is a backward incompatible change.

**EXAMPLES:**

```
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')

class ParentMethods
    Bases: object

    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']]

    j_transversal_of_idempotents()
        Returns a list of one idempotent per regular \( J \)-class

    EXAMPLES:
    sage: S = FiniteSemigroups().example(alphabet=('a', 'b', 'c'))
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
['a', 'ab', 'abc', 'ac', 'b', 'c', 'cb']
```

### 4.64 Finite sets

```python
class sage.categories.finite_sets.FiniteSets(base_category):
    Bases: CategoryWithAxiom_singleton
    The category of finite sets.
    EXAMPLES:

    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
class Algebras(category, *args):
    Bases: AlgebrasCategory
    EXAMPLES:

    sage: 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:

```python
sage: 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

class Subquotients(category, *args)
    Bases: SubquotientsCategory

element super_categories()
    EXAMPLES:
        sage: 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

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}

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()
    #optional - sage.rings.number_field
    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()
    #optional - sage.rings.number_field
    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: V = lie_conformal_algebras.Affine(QQ, 'A1')
        #optional - sage.combinat sage.modules
        sage: V.gens()
        (B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])
        sage: V.gen(0)
        #optional - sage.combinat sage.modules
        B[alpha[1]]
        sage: V.1
        #optional - sage.combinat sage.modules
        B[alphacheck[1]]

    ngens()

        The number of generators of this Lie conformal algebra.

        EXAMPLES:
sage: Vir = lie_conformal_algebras.Virasoro(QQ)  # optional - sage.combinat sage.modules
sage: Vir.ngens()  # optional - sage.combinat sage.modules
2
sage: V = lie_conformal_algebras.Affine(QQ, 'A2')  # optional - sage.combinat sage.modules
sage: V.ngens()  # optional - sage.combinat sage.modules
9
sage: V.some_elements()  # optional - sage.combinat sage.modules
Some elements of this Lie conformal algebra.
This method returns a list with elements containing at least the generators.

EXAMPLES:

sage: V = lie_conformal_algebras.Affine(QQ, 'A1',  # optional - sage.combinat sage.modules
....:
    names=('e', 'h', 'f'))
sage: V.some_elements()  # optional - sage.combinat sage.modules
[e, h, f, K, ...]
sage: all(v.parent() is V for v in V.some_elements())  # optional - sage.combinat sage.modules
True

4.67 Finitely Generated Lie Conformal Algebras

AUTHORS:

• Reimundo Heluani (2019-10-05): Initial implementation.

class sage.categories.finitely_generated_lie_conformal_algebras.FinitelyGeneratedLieConformalAlgebras(base_category):

    Bases: CategoryWithAxiom_over_base_ring

    The category of finitely generated Lie conformal algebras.

    EXAMPLES:

      sage: LieConformalAlgebras(QQbar).FinitelyGenerated()  # optional - sage.rings.number_field
      Category of finitely generated lie conformal algebras over Algebraic Field

      class Graded(base_category):

          Bases: GradedModulesCategory

          The category of H-graded finitely generated Lie conformal algebras.

          EXAMPLES:
class `ParentMethods`

Bases: `object`

`some_elements()`

Some elements of this Lie conformal algebra.

Returns a list with elements containing at least the generators.

EXAMPLES:

```python
sage: V = lie_conformal_algebras.Affine(QQ, 'A1',
          names=('e', 'h', 'f'))
[1]
sage: V.some_elements()
[1]
sage: all(v.parent() is V for v in V.some_elements())

True
```

class `Super(base_category)`

Bases: `SuperModulesCategory`

The category of super finitely generated Lie conformal algebras.

EXAMPLES:

```python
sage: LieConformalAlgebras(AA).FinitelyGenerated().Super()

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()

Category of H-graded super finitely generated lie conformal algebras
over Algebraic Field
```
4.68 Finitely generated magmas

class sage.categories.finitely_generated_magmas.FinitelyGeneratedMagmas(base_category):
    Bases: CategoryWithAxiom_singleton

    The category of finitely generated (multiplicative) magmas.

    See Magmas.SubcategoryMethods.FinitelyGeneratedAsMagma() for details.

    EXAMPLES:

    sage: C = Magmas().FinitelyGeneratedAsMagma(); C
    Category of finitely generated magmas
    sage: C.super_categories()
    [Category of magmas]
    sage: sorted(C.axioms())
    ['FinitelyGeneratedAsMagma']

    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

class sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups(base_category):
    Bases: CategoryWithAxiom_singleton

    The category of finitely generated (multiplicative) semigroups.

    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:

    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']
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:

    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

    class ParentMethods
    Bases: object

        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:

        semigroup_generators()

        INPUT:
        • gens – a list (or iterable) of elements of self
        • side – [default: “twosided”] “left”, “right” or “twosided”

        EXAMPLES:

        sage: S = FiniteSemigroups().example()
        sage: sorted(S.ideal([S('cab')], side="left"))
        sage: list(S.ideal([S('cab')], side="right"))
        ['cab', 'cabd']
        sage: sorted(S.ideal([S('cab')], side="twosided"))
'adcb', 'bac', 'bacd', 'badc', 'bca', 'bcad', 'bdac', 'bdca', 'cab', 'cabd', 'cadb', 'cbad', 'cbda', 'cdab', 'cda', 'dbac', 'dbca', 'dcab', 'dcba']
sage: sorted(S.ideal([S('cab')]))
['abc', 'abcd', 'abc', 'acbd', 'acdb', 'adbc', 'adcb', 'bac', 'bacd', 'badc', 'bcda', 'bdac', 'bdca', 'cab', 'cabd', 'cadb', 'cbad', 'cbda', 'cdab', 'cda', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']

semigroup_generators()
Return distinguished semigroup generators for self.
OUTPUT: a finite family
This method should be implemented by all semigroups in FinitelyGeneratedSemigroups.
EXAMPLES:
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?

EXAMPLES:
sage: S = FiniteSemigroups().example()
sage: S.succ_generators("left") (S('ca'))
('ac', 'bca', 'ca', 'dca')
sage: S.succ_generators("right") (S('ca'))
('ca', 'cab', 'ca', 'cad')
sage: S.succ_generators("twosided") (S('ca'))
('ac', 'bca', 'ca', 'dca', 'ca', 'cab', 'ca', 'cad')

example()
EXAMPLES:
sage: Semigroups().FinitelyGenerated().example()
An example of a semigroup: the free semigroup generated
by ('a', 'b', 'c', 'd')
extra_super_categories()

State that a finitely generated semigroup is endowed with a default enumeration.

EXAMPLES:

```
sage: Semigroups().FinitelyGenerated().extra_super_categories()
[Category of enumerated sets]
```

### 4.70 Function fields

**class** `sage.categories.function_fields.FunctionFields(s=None)`

Bases: `Category`

The category of function fields.

EXAMPLES:

We create the category of function fields:

```
sage: C = FunctionFields()
sage: 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]
```

### 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)  #optional - sage.groups
sage: GSets(S)  #optional - sage.groups
Category of G-sets for Symmetric group of order 3! as a permutation group
```

4.70. Function fields
Category Framework, Release 10.1

TODO: should this derive from Category_over_base?

classmethod an_instance()
    Returns an instance of this class.

    EXAMPLES:
    sage: GSets.an_instance()  # indirect doctest
    Category of G-sets for Symmetric group of order 8! as a permutation group

super_categories()
    EXAMPLES:
    sage: GSets(SymmetricGroup(8)).super_categories()  # indirect doctest
    [Category of sets]

4.72 Gcd domains

class sage.categories.gcd_domains.GcdDomains(s=None)
    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:
    sage: GcdDomains()
    Category of gcd domains
    sage: 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:
4.73 Generalized Coxeter Groups

class sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups(s=None)
    Bases: Category_singleton
    The category of generalized Coxeter groups.
    A generalized Coxeter group is a group with a presentation of the following form:

    \[ \langle s_i \mid s_i^{p_i}, s_is_j \cdots = s_js_i \cdots \rangle, \]

    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

    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:
```python
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: GeneralizedCoxeterGroups().additional_structure()
```

```python
sage: GeneralizedCoxeterGroups().additional_structure()
```

**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]
```

### 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]
```

**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:**

```python
sage: m = SymmetricFunctions(QQ).m()
    #optional - sage.combinat
sage: m.graded_algebra() is m
    #optional - sage.combinat
True
```

**class** `SignedTensorProducts(category, *args)`

**Bases:** `SignedTensorProductsCategory`

**extra_super_categories()**

**EXAMPLES:**

```python
```
class SubcategoryMethods
   Bases: object

SignedTensorProducts()
   Return the full subcategory of objects of self constructed as signed tensor products.

   See also:
   • SignedTensorProductsCategory
   • CovariantFunctorialConstruction

   EXAMPLES:

   sage: 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]

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: NCSF = NonCommutativeSymmetricFunctions(QQ) # optional - sage.combinat
   ...
sage: S = NCSF.Complete()  # optional - sage.combinat
    sage: L = S.formal_series_ring()  # optional - sage.combinat
sage: L  # optional - sage.combinat
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:

sage: NCSF = NonCommutativeSymmetricFunctions(QQ)  # optional - sage.combinat
sage: S = NCSF.Complete()  # optional - sage.combinat
sage: L = S.formal_series_ring()  # optional - sage.combinat
sage: L  # optional - sage.combinat
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:

sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])  # optional - sage.modules
sage: Cl = CliffordAlgebra(Q)  # optional - sage.combinat sage.modules
sage: M = Cl.free_graded_module((0, 2, 3))  # optional - sage.combinat sage.modules
sage: M.gens()  # optional - sage.combinat sage.modules
(g[0], g[2], g[3])
sage: N.<xy, z> = Cl.free_graded_module((1, 2))  # optional - sage.combinat sage.modules
sage: N.generators()  # optional - sage.combinat sage.modules
(x, y)

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()  # optional - sage.combinat
sage: m.graded_algebra() is m  # optional - 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: A.<x, y> = ExteriorAlgebra(QQ)  # optional - sage.combinat sage.modules
sage: A.one_basis()  #optional - sage.combinat sage.modules
0
sage: B = tensor((A, A, A))  #optional - sage.combinat sage.modules
sage: B.one_basis()  #optional - sage.combinat sage.modules
(0, 0, 0)
sage: B.one()  #optional - sage.combinat sage.modules
1

product_on_basis(i0, i1)

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: A = SteenrodAlgebra(3)  # optional - sage.combinat sage.modules
sage: x = A.Q(0)  # optional - sage.combinat sage.modules
sage: y = x.coproduct()  # optional - sage.combinat sage.modules
sage: y^2  # optional - sage.combinat sage.modules
0
```

TODO: optimize this implementation!

```python
extra_super_categories()
```

EXAMPLES:

```python
sage: Cat = AlgebrasWithBasis(QQ).Graded()
sage: Cat.SignedTensorProducts().extra_super_categories()  # optional - sage.combinat sage.modules

[Category of graded algebras with basis over Rational Field]
sage: Cat.SignedTensorProducts().super_categories()  # optional - sage.combinat sage.modules

[Category of graded algebras with basis over Rational Field, Category of signed tensor products of graded algebras over Rational Field]
```

## 4.76 Graded bialgebras

sage.categories.graded_bialgebras.**GradedBialgebras**(base_ring)

The category of graded bialgebras

EXAMPLES:

```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()  # optional - sage.categories
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

EXAMPLES:

```python
sage: C = GradedBialgebrasWithBasis(QQ); C
Join of Category of graded bialgebras 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]
```
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

extra_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:
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:
Category Framework, Release 10.1

```python
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
```

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

    \[ c(i) := \begin{cases} 
    1 & \text{if } i \text{ indexes the } 1 \text{ of the algebra} \\
    0 & \text{otherwise.} 
    \end{cases} \]

    EXAMPLES:

    ```python
    sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
    # optional - sage.modules
    sage: H.monomial(4).counit() # indirect doctest
    # optional - sage.modules
    0
    sage: H.monomial(0).counit() # indirect doctest
    # optional - sage.modules
    1
    ```

    example()
    Return an example of a graded connected Hopf algebra with a distinguished basis.

4.81. Graded Hopf algebras with 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 \( L = \bigoplus_{k=1}^{M} L_k \) (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:
```python
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:

        ```python
        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:

    ```python
    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
    ```

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:

```python
sage: C = LieConformalAlgebras(QQbar).Graded(); C
# optional - sage.rings.number_field
Category of H-graded Lie conformal algebras over Algebraic Field

sage: CS = LieConformalAlgebras(QQ).Graded().Super(); CS
# optional - sage.rings.number_field
Category of H-graded super Lie conformal algebras over Rational Field

sage: CS is LieConformalAlgebras(QQ).Super().Graded()
True
```

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: C = LieConformalAlgebras(QQbar)
# optional - sage.rings.number_field
sage: C.Graded().Super() is C.Super().Graded()
# optional - sage.rings.number_field
True

sage: Cp = C.WithBasis()
# optional - sage.rings.number_field
sage: Cp.Graded().Super() is Cp.Super().Graded()
# optional - sage.rings.number_field
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 \( M = \bigoplus M_i \) as a filtered module under the (natural) filtration given by

\[
F_i = \bigoplus_{j < i} M_j.
\]
EXAMPLES:

```python
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:

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

```python
class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

class sage.categories.graded_modules.GradedModulesCategory(base_category)
    Bases: RegressiveCovariantConstructionCategory, Category_over_base_ring
```

EXAMPLES:

```python
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]
```

```python
sage: AlgebrasWithBasis(QQ).Graded().base_ring()
Rational Field
sage: GradedHopfAlgebrasWithBasis(QQ).base_ring()
Rational Field
```

```python
classmethod 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 $G$:
This resulted from the following call:

```python
sage: sage.categories.graded_modules.GradedModulesCategory.default_super_categories(Algebras(QQ))
Join of Category of filtered algebras over Rational Field and Category of graded vector spaces over Rational Field
```

### 4.86 Graded modules with basis

```python
class sage.categories.graded_modules_with_basis.GradedModulesWithBasis(base_category):
    Bases: GradedModulesCategory

    The category of graded modules with a distinguished basis.

    EXAMPLES:

    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
```

```python
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:

    sage: E.<a,b> = ExteriorAlgebra(QQ)  # optional - sage.combinat sage.modules
    sage: ((1 + a) * (1 + b)).degree_negation()  # optional - sage.combinat sage.modules
    a*b - a - b + 1
    sage: E.zero().degree_negation()  # optional - sage.combinat sage.modules
    0
```

(continues on next page)
the free module on partitions over Integer Ring

```python
sage: pbp = lambda x: P.basis()[Partition(list(x))]  #
                      --optional -- sage.combinat sage.modules
sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])     #
                      --optional -- sage.combinat sage.modules
sage: p.degree_negation()  #
                      --optional -- sage.combinat sage.modules
```

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)  #
                      --optional -- sage.combinat sage.modules
sage: E.degree_negation((1 + a) * (1 + b))  #
                      --optional -- sage.combinat sage.modules
a*b - a - b + 1
sage: E.degree_negation(E.zero())  #
                      --optional -- sage.combinat sage.modules
0
sage: P = GradedModulesWithBasis(ZZ).example(); P  #
                      --optional -- 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))]  #
                      --optional -- sage.combinat sage.modules
sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])     #
                      --optional -- sage.combinat sage.modules
sage: P.degree_negation(p)  #
                      --optional -- 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 \( \text{self} \).

**EXAMPLES:**

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)  # optional - sage.combinat sage.modules

\rightarrow

sage: S = E.submodule([x + y, x^2*y - y^2*z, y])  # optional - sage.combinat sage.modules

\rightarrow

sage: Q = E.quotient_module(S)  # optional - sage.combinat sage.modules

\rightarrow

sage: Q.category()  # optional - sage.combinat sage.modules

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 vector spaces with basis over Rational Field
```

See also:

- Modules.WithBasis.ParentMethods.submodule()
- Rings.ParentMethods.quotient()
- sage.modules.with_basis.subquotient.QuotientModuleWithBasis

```python
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 \( \text{self} \)
- `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 \( \text{gens}[i] \).

See also:

- ModulesWithBasis.FiniteDimensional.ParentMethods.quotient_module()
- sage.modules.with_basis.subquotient.SubmoduleWithBasis
EXAMPLES:

A graded submodule of a graded module generated by homogeneous elements is naturally graded:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
     #optional - sage.combinat sage.modules
sage: S = E.submodule([x + y, x*y - y*z])
     #optional - sage.combinat sage.modules
sage: S.category()
Join of
    Category of graded vector spaces with basis over Rational Field and
    Category of subobjects of 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])
     #optional - sage.combinat sage.modules
sage: Sp.category()
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 vector spaces with basis over Rational Field
sage: Sp.basis()[0].degree()
1
sage: Sp.basis()[1].degree()
2
```

If it is generated by inhomogeneous elements, then it is filtered by default:

```
sage: F = E.submodule([x + y*z, x*z + y*x])
     #optional - sage.combinat sage.modules
sage: F.category()
Join of
    Category of subobjects of filtered modules with basis over Rational Field
    and
    Category of filtered vector spaces with basis over Rational Field
    and
    Category of finite dimensional 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):

```
sage: SM = E.submodule([x + y, x*y - y*z],
       ....: category=ModulesWithBasis(QQ))
sage: SM.category()
Join of
    Category of graded vector spaces with basis over Rational Field
```

(continues on next page)
Category of finite dimensional vector spaces with basis over Rational Field and Category of subobjects of sets

```
sage: FM = E.submodule([x + 1, x*y - x*y*z],
    ...: category=ModulesWithBasis(QQ))
```

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:

```
sage: Cat = ModulesWithBasis(QQ).Graded().Subobjects()
sage: E.submodule([x + y, x*y - 1], category=Cat)
Traceback (most recent call last):
  ...:
ValueError: all of the generators must be homogeneous
```

```python
class Quotients(category, *args):
    Bases: QuotientsCategory
class ElementMethods:
    Bases: object
def degree():
        Return the degree of self.

    EXAMPLES:
    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
degree_on_basis(m)
```

Return the degree of the basis element indexed by \( m \) in \( \text{self} \).

**EXAMPLES:**

```python
sage: E.<x,y,z> = ExteriorAlgebra(QQ)  #optional - sage.combinat sage.modules
sage: S = E.submodule([x + y, x*y - y*z, y])  #optional - sage.combinat sage.modules
sage: Q = E.quotient_module(S)  #optional - sage.combinat sage.modules
sage: B = Q.basis()  #optional - sage.combinat sage.modules
sage: [B[i].lift() for i in Q.indices()]  #optional - sage.combinat sage.modules
[1, z, x*z, y*z, x*y*z]
sage: [Q.degree_on_basis(i) for i in Q.indices()]  #optional - sage.combinat sage.modules
[0, 1, 2, 2, 3]
```

### 4.87 Graphs

**class** `sage.categories.graphs.Graphs(s=None)`

Bases: `Category_singleton`

The category of graphs.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: 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()
sage: TestSuite(C).run()
```

**extra_super_categories()**

Return the extra super categories of \( \text{self} \).

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,
```
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
```

dges()

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)]
```

daces()

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)]
```
acets()

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)]
```

vertices()

Return the vertices of self.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
```
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
- Martin Raum (2009-08): update to use new coercion model – see github issue #6670.
- John Palmieri (2011-07): more updates to coercion, categories, etc., group algebras constructed using CombinatorialFreeModule – see github issue #6670.
- Nicolas M. Thiéry (2010-2017), Travis Scrimshaw (2017): generalization to a covariant functorial construction for monoid algebras, and beyond – see e.g. github issue #18700.

class `sage.categories.group_algebras.GroupAlgebras`(category, *args)

Bases: `AlgebrasCategory`

The category of group algebras over a given base ring.

EXAMPLES:

```
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:

```
sage: C is GroupAlgebras(ZZ)
True
```

Here is how to create the group algebra of a group $G$:

```
sage: G = DihedralGroup(5)  # optional - sage.groups
sage: QG = G.algebra(QQ); QG  # optional - sage.groups sage.modules
Algebra of
Dihedral group of order 10 as a permutation group over Rational Field
```

and an example of computation:
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 coordinates in the canonical basis of the center. See Groups.Algebras.ParentMethods.center_basis() for details.

Warning:

- This method requires the underlying group to have a method conjugacy_classes_representatives (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: QS3 = SymmetricGroup(3).algebra(QQ)  #←
←optional - sage.groups sage.modules
sage: A = QS3([2,3,1]) + QS3([3,1,2])  #←
←optional - sage.groups sage.modules
sage: A.central_form()  #←
←optional - sage.groups sage.modules
B[(1,2,3)]
sage: QS4 = SymmetricGroup(4).algebra(QQ)  #←
←optional - sage.groups sage.modules
sage: B = sum(len(s.cycle_type()) * QS4(s) for s in Permutations(4))  #←
←optional - sage.groups sage.modules
sage: B.central_form()  #←
←optional - sage.groups sage.modules
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 github issue #28544):

```python
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(3,4)]]
    #←optional - sage.groups sage.modules
sage: QG = GroupAlgebras(QQ).example(G)
    #←optional - sage.groups sage.modules
sage: s = sum(QG.basis())
    #←optional - sage.groups sage.modules
sage: s.central_form()  # not tested
    #←optional - sage.groups sage.modules
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()`

```python
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:

        sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
            #←optional - 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
            #←optional - sage.groups sage.modules
        (1,2,3,4,5,6)
        sage: A.antipode_on_basis(g)
            #←optional - sage.groups sage.modules
        (1,6,5,4,3,2)
        sage: a = A.an_element(); a
            #←optional - sage.groups sage.modules
        () + 3*(1,2,3,4,5,6) + 3*(1,3,5)(2,4,6)
        sage: a.antipode()
            #←optional - sage.groups sage.modules
        () + 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
```

4.88. Group Algebras 545
Warning:

- This method requires the underlying group to have a method `conjugacy_classes` (every permutation group has one, thanks GAP!).

EXAMPLES:

```
sage: SymmetricGroup(3).algebra(QQ).center_basis()  # weakened
(...) optional - 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:

```
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A  # weakened
(...) optional - 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  # weakened
(...) optional - sage.groups sage.modules
(1,2,3,4,5,6)

sage: A.coproduct_on_basis(g)  # weakened
(...) optional - sage.groups sage.modules
(1,2,3,4,5,6) # (1,2,3,4,5,6)
```

`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:

```
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A  # weakened
(...) optional - sage.groups sage.modules
Algebra of
Cyclic group of order 6 as a permutation group over Integer Ring

sage: a = A.an_element(); a  # weakened
(...) optional - sage.groups sage.modules
() + 3*(1,2,3,4,5,6) + 3*(1,3,5)(2,4,6)

sage: a.counit()  # weakened
(...) optional - sage.groups sage.modules
() # () + 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_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
# optional - 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
# optional - sage.groups sage.modules
(1,2,3,4,5,6)
```

```sage
A.counit_on_basis(g)
# optional - sage.groups sage.modules
1
```

group()

Return the underlying group of the group algebra.

EXAMPLES:

```sage
GroupAlgebras(QQ).example(GL(3, GF(11))).group()
# optional - sage.groups sage.modules
General Linear Group of degree 3 over Finite Field of size 11
```

```sage
SymmetricGroup(10).algebra(QQ).group()
# optional - 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 no 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
S2 = SymmetricGroup(2)
# optional - sage.groups
sage: GroupAlgebra(S2).is_integral_domain()
# optional - sage.groups sage.modules
False
```

```sage
S1 = SymmetricGroup(1)
# optional - sage.groups
sage: GroupAlgebra(S1).is_integral_domain()
# optional - sage.groups sage.modules
True
```
example($G=None$)

Return an example of group algebra.

EXAMPLES:

```python
sage: GroupAlgebras(QQ['x']).example()  # optional - sage.groups sage.modules
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))  # optional - sage.groups sage.modules
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()  # optional - sage.groups sage.modules
[Category of hopf algebras over Rational Field]

sage: sorted(C.super_categories(), key=str)  # optional - sage.groups sage.modules
[Category of hopf algebras with basis over Rational Field, Category of monoid algebras over Rational Field]
```

### 4.89 Groupoid

```python
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:

```python
sage: Groupoid(DihedralGroup(3))
Groupoid with underlying set Dihedral group of order 6 as a permutation group
```

```python
classmethod an_instance()
Returns an instance of this class.

EXAMPLES:

```python
sage: Groupoid.an_instance() # indirect doctest
Groupoid with underlying set Symmetric group of order 8! as a permutation group
```

```python
super_categories()
EXAMPLES:

```python
sage: Groupoid(DihedralGroup(3)).super_categories()
[Category of sets]
```

### 4.90 Groups

```python
class sage.categories.groups.Groups(base_category)
Bases: CategoryWithAxiom_singleton
The category of (multiplicative) groups, i.e. monoids with inverses.

EXAMPLES:

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

**Algebras**
alias of GroupAlgebras

```python
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: C5 = CyclicPermutationGroup(5)  # optional - sage.groups
sage: C4 = CyclicPermutationGroup(4)  # optional - sage.groups
sage: S4 = SymmetricGroup(3)  # optional - sage.groups
sage: C = cartesian_product([C5, C4, S4])  # optional - sage.groups
sage: C.group_generators()  # optional - sage.groups
Family (((1,2,3,4,5), (), ()),
    ((), (1,2,3,4), ()),
    ((), (), (1,2)),
    ((), (), (2,3)))

We check the other portion of github issue #16718 is fixed:

sage: len(C.j_classes())  # optional - sage.groups
1

An example with an infinitely generated group (a better output is needed):

sage: G = Groups.free([1,2])  # optional - sage.groups
sage: H = Groups.free(ZZ)  # optional - sage.groups
sage: C = cartesian_product([G, H])  # optional - sage.groups
sage: C.monoid_generators()  # optional - sage.groups
Lazy family (gen(i))_{i in The Cartesian product of (...)}

order()

Return the cardinality of self.

EXAMPLES:

sage: C = cartesian_product([SymmetricGroup(10), SL(2, GF(3))])  # optional - sage.groups sage.rings.finite_rings
sage: C.order()  # optional - sage.groups sage.rings.finite_rings
87091200

Todo: this method is just here to prevent FiniteGroups.ParentMethods to call _cardinality_from_iterator.

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]

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
          $\{0, 1, \ldots, n - 1\}$
        • names – a string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix

        EXAMPLES:

        sage: Groups.Commutative.free(index_set=ZZ)  # optional - sage.groups
        Free abelian group indexed by Integer Ring
        
        sage: Groups().Commutative().free(ZZ)  # optional - sage.groups
        Free abelian group indexed by Integer Ring
        
        sage: Groups().Commutative().free(5)  # optional - sage.groups
        Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
        
        sage: F.<x,y,z> = Groups().Commutative().free(); F  # optional - sage.groups
        Multiplicative Abelian group isomorphic to Z x Z x Z

    class ElementMethods
        Bases: object

    conjugacy_class()
        Return the conjugacy class of self.

        EXAMPLES:

        sage: D = DihedralGroup(5)  # optional - sage.groups
        sage: g = D((1,3,5,2,4))  # optional - sage.groups
        sage: g.conjugacy_class()  # optional - 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]), # optional - sage.groups
                          (continues on next page)
optional - sage.groups sage.rings.finite_rings sage.modules

sage: h = H(matrix(GF(5), 2, [1,1, 0,1]))

sage: 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] )

sage: G = SL(2, GF(2))

sage: g = G.gens()[0]

sage: g.conjugacy_class()
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)

sage: g = G([[1,1], [0,1]])

sage: 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

\textbf{cayley_table}(\textit{names='letters', elements=None})

Return the “multiplication” table of this multiplicative group, which is also known as the “Cayley table”.

\textbf{Note}: The order of the elements in the row and column headings is equal to the order given by the table’s \texttt{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 \texttt{translation()} method.

For groups, this routine should behave identically to the \texttt{multiplication_table()} method for magmas, which applies in greater generality.

\textbf{INPUT}:

- \textit{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 \texttt{list()}, padded to a common width with leading 'a's.
– 'digits' - base 10 representation of the elements' positions in the list given by \texttt{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.

\begin{itemize}
\item \texttt{elements} - default = \texttt{None}. A list of elements of the group, in forms that can be coerced into the structure, e.g. 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 \texttt{column_keys()} method. Or the \texttt{elements} can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.
\end{itemize}

\textbf{OUTPUT:}

An object representing the multiplication table. This is an \texttt{OperationTable} object and even more documentation can be found there.

\textbf{EXAMPLES:}

Permutation groups, matrix groups and abelian groups can all compute their multiplication tables.

\begin{verbatim}
sage: G = DiCyclicGroup(3)  # optional - sage.groups
sage: T = G.cayley_table()  # optional - sage.groups
sage: T.column_keys()       # optional - sage.groups
(((), (5,6,7), ..., (1,4,2,3)(5,7))
sage: T                  # optional - sage.groups
* 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| 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 h
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 b
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
l| l j k i g h b c a e f d
\end{verbatim}

\begin{verbatim}
sage: M = SL(2, 2)             # optional - sage.groups
sage: M.cayley_table()       # optional - sage.groups
* a b c d e f
+---------
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
\end{verbatim}

\texttt{(continues on next page)}
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.

```
sage: C = CyclicPermutationGroup(11)  # optional - sage.groups
sage: C.cayley_table(names='digits')  # optional - sage.groups
*  00  01  02  03  04  05  06  07  08  09  10
+---------------------------------
00|  00  01  02  03  04  05  06  07  08  09  10
01|  01  02  03  04  05  06  07  08  09  10  00
02|  02  03  04  05  06  07  08  09  10  00  01
03|  03  04  05  06  07  08  09  10  00  01  02
04|  04  05  06  07  08  09  10  00  01  02  03
05|  05  06  07  08  09  10  00  01  02  03  04
06|  06  07  08  09  10  00  01  02  03  04  05
07|  07  08  09  10  00  01  02  03  04  05  06
08|  08  09  10  00  01  02  03  04  05  06  07
09|  09  10  00  01  02  03  04  05  06  07  08
10|  10  00  01  02  03  04  05  06  07  08  09
```

```
sage: G = QuaternionGroup()  # optional - sage.groups
sage: names = ['1', 'I', '-1', '-I', 'J', '-K', '-J', 'K']  # optional - sage.groups
sage: G.cayley_table(names=names)  # optional - sage.groups
*  1   I  -1  -I   J  -K  -J  K
+------------------------
 1| 1   I  -1  -I   J  -K  -J  K
 I| I  -1  -I   1   K   J  -K  -J
-1| -1  -I   1   I   J   K   J  -K
-1| -I  1   I  -1  -K  -J   K   J
 J| J  -K  -J   K  -1  -I   1   I
```

(continues on next page)
-K| -K -J K J I -1 -I 1
-J| -J K J -K 1 I -1 -I
K| K J -K -J -I 1 I -1

sage: A = AbelianGroup([2, 2])  #
˓→optional - sage.groups
sage: A.cayley_table(names='elements')  #
˓→optional - sage.groups
* 1 f1 f0 f0*f1
+------------------------
1| 1 f1 f0 f0*f1
f1| f1 1 f0*f1 f0
f0| f0 f0*f1 1 f1
f0*f1| f0*f1 f0 f1 1

The change_names() routine behaves similarly, but changes an existing table “in-place.”

sage: G = AlternatingGroup(3)  #
˓→optional - sage.groups
sage: T = G.cayley_table()  #
˓→optional - sage.groups
sage: T.change_names('digits')  #
˓→optional - sage.groups
sage: T  #
˓→optional - sage.groups
* 0 1 2
+----
0| 0 1 2
1| 1 2 0
2| 2 0 1

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.

sage: G = SL(2,ZZ)  #
˓→optional - sage.groups
sage: G  #
˓→optional - sage.groups
Special Linear Group of degree 2 over Integer Ring
sage: identity = matrix(ZZ, [[1,0], [0,1]])  #
˓→optional - sage.groups
sage: G.cayley_table(elements=[identity, -identity])  #
˓→optional - sage.groups
* 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. 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.
sage: from sage.matrix.operation_table import OperationTable  # doc
˓→optional - sage.groups
sage: G = DiCyclicGroup(3)  # doc
˓→optional - sage.groups
sage: commutator = lambda x, y: x*y*x^-1*y^-1  # doc
˓→optional - sage.groups
sage: T = OperationTable(G, commutator)  # doc
˓→optional - sage.groups
sage: T  # doc
˓→optional - sage.groups
. 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
b| a a a a a c c c c c c
c| a a a a a a b b b b b b
d| 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
f| a a a a a a b b b b b b
g| a b c a b c a c b a c b
h| a b c a b c a c b a c b
i| a b c a b c b a c b a c
j| a b c a b c a c b a c b
k| a b c a b c a c b a c b
l| a b c a b c b a c b a

sage: trans = T.translation()  # doc
˓→optional - sage.groups
sage: comm = [trans['a'], trans['b'], trans['c']]  # doc
˓→optional - sage.groups
sage: comm  # doc
˓→optional - sage.groups
[(), (5,6,7), (5,7,6)]

sage: P = G.cayley_table(elements=comm)  # doc
˓→optional - sage.groups
sage: P  # doc
˓→optional - sage.groups
* a b c
+-----
a| a b c
b| b c a
c| c a b

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 github 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])  # optional - sage.groups
sage: c = A.conjugacy_class(A.an_element())  # optional - sage.groups
sage: type(c)  # optional - sage.groups
<class 'sage.groups.conjugacy_classes.ConjugacyClass_with_category'>
```

\textbf{group\_generators()}\smallskip

Return group generators for \textit{self}.

This default implementation calls \texttt{gens()}, for backward compatibility.

EXAMPLES:

```
sage: A = AlternatingGroup(4)  # optional - sage.groups
sage: A.group_generators()  # optional - sage.groups
Family ((2,3,4), (1,2,3))
```

\textbf{holomorph()}\smallskip

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:

```
sage: G = Groups().example()  # optional - sage.groups
sage: G.holomorph()  # optional - sage.groups
Traceback (most recent call last):
...
NotImplementedError: holomorph of General Linear Group of degree 4 over Rational Field not yet implemented
```

\textbf{monoid\_generators()}\smallskip

Return the generators of \textit{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:

```
sage: A = AlternatingGroup(4)  # optional - sage.groups
sage: A.monoid_generators()  # optional - sage.groups
Family ((2,3,4), (1,2,3))
```

(continues on next page)
semidirect_product($N, mapping, check=True$)

The semi-direct product of two groups

EXAMPLES:

```
sage: G = Groups().example()
#...
optional - sage.groups
sage: G.semidirect_product(G, Morphism(G, G))
#...
optional - sage.groups
Traceback (most recent call last):
...
NotImplementedError: 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

documentation

example()

EXAMPLES:

```
sage: Groups().example()
#...
optional - sage.groups
```

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: Groups.free(index_set=ZZ)
#...
optional - sage.groups
```

(continues on next page)
4.91 Hecke modules

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
sage: C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
```

TODO: check that this is what we want:

```sage
sage: 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
sage: HeckeModules(IntegerRing())
Category of Hecke modules over Integer Ring
sage: HeckeModules(FiniteField(5))
    #optional - sage.rings.finite_rings
Category of Hecke modules over Finite Field of size 5
```

The base ring doesn’t have to be a principal ideal domain:

```sage
sage: HeckeModules(PolynomialRing(IntegerRing(), 'x'))
Category of Hecke modules over Univariate Polynomial Ring in x over Integer Ring
```

class Homsets(category, *args)
    Bases: HomsetsCategory
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
alias of 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(s=None)

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 generates the component.

**EXAMPLES:**

```python
c sage: C = HighestWeightCrystals()
c sage: C
Category of highest weight crystals
c sage: C.super_categories()
[Category of crystals]
c sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

**class ElementMethods**

Bases: object

```python
def string_parameters(word=None)
    Return the string parameters of self corresponding to the reduced word word.
    Given a reduced expression \( w = s_{i_1} \cdots s_{i_k} \), 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: mg.f_string([1,1,1,2,2]) == mg.f_string([1,1,2,2,1])
True

sage: x = mg.f_string([1,1,1,2,2,1,3,3,2,1,1,1])

sage: x.string_parameters(w0)
[4, 1, 1, 2, 2, 2]

sage: x.string_parameters([3,2,1,3,2,3])
[2, 3, 7, 0, 0, 0]

sage: x == mg.f_string([1]*7 + [2]*3 + [3]*2)
True

sage: B = crystals.infinity.Tableaux("A5")

sage: 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],
.....: [3,3,3,3,3,3,3,5],
.....: [4,4,4,6,6,6],
.....: [5,6]])

sage: b.string_parameters([1,2,1,3,2,1,4,3,2,1,5,4,3,2,1])
[0, 1, 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]])

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])
[7, 12, 15, 8, 10, 0]

sage: C = crystals.Tableaux(["C",2], shape=[2,1])

sage: mg = C.highest_weight_vector()

sage: lw = C.lowest_weight_vectors()[0]

sage: lw.e_string([2,1,1,1,2,2,1]) == mg
True

sage: lw.e_string([1,2,2,1,1,1,2]) == mg
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:

        sage: C = crystals.Letters(['A',5])
        sage: C.highest_weight_vectors()
        (1,)
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C, 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)])

\textbf{digraph}(\text{subset}=\text{None}, \text{index}\_\text{set}=\text{None}, \text{depth}=\text{None})

Return the DiGraph associated to self.

\textbf{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

\textbf{EXAMPLES:}

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

\textbf{highest\_weight\_vector()}

Returns the highest weight vector if there is a single one; otherwise, raises an error.

Caveat: this assumes that \textit{highest\_weight\_vectors()} returns a list or tuple.

\textbf{EXAMPLES:}

sage: C = crystals.Letters(['A',5])
sage: C.highest_weight_vector()
1

\textbf{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.

\textbf{EXAMPLES:}

sage: C = crystals.Letters(['A',5])
sage: C.highest_weight_vectors()
(1,)

sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C, 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)])
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.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: 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])
```

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 $\langle \rho', \lambda - \mu \rangle$ where $\langle \rho', \alpha_i \rangle = 1$ for all $i \in I$ (in particular, take $\rho' = \sum_{i \in I} h_i$).

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

$$
\dim_q B(\lambda) = \prod_{\alpha' \in \Delta_+^\vee} \left( \frac{1 - q^{\langle \lambda + \rho, \alpha' \rangle}}{1 - q^{\langle \rho, \alpha' \rangle}} \right)^{\text{mult } \alpha},
$$

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 $q$
- $prec$ – (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.
- $use_product$ – (default: False) if we have a finite crystal and True, use the product formula

EXAMPLES:

```python
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 + 6*q^7 + 6*q^6 + 6*q^5 + 6*q^4 + 5*q^3 + 4*q^2 + 2*q + 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

We check with a finite tensor product:

sage: TP = crystals.TensorProduct(C, C)
sage: TP.cardinality()
25600

sage: qdim = TP.q_dimension(use_product=True); qdim
# long time
q^32 + 2*q^31 + 8*q^30 + 15*q^29 + 34*q^28 + 63*q^27 + 110*q^26 + 175*q^25 + 276*q^24 + 389*q^23 + 550*q^22 + 725*q^21 + 930*q^20 + 1131*q^19 + 1362*q^18 + 1548*q^17 + 1736*q^16 + 1858*q^15 + 1947*q^14 + 1944*q^13 + 1918*q^12 + 1777*q^11 + 1628*q^10 + 1407*q^9 + 1186*q^8 + 928*q^7 + 720*q^6 + 498*q^5 + 342*q^4 + 201*q^3 + 117*q^2 + 48*q + 26
sage: qdim(1)
# long time
25600

sage: TP.q_dimension() == qdim
# long time
True

The \(q\)-dimensions of infinite crystals are returned as formal power series:

sage: C = crystals.LSPaths(['A',2,1], [1,0,0])
sage: C.q_dimension(prec=5)
1 + q + 2*q^2 + 2*q^3 + 4*q^4 + 0(q^5)

sage: C.q_dimension(prec=10)
1 + q + 2*q^2 + 2*q^3 + 4*q^4 + 5*q^5 + 7*q^6 + 9*q^7 + 13*q^8 + 16*q^9 + 0(q^10)
**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:**

```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: T.highest_weight_vectors()
[[[[1]], [[1]], [2]],
 [[3]], [[1]], [2]],
 [[[-2]], [[1]], [2]]]
sage: L = filter(lambda x: x.is_highest_weight(), T)
sage: 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]],
 [[3]], [[1]], [2]],
 [[[-2]], [[1]], [2]]]
sage: L = filter(lambda x: x.is_highest_weight(), T)
sage: tuple(L) == tuple(T.highest_weight_vectors_iterator())
True
```

**extra_super_categories()**

**EXAMPLES:**
additional_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.additional_structure()

todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

example()

Returns an example of highest weight crystals, as per Category.example().

EXAMPLES:

super_categories()

EXAMPLES:

4.93 Hopf algebras

class sage.categories.hopf_algebras.HopfAlgebras(base, name=None)

Bases: Category_over_base_ring

The category of Hopf algebras.

EXAMPLES:

sage: HopfAlgebras(QQ)
Category of hopf algebras over Rational Field
sage: HopfAlgebras(QQ).super_categories()
[Category of bialgebras over Rational Field]

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
A = HopfAlgebrasWithBasis(QQ).example(); A
#optional - 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
sage: [a,b] = A.algebra_generators()
#optional - sage.groups
sage: a, a.antipode()
#optional - sage.groups
(B[(1,2,3)], B[(1,3,2)])
sage: b, b.antipode()
#optional - sage.groups
(B[(1,3)], B[(1,3)])
```

class Morphism(s=None)
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:

```sage
N = NonCommutativeSymmetricFunctions(QQ)
#optional - sage.combinat
sage: R = N.ribbon()
#optional - sage.combinat
sage: R.antipode_by_coercion.__module__
'sage.categories.hopf_algebras'
sage: R.antipode_by_coercion(R[1,3,1])
#optional - sage.combinat
-R[2, 1, 2]
```
class Super\((\text{base\_category})\)

Bases: SuperModulesCategory

The category of super Hopf algebras.

**Note:** A super Hopf algebra is *not* simply a Hopf algebra with a \(\mathbb{Z}/2\mathbb{Z}\) grading due to the signed bialgebra compatibility conditions.

class ElementMethods

Bases: object

antipode()

Return the antipode of self.

EXAMPLES:

```
\[
\begin{align*}
\text{sage: } & A = \text{SteenrodAlgebra}(3) \quad \#\text{optional - sage.combinat sage.modules} \\
\text{sage: } & a = A.an_element() \quad \#\text{optional - sage.combinat sage.modules} \\
\text{sage: } & a, a\text{.antipode()} \quad \#\text{optional - sage.combinat sage.modules} \\
\end{align*}
\]
\[
(2 \, Q_1 \, Q_3 \, P(2,1), \, Q_1 \, Q_3 \, P(2,1))
\]```

dual()

Return the dual category.

EXAMPLES:

The category of super Hopf algebras over any field is self dual:

```
\[
\begin{align*}
\text{sage: } & C = \text{HopfAlgebras}(\text{QQ}).\text{Super()} \\
\text{sage: } & C\text{.dual()} \\
\end{align*}
\]
Category of super hopf algebras over Rational Field
```

class TensorProducts\((\text{category, *args})\)

Bases: TensorProductsCategory

The category of Hopf algebras constructed by tensor product of Hopf algebras

class ElementMethods

Bases: object

class ParentMethods

Bases: object

extra_super_categories()

EXAMPLES:

```
\[
\begin{align*}
\text{sage: } & C = \text{HopfAlgebras}(\text{QQ}).\text{TensorProducts()} \\
\text{sage: } & C\text{.extra\_super\_categories()} \\
\text{sage: } & \text{sorted}(C\text{.super\_categories()}, \text{key=str}) \\
\end{align*}
\]
\[
\begin{align*}
\text{[Category of hopf algebras over Rational Field,} \\
\text{Category of tensor products of algebras over Rational Field,} \\
\text{Category of tensor products of coalgebras over Rational Field]} \\
\end{align*}
\]```
**WithBasis**

alias of *HopfAlgebrasWithBasis*

**dual()**

Return the dual category

**EXAMPLES:**

The category of Hopf algebras over any field is self dual:

```
sage: C = HopfAlgebras(QQ)
sage: C.dual()
Category of hopf algebras over Rational Field
```

**super_categories()**

**EXAMPLES:**

```
sage: 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:**

```
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]
```

We now show how to use a simple Hopf algebra, namely the group algebra of the dihedral group (see also `AlgebrasWithBasis`):

```
sage: A = C.example(); A                      # optional - 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
sage: A.__custom_name = "A"                      # optional - sage.groups
sage: A.category()                               # optional - sage.groups
Category of finite dimensional hopf algebras with basis over Rational Field
sage: A.one_basis()                              # optional - sage.groups
()                                              # optional - sage.groups
sage: A.one()                                    # optional - sage.groups
(continues on next page)
```
optional - sage.groups
B[()]

sage: A.base_ring()  #...

optional - sage.groups
Rational Field
sage: A.basis().keys()  #...

optional - sage.groups
Dihedral group of order 6 as a permutation group

sage: [a, b] = A.algebra_generators()  #...

optional - sage.groups
sage: a, b  #...

optional - sage.groups
(B[(1,2,3)], B[(1,3)])

sage: a^3, b^2  #...

optional - sage.groups
(B[()], B[()])

sage: a*b  #...

optional - sage.groups
B[(1,2)]

sage: A.product  # todo: not quite ...

optional - sage.groups
<bound method MagmaticAlgebras.WithBasis.ParentMethods._product_from_product_on_ basis_multiply of A>

sage: A.product(b, b)  #...

optional - sage.groups
B[()]

sage: A.zero().coproduct()  #...

optional - sage.groups
0

sage: A.zero().coproduct().parent()  #...

optional - sage.groups
A # A

sage: a.coproduct()  #...

optional - sage.groups
B[(1,2,3)] # B[(1,2,3)]

sage: TestSuite(A).run(verbose=True)  #...

optional - 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_new() . . . pass
running ._test_nonzero_equal() . . . 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_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass

```
sage: A.__class__
# todo: not implemented
˓→optional - sage.groups
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_˓→category'>
sage: A.element_class
# todo: not implemented
˓→optional - sage.groups
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_˓→category.element_class'>
```

Let us look at the code for implementing `A`:

```
sage: A[PPP]  # todo: not implemented
˓→optional - sage.groups
```

class ElementMethods
    Bases: object
    Filtered
        alias of FilteredHopfAlgebrasWithBasis
    FiniteDimensional
        alias of FiniteDimensionalHopfAlgebrasWithBasis
    Graded
        alias of GradedHopfAlgebrasWithBasis

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:

```python
sage: A = HopfAlgebrasWithBasis(ZZ).example(); A
optional - sage.groups
An example of Hopf algebra with basis: the group algebra of the
Dihedral group of order 6 as a permutation group over Integer Ring
sage: A = HopfAlgebrasWithBasis(QQ).example()
optional - sage.groups
sage: [a,b] = A.algebra_generators()
optional - sage.groups
sage: a, A.antipode(a)
(1,2,3), B[(1,3,2)]
optional - sage.groups
sage: b, A.antipode(b)
(1,3), B[(1,3)]
```

**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: A = HopfAlgebrasWithBasis(QQ).example()
optional - sage.groups
sage: W = A.basis().keys(); W
Dihedral group of order 6 as a permutation group
sage: w = W.gen(0); w
B[(1,2,3)]
optional - sage.groups
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:
example($G=None$)

Returns an example of algebra with basis:

```
sage: HopfAlgebrasWithBasis(QQ['x']).example() # optional - 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)) # optional - 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
sage: Semigroups().HTrivial().Inverse_extra_super_categories()
[Category of j trivial semigroups]
sage: Monoids().HTrivial().Inverse()
Category of h trivial groups
```

## 4.96 Infinite Enumerated Sets

**AUTHORS:**


```python
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
    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]
    ```

    class ParentMethods
        Bases: object

        `list()`
        Raise an error because `self` is an infinite enumerated set.

        EXAMPLES:

        ```sage
        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

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

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

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: QQ.is_integral_domain()
True
sage: Parent(QQ, category=IntegralDomains()).is_integral_domain()
True

(continues on next page)
4.98 J-trivial semigroups

```python
sage: L.<z> = LazyLaurentSeriesRing(QQ) # optional - sage.combinat
sage: L.is_integral_domain() # optional - sage.combinat
True
sage: L.is_integral_domain(proof=True) # optional - sage.combinat
True
```

4.99 Kac-Moody Algebras

AUTHORS:
- Travis Scrimshaw (07-15-2017): Initial implementation

```python
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:
```
```python
sage: L = LieAlgebra(QQ, cartan_type=['A', 2]) # optional - sage.combinat sage.modules
sage: L.cartan_type() # optional - sage.combinat sage.modules
['A', 2]
```
```python
weyl_group()
        Return the Weyl group of self.
        EXAMPLES:
```
```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2])  # optional - sage.combinat sage.modules

sage: L.weyl_group()  # optional - sage.combinat sage.modules
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)
```

**example** ($n=2$)

Return an example of a Kac-Moody algebra as per `Category.example`.

**EXAMPLES:**

```
sage: from sage.categories.kac_moody_algebras import KacMoodyAlgebras
sage: KacMoodyAlgebras(QQ).example()  # optional - sage.combinat sage.modules
Lie algebra of ['A', 2] in the Chevalley basis
```

We can specify the rank of the example:

```
sage: KacMoodyAlgebras(QQ).example(4)  # optional - sage.combinat sage.modules
Lie algebra of ['A', 4] in the Chevalley basis
```

**super_categories**()

**EXAMPLES:**

```
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 $T^{(j)} = \frac{T^j}{j!}$.

**EXAMPLES:**

```python
```
\texttt{sage: Vir = lie_conformal_algebras.Virasoro(QQ)}
\texttt{#optional - sage.combinat sage.modules}
\texttt{sage: Vir.inject_variables()} \texttt{#optional - sage.combinat sage.modules}

Defining \texttt{L}, \texttt{C}
\texttt{sage: L.T()} \texttt{#optional - sage.combinat sage.modules}
\texttt{TL}
\texttt{sage: L.T(3)} \texttt{#optional - sage.combinat sage.modules}
\texttt{6*T^*(3)L}
\texttt{sage: C.T()} \texttt{#optional - sage.combinat sage.modules}
\texttt{0}

\textbf{bracket}(\textit{rhs})

The $\lambda$-bracket of these two elements.

EXAMPLES:

The brackets of the Virasoro Lie conformal algebra:

\texttt{sage: Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0}
\texttt{#optional - sage.combinat sage.modules}
\texttt{sage: L.bracket(L)} \texttt{#optional - sage.combinat sage.modules}
\texttt{{0: TL, 1: 2*L, 3: 1/2*C}}
\texttt{sage: L.bracket(L.T())} \texttt{#optional - sage.combinat sage.modules}
\texttt{{0: 2*T^*(2)L, 1: 3*TL, 2: 4*L, 4: 2*C}}

Now with a current algebra:

\texttt{sage: V = lie_conformal_algebras.Affine(QQ, 'A1')} \texttt{#optional - sage.combinat sage.modules}
\texttt{sage: V.gens()} \texttt{#optional - sage.combinat sage.modules}
\texttt{(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])}
\texttt{sage: E = V.0; H = V.1; F = V.2} \texttt{#optional - sage.combinat sage.modules}
\texttt{sage: H.bracket(H)} \texttt{#optional - sage.combinat sage.modules}
\texttt{{1: 2*B['K']}}
\texttt{sage: E.bracket(F)} \texttt{#optional - sage.combinat sage.modules}
\texttt{{0: B[alphacheck[1]], 1: B['K']}}

\textbf{nproduct}(\textit{rhs}, \textit{n})

The $n$-th product of these two elements.

EXAMPLES:

\texttt{sage: Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0} \texttt{#optional - sage.combinat sage.modules}

(continues on next page)
sage: L.nproduct(L, 3)  # optional - sage.combinat sage.modules
1/2*C
sage: L.nproduct(L.T(), 0)  # optional - sage.combinat sage.modules
2*T^(2)L
sage: V = lie_conformal_algebras.Affine(QQ, 'A1')  # optional - sage.combinat sage.modules
sage: E = V.0; H = V.1; F = V.2  # optional - sage.combinat sage.modules
sage: E.nproduct(H, 0) == - 2*E  # optional - sage.combinat sage.modules
True
sage: E.nproduct(F, 1)  # optional - sage.combinat sage.modules
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)  # optional - sage.combinat sage.modules
    sage: Vir.ideal()  # optional - 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.

    EXAMPLES:
    sage: LieConformalAlgebras(QQ).FinitelyGenerated()
Category of finitely generated lie conformal algebras over Rational Field

    FinitelyGeneratedAsLambdaBracketAlgebra()
        The category of finitely generated Lambda bracket algebras.

    EXAMPLES:
sage: 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:
sage: from sage.categories.lambda_bracket_algebras import LambdaBracketAlgebras
sage: 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(base_category)
Bases: CategoryWithAxiom_over_base_ring

The category of Lambda bracket algebras with basis.

EXAMPLES:
sage: LieConformalAlgebras(QQbar).WithBasis()  #...
sage: optional - 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:
sage: V = lie_conformal_algebras.NeveuSchwarz(QQ)  #...
    optional - sage.combinat sage.modules
sage: V.inject_variables()  #...
    optional - sage.combinat sage.modules
Defining L, G, C
sage: G.T(3).index()  #...
    optional - sage.combinat sage.modules
('G', 3)
sage: v = V.an_element(); v  #...
    optional - sage.combinat sage.modules
L + G + C
sage: v.index()  #...
    optional - sage.combinat sage.modules
Traceback (most recent call last):

(continues on next page)
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:

sage: C = LieConformalAlgebras(QQbar)  #
\texttt{optional - sage.rings.number_field}
sage: C1 = C.WithBasis().FinitelyGenerated(); C1  #
\texttt{optional - sage.rings.number_field}
Category of finitely generated Lie conformal algebras with basis
over Algebraic Field
sage: C2 = C.FinitelyGenerated().WithBasis(); C2  #
\texttt{optional - sage.rings.number_field}
Category of finitely generated Lie conformal algebras with basis
over Algebraic Field
sage: C1 is C2  #
\texttt{optional - sage.rings.number_field}
True

class Graded(base_category)

Bases: GradedModulesCategory

The category of H-graded finitely generated lambda bracket algebras with basis.

EXAMPLES:

sage: C = LieConformalAlgebras(QQbar)  #
\texttt{optional - sage.rings.number_field}
sage: C.WithBasis().FinitelyGenerated().Graded()  #
\texttt{optional - sage.rings.number_field}
Category of H-graded finitely generated Lie conformal algebras
with basis over Algebraic Field

class ParentMethods

Bases: object

def degree_on_basis(m)

Return the degree of the basis element indexed by \( m \) in \( \text{self} \).

EXAMPLES:

sage: V = lie_conformal_algebras.Virasoro(QQ)  #
\texttt{optional - sage.combinat sage.modules}
sage: V.degree_on_basis(('L', 2))  #
\texttt{optional - sage.combinat sage.modules}
4
4.102 Lattice posets

class sage.categories.lattice_posets.LatticePosets(s=None)
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: L = LatticePosets()
Category of lattice posets
sage: L.super_categories()
[Category of posets]
```

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")))
optional - sage.combinat
sage: D.join( D(6), D(10) )
30
```

meet(x, y)
Returns the meet of x and y in this lattice

INPUT:
• x, y – elements of self

EXAMPLES:

```python
sage: D = LatticePoset((divisors(30), attrcall("divides")))
optional - sage.combinat
sage: D.meet( D(6), D(15) )
3
```

super_categories()
Returns a list of the (immediate) super categories of self, as per Category.super_categories().

EXAMPLES:
4.103 Left modules

```python
sage: LatticePosets().super_categories()
[Category of posets]
```

class `sage.categories.left_modules.LeftModules`

```
Bases: `Category_over_base_ring`

The category of left modules left modules over an rng (ring not necessarily with unit), i.e. an abelian group with
left multiplication by elements of the rng

EXAMPLES:
```
```
sage: LeftModules(ZZ)
Category of left modules over Integer Ring
sage: LeftModules(ZZ).super_categories()
[Category of commutative additive groups]
```

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

```python
class `sage.categories.lie_algebras.LieAlgebras`

```
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: 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

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]]

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
github issue #16820 (and github issue #16823).

class ElementMethods
    Bases: object

    bracket(rhs)

    Return the Lie bracket $[self, rhs]$.

    EXAMPLES:

    sage: # needs sage.combinat 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

    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: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: g = (X + Y + Z).exp(); g  #← needs sage.symbolic
exp(X + Y + Z)
sage: h = X.exp(); h  #← needs sage.symbolic
exp(X)
sage: g.parent()  #← needs sage.symbolic
Lie group G of Free Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: g.parent() == h.parent()  #← needs sage.symbolic
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()  #← needs sage.symbolic
Lie group H of Free Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: g.parent() == k.parent()  #← needs sage.symbolic
False
```

`killing_form(x)`

Return the Killing form of `self` and `x`.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #← needs sage.combinat sage.modules
sage: a, b, c = L.lie_algebra_generators()  #← needs sage.combinat sage.modules
sage: a.killing_form(b)  #← needs sage.combinat 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.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 3*a + b - c
sage: elt.lift()
3*b0 + b1 - b2
```
\begin{verbatim}
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)  # needs sage.combinat sage.modules

sage: x.lift()  # needs sage.combinat sage.modules
\end{verbatim}

### to_vector(order=None)

Return the vector in \texttt{g.module()} corresponding to the element \texttt{self} of \texttt{g} (where \texttt{g} is the parent of \texttt{self}).

Implement this if you implement \texttt{g.module()}. See \texttt{LieAlgebras.module()} for how this is to be done.

**EXAMPLES:**

\begin{verbatim}
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.combinat sage.modules
sage: u = L((1, 0, 0)).to_vector(); u  # needs sage.combinat sage.modules
(1, 0, 0)
sage: parent(u)  # needs sage.combinat sage.modules
Vector space of dimension 3 over Rational Field
\end{verbatim}

\begin{class}
\textbf{FiniteDimensional}(base\_category)
\end{class}

\begin{block}
\textbf{Bases:} CategoryWithAxiom\_over\_base\_ring
\end{block}

\begin{block}
\textbf{WithBasis}
\end{block}

\begin{block}
alias of \texttt{FiniteDimensional\_LieAlgebras\_WithBasis}
\end{block}

\begin{block}
\textbf{extra\_super\_categories}()
\end{block}

Implements the fact that a finite dimensional Lie algebra over a finite ring is finite.

**EXAMPLES:**

\begin{verbatim}
sage: LieAlgebras(IntegerModRing(4)).FiniteDimensional().extra\_super\_categories()
[Category of finite sets]
sage: LieAlgebras(ZZ).FiniteDimensional().extra\_super\_categories()
[]
sage: C = LieAlgebras(GF(5)).FiniteDimensional()

\end{verbatim}
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.step()
        # needs sage.combinat sage.modules
        2

class ParentMethods
    Bases: object

    baker_campbell_hausdorff(X, Y, 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 self has a coercion from the rationals.
        INPUT:
        • X – an element of self
        • Y – an element of self
        • 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_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:

```
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
sage: L.bch(X, Y, 4)                   # needs sage.combinat sage.modules
X + 1/12*[X, [X, Y]] + 1/24*[X, [[X, Y], Y]]
+ 1/2*[X, Y] + 1/12*[[X, Y], Y] + Y
```

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
```

```
bch(X, Y, 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 self has a coercion from the rationals.

**INPUT:**
- \( X \) – an element of self
- \( Y \) – an element of 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
```
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:

```
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
```

```
sage: L.bch(X, Y, 4)  # needs sage.combinat sage.modules
```

```
X + 1/12*[X, [X, Y]] + 1/24*[X, [[X, Y], Y]] + 1/2*[X, Y] + 1/12*[X, Y] + Y
```

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
```

```
```

\textbf{bracket}(lhs, rhs)

Return the Lie bracket \([lhs, rhs]\) after coercing \(lhs\) and \(rhs\) into elements of \texttt{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.

\textbf{EXAMPLES}:

```
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: L.bracket(x, x + y)
-1*[1, 3, 2] + [3, 2, 1]
sage: L.bracket(x, 0)
```

(continues on next page)
sage: L.bracket(0, x)
\emptyset

Constructing the product space:

```
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:

```
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
```

```
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:

```
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.combinat sage.modules
sage: a, b, c = L.lie_algebra_generators()
# needs sage.combinat sage.modules
sage: L.ideal([2*a - c, b + c])
```

(continues on next page)
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field
with basis matrix:
\[
\begin{bmatrix}
1 & 0 & -1/2 \\
0 & 1 & 1 \\
\end{bmatrix}
\]

```python
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: I = 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 \( g \) is abelian if \([x, y] = 0\) for all \( x, y \in g \).

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).example()
sage: L.is_abelian()
False
sage: R = QQ['x,y']
sage: L = LieAlgebras(QQ).example(R.gens())
sage: L.is_abelian()
True
```

**is_commutative()**

Return if self is commutative. This is equivalent to self being abelian.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).example()
sage: L.<x,y> = LieAlgebra(QQ, 2)
sage: L.is_commutative()
False
```

(continued from previous page)
```python
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
```

**is_ideal**(*A*)

Return if *self* is an ideal of *A*.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).example()  #˓→ needs sage.combinat sage.modules
sage: L.is_ideal(L)  #˓→ needs sage.combinat sage.modules
True
```

**is_nilpotent**()

Return if *self* is a nilpotent Lie algebra.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #˓→ needs sage.combinat sage.modules
sage: L.is_nilpotent()  #˓→ needs sage.combinat sage.modules
True
```

**is_solvable**()

Return if *self* is a solvable Lie algebra.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  #˓→ needs sage.combinat sage.modules
sage: L.is_solvable()  #˓→ needs sage.combinat sage.modules
True
```

**killing_form**(x, y)

Return the Killing form of *x* and *y*.

**EXAMPLES:**

```python
sage: a, b, c = L.lie_algebra_generators()  #˓→ needs sage.combinat sage.modules
sage: L.killing_form(a, b + c)  #˓→ needs sage.combinat 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:

```python
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
```

`lift()`

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 Q-algebra.

EXAMPLES:

```python
sage: # needs sage.combinat sage.modules
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
```

```python
sage: a, b, c = L.lie_algebra_generators()
```

```python
sage: lifted = L.lift(2*a + b - c); lifted
2*b0 + b1 - b2
```

```python
sage: lifted.parent() is L.universal_enveloping_algebra()
True
```

`module()`

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:
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.combinat sage.modules
sage: L.module()  # needs sage.combinat sage.modules
Vector space of dimension 3 over Rational Field

subalgebra(gens, names=None, index_set=None, category=None)

Return the subalgebra of self generated by gens.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.combinat sage.modules
sage: a, b, c = L.lie_algebra_generators()  # needs sage.combinat sage.modules
sage: L.subalgebra([2*a - c, b + c])  # needs sage.combinat 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: L = LieAlgebras(QQ).example()  # needs sage.combinat sage.modules
sage: x,y = L.lie_algebra_generators()  # needs sage.combinat sage.modules
sage: L.subalgebra([x + y])  # needs sage.combinat sage.modules
Traceback (most recent call last):
... Not ImplementedError: subalgebras not yet implemented: see #17416

universal_enveloping_algebra()

Return the universal enveloping algebra of self.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # needs sage.combinat sage.modules
sage: L.universal_enveloping_algebra()  # needs sage.combinat 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.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

WithBasis
    alias of LieAlgebrasWithBasis

element( gens=None )
    Return an example of a Lie algebra as per Category.example.

    EXAMPLES:

    sage: LieAlgebras(QQ).example()  # needs 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)

super_categories()
    EXAMPLES:

    sage: 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

```python
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

    def lift(self):
        # Lift self to the universal enveloping algebra.

    def to_vector(self, order=None):
        # Return the vector in g.module() corresponding to the element self of g (where g is the parent of self).
```

EXAMPLES:

```python
sage: S = SymmetricGroup(3).algebra(QQ)
    # optional - sage.groups
sage: L = LieAlgebra(associative=S)
    # optional - sage.groups
sage: x = L.gen(3)
    # optional - sage.groups
sage: y = L.gen(1)
    # optional - sage.groups
sage: x.lift()  # optional - sage.groups
b3
sage: y.lift()  # optional - sage.groups
b1
sage: x * y  # optional - sage.groups
b1*b3 + b4 - b5
```

`Todo:` Doctest this implementation on an example not overshadowed.
Graded
alias of \texttt{GradedLieAlgebrasWithBasis}

\textbf{class ParentMethods}
Bases: object

\textbf{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 \texttt{\_bracket\_()} for the elements must be overwritten.

\textbf{EXAMPLES:}
\begin{verbatim}
  sage: L = LieAlgebras(QQ).WithBasis().example()  # optional - sage.combinat
  sage: L.bracket_on_basis(Partition([3,1]), Partition([2,2,1,1]))  # optional - sage.combinat
  0
\end{verbatim}

\textbf{dimension}()
Return the dimension of \texttt{self}.

\textbf{EXAMPLES:}
\begin{verbatim}
  sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat
  sage: L.dimension()  # optional - sage.combinat
  3
  sage: L = LieAlgebra(QQ, 'x,y', {('x','y'): {'x':1}})  # optional - sage.combinat sage.modules
  sage: L.dimension()  # optional - sage.combinat sage.modules
  2
\end{verbatim}

\textbf{from_vector}(v, order=None, coerce=False)
Return the element of \texttt{self} corresponding to the vector \(v\) in \texttt{self.module()}. Implement this if you implement \texttt{module()}; see the documentation of \verb+sage.categories.lie_algebras.LieAlgebras.module()\ for how this is to be done.

\textbf{EXAMPLES:}
\begin{verbatim}
  sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  # optional - sage.combinat
  sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u  # optional - sage.combinat
  (1, 0, 0)
sage: parent(u) is L  # optional - sage.combinat
  True
\end{verbatim}

\textbf{module}()
Return an \(R\)-module which is isomorphic to the underlying \(R\)-module of \texttt{self}.
See \verb+sage.categories.lie_algebras.LieAlgebras.module()\ for an explanation.
**pbw_basis** *(basis_key=\texttt{None}, **kwds)*

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to *self*. 

**EXAMPLES:**

```
sage: L = lie_algebras.sl(QQ, 2)  
     # optional - sage.combinat sage.modules
sage: PBW = L.pbw_basis()  
     # optional - sage.combinat sage.modules
```

**poincare_birkhoff_witt_basis** *(basis_key=\texttt{None}, **kwds)*

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to *self*. 

**EXAMPLES:**

```
sage: L = lie_algebras.sl(QQ, 2)  
     # optional - sage.combinat sage.modules
sage: PBW = L.pbw_basis()  
     # optional - sage.combinat sage.modules
```

**example** *(gens=\texttt{None})*

Return an example of a Lie algebra as per *Category.example*. 

**EXAMPLES:**

```
sage: LieAlgebras(QQ).WithBasis().example()  
     # optional - sage.combinat
An example of a Lie algebra: the abelian Lie algebra on the generators indexed by Partitions over Rational Field
```

Another set of generators can be specified as an optional argument:

```
sage: LieAlgebras(QQ).WithBasis().example(Compositions())  
     # optional - sage.combinat
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:

   $$[a_\lambda [b_\mu c]] = [[a_\lambda + \mu b_\mu]_\mu c] + (-1)^{p(a) p(b)} [b_\mu [a_\lambda c]],$$

   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

$$[\beta \gamma] = -[\gamma \beta] = K$$

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:

$$[L\lambda] = \left( T + \frac{3}{2} \lambda \right) G \quad [G\lambda] = 2L + \frac{\lambda^2}{3} C$$

See also:

- sage.algebras.lie_conformal_algebras.lie_conformal_algebra
- sage.algebras.lie_conformal_algebras.examples

AUTHORS:


```python
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
```

Some subcategories:

```sage
LieConformalAlgebras(QQbar).FinitelyGenerated().WithBasis()  #
```

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
Category of H-graded super Lie conformal algebras over Algebraic Real Field
sage: CGS is LieConformalAlgebras(AA).Super().Graded()
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:
Entry: LieConformalAlgebras(QuaternionAlgebra(2))

Traceback (most recent call last):
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)
    sage: R.inject_variables()
    Defining L, G, C
    sage: G.is_even_odd()
    1

FinitelyGeneratedAsLambdaBracketAlgebra
    alias of FinitelyGeneratedLieConformalAlgebras

Graded
    alias of GradedLieConformalAlgebras

class ParentMethods
    Bases: object

Super
    alias of SuperLieConformalAlgebras

WithBasis
    alias of LieConformalAlgebrasWithBasis

element()
    An example of parent in this category.

    EXAMPLES:

    sage: LieConformalAlgebras(QQ).example()
    The Virasoro Lie conformal algebra over Rational Field

super_categories()
    The list of super categories of this category.

    EXAMPLES:
4.107 Lie Conformal Algebras With Basis

AUTHORS:


```python
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 semigroups,
 Category of 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 sets,
 Category of sets with partial maps,
 Category of objects]
```

4.107. Lie Conformal Algebras With Basis

The category of Lie conformal algebras with basis.

```python
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:

    sage: LieConformalAlgebras(QQbar).WithBasis()
    #optional - 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.
```

4.107. Lie Conformal Algebras With Basis
EXAMPLES:

```python
sage: C = LieConformalAlgebras(QQbar)  #
optional - sage.rings.number_field
sage: CWF = C.WithBasis().FinitelyGenerated(); CWF  #
optional - sage.rings.number_field
Category of finitely generated Lie conformal algebras with basis over Algebraic Field
sage: CWF is C.FinitelyGenerated().WithBasis()  #
optional - sage.rings.number_field
True
```

class Graded(base_category)

Bases: `GradedLieConformalAlgebrasCategory`

The category of H-graded finitely generated Lie conformal algebras with basis.

EXAMPLES:

```python
sage: LieConformalAlgebras(QQbar).WithBasis().FinitelyGenerated().Graded()  #
optional - 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:

```python
sage: LieConformalAlgebras(AA).WithBasis().FinitelyGenerated().Super()  #
optional - 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()  #
optional - sage.rings.number_field
sage: C.Graded().Super()  #
optional - 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()  #
optional - sage.rings.number_field
True
```

class Graded(base_category)

Bases: `GradedLieConformalAlgebrasCategory`

The category of H-graded Lie conformal algebras with basis.
EXAMPLES:

```python
sage: LieConformalAlgebras(QQbar).WithBasis().Graded()    #optional - sage.rings.number_field
Category of H-graded Lie conformal algebras with basis over Algebraic Field
```

class **Super**(base_category)

Bases: `SuperModulesCategory`

The category of super Lie conformal algebras with basis.

EXAMPLES:

```python
sage: LieConformalAlgebras(AA).WithBasis().Super()    #optional - sage.rings.number_field
Category of super Lie conformal algebras with basis over Algebraic Real Field
```

class **Graded**(base_category)

Bases: `GradedLieConformalAlgebrasCategory`

The category of H-graded super Lie conformal algebras with basis.

EXAMPLES:

```python
sage: LieConformalAlgebras(QQbar).WithBasis().Super().Graded()    #optional - 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:

```python
sage: from sage.categories.lie_groups import LieGroups
sage: 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:

```python
sage: from sage.categories.lie_groups import LieGroups
sage: LieGroups(QQ).additional_structure()
```

```python
super_categories()
```

EXAMPLES:

```python
sage: from sage.categories.lie_groups import LieGroups
sage: LieGroups(QQ).super_categories()
[Category of topological groups, Category of smooth manifolds over Rational Field]
```

### 4.109 Loop Crystals

class `sage.categories.loop_crystals.KirillovReshetikhinCrystals`(s=\texttt{None})

Bases: `\texttt{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 $\ell = \min\{c, \varphi(b) \mid b \in B\}$. 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:

```python
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)
```

```python
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)
```

`lusztig\_involution()`

Return the result of the classical Lusztig involution on `self`.

EXAMPLES:
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]
sage: mg.lusztig_involution()
[[[-2, -2, 1], [-1, -1, 2]]]
sage: elt = mg.f_string([2,1,3,2]); elt
[[[3, -2, 1], [4, -1, 2]]]
sage: elt.lusztig_involution()
[[[-4, -2, 1], [-3, -1, 2]]]

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:

    sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: L = crystals.KirillovReshetikhin(['A',2,1],2,1)
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, 1]], [[1, 1]], [[2]]],
 [[[1]], [[2, 2]], [[1, 2]], [[2]]],
 [[[1]], [[1, 3]], [[1], [1]], [[3]]],
 [[[1]], [[2, 3]], [[1, 2]], [[3]]],
 [[[1]], [[3, 3]], [[1, 3]], [[3]]],
 [[[[2]], [[1, 1]], [[1, 2]], [[1]]],
 [[[2]], [[1, 2]], [[2, 2]], [[1]]],
 [[[2]], [[1, 3]], [[2, 3]], [[1]]],
 [[[2]], [[2, 3]], [[2, 2]], [[3]]],
 [[[2]], [[3, 3]], [[2, 3]], [[3]]],
 [[[3]], [[1, 1]], [[1, 3]], [[1]]],
 [[[3]], [[1, 2]], [[1, 3]], [[2]]],
 [[[3]], [[2, 2]], [[2, 3]], [[2]]],
 [[[3]], [[1, 3]], [[3, 3]], [[1]]],
 [[[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]]]
Alternatively, one can compute the combinatorial $R$-matrix using the isomorphism method of digraphs:

```
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)
(True, {
    [[[1]], [[2], [3]]]: [[[1], [3]], [[2]]],
    [[[3]], [[2], [3]]]: [[[2], [3]], [[3]]],
    [[[3]], [[1], [3]]]: [[[1], [3]], [[3]]],
    [[[[1], [3]], [[1]]], [[[1]], [[1]]],
    [[[[1], [2]]], [[[1], [2]], [[1]]],
    [[[2], [[1], [2]]], [[[2], [2]], [[3]]],
    [[[1], [2]], [[2]], [[1]]],
    [[[2], [1], [3]]]: [[[1], [2]], [[3]],
    [[[2], [[2], [3]]]: [[[2], [3]], [[2]]]})
```

**affinization()**

Return the corresponding affinization crystal of `self`.

**EXAMPLES:**

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.affinization()
Affinization of Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r, →s)=(1,1)
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)
```

**b_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:**

```
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]
```

(continues on next page)
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]

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

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]]

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,)],)

is_perfect(ell=None)

Check if self is a perfect crystal of level ell.

A crystal $B$ is perfect of level $\ell$ if:
1. $B$ is isomorphic to the crystal graph of a finite-dimensional $U'_q(g)$-module.
2. $B \otimes B$ is connected.
3. There exists a $\lambda \in X$, such that $\text{wt}(B) \subset \lambda + \sum_{i \in I} \mathbb{Z}_{\leq 0} \alpha_i$ and there is a unique element in $B$ of classical weight $\lambda$.
4. For all $b \in B$, $\text{level}(\varepsilon(b)) \geq \ell$.
5. For all $\Lambda$ dominant weights of level $\ell$, there exist unique elements $b_\Lambda, b^\Lambda \in B$, such that $\varepsilon(b_\Lambda) = \Lambda = \varphi(b^\Lambda)$.

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:

```python
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
```

Todo: Implement a version for tensor products of KR crystals.

```python
level()
```

Return the level of self when self is a perfect crystal.

See also:

```python
is_perfect()
```

EXAMPLES:

```python
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
```

```python
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]]

module_generator()

Return the unique module generator of classical weight $s\Lambda_r$ of the Kirillov-Reshetikhin crystal $B^{r,s}$.

EXAMPLES:

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 self.

The $q$-dimension of a KR crystal is defined as the $q$-dimension of the underlying classical crystal.

EXAMPLES:

sage: KRC = crystals.KirillovReshetikhin(['A',2,1], 2,2)
sage: KRC.q_dimension()
q^4 + q^3 + 2*q^2 + q + 1
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: KRC.q_dimension()
q^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

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
Return the value $s$ in self written as $B^{r,s}$.

**EXAMPLES:**

```python
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,4)
sage: K.s()
sage: K.s()
4
```

### 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 `e_string_to_ground_state()`) and counting the number of affine Kashiwara operators $e_0$ applied on the way.

**OUTPUT:** an integer

**EXAMPLES:**

```python
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]
sage: t.affine_grading()
1

sage: K = crystals.KirillovReshetikhin(['A',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]], [[1]], [[1]]] 3
[[[2]], [[1]], [[1]]] 2
[[[1]], [[2]], [[1]]] 1
[[[3]], [[2]], [[1]]] 0

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]], [[1]], [[1]]] 2
[[[2]], [[1]], [[1]]] 1
[[[-1]], [[1]], [[1]]] 1
[[[1]], [[2]], [[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:

```sage
K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]
sage: t.e_string_to_ground_state()
(0, 2)
```

```sage
K = crystals.KirillovReshetikhin(['C',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]; t
[[[1]], [[1]]]
sage: t.e_string_to_ground_state()
(0,)
sage: x = t.e(0)
sage: x.e_string_to_ground_state()
()
sage: y = t.f_string([[2],[1]]; y
[[[2]], [[1]]]
sage: y.e_string_to_ground_state()
()
```

definition

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:

```python
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]]] 4
[[[1],[1]]] 3
[[[[1],[1]]]] 1
[[[1],[[1]]]] 0
[[[1],[2],[1]]] 1
[[[2],[2],[1]]] 2
[[[-1],[[1]]]] 2
[[[1],[-1],[1]]] 2
[[[2],[-1],[[1]]]] 2
]
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: T = crystals.TensorProduct(K)
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
```

```python
class ParentMethods
    Bases: object

```
cardinality()
Return the cardinality of self.

EXAMPLES:

```python
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.cardinality()
100
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
```

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:

```python
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]]])
```

maximal_vector()
Return the maximal vector of self.

EXAMPLES:

```python
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: T.maximal_vector()
[[[1]], [[1]], [[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:

```python
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
+ B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
sage: R.<t> = ZZ[]
sage: 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]]
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
....: == T.one_dimensional_configuration_sum())
True
```

```
extra_super_categories()
EXAMPLES:

```python
sage: from sage.categories.loop_crystals import KirillovReshetikhinCrystals
sage: KirillovReshetikhinCrystals().TensorProducts().extra_super_categories()
[Category of finite regular loop crystals]
```

```
super_categories()
EXAMPLES:

```python
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

$$H(\epsilon_0(b \otimes b')) = H(b \otimes b') + \begin{cases} 
1 & \text{if } i = 0 \text{ and LL}, \\
-1 & \text{if } i = 0 \text{ and RR}, \\
0 & \text{otherwise},
\end{cases}$$

where LL (resp. RR) denote $\epsilon_0$ 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
- $B'$ – 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:
    b, H(b)
([[], [[1], [2]]], 1)
([[[1, 1]], [[1], [2]]], 0)
([[[2, -2]], [[1], [2]]], 1)
([[[1, -2]], [[1], [2]]], 1)
```

REFERENCES:

[KKMMNN1992]

class sage.categories.loop_crystals.LoopCrystals(s=None)

Bases: Category_singleton

The category of $U_q'(g)$-crystals, where $g$ is of affine type.
The category is called loop crystals as we can also consider them as crystals corresponding to the loop algebra $g_0[t]$, where $g_0$ is the corresponding classical type.

EXAMPLES:

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: C = LoopCrystals()
sage: C
Category of loop crystals
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)
```

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:

```python
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:
```

(continues on next page)
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:

```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]
```

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)
```

super_categories()

EXAMPLES:

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: LoopCrystals().super_categories()
[Category of crystals]
```

class sage.categories.loop_crystals.RegularLoopCrystals(s=None)

Bases: Category_singleton

The category of regular $U'_q(g)$-crystals, where $g$ 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 = crystalsProjectedLevelZeroLSPaths(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))]
```
super_categories()

EXAMPLES:

```
sage: from sage.categories.loop_crystals import RegularLoopCrystals
sage: RegularLoopCrystals().super_categories()
[Category of regular crystals,
  Category of loop crystals]
```

### 4.110 L-trivial semigroups

```python
class sage.categories.l_trivial_semigroups.LTrivialSemigroups(base_category):
    Bases: CategoryWithAxiom

    @staticmethod
    def super_categories():
        return [Category of regular semigroups,]

    @staticmethod
    def extra_super_categories():
        return [Category of regular semigroups,]

    @staticmethod
    def RTrivial_extra_super_categories():
        return [Category of regular semigroups,]
```

### 4.111 Magmas

```python
class sage.categories.magmas.Magmas(s=None):
    Bases: Category_singleton

    The category of (multiplicative) magmas.  A magma is a set with a binary operation *.

    EXAMPLES:

    ```python
    sage: Magmas()
    Category of magmas
    sage: Magmas().super_categories()
    [Category of sets]
    ```
```

(continues on next page)
The following axioms are defined by this category:

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

```python
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()  #optional - sage.groups
        True
        sage: SymmetricGroup(1).algebra(ZZ).is_field()  #optional - sage.groups
        False
        sage: SymmetricGroup(2).algebra(QQ).is_field()  #optional - sage.groups
        False

        extra_super_categories()
        EXAMPLES:

        sage: MCA = Magmas().Commutative().Algebras(QQ)
sage: MCA.extra_super_categories()
        [Category of commutative magmas]
```

This implements the fact that the algebra of a commutative magma is commutative:
sage: 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

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: A = SymmetricGroupAlgebra(QQ, 3)     # optional - sage.groups
            sage: x = cartesian_product([A([1,3,2]), A([2,3,1])])     # optional - sage.groups
            sage: y = cartesian_product([A([1,3,2]), A([2,3,1])])     # optional - sage.groups
            sage: x = cartesian_product([A,A]).product(x,y)     # optional - sage.groups
            B[(0, [1, 2, 3])] + B[(1, [3, 1, 2])]
            sage: x*y     # optional - sage.groups
            B[(0, [1, 2, 3])] + B[(1, [3, 1, 2])]

documentation

example()
    Return an example of Cartesian product of magmas.

    EXAMPLES:

    sage: C = Magmas().CartesianProducts().example(); C
    The Cartesian product of (Rational Field, Integer Ring, Integer Ring)
    sage: C.category()
    Join of Category of Cartesian products of commutative rings and
    Category of Cartesian products of metric spaces
    sage: sorted(C.category().axioms())
    ['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
    (continues on next page)
'AdditiveUnital', 'Associative', 'Commutative',
'Distributive', 'Unital']

\texttt{sage: TestSuite(C).run()}

**extra_super_categories()**

This implements the fact that a subquotient (and therefore a quotient or subobject) of a finite set is finite.

**EXAMPLES:**

\texttt{sage: Semigroups().CartesianProducts().extra_super_categories()}

[Category of semigroups]

\texttt{sage: Semigroups().CartesianProducts().super_categories()}

[Category of semigroups, Category of Cartesian products of magmas]

**class Commutative** \texttt{(base\_category)}

**Bases:** \texttt{CategoryWithAxiom\_singleton}

**class Algebras** \texttt{(category, \_\_args)}

**Bases:** \texttt{Algebras\_Category}

**extra_super_categories()**

**EXAMPLES:**

\texttt{sage: MCA = Magmas().Commutative().Algebras(QQ)}

\texttt{sage: MCA.extra_super_categories()}

[Category of commutative magmas]

\texttt{sage: MCA.super_categories()}

[Category of magma algebras over Rational Field, 
Category of commutative magmas]

In particular, commutative monoid algebras are commutative algebras:

\texttt{sage: MoCA = Monoids().Commutative().Algebras(QQ)}

\texttt{sage: MoCA.is_subcategory(Algebras(QQ).Commutative())}

\texttt{True}

**class CartesianProducts** \texttt{(category, \_\_args)}

**Bases:** \texttt{CartesianProductsCategory}

**extra_super_categories()**

Implement the fact that a Cartesian product of commutative additive magmas is still an commu-
tative additive magmas.

**EXAMPLES:**

\texttt{sage: C = Magmas().Commutative().CartesianProducts()}

\texttt{sage: C.extra_super_categories()}

[Category of commutative magmas]

\texttt{sage: C.axioms()}

\texttt{frozenset\{(\textquoteleft\textquoteleft Commutative\textquoteleft\textquoteleft)\}}
class ParentMethods
    Bases: object
    
is_commutative()
    Return True, since commutative magmas are commutative.
    EXAMPLES:

    sage: Parent(QQ, category=CommutativeRings()).is_commutative()
    True

class ElementMethods
    Bases: object
    
is_idempotent()
    Test whether self is idempotent.
    EXAMPLES:

    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

    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

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
        1. '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.

```python
sage: G = CyclicPermutationGroup(5)         #optional - sage.groups
sage: G.multiplication_table()             #optional - sage.groups
* a b c d e
+--------
a| a b c d e
b| b c d e a
. . .
```
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.

```python
sage: from sage.categories.examples.finite_semigroups import LeftRegularBand
sage: L = LeftRegularBand(('a', 'b'))
sage: T = L.multiplication_table(names='digits')            #optional - sage.modules
sage: T.column_keys()                                        #optional - sage.modules
('a', 'ab', 'b', 'ba')
sage: T                                                        #optional - 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
```
Specifying the elements in an alternative order can provide more insight into how the operation behaves.
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 = sorted(L.list())
sage: L.multiplication_table(elements=elts)   # optional - sage.modules
  * a b c d e f g h i j k l m n o
  +---------------------------
   a| a b c d e b b c c d e e e
   b| b b c c b b c c c c c c c c
   c| c c c c c c c c c c c c c c
   d| d e e e e e e e e e e e e e e e e
   e| e e e e e e e e e e e e e e e e e e e e e
   f| g g h h h f g h i j i j i j
   g| g g h h g g h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h h
sage: T.translation()  # optional - sage.groups sage.modules
{'x': (), 'y': (1, 2, 3), 'z': (1, 3, 2)}
sage: T  # optional - sage.groups sage.modules
* x y z
+------
x| x y z
y| y z x
z| z x y

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)
'ab'

A parent in $\text{Magma}()$ 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 $S.\text{product}(x,y)$, and reciprocally.

As a bonus, $S.\text{product}$ models the binary function from $S$ to $S$:

sage: bin = S.product
sage: bin(x, y)
'ab'

Currently, $S.\text{product}$ 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 $S.\text{product}$ 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

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 \(\text{Magmas}()\) must either implement \texttt{product()} \(\) in the parent class or \texttt{\_mul\_} in the element class. By default, the addition method on elements \(x\texttt{.\_mul\_}(y)\) calls \(S\texttt{.product}(x,y)\), and reciprocally.

As a bonus, \(S\texttt{.product}\) models the binary function from \(S\) to \(S\):

```
sage: bin = S.product
sage: bin(x,y)
'ab'
```

Currently, \(S\texttt{.product}\) 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 \(S\texttt{.product}\) 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
```

**class Realizations**(\texttt{category, \*args})

Bases: \texttt{RealizationsCategory}

**class ParentMethods**

Bases: object

\texttt{product\_by\_coercion}(\texttt{left, right})

Default implementation of product for realizations.

This method coerces to the realization specified by \texttt{self.realization\_of()}. \texttt{\_a\_realization()}, computes the product in that realization, and then coerces back.

EXAMPLES:

```
sage: Out = Sets().WithRealizations().example().Out(); Out  # optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field
  in the Out basis
sage: Out.product  # optional - sage.combinat sage.modules
<bound method Magmas.Realizations.ParentMethods.product\_by\_coercion
  of The subset algebra of {1, 2, 3} over Rational Field
```

(continues on next page)
class SubcategoryMethods

    Bases: object

    Associative()

    Return the full subcategory of the associative objects of self.

    A (multiplicative) magma Magmas $M$ is associative if, for all $x, y, z \in M$,

    \[ x \ast (y \ast z) = (x \ast y) \ast z \]

    See also:

    Wikipedia article Associative_property

    EXAMPLES:

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

    Commutative()

    Return the full subcategory of the commutative objects of self.

    A (multiplicative) magma Magmas $M$ is commutative if, for all $x, y \in M$,

    \[ x \ast y = y \ast x \]

    See also:

    Wikipedia article Commutative_property

    EXAMPLES:

    sage: Magmas().Commutative()
    Category of commutative magmas

    sage: Monoids().Commutative()
    Category of commutative monoids

    Distributive()

    Return the full subcategory of the objects of self where $\ast$ is distributive on $+$.

    INPUT:

    * self – a subcategory of Magmas and AdditiveMagmas
Given that Sage does not yet know that the category \texttt{MagmasAndAdditiveMagmas} is the intersection of the categories \texttt{Magmas} and \texttt{AdditiveMagmas}, the method \texttt{MagmasAndAdditiveMagmas}.\texttt{SubcategoryMethods.Distributive()} is not available, as would be desirable, for this intersection.

This method is a workaround. It checks that \texttt{self} is a subcategory of both \texttt{Magmas} and \texttt{AdditiveMagmas} and upgrades it to a subcategory of \texttt{MagmasAndAdditiveMagmas} before applying the axiom. It complains otherwise, since the \texttt{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
```

\textbf{FinitelyGenerated()}

Return the subcategory of the objects of \texttt{self} that are endowed with a distinguished finite set of (multiplicative) magma generators.

**EXAMPLES:**

This is a shorthand for \texttt{FinitelyGeneratedAsMagma()}, which see:

```python
sage: Magmas().FinitelyGenerated()
Category of finitely generated magmas
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated enumerated groups
```

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:

```python
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:

```python
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated enumerated groups
```

```python
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 self.

This axiom is in fact only meaningful for semigroups. This stub definition is here as a workaround for github 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 self.
A (multiplicative) magma \textit{Magmas} \( M \) is \textit{unital} if it admits an element 1, called \textit{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 \( M \text{.one()} \).

\textbf{See also:}

Wikipedia article \textit{Unital_magma#unital}

\textbf{EXAMPLES:}

```sage
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
```

class Subquotients(category, *args)

\textit{Bases:} SubquotientsCategory

The category of subquotient magmas.

See \textit{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 \( A \) can be rewritten as follows:

\[
\text{• for any two } a, b \in S \text{ the identity } a \times_S b = r(l(a) \times_G l(b)) \text{ holds.}
\]

This is used by this category to implement the product \( \times_S \) of \( S \) from \( l \) and \( r \) and the product of \( G \).

\textbf{EXAMPLES:}

```sage
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]
```

class ParentMethods

\textit{Bases:} object

\textbf{product}(x, y)

Return the product of two elements of \textit{self}.

\textbf{EXAMPLES:}

```sage
s = Semigroups().Subquotients().example()
s
An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup
s.product(S(19), S(3))
19
```
Here is a more elaborate example involving a sub algebra:

```sage
sage: Z = SymmetricGroup(5).algebra(QQ).center()  # optional - sage.groups
sage: B = Z.basis()  # optional - sage.groups
```

```python
class Unital(base_category):
    Bases: CategoryWithAxiom_singleton

class Algebras(category, *args):
    Bases: AlgebrasCategory
    extra_super_categories()

    EXAMPLES:

    sage: MCA = Magmas().Commutative().Algebras(QQ)
    sage: MCA.extra_super_categories()
    [Category of commutative magmas]

    This implements the fact that the algebra of a commutative magma is commutative:

    sage: 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

class CartesianProducts(category, *args):
    Bases: CartesianProductsCategory

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object
    one()
    Return the unit of this Cartesian product.
    It is built from the units for the Cartesian factors of self.

    EXAMPLES:

    sage: cartesian_product([QQ, ZZ, RR]).one()
    (1, 1, 1.00000000000000)

    extra_super_categories()
    Implement the fact that a Cartesian product of unital magmas is a unital magma

    EXAMPLES:
The category of Cartesian products of unital magmas does not contain the category of unital magmas directly. However, it is a supercategory of the category of unital magmas.

```python
 sage: C = Magmas().Unital().CartesianProducts()
 sage: C.extra_super_categories()
 [Category of unital magmas]
 sage: C.axioms()
 frozenset({'Unital'})
 sage: Monoids().CartesianProducts().is_subcategory(Monoids())
 True
```

### Class `ElementMethods`

Bases: `object`

```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:
        ```python
        sage: C = Magmas().Unital().Inverse().CartesianProducts()
        sage: C.extra_super_categories()
        [Category of inverse unital magmas]
        sage: sorted(C.axioms())
        ['Inverse', 'Unital']
        ```
```

### Class `ParentMethods`

Bases: `object`

```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:
        ```python
        sage: S = SymmetricGroup(2)  # optional - sage.groups
        sage: S.is_empty()           # optional - sage.groups
        False
        sage: M = Monoids().example()
        sage: M.is_empty()
        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:

```
sage: M = Monoids().example(); M
An example of a monoid:
the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.one()
```

class `Realizations`(*category, *args)

```
Bases: `RealizationsCategory`
```

class `ParentMethods`

```
Bases: object
```

```
one()
Return the unit element of `self`.
```

EXAMPLES:

```
sage: from sage.combinat.root_system.extended_affine_weyl_group import*
        ExtendedAffineWeylGroup # optional - sage.combinat sage.groups
sage: PvW0 = ExtendedAffineWeylGroup(["A",2,1]).PvW0() # optional - sage.combinat sage.groups
sage: PvW0 in Magmas().Unital().Realizations() # optional - sage.combinat sage.groups
True
sage: PvW0.one() # optional - sage.combinat sage.groups
1
```

class `SubcategoryMethods`

```
Bases: object
```

```
Inverse()
```

```
Return the full subcategory of the inverse objects of `self`.
```

An inverse `:class:` (multiplicative) magma `<Magmas>` is a *unital magma* such that every element admits both an inverse on the left and on the right. Such a magma is also called a *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()

EXAMPLES:

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

```
sage: super_categories()
EXAMPLES:
```
```
sage: Magmas().super_categories()
[Category of sets]
```

### 4.112 Magmas and Additive Magmas

class sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas(s=None)

Bases: Category_singleton

The category of sets \((S, +, \ast)\) with an additive operation ‘+’ and a multiplicative operation \(\ast\)

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 rings
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().
  →AdditiveUnital().Unital()
Category of semirings
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().
  →AdditiveUnital().AdditiveInverse().Unital()
Category of rings
```

This category is really meant to represent the intersection of the categories of \(\text{Magmas}\) and \(\text{AdditiveMagmas}\); 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
```

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 $x, y, z \in M$,

\[
x \ast (y + z) = x \ast y + x \ast z \quad \text{and} \quad (x + y) \ast z = x \ast z + y \ast z
\]

EXAMPLES:

```python
sage: from sage.categories.magas_and_additive_magas import...
  ...MagmasAndAdditiveMagmas
sage: 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:

```python
sage: Magmas() & AdditiveMagmas()
Join of Category of magmas and Category of additive magmas
```

Still, the natural syntax works:

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

thanks to a workaround implemented in Magmas.SubcategoryMethods.Distributive():

```python
sage: (Magmas() & AdditiveMagmas()).Distributive.__module__
'sage.categories.magas'
```

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:

```python
sage: from sage.categories.magas_and_additive_magas import...
  ...MagmasAndAdditiveMagmas
sage: MagmasAndAdditiveMagmas().additional_structure()
```
super_categories()

EXAMPLES:

```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: MagmasAndAdditiveMagmas().super_categories()
[Category of magmas, Category of additive magmas]
```

### 4.113 Non-unital non-associative algebras

```python
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 github issue #15043).

EXAMPLES:

```
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]
```

**Associative**

alias of `AssociativeAlgebras`

```python
class ParentMethods
    Bases: object

    algebra_generators()
        Return a family of generators of this algebra.
        EXAMPLES:
```

```
sage: F = AlgebrasWithBasis(QQ).example(); F
# optional - sage.combinat sage.modules
An example of an algebra with basis:
    the free algebra on the generators ('a', 'b', 'c') over Rational Field
sage: F.algebra_generators()
# optional - sage.combinat sage.modules
Family (B[word: a], B[word: b], B[word: c])
```

**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

def 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: p_mult = matrix([[0,0,0], [0,0,-1], [0,0,0]])
# optional - sage.combinat sage.modules
sage: q_mult = matrix([[0,0,1], [0,0,0], [0,0,0]])
# optional - sage.combinat sage.modules
sage: A = algebras.FiniteDimensional(QQ,
       \[p_mult, q_mult, matrix(QQ, 3, 3)], 'p,q,z')
# optional - sage.combinat sage.modules
....:
    [p_mult, q_mult, matrix(QQ, 3, 3)], 'p,q,z')
sage: A.inject_variables()
# optional - sage.combinat sage.modules
Defining p, q, z
sage: p * q
# optional - sage.combinat sage.modules
z
sage: q * p
# optional - sage.combinat sage.modules
-z
sage: A.derivations_basis()
# optional - sage.combinat sage.modules
(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: A = algebras.Exterior(QQ, 1)
# optional - sage.combinat sage.modules
sage: A.derivations_basis()
# optional - sage.combinat sage.modules
(0 0) 
(0 1) 
)
```

(continues on next page)
Category Framework, Release 10.1

(continued from previous page)

```python
sage: D = A.module_morphism(matrix=A.derivations_basis()[0],  #
          codomain=A)
...

sage: one, e = A.basis()  #
...

sage: 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)

```python
class ParentMethods
    Bases: object

    algebra_generators()

    Return generators for this algebra.
    This default implementation returns the basis of this algebra.
    OUTPUT: a family

    See also:
    • basis()
    • MagmaticAlgebras.ParentMethods.algebra_generators()

    EXAMPLES:

    sage: D4 = DescentAlgebra(QQ, 4).B()  #
    ...
    sage: D4.algebra_generators()  #
    ...
    Lazy family (...)_{i in Compositions of 4}

    sage: R.<x> = ZZ[]
    sage: P = PartitionAlgebra(1, x, R)  #
    ...
    sage: P.algebra_generators()  #
    ...
    Lazy family (Term map
    from Partition diagrams of order 1
    to Partition Algebra of rank 1 with parameter x
    over Univariate Polynomial Ring in x
    over Integer Ring(i))_{i in Partition diagrams of order 1}
```

```python
product()

    The product of the algebra, as per Magmas.ParentMethods.product()

    By default, this is implemented using one of the following methods, in the specified order:
    • product_on_basis()
    • product_by_coercion()

    EXAMPLES:
```
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: a, b, c = A.algebra_generators()  # optional - sage.combinat sage.modules
sage: A.product(a + 2*b, 3*c)  # optional - sage.combinat sage.modules
3*B[word: ac] + 6*B[word: bc]

**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:**

```python
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: Word = A.basis().keys()  # optional - sage.combinat sage.modules
sage: A.product_on_basis(Word("abc"), Word("cba"))  # optional - sage.combinat sage.modules
B[word: abccba]
```

**additional_structure()**

Return None.

Indeed, the category of (magmatic) algebras defines no new structure: a morphism of modules and of magmas between two (magmatic) algebras is a (magmatic) algebra morphism.

**See also:**

`Category.additional_structure()`

**Todo:** This category should be a `CategoryWithAxiom`, the axiom specifying the compatibility between the magma and module structure.

**EXAMPLES:**

```python
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(QQ).additional_structure()
```

**super_categories()**

**EXAMPLES:**

```python
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MA = MagmaticAlgebras(QQ)
sage: MA.super_categories()
```

(continues on next page)
```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: 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.

**super_categories()**

EXAMPLES:

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

**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:**

```
sage: from sage.categories.manifolds import Manifolds
c
```

**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:**
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:
        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:
        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:
        sage: from sage.categories.manifolds import Manifolds
        sage: C = Manifolds(RR).FiniteDimensional()
        sage: 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:

        sage: from sage.categories.manifolds import Manifolds
        sage: Manifolds(RR).AlmostComplex()
        Category of almost complex manifolds over Real Field with 53 bits of precision

    Analytic()
        Return the subcategory of the analytic objects of self.
        
        EXAMPLES:

        sage: from sage.categories.manifolds import Manifolds
        sage: Manifolds(RR).Analytic()
        Category of analytic manifolds over Real Field with 53 bits of precision

    Complex()
        Return the subcategory of manifolds over C of self.
        
        EXAMPLES:

        sage: from sage.categories.manifolds import Manifolds
        sage: Manifolds(CC).Complex()
        Category of complex manifolds over Complex Field with 53 bits of precision

4.114. Manifolds
Connected()  
Return the full subcategory of the connected objects of self.

EXAMPLES:

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Connected()
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()
Category of differentiable manifolds over Real Field with 53 bits of precision
```

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
Category of finite dimensional connected manifolds over Real Field with 53 bits of precision
```

Smooth()  
Return the subcategory of the smooth objects of self.

EXAMPLES:

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Smooth()
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:

```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).additional_structure()
```

super_categories()  
EXAMPLES:
4.115 Matrix algebras

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

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
```

super_categories()

EXAMPLES:

```python
sage: MatrixAlgebras(QQ).super_categories()
[Category of algebras over Rational Field]
```

4.116 Metric Spaces

```python
class `sage.categories.metric_spaces.MetricSpaces`(category, *args)

Bases: `MetricSpacesCategory`

The category of metric spaces.

A metric on a set $S$ is a function $d : S \times S \to \mathbb{R}$ such that:

- $d(a, b) \geq 0$,
- $d(a, b) = 0$ if and only if $a = b$.

A metric space is a set $S$ with a distinguished metric.

Implementation

Objects in this category must implement either a `dist` on the parent or the elements or `metric` on the parent; otherwise this will cause an infinite recursion.

Todo:

- Implement a general geodesics class.
- Implement a category for metric additive groups and move the generic distance $d(a, b) = |a - b|$ there.
- Incorporate the length of a geodesic as part of the default distance cycle.

EXAMPLES:
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((0, 0), (2, 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()

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()
```

```python
sage: C.extra_super_categories()
[Category of complete metric spaces]
```

```python
sage: C.super_categories()
[Category of Cartesian products of metric spaces,
 Category of complete metric spaces]
```

```python
sage: C.axioms()
frozenset({'Complete'})
```

```python
sage: R2 = RR.cartesian_product(RR)
sage: R2 in MetricSpaces()
True
```

```python
sage: R2 in MetricSpaces().Complete()
True
```

```python
sage: QR = QQ.cartesian_product(RR)
sage: QR in MetricSpaces()
True
```

```python
sage: QR in MetricSpaces().Complete()
False
```

---

```python
class ElementMethods
    Bases: object

    abs()
    Return the absolute value of self.

    EXAMPLES:

    sage: CC(I).abs()
    1.00000000000000
```

```python
dist(b)
    Return the distance between self and other.

    EXAMPLES:

    sage: UHP = HyperbolicPlane().UHP()
    ...: sage.symbolic
    sage: p1 = UHP.get_point(5 + 7*I)
    ...: sage.symbolic
    sage: p2 = UHP.get_point(1 + I)
    ...: sage.symbolic
    sage: p1.dist(p2)
    ...: sage.symbolic
    arccosh(33/7)
```

---

```python
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.
```

---

4.116. Metric Spaces
class ElementMethods
Bases: object

class ParentMethods
Bases: object
dist(a, b)
Return the distance between a and b in self.

EXAMPLES:

sage: UHP = HyperbolicPlane().UHP()  # optional - sage.symbolic
sage: p1 = UHP.get_point(5 + 7*I)    # optional - sage.symbolic
sage: p2 = UHP.get_point(1.0 + I)    # optional - sage.symbolic
sage: UHP.dist(p1, p2)              # optional - sage.symbolic
2.23230104635820

sage: PD = HyperbolicPlane().PD()  # optional - sage.symbolic
sage: PD.dist(PD.get_point(0), PD.get_point(I/2))  # optional - sage.symbolic
arccosh(5/3)

metric(*args, **kwds)
Deprecated: Use metric_function() instead. See github issue #30062 for details.

metric_function()
Return the metric function of self.

EXAMPLES:

sage: UHP = HyperbolicPlane().UHP()  # optional - sage.symbolic
sage: m = UHP.metric_function()      # optional - sage.symbolic
sage: p1 = UHP.get_point(5 + 7*I)    # optional - sage.symbolic
sage: p2 = UHP.get_point(1.0 + I)    # optional - sage.symbolic
sage: m(p1, p2)                     # optional - sage.symbolic
2.23230104635820

class SubcategoryMethods
Bases: object

Complete()
Return the full subcategory of the complete objects of self.

EXAMPLES:
class \texttt{WithRealizations}(\texttt{category}, *\texttt{args})

\texttt{class ParentMethods}

\texttt{class sage.categories.metric_spaces.MetricSpacesCategory}(\texttt{category}, *\texttt{args})

\texttt{classmethod default_super_categories}(\texttt{category})
4.117 Modular abelian varieties

```
class sage.categories.modular_abelian_varieties.ModularAbelianVarieties(Y):
    Bases: Category_over_base

    The category of modular abelian varieties over a given field.

    EXAMPLES:
    sage: ModularAbelianVarieties(QQ)
    Category of modular abelian varieties over Rational Field
```

```
class Homsets(category, *args):
    Bases: HomsetsCategory

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]
```

```
base_field() :
    EXAMPLES:
    sage: ModularAbelianVarieties(QQ).base_field()
    Rational Field
```

```
super_categories() :
    EXAMPLES:
    sage: 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:

    $r * (x * s) = (r * x) * s \quad \forall r, s \in R \text{ and } x \in M$
```
INPUT:

- **base_ring** – a ring \( R \) or subcategory of \( \text{Rings()} \)
- **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
```

**Todo:**

- Clarify the distinction, if any, with 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:

\[
r \ast x = x \ast r \quad \forall r \in R \text{ and } x \in M
\]

- 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 FreeModules($R$) category, when so prompted by a concrete use case: e.g. modeling a free module with several bases (using Sets.SubcategoryMethods.Realizations()) or with an atlas of local maps (see e.g. github issue #15916).

```
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#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:
    sage: Modules(ZZ).CartesianProducts().extra_super_categories()
    [Category of modules over Integer Ring]
    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

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())
# optional - sage.rings.finite_rings
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
```

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())
# optional - sage.rings.finite_rings
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
```

Graded
alias of GradedModules
**class** Homsets\((\text{category, *args})\)

**Bases:** HomsetsCategory

The category of homomorphism sets \(\text{hom}(X, Y)\) for \(X, Y\) modules.

**class** Endset\((\text{base_category})\)

**Bases:** CategoryWithAxiom_over_base_ring

The category of endomorphism sets \(\text{End}(X)\) 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:**

```python
sage: Modules(ZZ).Endsets().\text{extra_super_categories}()
[\text{Category of magmatic algebras over Integer Ring}]
```

**class** ParentMethods

**Bases:** object

**base_ring()**

Return the base ring of \(self\).

**EXAMPLES:**

```python
sage: E = CombinatorialFreeModule(ZZ, [1,2,3])  
optional - sage.modules
sage: F = CombinatorialFreeModule(ZZ, [2,3,4])  
optional - sage.modules
sage: H = Hom(E, F)  
optional - sage.modules
sage: H.\text{base_ring}()  
optional - sage.modules
Integer Ring
```

This \text{base_ring} method is actually overridden by \text{sage.structure.category_object.}

**CategoryObject.base_ring():**

```python
sage: H.\text{base_ring}.__module__  
optional - sage.modules
```

Here we call it directly:

```python
sage: method = H.\text{category}().parent_class.base_ring  
optional - sage.modules
sage: method.\text{__get__}(H)()  
optional - sage.modules
Integer Ring
```
zero()  
EXAMPLES:

```sage
e = CombinatorialFreeModule(ZZ, [1,2,3])
# optional - sage.modules
f = CombinatorialFreeModule(ZZ, [2,3,4])
# optional - sage.modules
H = Hom(e, f)
# optional - sage.modules
f = H.zero()
# optional - sage.modules
f
# optional - sage.modules
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(2))
# optional - sage.modules
0
f(e.monomial(3)) == f.zero()
# optional - sage.modules
True
```

base_ring()  
EXAMPLES:

```sage
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]
```

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 `iter_of_elements_coeff` iterates through the sequence \((\lambda_1, v_1), \ldots, (\lambda_k, v_k)\).

INPUT:
- `iter_of_elements_coeff` – iterator of pairs `(element, coeff)` with `element` in `self` and `coeff` in `self.base_ring()`
- `factor_on_left` (optional) if True, the coefficients are multiplied from the left; if False, the coefficients are multiplied from the right

EXAMPLES:
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: \( f \).domain() if it’s defined; otherwise it must be specified)
- category -- a category or None (default: None)

EXAMPLES:

sage: V = FiniteRankFreeModule(QQ, 2)  # optional - sage.modules
sage: e = V.basis('e'); e  # optional - sage.modules
Basis \( \langle e_0, e_1 \rangle \) on the
2-dimensional vector space over the Rational Field
sage: neg = V.module_morphism(function=operator.neg, codomain=V); neg  # optional - sage.modules
Generic endomorphism of
2-dimensional vector space over the Rational Field
sage: neg(e[0])  # optional - sage.modules
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:

sage: C = CombinatorialFreeModule(QQ, ['a','b','c'])  # optional - sage.modules
sage: TA = TensorAlgebra(C)  # optional - sage.modules
sage: TA.quotient  # optional - sage.modules

(continues on next page)
Optional - 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:

sage: A = HopfAlgebrasWithBasis(QQ).example()  #...
optional - sage.groups sage.modules
sage: A.tensor_square()  #...
optional - 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 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 construction (see github issue #15647). For now the two constructions are not distinguished which is an oversimplified model.

See also:
- dual.DualObjectsCategory
  - CovariantFunctorialConstruction.

EXAMPLES:

sage: VectorSpaces(QQ).DualObjects()
Category of duals of vector spaces over Rational Field

The dual of a vector space is a vector space:
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]
```

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]
```

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
```

```
Filtered(base_ring=None)
```

Return the subcategory of the filtered objects of `self`.

**INPUT:**
- `base_ring` – this is ignored

**EXAMPLES:**

```python
sage: Modules(ZZ).Filtered()
Category of filtered modules over Integer Ring
```

```python
sage: Coalgebras(QQ).Filtered()
Category of filtered coalgebras over Rational Field
```

```python
sage: 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:**

```python
sage: Modules(ZZ).FiniteDimensional()
Category of finite dimensional modules over Integer Ring
```

```python
sage: Coalgebras(QQ).FiniteDimensional()
Category of finite dimensional coalgebras over Rational Field
```

```python
sage: AlgebrasWithBasis(QQ).FiniteDimensional()
Category of finite dimensional algebras with basis over Rational Field
```

```
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
sage: sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
[Category of coalgebras over Rational Field,
 Category of duals of vector spaces over Rational Field]
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
sage: VectorSpaces(QQ).dual()
Category of duals of vector spaces over Rational Field
```

```
Filtered(base_ring=None)
```

Return the subcategory of the filtered objects of `self`.

**INPUT:**
- `base_ring` – this is ignored

**EXAMPLES:**

```python
sage: Modules(ZZ).Filtered()
Category of filtered modules over Integer Ring
```

```python
sage: Coalgebras(QQ).Filtered()
Category of filtered coalgebras over Rational Field
```

```python
sage: 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:**

```python
sage: Modules(ZZ).FiniteDimensional()
Category of finite dimensional modules over Integer Ring
```

```python
sage: Coalgebras(QQ).FiniteDimensional()
Category of finite dimensional coalgebras over Rational Field
```

```python
sage: 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)
˓→ optional - sage.combinat sage.modules
sage: from sage.modules.fp_graded.module import FPModule
˓→ optional - sage.combinat sage.modules
sage: FPModule(A, [0, 1], [[Sq(2), Sq(1)]]).category()
˓→ optional - sage.combinat sage.modules
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
```

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:

```
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 \texttt{WithBasis()}
- Improve the support for covariant functorial constructions categories over a base ring so as to get rid of the \texttt{base\_ring} argument.

\textbf{TensorProducts()}

Return the full subcategory of objects of \texttt{self} constructed as tensor products.

See also:

- \texttt{tensor.TensorProductsCategory}
- \texttt{RegressiveCovariantFunctorialConstruction}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: ModulesWithBasis(QQ).TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
\end{verbatim}

\textbf{WithBasis()}

Return the full subcategory of the objects of \texttt{self} with a distinguished basis.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Modules(ZZ).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
\end{verbatim}

\textbf{base\_ring()}

Return the base ring (category) for \texttt{self}.

This implements a \texttt{base\_ring} method for all subcategories of \texttt{Modules(K)}.

\textbf{EXAMPLES:}

\begin{verbatim}
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: C = Modules(Rings()) & Semigroups(); C
Join of Category of semigroups and Category of modules over rings
sage: C.base_ring()
Category of rings
sage: C.base_ring.__module__
'sage.categories.modules'
sage: C = DescentAlgebra(QQ,3).B().category()  #optional - sage.combinat sage.modules
  ... optional - sage.combinat sage.modules
sage: C.base_ring.__module__
\end{verbatim}

(continues on next page)
dual()

Return the category of spaces constructed as duals of spaces of \texttt{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 \texttt{self}, endowed with the appropriate additional structure.

**Warning:**

- This semantic of \texttt{dual} and \texttt{DualObject} is imposed on all subcategories, in particular to make \texttt{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 \texttt{graded_dual} construction (see github issue \#15647). For now the two constructions are not distinguished which is an oversimplified model.

See also:

- \texttt{dual.DualObjectsCategory}
- \texttt{CovariantFunctorialConstruction}.

**EXAMPLES:**

```
sage: VectorSpaces(QQ).DualObjects()
Category of duals of vector spaces over Rational Field
```

The dual of a vector space is a vector space:

```
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```

The dual of an algebra is a coalgebra:
The dual of a coalgebra is an algebra:

```
sage: 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:

```
sage: 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)             # optional - sage.combinat sage.modules
sage: T = A.tensor(A)                   # optional - sage.combinat sage.modules
sage: T.construction()                  # optional - sage.combinat sage.modules
(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: F = CombinatorialFreeModule(ZZ, [1,2])                # optional - sage.modules
sage: F.rename("F")                                         # optional - sage.modules
sage: G = CombinatorialFreeModule(ZZ, [3,4])                # optional - sage.modules
sage: G.rename("G")                                         # optional - sage.modules
(continues on next page)
sage: T = tensor([F, G]); T
#optional - sage.modules
F # G
sage: T.tensor_factors()
#optional - sage.modules
(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]

Nota bene:

sage: Modules(QQ)
Category of vector spaces over Rational Field
sage: Modules(QQ).super_categories()
[Category of modules over Rational Field]
AUTHORS:

• Nicolas M. Thiery (2008-2014): initial revision, axiomatization
• Jason Bandlow and Florent Hivert (2010): Triangular Morphisms
• Christian Stump (2010): github issue #9648 module_morphism’s to a wider class of codomains

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()
[Category of modules with basis over Rational Field, Category of vector spaces over Rational Field]
```

Let $X$ and $Y$ be two modules with basis. We can build $\text{Hom}(X, Y)$:

```python
sage: X = CombinatorialFreeModule(QQ, [1,2]); X.__custom_name = "X"
# optional - sage.modules
sage: Y = CombinatorialFreeModule(QQ, [3,4]); Y.__custom_name = "Y"
# optional - sage.modules
sage: H = Hom(X, Y); H
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:

```python
sage: H.zero() # todo: move this test into module once we have an example  #
# optional - sage.modules
Generic morphism:
  From: X
  To: Y
```

which we can apply to elements of $X$:

```python
sage: x = X.monomial(1) + 3 * X.monomial(2)
# optional - sage.modules
```
EXAMPLES:

We now construct a more interesting morphism by extending a function by linearity:

```python
sage: phi = H(on_basis=lambda i: Y.monomial(i + 2)); phi
optional - sage.modules

Generic morphism:
  From: X
  To:  Y
  sage: phi(x)
optional - sage.modules
```

We can retrieve the function acting on indices of the basis:

```python
sage: f = phi.on_basis()
optional - sage.modules
sage: f(1), f(2)
optional - sage.modules
(B[3], B[4])
```

\( \text{Hom}(X,Y) \) 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
sage: H in ModulesWithBasis(QQ), H in Modules(QQ)
optional - sage.modules
(False, True)
```

Some more playing around with categories and higher order homsets:

```python
sage: H.category()
optional - sage.modules
Category of homsets of modules with basis over Rational Field
sage: Hom(H, H).category()
optional - sage.modules
Category of endsets of homsets of modules with basis over Rational Field
```

Todo: End\((X)\) is an algebra.

Note: This category currently requires an implementation of an element method \texttt{support}. Once \texttt{github issue #18066} is merged, an implementation of an \texttt{items} method will be required.

```python
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 vector spaces with basis over Rational Field,
Category of Cartesian products of vector spaces over Rational Field]
```

class DualObjects(category, *args)
Bases: DualObjectsCategory

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]
```

class ElementMethods
Bases: object

coefficient(m)

Return the coefficient of m in self and raise an error if m is not in the basis indexing set.

INPUT:
* m -- a basis index of the parent of self

OUTPUT:
The B[m]-coordinate of self with respect to the basis B. Here, B denotes the given basis of the parent of self.

EXAMPLES:

```
sage: s = CombinatorialFreeModule(QQ, Partitions())  #...
    --optional - sage.combinat sage.modules
sage: z = s([4]) - 2*s([2,1]) + s([1,1,1]) + s([1])  #...
    --optional - sage.combinat sage.modules
sage: z.coefficient([4])  #...
    --optional - sage.combinat sage.modules
1
sage: z.coefficient([2,1])  #...
    --optional - sage.combinat sage.modules
-2
sage: z.coefficient(Partition([2,1]))  #...
    --optional - sage.combinat sage.modules
-2
sage: z.coefficient([1,2])  #...
    --optional - sage.combinat sage.modules
```
Traceback (most recent call last):
...
AssertionError: [1, 2] should be an element of Partitions

sage: z.coefficient(Composition([2,1]))           #→
...
AssertionError: [2, 1] should be an element of Partitions

Test that coefficient also works for those parents that do not have an element_class:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: H = pAdicWeightSpace(3)</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules sage.rings.padics</td>
<td></td>
</tr>
<tr>
<td>sage: F = CombinatorialFreeModule(QQ, H)</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules sage.rings.padics</td>
<td></td>
</tr>
<tr>
<td>sage: hasattr(H, &quot;element_class&quot;)</td>
<td>False</td>
</tr>
<tr>
<td>#→ optional - sage.modules sage.rings.padics</td>
<td></td>
</tr>
<tr>
<td>sage: h = H.an_element()</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules sage.rings.padics</td>
<td></td>
</tr>
<tr>
<td>sage: (2*F.monomial(h)).coefficient(h)</td>
<td>2</td>
</tr>
<tr>
<td>#→ optional - sage.modules sage.rings.padics</td>
<td></td>
</tr>
</tbody>
</table>

**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()

**EXAMPLES:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: B = F.basis()</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: f = B['a'] - 3*B['c']</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: f.coefficients()</td>
<td>[1, -3]</td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: f = B['c'] - 3*B['a']</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: f.coefficients()</td>
<td>[-3, 1]</td>
</tr>
<tr>
<td>#→ optional - sage.modules</td>
<td></td>
</tr>
<tr>
<td>sage: s = SymmetricFunctions(QQ).schur()</td>
<td></td>
</tr>
<tr>
<td>#→ optional - sage.combinat sage.modules</td>
<td></td>
</tr>
</tbody>
</table>

(continues on next page)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])  # optional - sage.combinat sage.modules

sage: z.coefficients()  # optional - sage.combinat sage.modules
[1, 1, 1, 1]

is_zero()

Return True if and only if self == 0.

EXAMPLES:

sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])  # optional - sage.modules

sage: B = F.basis()  # optional - sage.modules

sage: f = B['a'] - 3*B['c']  # optional - sage.modules

sage: f.is_zero()  # optional - sage.modules
False

sage: F.zero().is_zero()  # optional - sage.modules
True

sage: s = SymmetricFunctions(QQ).schur()  # optional - sage.combinat sage.modules

sage: s([2,1]).is_zero()  # optional - sage.combinat sage.modules
False

sage: s(0).is_zero()  # optional - sage.combinat sage.modules
True

sage: (s([2,1]) - s([2,1])).is_zero()  # optional - sage.combinat sage.modules
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: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")  # optional - sage.modules

sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)  # optional - sage.modules

sage: x.leading_coefficient()  # optional - sage.modules
1

sage: def key(x): return -x  # optional - sage.modules

(continues on next page)
optional - sage.modules

sage: x.leading_coefficient(key=key)  #-

optional - sage.modules

3

sage: s = SymmetricFunctions(QQ).schur()  #-

optional - sage.combinat sage.modules


optional - sage.combinat sage.modules

sage: f.leading_coefficient()  #-

optional - sage.combinat sage.modules

-5

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:

sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")  #-

optional - sage.modules

sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)  #-

optional - sage.modules

sage: x.leading_item()  #-

optional - sage.modules

(3, 4)

sage: def key(x): return -x  #-

optional - sage.modules

sage: x.leading_item(key=key)  #-

optional - sage.modules

(1, 3)

sage: s = SymmetricFunctions(QQ).schur()  #-

optional - sage.combinat sage.modules


optional - sage.combinat sage.modules

sage: f.leading_item()  #-

optional - sage.combinat sage.modules

([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: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X") #optional - sage.modules
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3) #optional - sage.modules
sage: x.leading_monomial() #optional - sage.modules
B[3]
sage: def key(x): return -x #optional - sage.modules
sage: x.leading_monomial(key=key) #optional - sage.modules
B[1]
sage: s = SymmetricFunctions(QQ).schur() #optional - sage.combinat sage.modules
sage: f.leading_monomial() #optional - 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
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]) #optional - sage.modules
sage: X.rename("X"); x = X.basis() #optional - sage.modules
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3) #optional - sage.modules
sage: x.leading_support() #optional - sage.modules
3
sage: def key(x): return -x #optional - sage.modules
sage: x.leading_support(key=key) #optional - sage.modules
1
sage: s = SymmetricFunctions(QQ).schur() #optional - sage.combinat sage.modules
sage: f.leading_support() #optional - sage.combinat sage.modules
s[3]
```
**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:**

```python
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")  # optional - sage.modules
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)  # optional - sage.modules
sage: x.leading_term()  # optional - sage.modules
B[3]
sage: def key(x): return -x  # optional - sage.modules
sage: x.leading_term(key=key)  # optional - sage.modules
3*B[1]
sage: s = SymmetricFunctions(QQ).schur()  # optional - sage.combinat sage.modules
sage: f.leading_term()  # optional - sage.combinat sage.modules
-5*s[3]
```

**length()**

Return the number of basis elements whose coefficients in `self` are nonzero.

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])  # optional - sage.modules
sage: B = F.basis()  # optional - sage.modules
sage: f = B['a'] - 3*B['c']  # optional - sage.modules
sage: f.length()  # optional - sage.modules
2
sage: s = SymmetricFunctions(QQ).schur()  # optional - sage.combinat sage.modules
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])  # optional - sage.combinat sage.modules
sage: z.length()  # optional - sage.combinat sage.modules
4
```

**map_coefficients(f)**

Mapping a function on coefficients.
INPUT:
- \( f \) – an endofunction on the coefficient ring of the free module
Return a new element of \( \text{self.parent()} \) obtained by applying the function \( f \) to all of the coefficients of \( \text{self} \).

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
#optional - sage.modules
sage: B = F.basis()
#optional - sage.modules
sage: f = B['a'] - 3*B['c']
#optional - sage.modules
sage: f.map_coefficients(lambda x: x + 5)
#optional - sage.modules
6*B['a'] + 2*B['c']
```

Killed coefficients are handled properly:

```python
sage: f.map_coefficients(lambda x: 0)
#optional - sage.modules
0
sage: list(f.map_coefficients(lambda x: 0))
#optional - sage.modules
[]
```

### 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:

```python
sage: s = SymmetricFunctions(QQ).schur()
#optional - sage.combinat sage.modules
sage: a = s([2,1]) + 2*s([3,2])
#optional - sage.combinat sage.modules
sage: a.map_coefficients(lambda x: x * 2)
#optional - sage.combinat sage.modules
2*s[2, 1] + 4*s[3, 2]
```

\( f \) needs not be injective:
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:

```python
sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])

# optional - sage.modules
sage: x = B.an_element(); x

# optional - sage.modules
2*B[-1] + 2*B[0] + 3*B[1]

sage: x.map_support(lambda i: -i)

# optional - sage.modules
3*B[-1] + 2*B[0] + 2*B[1]

f needs not be injective:

```python
sage: x.map_support(lambda i: 1)

# optional - sage.modules
7*B[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:
sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])  # optional - sage.modules
sage: x = B.an_element(); x  # optional - sage.modules
2*B[-1] + 2*B[0] + 3*B[1]
sage: x.map_support_skip_none(lambda i: -i if i else None)  # optional - sage.modules
3*B[-1] + 2*B[1]

f needs not be injective:
sage: x.map_support_skip_none(lambda i: 1 if i else None)  # optional - 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:
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])  # optional - sage.modules
sage: B = F.basis()  # optional - sage.modules
sage: f = B['a'] + 3*B['c']  # optional - sage.modules
sage: d = f.monomial_coefficients()  # optional - sage.modules
sage: d['a']  # optional - sage.modules
1
sage: d['c']  # optional - sage.modules
3

monomials()
Return a list of the monomials of self (in an arbitrary order).
The monomials of an element $a$ are defined to be the basis elements whose corresponding coefficients
of $a$ are non-zero.

EXAMPLES:
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])  # optional - sage.modules
sage: B = F.basis()  # optional - sage.modules
sage: f = B['a'] + 2*B['c']  # optional - sage.modules
sage: f.monomials()  # optional - sage.modules
(continues on next page)
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.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
  #← optional - sage.modules
sage: B = F.basis()
  #← optional - sage.modules
sage: f = B['a'] - 3*B['c']
  #← optional - sage.modules
sage: sorted(f.support())
  #← optional - sage.modules
['a', 'c']
```

support_of_term()

Return the support of self, where self is a monomial (possibly with coefficient).

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3,4]); X.rename("X")
  #← optional - sage.modules
sage: X.monomial(2).support_of_term()
  #← optional - sage.modules
2
sage: X.term(3, 2).support_of_term()
  #← optional - sage.modules
3
```

An exception is raised if self has more than one term:

```
sage: (X.monomial(2) + X.monomial(3)).support_of_term()
  #← optional - sage.modules
Traceback (most recent call last):
  ...
```
tensor(*elements)

Return the tensor product of its arguments, as an element of the tensor product of the parents of those elements.

EXAMPLES:

```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()  # optional - sage.combinat sage.modules
sage: a, b, c = A.algebra_generators()  # optional - sage.combinat sage.modules
sage: a.tensor(b, c)  # optional - sage.combinat sage.modules
```

FIXME: is this a policy that we want to enforce on all parents?

terms()

Return a list of the (non-zero) terms of self (in an arbitrary order).

See also:

monomials()

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])  # optional - sage.modules
sage: B = F.basis()  # optional - sage.modules
sage: f = B['a'] + 2*B['c']  # optional - sage.modules
sage: f.terms()  # optional - sage.modules
[B['a'], 2*B['c']]
```

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:

```python
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")  # optional - sage.modules
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)  # optional - sage.modules
sage: x.trailing_coefficient()  # optional - sage.modules
3
sage: def key(x): return -x  # optional - sage.modules
sage: x.trailing_coefficient(key=key)  # optional - sage.modules
```

(continues on next page)
trailing_item(*args, **kwds)

Return the pair \((c, k)\) where \(c \cdot \text{self.parent().monomial}(k)\) is the trailing term of \text{self}.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when \text{self} is printed.

If the default term ordering is not what is desired, a comparison key \text{key}(x), can be provided.

EXAMPLES:

\[
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
\]

\[
sage: x = 3 \cdot X.monomial(1) + 2 \cdot X.monomial(2) + X.monomial(3)
\]

\[
sage: x.trailing_item()
\]

\[
(1, 3)
\]

\[
sage: def key(x): return -x
\]

\[
sage: x.trailing_item(key=key)
\]

\[
(3, 1)
\]

\[
sage: s = SymmetricFunctions(QQ).schur()
\]

\[
sage: f = 2 \cdot s[1] + 3 \cdot s[2,1] - 5 \cdot s[3]
\]

\[
sage: f.trailing_item()
\]

\[
([1], 2)
\]

trailing_monomial(*args, **kwds)

Return the trailing monomial of \text{self}.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when \text{self} is printed.

If the default term ordering is not what is desired, a comparison key \text{key}(x), can be provided.

EXAMPLES:

\[
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
\]

\[
sage: x = 3 \cdot X.monomial(1) + 2 \cdot X.monomial(2) + X.monomial(3)
\]

\[
sage: x.trailing_monomial()
\]

(continues on next page)
sage: x.trailing_monomial()  #→
˓→optional - sage.modules
B[1]
sage: def key(x): return -x  #→
˓→optional - sage.modules
sage: x.trailing_monomial(key=key)  #→
˓→optional - sage.modules
B[3]
sage: s = SymmetricFunctions(QQ).schur()  #→
˓→optional - sage.combinat sage.modules
˓→optional - sage.combinat sage.modules
sage: f.trailing_monomial()  #→
˓→optional - sage.combinat sage.modules
s[1]

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")  #→
˓→optional - sage.modules
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)  #→
˓→optional - sage.modules
sage: x.trailing_support()  #→
˓→optional - sage.modules
1
sage: def key(x): return -x  #→
˓→optional - sage.modules
sage: x.trailing_support(key=key)  #→
˓→optional - sage.modules
3
sage: s = SymmetricFunctions(QQ).schur()  #→
˓→optional - sage.combinat sage.modules
˓→optional - sage.combinat sage.modules
sage: f.trailing_support()  #→
˓→optional - sage.combinat 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:

```
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_term()
#...
sage: def key(x):
...    return -x
#...
sage: x.trailing_term(key=key)
#...
sage: s = SymmetricFunctions(QQ).schur()
#...
#...
sage: f.trailing_term()
#...
```

Filtered

alias of `FilteredModulesWithBasis`

FiniteDimensional

alias of `FiniteDimensionalModulesWithBasis`

Graded

alias of `GradedModulesWithBasis`

class `Homsets`(category, *args)

Bases: `HomsetsCategory`

class `ParentMethods`

Bases: object

class `MorphismMethods`

Bases: object

```
on_basis()
Return the action of this morphism on basis elements.
```

OUTPUT:

- a function from the indices of the basis of the domain to the codomain

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); X.rename("X")
#...
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4]); Y.rename("Y")
#...
sage: H = Hom(X, Y)
#...
sage: x = X.basis()
#...
```
class ParentMethods
Bases: object

basis()

Return the basis of self.

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
```

```python
sage: F.basis()
```

```python
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```python
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
```

```python
sage: list(QS3.basis())
```

```python
[[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:

```python
sage: S = SymmetricGroupAlgebra(QQ, 4)
```

```python
sage: S.cardinality()
```

```
+Infinity
```

```python
sage: S = SymmetricGroupAlgebra(GF(2), 4) # not tested -- MRO bug trac
```

```
#15475    # optional - sage.groups sage.rings.finite_rings sage.modules
```

```
sage: S.cardinality() # not tested -- MRO bug trac
```

```
#15475    # optional - sage.groups sage.rings.finite_rings sage.modules
```

(continues on next page)
dimension()

Return the dimension of self.

EXAMPLES:

```python
sage: A.<x,y> = algebras.DifferentialWeyl(QQ)  #...
  # optional - sage.combinat sage.modules
sage: A.dimension()  #...
  # optional - sage.combinat sage.modules
+Infinity
```

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: R.<x,y> = QQ[]
sage: C = CombinatorialFreeModule(R, ZZ, prefix='z')  #...
  # optional - sage.modules
sage: z = C.basis()  #...
  # optional - sage.modules
sage: C.echelon_form([z[0] - z[1], 2*z[1] - 2*z[2], z[0] - z[2]])  #...
```

**is_finite()**

Return whether self is finite.

This is true if and only if self.basis().keys() and self.base_ring() are both finite.

**EXAMPLES:**

```python
sage: GroupAlgebra(SymmetricGroup(2), IntegerModRing(10)).is_finite()  # optional - sage.groups sage.modules
True
sage: GroupAlgebra(SymmetricGroup(2)).is_finite()  # optional - sage.groups sage.modules
False
sage: GroupAlgebra(AbelianGroup(1), IntegerModRing(10)).is_finite()  # optional - sage.groups sage.modules
False
```

**module_morphism** *(on_basis=None, matrix=None, function=None, diagonal=None, triangular=None, unitriangular=False, **keywords)*

Construct a module morphism from self to codomain.

Let 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:**

- **self** – a parent \( X \) in ModulesWithBasis(R) with basis \( x = (x_i)_{i \in I} \).

Exactly one of the four following options must be specified in order to define the morphism:

- **on_basis** – a function \( f \) from \( I \) to \( Y \)
- **diagonal** – a function \( d \) from \( I \) to \( R \)
- **function** – a function \( f \) from \( X \) to \( Y \)
- **matrix** – a matrix of size \( \dim Y \times \dim X \) (if the keyword side is set to 'left') or \( \dim Y \times \dim X \) (if this keyword is 'right')

Further options include:

- **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)
- **zero** – the zero of the codomain (default: codomain.zero()); can be used (with care) to define affine maps. Only meaningful with on_basis.
- **position** – 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 on_basis.
- **triangular** – (default: None) "upper" or "lower" or None:
  - "upper" - if the leading_support() of the image of the basis vector \( x_i \) is \( i \), or
  - "lower" - if the trailing_support() of the image of the basis vector \( x_i \) is \( i \).
- **unitriangular** – (default: False) a boolean. Only meaningful for a triangular morphism. As a shorthand, one may use unitriangular="lower" for triangular="lower", unitriangular=True.
- **side** – “left” or “right” (default: “left”) Only meaningful for a morphism built from a matrix.

**EXAMPLES:**

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:
\[ g(a, \sum \lambda_i x_i, c) = \sum \lambda_i f(a, i, c). \]

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); X.rename("X")  # optional - sage.modules
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4]); Y.rename("Y")  # optional - sage.modules
sage: def f(i):
....:     return Y.monomial(i) + 2*Y.monomial(i+1)
sage: phi = X.module_morphism(f, codomain=Y)  # optional - sage.modules
sage: x = X.basis(); y = Y.basis()  # optional - sage.modules
sage: phi(x[1] + x[3])
sage: phi  # optional - sage.modules
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()  # optional - sage.modules
Category of finite dimensional vector spaces with basis over Rational Field
```

With the `zero` argument, one can define affine morphisms:

```
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])  # optional - sage.modules
sage: phi(x[1] + x[3])
```

In this special case, the default category is `Sets()`:

```
sage: phi.category_for()  # optional - sage.modules
Category of sets
```

One can construct morphisms with the base ring as codomain:

```
sage: X = CombinatorialFreeModule(ZZ, [1, -1])  # optional - sage.modules
(continues on next page)```
sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=ZZ)  # ˓→optional - sage.modules

sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))  # ˓→optional - sage.modules
-1

sage: phi.category_for()  # ˓→optional - sage.modules
Category of commutative additive semigroups

sage: phi.category_for()  # todo: not implemented (ZZ is currently not in Modules(ZZ)) ˓→optional - sage.modules
Category of modules over Integer Ring

Or more generally any ring admitting a coercion map from the base ring:

sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=RR)  # ˓→optional - sage.modules

sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))  # ˓→optional - sage.modules
-1.00000000000000

sage: phi.category_for()  # ˓→optional - sage.modules
Category of commutative additive semigroups

sage: phi.category_for()  # todo: not implemented (RR is currently not in Modules(ZZ)) ˓→optional - sage.modules
Category of modules over Integer Ring

sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=Zmod(4))  # ˓→optional - sage.modules

sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))  # ˓→optional - sage.modules
3

sage: phi = Y.module_morphism(on_basis=lambda i: i, codomain=Zmod(4))  # ˓→optional - sage.modules

Traceback (most recent call last):
...
ValueError: codomain(Ring of integers modulo 4) should be a module over the base ring of the domain(Y)

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 $Y = \mathbb{C}$:

sage: X = CombinatorialFreeModule(RR, ['x', 'y'])  # ˓→optional - sage.modules

sage: Y = CombinatorialFreeModule(CC, ['z'])  # ˓→optional - sage.modules

sage: x = X.monomial('x')  # ˓→optional - sage.modules

sage: y = X.monomial('y')  # ˓→optional - sage.modules

sage: z = Y.monomial('z')  # ˓→optional - sage.modules

(continues on next page)
```python
sage: def on_basis(a):
    if a == 'x':
        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
3.00000000000000*B['x'] + 2.00000000000000*B['y']

sage: phi(v)
(3.00000000000000+2.00000000000000*I)*B['z']
```

Of course, there should be a coercion between the respective base rings of the domain and the codomain for this to be meaningful:

```python
sage: Y = CombinatorialFreeModule(QQ, ['z'])

sage: phi = X.module_morphism(on_basis=on_basis, codomain=Y)

Traceback (most recent call last):
...
ValueError: codomain(=Free module generated by {'z'} over Rational Field) should be a module over the base ring of the domain(=Free module generated by {'x', 'y'} over Real Field with 53 bits of precision)
```

```python
sage: Y = CombinatorialFreeModule(RR['q'], ['z'])

sage: phi = Y.module_morphism(on_basis=on_basis, codomain=X)

Traceback (most recent call last):
...
ValueError: codomain(=Free module generated by {'x', 'y'} over Real Field with 53 bits of precision) should be a module over
```

(continues on next page)
With the `diagonal=d` argument, this constructs the module morphism \( g \) such that

\[
g(x_i) = d(i) y_i.
\]

This assumes that the respective bases \( x \) and \( y \) of \( X \) and \( Y \) have the same index set \( I \):

```
sage: X = CombinatorialFreeModule(ZZ, [1, 2, 3]); X.rename("X")
--- optional - sage.modules
sage: from sage.arith.misc import factorial
--- optional - sage.modules
sage: phi = X.module_morphism(diagonal=factorial, codomain=X)
--- optional - sage.modules
sage: x = X.basis()
--- optional - sage.modules
sage: phi(x[1]), phi(x[2]), phi(x[3])
--- optional - sage.modules
(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")
--- optional - sage.modules
sage: x = X.basis()
--- optional - sage.modules
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); Y.rename("Y")
--- optional - sage.modules
sage: y = Y.basis()
--- optional - sage.modules
sage: m = matrix([[[0,1,2], [3,5,0]]])
--- optional - sage.modules
sage: phi = X.module_morphism(matrix=m, codomain=Y)
--- optional - sage.modules
sage: phi(x[1])
--- optional - sage.modules
3*B[4]
sage: phi(x[2])
--- optional - sage.modules
```

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")
--- optional - sage.modules
sage: x = X.basis()
--- optional - sage.modules
```

sage: Y = CombinatorialFreeModule(QQ, I); Y.rename("Y")  # optional - sage.modules
sage: f = Y.sum_of_monomials * divisors  # optional - sage.modules
sage: phi = X.module_morphism(f, triangular="upper", codomain=Y)  # optional - sage.modules
sage: phi(x[2])  # optional - sage.modules
sage: phi(x[6])  # optional - sage.modules
sage: phi(x[30])  # optional - sage.modules
sage: phi.preimage(y[2])  # optional - sage.modules
sage: phi.preimage(y[6])  # optional - sage.modules
sage: phi.preimage(y[30])  # optional - sage.modules

Since github issue #8678, one can also define a triangular morphism from a function:

sage: X = CombinatorialFreeModule(QQ, [0,1,2,3,4]); x = X.basis()  # optional - sage.modules
sage: from sage.modules.with_basis.morphism import TriangularModuleMorphismFromFunction  # optional - sage.modules
sage: def f(x): return x + X.term(0, sum(x.coefficients()))  # optional - sage.modules
sage: phi = X.module_morphism(function=f, codomain=X, ...: triangular="upper")  # optional - sage.modules
sage: phi(x[2] + 3*x[4])  # optional - sage.modules
sage: phi.preimage(_)  # optional - sage.modules

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 \( \text{Hom}(\text{codomain}, \text{domain}, \text{category}) \). This is only correct for unary functions.
Todo:
- Should codomain be `self` by default in the diagonal, triangular, and matrix cases?
- Support for diagonal morphisms between modules not sharing the same index set

`monomial(i)`
Return the basis element indexed by `i`.

**INPUT:**
- `i` – an element of the index set

**EXAMPLES:**

```sage
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
#...
˓→optional - sage.modules
sage: F.monomial('a')
˓→optional - sage.modules
B['a']
```

F.monomial is in fact (almost) a map:

```sage
sage: F.monomial
˓→optional - sage.modules
Term map from {'a', 'b', 'c'}
to Free module generated by {'a', 'b', 'c'} over Rational Field
```

`monomial_or_zero_if_none(i)`

**EXAMPLES:**

```sage
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
#...
˓→optional - sage.modules
sage: F.monomial_or_zero_if_none('a')
˓→optional - sage.modules
B['a']
sage: F.monomial_or_zero_if_none(None)
˓→optional - sage.modules
0
```

`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()`

**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:**
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")  #...
˓→optional - sage.modules
sage: x = X.basis()  #...
˓→optional - sage.modules
sage: Y = X.quotient_module([x[0] - x[1], x[1] - x[2]],
˓→optional - sage.modules
...:
already_echelonized=True)
sage: Y.print_options(prefix='y'); Y  #...
˓→optional - sage.modules
Free module generated by {2} over Rational Field
sage: y = Y.basis()  #...
˓→optional - sage.modules
sage: y[2]  #...
˓→optional - sage.modules
y[2]
sage: y[2].lift()  #...
˓→optional - sage.modules
x[2]
sage: Y.retract(x[0] + 2*x[1])  #...
˓→optional - sage.modules
3*y[2]
sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')  #...
˓→optional - sage.modules
sage: x = C.basis()  #...
˓→optional - sage.modules
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
˓→optional - sage.modules
sage: Y = C.quotient_module(gens)  #...
˓→optional - sage.modules
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:

sage: x = DihedralGroup(6).algebra(QQ).random_element()  #...
˓→optional - sage.groups sage.modules
sage: x.parent() is DihedralGroup(6).algebra(QQ)  #...
˓→optional - 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) # optional - sage.libs.gap
sage: m = SU(2, 13).algebra(QQ).random_element(1)  # optional - sage.groups sage.modules
sage: m.parent() is SU(2, 13).algebra(QQ)  # optional - sage.groups sage.modules
True
sage: p = CombinatorialFreeModule(ZZ, Partitions(4)).random_element()  # optional - sage.combinat sage.modules
sage: p.parent() is CombinatorialFreeModule(ZZ, Partitions(4))  # optional - sage.combinat sage.modules
True
```

```python
submodule(gens, check=True, already_echelonized=False, unitriangular=False, support_order=None, category=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

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:**
- `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: X = CombinatorialFreeModule(QQ, range(3), prefix="x") # optional - sage.modules
sage: x = X.basis()  # optional - sage.modules
```
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:

```python
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
optional - sage.modules
sage: x = X.basis()
optional - sage.modules
sage: gens = Family({1: x[0] - x[1], 3: x[1] - x[2]}); gens
optional - sage.modules
Finite family {1: x[0] - x[1], 3: x[1] - x[2]}
sage: Y = X.submodule(gens, already_echelonized=True)
optional - sage.modules
sage: Y.print_options(prefix='y'); Y
optional - sage.modules
Free module generated by {1, 3} over Rational Field
sage: y = Y.basis()
optional - sage.modules
sage: y[1]
optional - sage.modules
y[1]
sage: y[1].lift()
optional - sage.modules
x[1] - x[2]
sage: Y.retract(x[0] - x[2])
optional - sage.modules
y[0] + y[1]
sage: Y.retract(x[0])
optional - sage.modules
Traceback (most recent call last):
... ValueError: x[0] is not in the image
```

(continues on next page)
It is not necessary that the generators of the submodule form a basis (an explicit basis will be computed):

\begin{verbatim}
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]; gens
[2*x[1] - 2*x[2], x[0] - x[1], x[0] - x[2]]
sage: Y = X.submodule(gens, already_echelonized=False)
sage: Y.print_options(prefix='y')

Free module generated by {0, 1} over Rational Field
sage: [b.lift() for b in Y.basis()]
[2*x[0] - 2*x[2], x[1] - x[2]]
\end{verbatim}

We now implement by hand the center of the algebra of the symmetric group $S_3$:

\begin{verbatim}
sage: S3 = SymmetricGroup(3)
sage: S3A = S3.algebra(QQ)
sage: basis = S3A.annihilator_basis(S3A.algebra_generators(),
....: S3A.bracket)
\end{verbatim}

Free module generated by {0, 1, 2} over Rational Field
sage: center = S3A.submodule(basis,
....: category=AlgebrasWithBasis(QQ).Subobjects(),
....: already_echelonized=True)
\end{verbatim}

Free module generated by {0, 1, 2} over Rational Field
sage: center in Algebras
True
sage: center.print_options(prefix='c')  # optional - sage.groups sage.modules
sage: c = center.basis()  # optional - sage.groups sage.modules
sage: c[1].lift()  # optional - sage.groups sage.modules
(1,2,3) + (1,3,2)
sage: c[0]^2  # optional - sage.groups sage.modules
c[0]
sage: e = 1/6 * (c[0]+c[1]+c[2])  # optional - sage.groups sage.modules
sage: e.is_idempotent()  # optional - sage.groups sage.modules
True

Of course, this center is best constructed using:

sage: center = S3A.center()  # optional - sage.groups sage.modules

We can also automatically construct a basis such that the lift morphism is (lower) unitriangular:

```python
sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')  # optional - sage.modules
sage: x = C.basis()  # optional - sage.modules
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]  # optional - sage.modules
sage: Y = C.submodule(gens, unitriangular=True)  # optional - sage.modules
sage: Y.lift.matrix()  # optional - sage.modules
[ 1 0]
[ 0 1]
[-1 -1]
```

We now construct a (finite-dimensional) submodule of an infinite dimensional free module:

```python
sage: C = CombinatorialFreeModule(QQ, ZZ, prefix='z')  # optional - sage.modules
sage: z = C.basis()  # optional - sage.modules
sage: gens = [z[0] - z[1], 2*z[1] - 2*z[2], z[0] - z[2]]  # optional - sage.modules
sage: Y = C.submodule(gens)  # optional - sage.modules
sage: Y.lift(b) for b in Y.basis()  # optional - sage.modules
[z[0] - z[2], z[1] - z[2]]
```

```python
sum_of_monomials()
```
Return the sum of the basis elements with indices in indices.

INPUT:
• indices – an list (or iterable) of indices of basis elements

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
    # optional - sage.modules
sage: F.sum_of_monomials(['a', 'b'])
    # optional - sage.modules
B['a'] + B['b']

sage: F.sum_of_monomials(['a', 'b', 'a'])
    # optional - sage.modules
2*B['a'] + B['b']
```

F.sum_of_monomials is in fact (almost) a map:

```
sage: F.sum_of_monomials
    # optional - 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]])
    # optional - sage.modules
sage: J.<a,b,c> = JordanAlgebra(m)
    # optional - sage.combinat sage.modules
sage: J.sum_of_terms(((0, 2), (2, -3)))
    # optional - 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")
    # optional - sage.combinat sage.modules
sage: A.tensor(A, A)
    # optional - sage.combinat sage.modules
A # A # A
sage: A.rename(None)
    # optional - 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:**

coeff * B[index], where B is the basis of *self*.

**EXAMPLES:**

```
sage: m = matrix([[0,1], [1,1]])
optional - sage.modules
sage: J.<a,b,c> = JordanAlgebra(m)
optional - sage.combinat sage.modules
sage: J.term(1, -2)
optional - 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*:

```
sage: A = CombinatorialFreeModule(ZZ, [1,2], prefix="A")
optional - sage.modules
sage: A.rename("A")
optional - sage.modules
sage: B = CombinatorialFreeModule(ZZ, [3,4], prefix="B")
optional - sage.modules
```

(continues on next page)
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: a = A.monomial(1) + 2 * A.monomial(2); a
#optional - sage.modules
sage: b = B.monomial(3) - 2 * B.monomial(4); b
#optional - sage.modules
sage: f(a, b)
#optional - sage.modules
sage: tensor([a, b]).apply_multilinear_morphism(f)
#optional - sage.modules
```

$f$ may be a bilinear morphism to any module over the base ring of $A$ and $B$. Here the codomain is $\mathbb{Z}$:

```python
sage: def f(a, b):
    ....: return sum(a.coefficients(), 0) * sum(b.coefficients(), 0)
sage: f(a, b)
#optional - sage.modules
-3
sage: tensor([a, b]).apply_multilinear_morphism(f)
#optional - sage.modules
-3
```

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())
sage: type(f(A.zero(), B.zero()))
<... 'int'>
sage: tensor([a, b]).apply_multilinear_morphism(f)
#optional - sage.modules
Traceback (most recent call last):
  ...
AttributeError: 'int' object has no attribute 'parent'
```

Which would be wrong and break this method:

```python
sage: tensor([a, b]).apply_multilinear_morphism(f)
#optional - sage.modules
Traceback (most recent call last):
  ...
AttributeError: 'int' object has no attribute 'parent'
```

Here we consider an example where the codomain is a module with basis with a different base ring:
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:

```python
sage: Sym = SymmetricFunctions(QQ)  #
optional - sage.combinat sage.modules
sage: s = Sym.schur()  #
optional - sage.combinat sage.modules
sage: def f(a, b): return a * b.antipode()  #
optional - sage.combinat sage.modules
sage: x = 4 * s.an_element(); x  #
optional - sage.combinat sage.modules
8*s[] + 8*s[1] + 12*s[2]
sage: x.coproduct().apply_multilinear_morphism(f)  #
optional - sage.combinat sage.modules
8*s[]
sage: x.coproduct().apply_multilinear_morphism(f) == x.counit()  #
optional - sage.combinat sage.modules
True
```

We recover the constant term of $x$, as desired.

**Todo:** Extract a method to linearize a multilinear morphism, and delegate the work there.

```python
class ParentMethods
Bases: object

Implements operations on tensor products of modules with basis.

extra_super_categories()

EXAMPLES:

```
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
```

4.120 Monoid algebras

```
sage.categories.monoid_algebras.MonoidAlgebras(base_ring)
The category of monoid algebras over base_ring.
```

EXAMPLES:

```
sage: C = MonoidAlgebras(QQ); C
Category of monoid algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of bialgebra 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:

```
sage: C is Monoids().Algebras(QQ)
True
```

4.121 Monoids

```
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()
Category of monoids
sage: Monoids().super_categories()
[Category of semigroups, Category of unital magmas]
sage: Monoids().all_super_categories()
[Category of monoids, 
 Category of semigroups, 
 Category of unital magmas, Category of magmas, 
 Category of sets, 
 (continues on next page)
Category of sets with partial maps,
Category of objects

```python
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')
```

class **Algebras**(*category*, *args)

Bases: `AlgebrasCategory`

```python
class ElementMethods

Bases: object

```is_central()```

Return whether the element `self` is central.

EXAMPLES:

```python
sage: SG4 = SymmetricGroupAlgebra(ZZ,4)  # optional - sage.groups sage.modules
sage: SG4(1).is_central()                  # optional - sage.groups sage.modules
True
sage: SG4(Permutation([1,3,2,4])).is_central()  # optional - sage.groups sage.modules
False
sage: A = GroupAlgebras(QQ).example(); A       # optional - sage.groups sage.modules
Algebra of Dihedral group of order 8 as a permutation group over Rational Field
sage: sum(A.basis()).is_central()             # optional - sage.groups sage.modules
True
```

class **ParentMethods**

Bases: object

```python
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:**


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() #
...optional - 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.semigroup_generators()
Family (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
sage: Z12.algebra(QQ).algebra_generators() #
...optional - 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])

sage: A10 = AlternatingGroup(10) #
...optional - sage.groups
sage: GroupAlgebras(QQ).example(A10).algebra_generators() #
...optional - sage.groups sage.modules
Family ((8,9,10), (1,2,3,4,5,6,7,8,9))

sage: A = DihedralGroup(3).algebra(QQ); A #
...optional - sage.groups sage.modules
Algebra of Dihedral group of order 6 as a permutation group
over Rational Field
sage: A.algebra_generators() #
...optional - sage.groups sage.modules
Family ((1,2,3), (1,3))

one_basis() 
Return the unit of the monoid, which indexes the unit of this algebra, as per AlgebrasWithBasis.
ParentMethods.one_basis().

EXAMPLES:

sage: A = Monoids().example().algebra(ZZ) #
...optional - sage.modules
sage: A.one_basis() #
...optional - sage.modules
...
sage: A.one() #
...optional - sage.modules
B[]
sage: A(3)                      #optional - sage.modules
3*B['']

extra_super_categories()

The algebra of a monoid is a bialgebra and a monoid.

EXAMPLES:

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]

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:

sage: G1 = SymmetricGroup(3)             #optional - sage.groups sage.modules
sage: G2 = SL(2, 3)                      #optional - sage.groups sage.modules
sage: G = cartesian_product([G1, G2])   #optional - sage.groups sage.modules
sage: G((G1.gen(0), G2.gen(1))).multiplicative_order()  #optional - sage.groups sage.modules
12

class ParentMethods

Bases: object

monoid_generators()

Return the generators of self.

EXAMPLES:

sage: M = Monoids.free([1, 2, 3])        #optional - sage.groups
sage: N = Monoids.free(['a', 'b'])       #optional - sage.groups

(continues on next page)
An example with an infinitely generated group (a better output is needed):

```
sage: N = Monoids().free(ZZ)  #
```

```
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’]))
```

```
Lazy family (gen(i))_{i in The Cartesian product of (...)}
```

```
element multiplication
```

```
extra_super_categories()
```

A Cartesian product of monoids is endowed with a natural group structure.

EXAMPLES:

```
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

Category of commutative (abelian) monoids.

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:**

```
sage: Monoids().Commutative().free(index_set=ZZ)  #
```

```
Free abelian monoid indexed by Integer Ring
```

```
sage: Monoids().Commutative().free(ZZ)  #
```

(continues on next page)
Free abelian monoid indexed by Integer Ring

```python
sage: F.<x,y,z> = Monoids().Commutative().free(); F
```

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.

EXAMPLES:

```python
sage: AA(sqrt(~2)).inverse()
```

1.414213562373095?

```python
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:

```python
sage: A = Matrix([[1, 1], [-1, 0]])
```

Finite alias of `FiniteMonoids`

Inverse alias of `Groups`

```python
class ParentMethods
Bases: object

prod(args)

n-ary product of elements of self.

INPUT:

- args – a list (or iterable) of elements of self

Returns the product of the elements in args, as an element of self.

EXAMPLES:

```
4.121. Monoids
sage: S = Monoids().example()
sage: S.prod([S('a'), S('b')))
'ab'

**semigroup_generators()**
Return the generators of self as a semigroup.

The generators of a monoid $M$ as a semigroup are the generators of $M$ as a monoid and the unit.

**EXAMPLES:**

```python
sage: M = Monoids().free([1,2,3])
# optional - sage.groups
sage: M.semigroup_generators()
# optional - sage.groups
Family (1, F[1], F[2], F[3])
```

**submonoid(generators, category=None)**
Return the multiplicative submonoid generated by generators.

**INPUT:**
- **generators** – a finite family of elements of self, or a list, iterable, … that can be converted into one (see Family).
- **category** – a category

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
# optional - sage.groups
A submonoid of (Ring of integers modulo 15) with 2 generators
sage: M.list()
# optional - sage.groups
[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
# optional - sage.groups
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: S.list()
# optional - sage.groups
[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()); M2
# optional - sage.groups
sage: M2 is M
# optional - sage.groups
True
```

**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:

```
sage: S = Monoids().Subquotients().example() # todo: not implemented
sage: 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  # todo: not implemented
  optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field
sage: A.one()  # todo: not implemented
  optional - sage.combinat sage.modules
'sage.categories.monoids'
sage: A.one()  # todo: not implemented
  optional - sage.combinat 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:

```
sage: Monoids().free(index_set=ZZ)          # todo:
  optional - sage.groups
Free monoid indexed by Integer Ring
sage: Monoids().free(ZZ)                   # todo:
  optional - sage.groups
Free monoid indexed by Integer Ring
```

(continues on next page)
4.122 Number fields

class sage.categories.number_fields.NumberFields(s=None)
    Bases: Category_singleton
    The category of number fields.

    EXAMPLES:
    We create the category of number fields:

    sage: C = NumberFields()
    sage: C
    Category of number fields

    By definition, it is infinite:

    sage: NumberFields().Infinite() is NumberFields()
    True

    Notice that the rational numbers $\mathbb{Q}$ are considered as an object in this category:

    sage: RationalField() in C
    True

    However, we can define a degree 1 extension of $\mathbb{Q}$, which is of course also in this category:

    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:

    sage: K = NumberField(x^2 + 1, 'a')
    sage: K in C
    True

    class ElementMethods
        Bases: object

    class ParentMethods
        Bases: object
zeta_function(prec=53, max_imaginary_part=0, max_asympt_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_asympt_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)  # optional - sage.rings.number_field
sage: Z = K.zeta_function(); Z  # optional - sage.rings.number_field sage.symbolic
PARI zeta function associated to Number Field in a with defining polynomial x^2 + x - 1
sage: Z(-1)  # optional - 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])  # optional - sage.rings.number_field
sage: Z = L.zeta_function()  # optional - sage.rings.number_field sage.symbolic
sage: Z(5)  # optional - 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)  # optional - sage.rings.number_field
sage: Z = K.zeta_function(algorithm="pari")  # optional - sage.rings.number_field sage.symbolic
sage: Z(-1)  # optional - 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])  # optional - sage.rings.number_field
sage: Z = L.zeta_function(algorithm="pari")  # optional - sage.rings.number_field sage.symbolic
sage: Z(5)  # optional - sage.rings.number_field sage.symbolic
1.00199015670185
```

(continues on next page)
optional - sage.rings.number_field sage.symbolic
1.00199015670185

super_categories()

EXAMPLES:

```
sage: NumberFields().super_categories()
[Category of infinite fields]
```

4.123 Objects

class sage.categories.objects.Objects(s=None)

Bases: Category_singleton

The category of all objects the basic category

EXAMPLES:

```
sage: Objects()
Category of objects
sage: 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
```

See also:

• Homsets()

Homsets()

Return the category of homsets between objects of this category.

EXAMPLES:

```
sage: Sets().Homsets()
Category of homsets of sets
```
sage: 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:**

```sage
sage: Objects().additional_structure()
```

**super_categories()**

**EXAMPLES:**

```sage
sage: Objects().super_categories()
[]
```
4.124 Partially ordered monoids

class sage.categories.partially_ordered_monoids.PartiallyOrderedMonoids(s=None)
    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 \( x * z \leq y * z \) and \( z * x \leq z * y \).

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(s=None)
    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]
```

The category of permutation groups defines additional structure that should be preserved by morphisms, namely the distinguished action:
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]
```

### 4.126 Pointed sets

**class** `sage.categories.pointed_sets.PointedSets(s=None)`

**Bases:** `Category_singleton`

The category of pointed sets.

**EXAMPLES:**
```
sage: PointedSets()
Category of pointed sets
```

**super_categories()**

**EXAMPLES:**
```
sage: 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 *Q*:
```
sage: 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]
```
4.128 Posets

class sage.categories.posets.Posets(s=None)
    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

    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 `Sets.Facade`), one can compare directly its elements using the usual Python operators:

    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 github issue #10130 will be resolved, this will be automatically provided by this category:
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:

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]

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:

sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.ge(6, 3)
True
sage: D.ge(3, 3)

(continues on next page)
gt(x, y)

Return whether $x > y$ in the poset self.

INPUT:

• $x, y$ – elements of self.

This default implementation delegates the work to $\lt()$.

EXAMPLES:

```
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: 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
```

(continues on next page)
is_chain_of_poset(o, ordered=False)

Return whether an iterable \( o \) is a chain of \( \textbf{self} \), including a check for \( o \) being ordered from smallest to largest element if the keyword \texttt{ordered} is set to \texttt{True}.

INPUT:

- \( o \) – an iterable (e. g., list, set, or tuple) containing some elements of \( \textbf{self} \)
- \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 \( \textbf{self} \) formed by the elements of \( o \) is a chain in \( \textbf{self} \).

If \texttt{ordered} is set to \texttt{True}, the truth value of the following assertion is returned: Every element of the list \( o \) is (strictly!) smaller than its successor in \( \textbf{self} \). (This makes no sense if \texttt{ordered} is a set.)

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_chain_of_poset([1, 3])
True
sage: P.is_chain_of_poset([3, 1])
True
sage: P.is_chain_of_poset([1, 3], ordered=True)
True
sage: P.is_chain_of_poset([3, 1], ordered=True)
False
sage: P.is_chain_of_poset([])
True
sage: P.is_chain_of_poset([], ordered=True)
True
sage: P.is_chain_of_poset((2, 12, 6))
True
sage: P.is_chain_of_poset((2, 6, 12), ordered=True)
True
sage: P.is_chain_of_poset((2, 12, 6), ordered=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=True)
False
sage: Q.is_chain_of_poset([1, 2])
True
sage: Q.is_chain_of_poset([2, 1], ordered=True)
True
sage: Q.is_chain_of_poset([2, 1, 1], ordered=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=True)
False
sage: Q.is_chain_of_poset([2, 3, 4], ordered=True)
True

Examples with infinite posets:

sage: from sage.categories.examples.posets import FiniteSetsOrderedByInclusion
sage: R = FiniteSetsOrderedByInclusion()
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

sage: from sage.categories.examples.posets import PositiveIntegersOrderedByDivisibilityFacade
sage: T = PositiveIntegersOrderedByDivisibilityFacade()
sage: T.is_chain_of_poset((T(3), T(4), T(7)))
False
sage: T.is_chain_of_poset((T(3), T(6), T(3)))
True
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

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({})
True
sage: P.is_order_filter({3, 4, 12})
False
sage: P.is_order_filter({3, 6, 12})
True

is_order_ideal(o)

4.128. Posets 717
Return whether \( o \) is an order ideal of \( \text{self} \), assuming \( \text{self} \) has no infinite descending path.

**INPUT:**
- \( o \) – a list (or set, or tuple) containing some elements of \( \text{self} \)

**EXAMPLES:**

```python
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
```

\( \text{le}(x, y) \)

Return whether \( x \leq y \) in the poset \( \text{self} \).

**INPUT:**
- \( x, y \) – elements of \( \text{self} \).

**EXAMPLES:**

```python
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
```

\( \text{lower_covers}(x) \)

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]
```

\( \text{lt}(x, y) \)

Return whether \( x < y \) in the poset \( \text{self} \).

**INPUT:**
- \( x, y \) – elements of \( \text{self} \).

This default implementation delegates the work to \( \text{le}(\cdot) \).

**EXAMPLES:**

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lt( 3, 6 )
(continues on next page)```
order_filter\(\text{(elements)}\)

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 \(y \geq x\), 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\(\text{(elements)}\)

Return the order ideal in \(self\) generated by the elements of an iterable \(\text{elements}\).

A subset \(I\) of a poset is said to be an order ideal if, for any \(x\) in \(I\) and \(y\) such that \(y \leq x\), 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 \(I \cup \{v\}\), if \(v \notin I\) but every element of \(P\) smaller than \(v\) is in \(I\);
- the set \(I \setminus \{v\}\), if \(v \in I\) 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()
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)
```
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_order_filter(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:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.upper_covers(3)
[6, 15]
```

**example**(choice=None)
Return examples of objects of Posets(), as per Category.example().

EXAMPLES:

```
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:

```
sage: Posets().super_categories()
[Category of sets]
```

### 4.129 Principal ideal domains

**class** sage.categories.principal_ideal_domains.PrincipalIdealDomains(s=None)**
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:
Category Framework, Release 10.1

sage: PrincipalIdealDomains()
Category of principal ideal domains
sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]

See also Wikipedia article Principal_ideal_domain

class ElementMethods
    Bases: object

class ParentMethods
    Bases: object

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()

super_categories()
    EXAMPLES:

sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]

4.130 Quotient fields

class sage.categories.quotient_fields.QuotientFields(s=None)
    Bases: Category_singleton

The category of quotient fields over an integral domain

EXAMPLES:

sage: QuotientFields()
Category of quotient fields
sage: QuotientFields().super_categories()
[Category of fields]

class ElementMethods
    Bases: object

denominator()
    Constructor for abstract methods

    EXAMPLES:

sage: def f(x):
    ....:     "doc of f"
    ....:     return 1

(continues on next page)
sage: x = abstract_method(f); x
<abstract method f at ...>
sage: x.__doc__
'doc of f'
sage: x.__name__
'f'
sage: x.__module__
'__main__'

**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:**

```
sage: F.<x> = Frac(QQ['x'])
sage: (1/x).derivative()
-1/x^2
```

```
sage: (x+1/x).derivative(x, 2)
2/x^3
```

```
sage: F.<x,y> = Frac(QQ['x,y'])
sage: (1/(x+y)).derivative(x,y)
2/(x^3 + 3*x^2*y + 3*x*y^2 + y^3)
```

**factor(***args, **kwds*)**

Return the factorization of `self` over the base ring.

**INPUT:**

- *args - Arbitrary arguments suitable over the base ring
- **kwds - Arbitrary keyword arguments suitable over the base ring

**OUTPUT:**

- Factorization of `self` over the base ring

**EXAMPLES:**

```
sage: K.<x> = QQ[]
sage: f = (x^3+x)/(x-3)
sage: f.factor()  # optional - sage.libs.pari
(x - 3)^-1 * x * (x^2 + 1)
```

Here is an example to show that github issue #7868 has been resolved:

```
sage: R.<x,y> = GF(2)[]  # optional - sage.rings.finite_rings
sage: f = x*y/(x+y)  # optional - sage.rings.finite_rings
```

(continues on next page)
\texttt{sage: f.factor()} 
\texttt{#optional - sage.rings.finite_rings} 
\((x + y)^{-1} * y * x\)

\textbf{gcd}(\texttt{other})

Greatest common divisor

\textbf{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).

\textbf{AUTHOR:}

• Simon King (2011-02): See github issue #10771

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: R.<x> = QQ['x']}
\texttt{sage: p = (1+x)^3*(1+2*x^2)/(1-x^5)}
\texttt{sage: q = (1+x)^2*(1+3*x^2)/(1-x^4)}
\texttt{sage: factor(p)}
\texttt{#optional - sage.libs.pari}
\((-2) * (x - 1)^{-1} * (x + 1)^3 * (x^2 + 1/2) * (x^4 + x^3 + x^2 + x + 1)^{-1}
\texttt{sage: factor(q)}
\texttt{#optional - sage.libs.pari}
\((-3) * (x - 1)^{-1} * (x + 1) * (x^2 + 1)^{-1} * (x^2 + 1/3)
\texttt{sage: gcd(p, q)}
(x + 1)/(x^5 + x^4 - x^2 - 1)
\texttt{sage: factor(gcd(p, q))}
\texttt{#optional - sage.libs.pari}
(x - 1)^{-1} * (x + 1) * (x^4 + x^3 + x^2 + x + 1)^{-1}
\texttt{sage: factor(gcd(p, 1 + x))}
\texttt{#optional - sage.libs.pari}
(x - 1)^{-1} * (x + 1)^{-1} * (x^4 + x^3 + x^2 + x + 1)^{-1}
\texttt{sage: factor(gcd(1 + x, q))}
\texttt{#optional - sage.libs.pari}
(x - 1)^{-1} * (x + 1) * (x^2 + 1)^{-1}
\end{verbatim}

\textbf{lcm}(\texttt{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 does not appear in the decomposition of \(x\) or \(y\) are considered to have exponent zero).
AUTHOR:
  • Simon King (2011-02): See github issue #10771

EXAMPLES:

```python
sage: lcm(2/3, 1/5)
2
```
Indeed $2/3 = 2^13^{-1}5^0$ and $1/5 = 2^03^05^{-1}$, so $lcm(2/3, 1/5) = 2^13^05^0 = 2$.

```python
sage: lcm(1/3, 1/5) 1 sage: lcm(1/3, 1/6) 1/3
```

Some more involved examples:

```python
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) # optional - sage.libs.pari
(-2) * (x - 1)^-1 * (x + 1)^3 * (x^2 + 1/2) * (x^4 + x^3 + x^2 + x + 1)^-1
sage: factor(q) #-optional - sage.libs.pari
(-3) * (x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^2 + 1/3)
sage: factor(lcm(p, q)) #-optional - sage.libs.pari
(x - 1)^-1 * (x + 1)^3 * (x^2 + 1/3) * (x^2 + 1/2)
sage: factor(lcm(p, 1 + x)) #-optional - sage.libs.pari
(x + 1)^3 * (x^2 + 1/2)
sage: factor(lcm(1 + x, q)) #-optional - sage.libs.pari
(x + 1) * (x^2 + 1/3)
```

`numerat()`
Constructor for abstract methods

```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__'
```

`partial_fraction_decomposition(decompose_powers=True)`
Decomposes 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` – 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:
  • Robert Bradshaw (2007-05-31)
EXAMPLES:

```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
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
sage: q = 2*t / (t + 3)^2
sage: q.partial_fraction_decomposition()
(0, [2*t/(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)])
```

We can decompose over a given algebraic extension:

```sage
R.<x> = QQ[sqrt(2)]
sage: r = 1/(x^4+1)
sage: r.partial_fraction_decomposition()
(0, [(1/4*sqrt2*x + 1/2)/(x^2 - sqrt2*x + 1),
     (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)
```

(continues on next page)
We can also ask Sage to find the least extension where the denominator factors in linear terms:

```python
sage: R.<x> = QQ[]
sage: r = 1/(x^4+2)
sage: N = r.denominator().splitting_field('a'); N
Number Field in a with defining polynomial x^8 - 8*x^6 + 28*x^4 + 16*x^2 + 36
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:

```python
sage: R.<x> = QQbar[]
sage: r = 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)
```

We do the best we can over inexact fields:

```python
sage: R.<x> = RealField(20)[]
sage: q = 1/(x^2 + x + 2)^2 + 1/(x-1); q
```

(continues on next page)
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))

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 \text{xgcd} and \text{lcm}, it is possible to be a bit more specific and define the \text{gcd} uniquely up to a unit of the base ring (rather than in the fraction field).

EXAMPLES:

sage: QQ(3).xgcd(QQ(2))
(1, 1, -1)
sage: QQ(3).xgcd(QQ(1/2))
(1/2, 0, 1)
sage: QQ(1/3).xgcd(QQ(2))
(1/3, 1, 0)
sage: QQ(3/2).xgcd(QQ(5/2))
(1/2, 2, -1)

sage: R.<x> = QQ['x']
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)  
(-3) * (x - 1)^-1 * (x + 1) * (x^2 + 1)^-1

sage: g, s, t = xgcd(p, q)
sage: g
(x + 1)/(x^7 + x^5 - x^2 - 1)
sage: g == s*p + t*q
True

An example without a well defined gcd or xgcd on its base ring:

sage: K = QuadraticField(5)  
(continues on next page)
class ParentMethods
    Bases: object

super_categories()

EXAMPLES:

    sage: QuotientFields().super_categories()
    [Category of fields]

4.131 Quantum Group Representations

AUTHORS:

- Travis Scrimshaw (2018): initial version

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
˓→ # optional - sage.combinat sage.modules
sage: C = crystals.Tableaux(['C',4], shape=[1]) #optional - sage.combinat sage.modules
sage: R = ZZ['q'].fraction_field() #optional - sage.combinat sage.modules
sage: V = MinusculeRepresentation(R, C) #optional - sage.combinat sage.modules
sage: V.cartan_type() #optional - sage.combinat sage.modules

index_set()

Return the index set of self.

EXAMPLES:
```
sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
˓→ # optional - sage.combinat sage.modules
sage: C = crystals.Tableaux(['C',4], shape=[1]) #optional - sage.combinat sage.modules
sage: R = ZZ['q'].fraction_field() #optional - sage.combinat sage.modules
sage: V = MinusculeRepresentation(R, C) #optional - sage.combinat sage.modules
sage: V.index_set() #optional - sage.combinat sage.modules
(1, 2, 3, 4)
```
```
q()

Return the quantum parameter $q$ of self.

EXAMPLES:
```
sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
˓→ # optional - sage.combinat sage.modules
sage: C = crystals.Tableaux(['C',4], shape=[1]) #optional - sage.combinat sage.modules
sage: R = ZZ['q'].fraction_field() #optional - sage.combinat sage.modules
sage: V = MinusculeRepresentation(R, C) #optional - sage.combinat sage.modules
sage: V.q() #optional - sage.combinat sage.modules
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:

```sage
from sage.algebras.quantum_groups.representations import MinusculeRepresentation # optional - sage.combinat sage.modules
c = crystals.Tableaux(['C', 2], shape=[1]) # optional - sage.combinat sage.modules
R = ZZ['q'].fraction_field() # optional - sage.combinat sage.modules
V = MinusculeRepresentation(R, c) # optional - sage.combinat sage.modules
T = tensor([V, V]) # optional - sage.combinat sage.modules
T.cartan_type() # optional - sage.combinat sage.modules
['C', 2]
```

extra_super_categories()

EXAMPLES:

```sage
from sage.categories.quantum_group_representations import QuantumGroupRepresentations
cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
cat.TensorProducts().extra_super_categories()
[Category of quantum group representations over Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

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, power=1)

Return the action of $K_i$ on self to the power power.

INPUT:
• i – an element of the index set
• power – (default: 1) the power of $K_i$

EXAMPLES:

```sage
from sage.algebras.quantum_groups.representations import AdjointRepresentation # optional - sage.combinat sage.modules
K = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1) # optional - sage.combinat sage.modules
R = ZZ['q'].fraction_field() # optional - sage.combinat sage.modules
V = AdjointRepresentation(R, K) # optional - sage.combinat sage.modules
# (continues on next page)
Return the action of $e_i$ on self.

INPUT:

• i – an element of the index set

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{from sage.algebras.quantum_groups.representations import} \\
& \text{AdjointRepresentation} \quad \# \text{optional - sage.combinat sage.modules} \\
\text{sage: } & C = \text{crystals.Tableaux(['G',2], shape=[1,1])} \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
\text{sage: } & R = \text{ZZ['q'].fraction_field()} \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
\text{sage: } & V = \text{AdjointRepresentation}(R, C) \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
\text{sage: } & v = V.\text{an_element(); v} \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
2*B[[1], [2]] + 2*B[[1], [3]] + 3*B[[2], [3]] \\
\text{sage: } & v.\text{e}(1) \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
(3*q^4+3*q^2+3)/q^2)*B[[1], [3]] \\
\text{sage: } & v.\text{e}(2) \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
2*B[[1], [2]] \\
\end{align*}
\]

Return the action of $f_i$ on self.

INPUT:

• i – an element of the index set

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{from sage.algebras.quantum_groups.representations import} \\
& \text{AdjointRepresentation} \quad \# \text{optional - sage.combinat sage.modules} \\
\text{sage: } & K = \text{crystals.KirillovReshetikhin(['D',4,1], 2,1)} \\
& \quad \# \text{optional - sage.combinat sage.modules} \\
\end{align*}
\] (continues on next page)
sage: R = ZZ['q'].fraction_field()  #
...: optional - sage.combinat sage.modules
sage: V = AdjointRepresentation(R, K)  #
...: optional - sage.combinat sage.modules
sage: v = V.an_element(); v  #
...: optional - sage.combinat sage.modules
2*B[[[1], [2]]] + 2*B[[[1], [3]]] + 3*B[[[1], [3]]]
sage: v.f(0)  #
...: optional - sage.combinat sage.modules
((2*q^2+2)/q)*B[[[1], [2]]]
sage: v.f(1)  #
...: optional - sage.combinat sage.modules
3*B[[[2], [3]]]
sage: v.f(2)  #
...: optional - sage.combinat sage.modules
2*B[[[1], [3]]]
sage: v.f(3)  #
...: optional - sage.combinat sage.modules
3*B[[[1], [4]]]
sage: v.f(4)  #
...: optional - sage.combinat sage.modules
3*B[[[1], [-4]]]

class ParentMethods
Bases: object
tensor("factors")

Return the tensor product of self with the representations factors.

EXAMPLES:

sage: from sage.algebras.quantum_groups.representations import (  #
...: optional - sage.combinat sage.modules
....: MinusculeRepresentation, AdjointRepresentation)
sage: R = ZZ['q'].fraction_field()
sage: CM = crystals.Tableaux(['D',4], shape=[1])  #
...: optional - sage.combinat sage.modules
sage: CA = crystals.Tableaux(['D',4], shape=[1,1])  #
...: optional - sage.combinat sage.modules
sage: V = MinusculeRepresentation(R, CM)  #
...: optional - sage.combinat sage.modules
sage: V.tensor(V, V)  #
...: optional - sage.combinat sage.modules
V((1, 0, 0, 0)) # V((1, 0, 0, 0)) # V((1, 0, 0, 0))
sage: A = MinusculeRepresentation(R, CA)  #
...: optional - sage.combinat sage.modules
sage: V.tensor(A)  #
...: optional - sage.combinat sage.modules
V((1, 0, 0, 0)) # V((1, 1, 0, 0))
sage: B = crystals.Tableaux(['A',2], shape=[1])  #
...: optional - sage.combinat sage.modules
sage: W = MinusculeRepresentation(R, B)  #
...: optional - sage.combinat sage.modules

(continues on next page)
sage: tensor([W,V])
#optional - sage.combinat sage.modules
Traceback (most recent call last):
... ValueError: all factors must be of the same Cartan type
sage: tensor([V,A,W])
#optional - sage.combinat sage.modules
Traceback (most recent call last):
... ValueError: all factors must be of the same Cartan type

class TensorProducts(category, *args)

Bases: TensorProductsCategory

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 $power$.

INPUT:
• $i$ – an element of the index set
• $b$ – an element of basis keys
• $power$ – (default: 1) the power of $K_i$

EXAMPLES:

```python
sage: from sage.algebras.quantum_groups.representations import (  #
    optional - sage.combinat sage.modules
    MinusculeRepresentation, AdjointRepresentation)
...:

sage: R = ZZ['q'].fraction_field()
sage: CM = crystals.Tableaux(['A',2], shape=[1])
#optional - sage.combinat sage.modules
sage: VM = MinusculeRepresentation(R, CM)
#optional - sage.combinat sage.modules
sage: CA = crystals.Tableaux(['A',2], shape=[2,1])
#optional - sage.combinat sage.modules
sage: VA = AdjointRepresentation(R, CA)
#optional - sage.combinat sage.modules
sage: v = tensor([sum(VM.basis()), VA.module_generator()]); v
#optional - sage.combinat sage.modules
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
#optional - sage.combinat sage.modules
```
\begin{align*}
1/q^2 B[[[1]]] & \# B[[[1], [2]]] \\
+ 1/q^2 B[[[2]]] & \# B[[[1], [2]]] \\
+ B[[[3]]] & \# B[[[1], [2]]]
\end{align*}

\textbf{\texttt{e_on_basis}(i, b)}

Return the action of $e_i$ on the basis element indexed by $b$.

\textbf{INPUT:}

- $i$ – an element of the index set
- $b$ – an element of basis keys

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.algebras.quantum_groups.representations import (  
    Optional - sage.combinat sage.modules  
    MinusculeRepresentation, AdjointRepresentation)
sage: R = ZZ['q'].fraction_field()  
{sage: CM = crystals.Tableaux(['D',4], shape=[1])  
    Optional - sage.combinat sage.modules
{sage: VM = MinusculeRepresentation(R, CM)  
    Optional - sage.combinat sage.modules
{sage: CA = crystals.Tableaux(['D',4], shape=[1,1])  
    Optional - sage.combinat sage.modules
{sage: VA = AdjointRepresentation(R, CA)  
    Optional - sage.combinat sage.modules
{sage: v = tensor([VM.an_element(), VA.an_element()]); v  
    Optional - sage.combinat sage.modules
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[[[3]]] # B[[[1], [2]]] 
+ 6*B[[[3]]] # B[[[1], [3]]] + 9*B[[[3]]] # B[[[1], [3]]]
\end{verbatim}

\begin{verbatim}
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[[[1], [3]]]
\end{verbatim}

\begin{verbatim}
sage: v.e(2) # 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[[[1], [3]]]
\end{verbatim}

\begin{verbatim}
sage: v.e(3) # indirect doctest
0
\end{verbatim}

\begin{verbatim}
sage: v.e(4) # indirect doctest
0
\end{verbatim}

\textbf{\texttt{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 (  #
  #optional - sage.combinat sage.modules
  MinusculeRepresentation, AdjointRepresentation)

sage: R = ZZ['q'].fraction_field()
sage: KM = crystals.KirillovReshetikhin(['B',3,1], 3,1)  #
  #optional - sage.combinat sage.modules
sage: VM = MinusculeRepresentation(R, KM)  #
  #optional - sage.combinat sage.modules
sage: KA = crystals.KirillovReshetikhin(['B',3,1], 2,1)  #
  #optional - sage.combinat sage.modules
sage: VA = AdjointRepresentation(R, KA)  #
  #optional - sage.combinat sage.modules

sage: v = tensor([VM.an_element(), VA.an_element()]); v  #
  #optional - sage.combinat sage.modules
4*B[[++, [], ]] # B[[[], []]] + 4*B[[++, [], ]] # B[[[1], [2]]]
+ 6*B[[++, [], ]] # B[[[1], [3]]] + 4*B[[++, [], ]] # B[[1]]
+ 4*B[[++, [], ]] # B[[[1], [2]]]
+ 6*B[[++, [], ]] # B[[[1], [3]]] + 6*B[[++, [], ]] # B[[1]]
+ 6*B[[++, [], ]] # B[[[1], [2]]]
+ 9*B[[++, [], ]] # B[[[1], [3]]]
sage: v.f(0)  # indirect doctest  #
  #optional - sage.combinat sage.modules
((4*q^4+4)/q^2)*B[[++, [], ]] # B[[[1], [2]]]
+ ((4*q^4+4)/q^2)*B[[++, [], ]] # B[[[1], [2]]]
+ ((6*q^4+6)/q^2)*B[[++, [], ]] # B[[[1], [2]]]

sage: v.f(1)  # indirect doctest  #
  #optional - sage.combinat sage.modules
6*B[[++, [], ]] # B[[[2], [3]]]
+ 6*B[[++, [], ]] # B[[[2], [3]]]
+ 9*B[[++, [], ]] # B[[[2], [3]]]
+ 6*B[[++, [], ]] # B[[1]]
+ 6*B[[++, [], ]] # B[[[1], [2]]]
+ 9*q^2*B[[++, [], ]] # B[[[1], [3]]]

sage: v.f(2)  # indirect doctest  #
  #optional - sage.combinat sage.modules
4*B[[++, [], ]] # B[[[1], [3]]]
+ 4*B[[++, [], ]] # B[[[1], [3]]]
+ 4*B[[++, [], ]] # B[[1]]
+ 4*q^2*B[[++, [], ]] # B[[[1], [2]]]
+ ((6*q^2+6)/q^2)*B[[++, [], ]] # B[[[1], [3]]]

sage: v.f(3)  # indirect doctest  #
  #optional - sage.combinat sage.modules
6*B[[++, [], ]] # B[[[1], [0]]]
+ 4*B[[++, [], ]] # B[[1]]
+ 4*B[[++, [], ]] # B[[[1], [2]]]
+ 6*q^2*B[[++, [], ]] # B[[[1], [3]]]
+ 6*B[[++, [], ]] # B[[[1], [0]]]
```

(continues on next page)
extra_super_categories()

EXAMPLES:

```python
sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.WithBasis().TensorProducts().extra_super_categories()
[Category of quantum group representations with basis over Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

element()

Return an example of a quantum group representation as per `Category.example`.

EXAMPLES:

```python
sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.example()
# optional - sage.combinat sage.modules
V((2, 1, 0))
```

super_categories()

Return the super categories of `self`.

EXAMPLES:

```python
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(s=None)

Bases: `Category_singleton`

The category of regular crystals.

A crystal is called regular if every vertex $b$ satisfies

$$
\varepsilon_i(b) = \max\{k \mid e_k(b) \neq 0\} \quad \text{and} \quad \varphi_i(b) = \max\{k \mid f_k(b) \neq 0\}.
$$

**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:

```python
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
```

class ElementMethods

    Bases: object

    demazure_operator_simple(i, ring=None)

        Return the Demazure operator $D_i$ applied to self.

        INPUT:
        • i – an element of the index set of the underlying crystal
        • ring – (default: QQ) a ring

        OUTPUT:

        An element of the ring-free module indexed by the underlying crystal.

        Let $r = \langle \mathrm{wt}(b), \alpha_i^\vee \rangle$, then $D_i(b)$ is defined as follows:
        • If $r \geq 0$, this returns the sum of the elements obtained from self by application of $f_k^i$ for $0 \leq k \leq r$.
        • If $r < 0$, this returns the opposite of the sum of the elements obtained by application of $e_k^i$ for $0 < k < -r$.

    REFERENCES:
    • [Li1995]
    • [Ka1993]

    EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
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 $B$. 

Chapter 4. Individual Categories
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()`

**EXAMPLES:**

```python
sage: T = crystals.Tableaux(['A',3], shape=[2,2])
sage: G = T(2,1,4,3).dual_equivalence_class()
sage: G.edges(sort=True)
[([[1, 3], [2, 4]], [[1, 2], [3, 4]], 2),
 ([[1, 3], [2, 4]], [[1, 2], [3, 4]], 3)]
sage: T = crystals.Tableaux(['A',4], shape=[3,2])
sage: G = T(2,1,4,3,5).dual_equivalence_class()
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, 5], [2, 4]], [[1, 2, 5], [3, 4]], 3),
 ([[1, 3, 4], [2, 5]], [[1, 2, 4], [3, 5]], 2),
 ([[1, 2, 4], [3, 5]], [[1, 2, 3], [4, 5]], 3),
 ([[1, 2, 4], [3, 5]], [[1, 2, 3], [4, 5]], 4)]
```

**epsilon**($i$)

Return $\varepsilon_i$ of `self`.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```

**phi**($i$)

Return $\varphi_i$ of `self`.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).phi(1)
1
sage: C(2).phi(1)
0
```
**stembridgeDel_depth**(i, j)

Return the difference in the *j*-depth of self and *f*_\_\_i of 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=[[1,1],[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 self and *f*_\_\_i of 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
```

**stembridgeDelta_depth**(i, j)

Return the difference in the *j*-depth of self and *e*_\_\_i of 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=[[1,1],[2]])
sage: t.stembridgeDelta_depth(1,2)
0
sage: s = T(rows=[[1,3],[3]])
sage: s.stembridgeDelta_depth(1,2)
-1
```

**stembridgeDelta_rise**(i, j)

Return the difference in the *j*-rise of self and *e*_\_\_i of 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.stembridgeDelta_rise(1,2)
0
sage: s = T(rows=[[2,3],[3]])
sage: s.stembridgeDelta_rise(1,2)
-1
```
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 $b=$stembridgeDelta_depth$(x,i,j)$, and $c=$stembridgeDelta_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=[[1,2],[2]])
sage: t.stembridgeDelta_rise(1,2)
-1
sage: s = T(rows=[[2,3],[3]])
sage: s.stembridgeDelta_rise(1,2)
0
```

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, 0)
```

```
```
```
sage: B = crystals.LSPaths(La[0])
sage: La = A21.weight_lattice(extended=True).fundamental_weights()
sage: C = crystals.GeneralizedYoungWalls(2, La[0])
sage: H = Hom(B, C)

```python
from sage.categories.highest_weight_crystals import HighestWeightCrystalMorphism
class Psi(HighestWeightCrystalMorphism):
    ....: def is_strict(self):
    ....:     return True

class Psi(HighestWeightCrystalMorphism):
    ....: def is_strict(self):
    ....:     return True
```

```python
sage: psi = Psi(H, C.module_generators); psi
sage: psi.is_isomorphism()
```

```python
class ParentMethods
    Bases: object
demazure_operator(element, reduced_word)

    Returns 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

    OUTPUT:
    * an element of the free module indexed by the underlying crystal

    EXAMPLES:

```
```

```python
sage: T = crystals.Tableaux(["A",2], shape=[2,1])
sage: C = CombinatorialFreeModule(QQ, T)
sage: t = T.highest_weight_vector()
sage: b = 2*C(t)
sage: T.demazure_operator(b,[1,2,1])
```

```python
2*B[[[1, 1, 1, 1]]] + 2*B[[[1, 1, 1, 2]]] + 2*B[[[1, 1, 2, 2]]] + 2*B[[[1, 2, 2, 2]]] + 2*B[[[2, 2, 2, 2]]] + 2*B[[[1, 1, 1, 1]]] + 2*B[[[1, 1, 1, 2]]] + 2*B[[[1, 1, 2, 2]]] + 2*B[[[1, 2, 2, 2]]] + 2*B[[[2, 2, 2, 2]]]
```

The Demazure operator is idempotent:

```python
sage: T = crystals.Tableaux("A1", shape=[4])
sage: C = CombinatorialFreeModule(QQ, T)
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, 1, 1, 2]]] + B[[[1, 1, 2, 2]]] + B[[[1, 2, 2, 2]]] + B[[[2, 2, 2, 2]]]
sage: e == T.demazure_operator(e,[1])
```

```python
True
```

```python
sage: all(T.demazure_operator(T.demazure_operator(C(t),[1]),[1])
```

(continues on next page)
demazure_subcrystal\((element, \text{reduced\_word}, \text{only\_support}=\text{True})\)

Return the subcrystal corresponding to 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
- \text{only\_support} – (default: \text{True}) only include arrows corresponding to the support of \text{reduced\_word}

**OUTPUT:**
- the Demazure subcrystal

**EXAMPLES:**

```sage
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]]]
```

We construct an example where we don’t only want the arrows indicated by the support of the reduced word:

```sage
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]], [[2, 2]], 1]]
```

**dual_equivalence_graph\((X=None, \text{index\_set}=\text{None}, \text{directed}=\text{True})\)**

Return the dual equivalence graph indexed by \text{index\_set} on the subset \(X\) of \text{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
- \(b' = f_{i-1}f_if_{i-1}b\).

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 \( \epsilon(b, j) \) 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 \( G(\mathcal{X}) \) 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 \( \text{self} \) of weight 0)
- \( \text{index\_set} \) – (optional) the index set \( I \) (default: the whole index set of \( \text{self} \)); this has to be a subset of the index set of \( \text{self} \) (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 \( b' = f_{i-1}f_ie_i e_i^{-1}e_i b \)

**See also:**
- \text{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)]
sage: T = crystals.Tableaux(['A',4], shape=[3,2])
sage: G = T.dual_equivalence_graph()
sage: G.edges(sort=True)
[(\[1, 3, 5\], \[2, 4\]), (\[1, 2, 4\], \[3, 5\]), (\[1, 2, 3\], \[4, 5\]), (\[1, 2, 3\], \[4, 5\]), (\[1, 2, 4\], (\[1, 3, 5\], \[2, 4\]), 3),
 (\[1, 2, 4\], \[3, 5\], (\[1, 2, 3\], (\[1, 2, 4\], \[3, 5\]), 3),
 (\[1, 2, 3\], \[4, 5\]), (\[1, 2, 3\], \[4, 5\]), (\[1, 2, 4\], \[3, 5\]), 4)]
```

**class TensorProducts**(category, *args)

**Bases:** TensorProductsCategory

The category of regular crystals constructed by tensor product of regular crystals.

**extra_super_categories()**

**EXAMPLES:**
additional_structure()

Return None.

Indeed, the category of regular crystals defines no new structure: it only relates $\varepsilon_\alpha$ and $\varphi_\alpha$ to $e_\alpha$ and $f_\alpha$ respectively.

See also:

`Category.additional_structure()`

Todo: Should this category be a `CategoryWithAxiom`?

EXAMPLES:

```sage
sage: RegularCrystals().additional_structure()
```

example($n=3$)

Returns an example of highest weight crystals, as per `Category.example()`.

EXAMPLES:

```sage
sage: B = RegularCrystals().example(); B
Highest weight crystal of type $A_3$ of highest weight $\omega_1$
```

super_categories()

EXAMPLES:

```sage
sage: RegularCrystals().super_categories()
[Category of crystals]
```

### 4.133 Regular Supercrystals

**class** `sage.categories.regular_supercrystals.RegularSuperCrystals($s=None$)`

Bases: `Category_singleton`

The category of crystals for super Lie algebras.

EXAMPLES:

```sage
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 $\text{wt}(x)$)

**EXAMPLES:**

```python
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
dsage: abstract_methods_of_class(RegularSuperCrystals().element_class)
{'optional': [], 'required': ['e', 'f', 'weight']}
```

**class ElementMethods**

**Bases:** object

- **epsilon($i$)**
  
  Return $\varepsilon_i$ of self.

  **EXAMPLES:**

  ```python
  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
  ```

- **phi($i$)**

  Return $\varphi_i$ of self.

  **EXAMPLES:**

  ```python
  sage: C = crystals.Tableaux(['A', [1,2]], shape=[2,1])
sage: c = C.an_element(); c
  [[-2, -2], [-1]]
sage: c.phi(1)
  0
sage: c.phi(2)
  0
sage: c.phi(0)
  1
  ```

**class TensorProducts**

**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]

super_categories()

EXAMPLES:

sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: C = RegularSuperCrystals()
sage: C.super_categories()
[Category of finite super crystals]

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]

class ElementMethods

Bases: object

class ParentMethods

Bases: object

super_categories()

EXAMPLES:

sage: RightModules(QQ).super_categories()
[Category of commutative additive groups]

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])
sage: I
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]

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 github 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)  # optional - sage.libs.pari
sage: S(1).inverse_of_unit()  # optional - 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
```

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
```

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
```

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
sage: MS = MatrixSpace(ZZ, 2)  #optional - sage.modules
sage: MS.one().is_unit()        #optional - sage.modules
True
sage: MS.zero().is_unit()       #optional - sage.modules
False
sage: MS([1,2,3,4]).is_unit()   #optional - sage.modules
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:

```sage
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  #optional - sage.libs.singular
Ring endomorphism of Fraction Field of Univariate Polynomial Ring in x
    over Rational Field
    Defn: x |--> x + 1
sage: g(x)                                  #optional - sage.libs.singular
x + 1
sage: g(1/x)                                #optional - sage.libs.singular
1/(x + 1)
```

If this morphism is not injective, it does not extend to the fraction field and an error is raised:

```sage
sage: f = GF(5).coerce_map_from(ZZ)         #optional - sage.rings.finite_rings
sage: f.extend_to_fraction_field()         #optional - sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: the morphism is not injective
```
is_injective()

Return whether or not this morphism is injective.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: R.hom([x, y^2], R).is_injective()  # optional - sage.libs.singular
True
sage: R.hom([x, x^2], R).is_injective()  # optional - sage.libs.singular
False
sage: S.<u,v> = R.quotient(x^3*y)
```

```
sage: R.hom([v, u], S).is_injective()  # optional - sage.libs.singular
False
sage: S.hom([-u, v], S).is_injective()  # optional - sage.libs.singular
True
sage: S.cover().is_injective()  # optional - sage.libs.singular
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
```

Unless the codomain is the zero ring:

```
sage: codomain = Integers(1)
sage: f = QQ.hom([Zmod(1)(0)], check=False)
sage: f.is_injective()
False
```

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  # optional - sage.rings.finite_rings
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Finite Field of size 3
  Defn: x |--> 1
sage: f.is_injective()  # optional - sage.rings.finite_rings
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
```

A coercion to the fraction field is injective:

```
sage: R = ZpFM(3)  # optional - sage.rings.padics
sage: R.fraction_field().coerce_map_from(R).is_injective()  # optional - sage.rings.padics
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 self

EXAMPLES:

```
sage: F = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: F  # optional - 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, b, c = F.algebra_generators()  # optional - sage.combinat sage.modules
sage: F.bracket(a, b)  # optional - sage.combinat sage.modules
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:
characteristic()

Return the characteristic of this ring.

EXAMPLES:

```python
sage: QQ.characteristic()
0
sage: GF(19).characteristic()  # optional - sage.rings.finite_rings
19
sage: Integers(8).characteristic()  # optional - sage.rings.finite_rings
8
sage: Zp(5).characteristic()  # optional - sage.rings.padics
0
```

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:

```python
sage: R.<x> = QQ[[x]]
sage: V, from_V, to_V = R.free_module(R)  # optional - sage.modules
  └─optional - sage.combinat sage.modules
sage: v = to_V(1 + x); v  # optional - sage.modules
(1 + x)
sage: from_V(v)  # optional - sage.modules
1 + x
sage: W, from_W, to_W = R.free_module(R, basis=(1 - x))  # optional - sage.modules
  └─optional - sage.combinat sage.modules
sage: W is V  # optional - sage.modules
True
sage: w = to_W(1 + x); w  # optional - sage.modules
(1 - x^2)
sage: from_W(w)  # optional - sage.combinat sage.modules
```

(continues on next page)
ideal(*args, **kwds)

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 [github issue #7797](https://github.com/sagemath/sage/issues/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: MS = MatrixSpace(QQ, 2, 2)  # optional - sage.modules
sage: MS in Rings()                # optional - sage.modules
True
sage: MS.ideal(2)                 # optional - sage.modules
Twosided Ideal
(     [ 2 0 ]
[ 0 2 ]
) of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS.ideal([MS.0, MS.1], side='right')  # optional - sage.modules
Right Ideal
(     [ 1 0 ]
[ 0 0 ],

     [ 0 1 ]
[ 0 0 ]
) of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

**ideal_monoid()**

The monoid of the ideals of this ring.

**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 [github issue #7797](https://github.com/sagemath/sage/issues/7797).

**EXAMPLES:**
```python
sage: MS = MatrixSpace(QQ, 2, 2) # optional - sage.modules
sage: isinstance(MS, Ring) # optional - sage.modules
False
sage: MS in Rings() # optional - sage.modules
True
sage: MS.ideal_monoid() # optional - sage.modules
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

Note that the monoid is cached:
```
```python
sage: MS.ideal_monoid() is MS.ideal_monoid() # optional - sage.modules
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
```

**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()
False
sage: R.<x> = ZZ[]
sage: R.quo(1).is_zero()
True
sage: R.<x> = GF(101)[] # optional - sage.rings.finite_rings
sage: R.quo(77).is_zero() # optional - sage.rings.finite_rings
True
sage: R.quo(x^2 + 1).is_zero() # optional - sage.rings.finite_rings
False
```

**quo(I, names=None, **kwds)**

Quotient of a ring by a two-sided ideal.
Note: This is a synonym for quotient().

EXAMPERS:

```
sage: MS = MatrixSpace(QQ, 2)  # optionally - sage.modules
sage: I = MS * MS.gens() * MS  # optionally - sage.modules
```

MS is not an instance of Ring.
However 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)  # optionally - sage.modules
False
sage: isinstance(MS, Rings().parent_class)  # optionally - sage.modules
True
sage: MS.quo(I, names=['a', 'b', 'c', 'd'])  # optionally - sage.modules
Quotient of Full MatrixSpace of 2 by 2 dense matrices over Rational Field by the ideal

\[
\begin{bmatrix}
1 & 0 \\
0 & 0
\end{bmatrix},
\begin{bmatrix}
0 & 1 \\
0 & 0
\end{bmatrix},
\begin{bmatrix}
0 & 0 \\
1 & 0
\end{bmatrix},
\begin{bmatrix}
0 & 0 \\
0 & 1
\end{bmatrix}
\]
```

A test with a subclass of Ring:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)  # optionally - sage.libs.singular
sage: S.<a,b> = R.quo((x^2, y))  # optionally - sage.libs.singular
sage: S  # optionally - sage.libs.singular
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2, y)
sage: S.gens()  # optionally - sage.libs.singular
(a, 0)
sage: a == b  # optionally - sage.libs.singular
```

(continues on next page)
quotient(I, names=None, **kwds)

Quotient of a ring by a two-sided ideal.

INPUT:
• I – A two-sided ideal of this ring.
• 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 sage.rings.ring.Ring.quotient(). So, we need a bit of effort to make the following example work with the category framework:

```python
sage: F.<x,y,z> = FreeAlgebra(QQ)
     # optional - sage.combinat sage.modules
sage: from sage.rings.noncommutative_ideals import Ideal_nc
     # optional - sage.combinat sage.modules
sage: from itertools import product
sage: class PowerIdeal(Ideal_nc):
     # optional - sage.combinat sage.modules
         ....: 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(0))
sage: I = PowerIdeal(F, 3)
     # optional - sage.combinat sage.modules
sage: Q = Rings().parent_class.quotient(F, I); Q
     # optional - sage.combinat sage.modules
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*y*x, y*y*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*z*x, z*z*y, z*z^2, y^3)
```

```python
sage: Q.0
     # optional - sage.combinat sage.modules
xbar
sage: Q.1
     # optional - sage.combinat sage.modules
ybar
sage: Q.2
     # optional - sage.combinat sage.modules
zbar
sage: Q.0*Q.1
     # optional - sage.combinat sage.modules
xbar*ybar
sage: Q.0*Q.1*Q.0
     # optional - sage.combinat sage.modules
```

(continues on next page)
An example with polynomial rings:

```python
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,)
```

```python
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quotient((x^2, y))  # optional - sage.libs.singular
sage: S  # optional - sage.libs.singular
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2, y)
```

```python
sage: S.gens()  # optional - sage.libs.singular
(a,)
```

```python
sage: a == b  # optional - sage.libs.singular
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)  # optional - sage.modules
```

```python
sage: I = MS * MS.gens() * MS  # optional - 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:

```python
sage: isinstance(MS, sage.rings.ring.Ring)  # optional - sage.modules
False
sage: isinstance(MS, Rings().parent_class)  # optional - sage.modules
```

(continues on next page)
A test with a subclass of `Ring`:

```python
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,)
```

```python
sage: R.<x,y> = PolynomialRing(QQ,2)  # optional - sage.libs.singular
sage: S.<a,b> = R.quotient_ring((x^2, y))  # optional - sage.libs.singular
sage: S  # optional - sage.libs.singular
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2, y)
sage: S.gens()  # optional - sage.libs.singular
(a, 0)
sage: a == b  # optional - sage.libs.singular
False
```

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

NoZeroDivisors()

Return the full subcategory of the objects of self having 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:

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 $(S, +, \ast)$ 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 $(S, \ast)$, where $\ast$ distributes over $\cdot$.

EXAMPLES:

sage: C = Rngs(); C
Category of rngs

sage: sorted(C.super_categories(), key=str)
[Category of associative additive commutative additive associative additive unital left-distributive magmas and additive magmas,
 Category of commutative additive groups]

sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive']

sage: C is (CommutativeAdditiveGroups() & Semigroups()).Distributive()
True
sage: C.Unital()
Category of rings

Unital

alias of Rings
4.138 R-trivial semigroups

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]

    extra_super_categories()
    Implement the fact that a $R$-trivial semigroup is $H$-trivial.

    EXAMPLES:

    sage: Semigroups().RTrivial().extra_super_categories()
    [Category of h trivial semigroups]

4.139 Schemes

class sage.categories.schemes.Schemes(s=None):
    Bases: Category

    The category of all schemes.

    EXAMPLES:

    sage: 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

    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]
class sage.categories.schemes.Schemes_over_base(base, name=None)

Bases: Category_over_base

The category of schemes over a given base scheme.

EXAMPLES:

```python
sage: Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

def base_scheme()

EXAMPLES:

```python
sage: Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring
```

def super_categories()

EXAMPLES:

```python
sage: 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 * which is associative (see Wikipedia article Semigroup).

The operation * is not required to have a neutral element. A semigroup for which such an element exists is a monoid.

EXAMPLES:

```python
sage: C = Semigroups(); C
d
Category of semigroups
sage: C.super_categories()
[Category of magmas]
```

def all_super_categories()

```python
sage: C.all_super_categories()
[Category of semigroups, Category of magmas, Category of sets, Category of sets with partial maps, Category of objects]
```

def axioms()

```python
sage: C.axioms()
frozenset({'Associative'})
```

def example()

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

class Algebras(category, *args)

Bases: AlgebrasCategory

class ParentMethods

Bases: object
algebra_generators()

The generators of this algebra, as per MagmaticAlgebras.ParentMethods.algebra_generators().

They correspond to the generators of the semigroup.

EXAMPLES:

```python
sage: M = FiniteSemigroups().example(); M
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
sage: M.semigroup_generators()
Family ('a', 'b', 'c', 'd')
sage: M.algebra(ZZ).algebra_generators()  # optional - sage.modules
Family (B['a'], B['b'], B['c'], B['d'])
```

gen(i=0)

Return the i-th generator of self.

EXAMPLES:

```python
sage: A = GL(3, GF(7)).algebra(ZZ)  # optional - sage.groups sage.libs.pari sage.modules
sage: A.gen(0)  # optional - sage.groups sage.libs.pari sage.modules
[3 0 0]
[0 1 0]
[0 0 1]
```

gens()

Return the generators of self.

EXAMPLES:

```python
sage: a, b = SL2Z.algebra(ZZ).gens(); a, b  # optional - sage.groups sage.modules
([[ 0 -1]
  [ 1 0]],
 [[1 1]
 [0 1])
sage: 2*a + b  # optional - sage.groups sage.modules
2*[0 -1]
[1 0] +
[1 1]
[0 1]
```

ngens()

Return the number of generators of self.

EXAMPLES:

```python
sage: SL2Z.algebra(ZZ).ngens()  # optional - sage.groups sage.modules
(continues on next page)
```
sage: DihedralGroup(4).algebra(RR).ngens()  # optional - sage.groups sage.modules
2

**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 product of the corresponding elements of the group.

**EXAMPLES:**

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)  # optional - sage.modules
sage: a, b, c, d = A.algebra_generators()  # optional - sage.modules
sage: a * b + b * d * c * d  # optional - 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:**

sage: G = groups.permutation.Dihedral(4)  # optional - sage.groups
sage: A = G.algebra(QQ)  # optional - sage.groups sage.modules
sage: V = A.regular_representation()  # optional - sage.groups sage.modules
sage: V == G.regular_representation(QQ)  # optional - sage.groups sage.modules
True

**trivial_representation**\((side='twosided')\)

Return the trivial representation of \(self\).

**INPUT:**
- \(side\) – ignored

**EXAMPLES:**

sage: G = groups.permutation.Dihedral(4)  # optional - sage.groups
sage: A = G.algebra(QQ)  # optional - sage.groups sage.modules
sage: V = A.trivial_representation()  # optional - sage.groups sage.modules
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

Bases: CartesianProductsCategory

element_class

Implement the fact that a Cartesian product of semigroups is a semigroup.

EXAMPLES:

```
sage: Semigroups().CartesianProducts().extra_super_categories()
[Category of semigroups]
sage: 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

**EXAMPLES:**

We start with the (right) Cayley graphs of some classical groups:

```sage
d4 = DihedralGroup(4); d4
G = d4.cayley_graph()
show(G)
A5 = AlternatingGroup(5); A5
G = A5.cayley_graph()
G.show3d()
```

Alternative generators may be specified:

```sage
G = A5.cayley_graph(generators=[A5.gens()[0]])
G.num_edges()
g = PermutationGroup([(i + 1, j + 1]
```

(continues on next page)
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:

```python
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(4)), []))

sage: G = M.cayley_graph(elements=elements)
```

We now illustrate the side and simple options on a semigroup:

```python
sage: S = FiniteSemigroups().example(alphabet=('a', 'b'))
sage: g = S.cayley_graph(side="left", simple=True)
```

```python
sage: g = S.cayley_graph(side="twosided", simple=True)
```

(continues on next page)
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:

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')
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
```

regular_representation(base_ring=None, side='left')
Return the regular representation of self over base_ring.
- side – (default: "left") whether this is the "left" or "right" regular representation

EXAMPLES:

```python
sage: G = groups.permutation.Dihedral(4)
#optional - sage.groups
sage: G.regular_representation()
#optional - sage.groups
Left Regular Representation of Dihedral group of order 8
as a permutation group over Integer Ring
```

semigroup_generators()
Return distinguished semigroup generators for self.

OUTPUT: a family
This method is optional.

EXAMPLES:

```python
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:
By default, $M$ is just in the category of subsemigroups:

```python
sage: M in Semigroups().Subobjects()  # _
optional - sage.groups
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:

```python
sage: M = R.subsemigroup([R(-1)],  # _
optional - sage.groups
....: category=Monoids().Finite().Subobjects() & Groups()); M
A submonoid of (Ring of integers modulo 15) with 1 generators
sage: M.list()  # _
optional - sage.groups
[1, 14]
sage: M.one()  # _
optional - sage.groups
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:

```python
sage: M = R.subsemigroup([R(5)],  # _
optional - sage.groups
....: 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: M = R.subsemigroup([R(5)], one=R(10),  # _
optional - sage.groups
....: category=Semigroups().Finite().Subobjects() & Groups()); M
A subsemigroup of (Ring of integers modulo 15) with 1 generators
sage: M in Groups()  # _
optional - sage.groups
True
sage: M.list()  # _
optional - sage.groups
[10, 5]
sage: M.one()  # _
optional - sage.groups
10
```
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 \( \mathbb{Z} \)
- side – ignored

**EXAMPLES:**

```python
sage: G = groups.permutation.Dihedral(4)  # optional - sage.groups
sage: G.trivial_representation()  # optional - 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:**

```python
sage: S = FiniteSemigroups().Quotients().example().semigroup\_generators()  # todo: not implemented
```

**example()**

Return an example of quotient of a semigroup, as per Category.example().

**EXAMPLES:**

```python
sage: 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 \emph{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 $s^w s = s$.

**EXAMPLES:**

```python
sage: Semigroups().Aperiodic()
Category of aperiodic semigroups
```

An aperiodic semigroup is $H$-trivial:

```python
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
- `Semigroups.SubcategoryMethods.RTrivial`
- `Semigroups.SubcategoryMethods.LTrivial`
- `Semigroups.SubcategoryMethods.JTrivial`
- `Semigroups.SubcategoryMethods.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
- `Semigroups.SubcategoryMethods.RTrivial`
- `Semigroups.SubcategoryMethods.LTrivial`
- `Semigroups.SubcategoryMethods.JTrivial`
- `Semigroups.SubcategoryMethods.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:

$$x \leq_J y \iff x \in S y S$$

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:**
A semigroup is $J$-trivial if and only if it is $L$-trivial and $R$-trivial:

```
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'JTrivial', 'LTrivial', 'RTrivial']
sage: Semigroups().LTrivial().RTrivial()
Category of j trivial semigroups
```

For a commutative semigroup, all three axioms are equivalent:

```
sage: Semigroups().Commutative().LTrivial()
Category of commutative j trivial semigroups
sage: Semigroups().Commutative().RTrivial()
Category of commutative j trivial semigroups
```

See also:
- Wikipedia article Green%27s_relations
- Semigroups.SubcategoryMethods.LTrivial
- Semigroups.SubcategoryMethods.RTrivial
- Semigroups.SubcategoryMethods.HTrivial

$LTrivial()$

Return the full subcategory of the $L$-trivial objects of self.

Let $S$ be (multiplicative) semigroup. The $L$-preorder $\leq_L$ on $S$ is defined by:

$$x \leq_L y \iff x \in Sy$$

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:

```
sage: C = Semigroups().LTrivial(); C
Category of l trivial semigroups
```

A $L$-trivial semigroup is $H$-trivial:

```
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
c = Semigroups().RTrivial(); c
Category of r trivial semigroups
```

An $R$-trivial semigroup is $H$-trivial:

```python
c.sorted(c.axioms())
['Associative', 'HTrivial', 'RTrivial']
```

See also:

- Wikipedia article Green%27s_relations
- Semigroups.SubcategoryMethods.LTrivial
- Semigroups.SubcategoryMethods.JTrivial
- Semigroups.SubcategoryMethods.HTrivial

**class Subquotients(category, *args)**

Bases: SubquotientsCategory

The category of subquotient semi-groups.

**EXAMPLES:**

```python
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
Semigroups().Subquotients().example()
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:

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 Semirngs

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:

sage: Semirings()
Category of semirings
sage: Semirings().super_categories()
[Category of associative additive commutative additive associative additive unital \ldots distributive magmas and additive magmas, Category of monoids]
sage: sorted(Semirings().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveUnital', 'Associative', \ldots '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
```

Typically, finite group algebras are semisimple:

```
sage: DihedralGroup(5).algebra(QQ) in SemisimpleAlgebras
˓→ optional - sage.groups
True
```

Unless the characteristic of the field divides the order of the group:

```
sage: DihedralGroup(5).algebra(IntegerModRing(5)) in SemisimpleAlgebras
˓→ optional - sage.groups
False
```

```
sage: DihedralGroup(5).algebra(IntegerModRing(7)) in SemisimpleAlgebras
˓→ optional - sage.groups
True
```

See also:

Wikipedia article Semisimple_algebra

class FiniteDimensional(base_category)

Bases: CategoryWithAxiom_over_base_ring

WithBasis

alias of FiniteDimensionalSemisimpleAlgebrasWithBasis

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
sage: A = SymmetricGroup(4).algebra(QQ)     # optional - sage.groups
sage: A.radical_basis()                      # optional - sage.groups
()                                          
```

```
sage: super_categories()                      
EXAMPLES:
```

```sage
sage: Algebras(QQ).Semisimple().super_categories()                                                           
[Category of algebras over Rational Field]
```

## 4.143 Sets

### exception
sage.categories.sets_cat.EmptySetError

Exception raised when some operation can’t be performed on the empty set.

EXAMPLES:

```sage
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
```

### class
tsage.categories.sets_cat.Sets(s=None)

The category of sets.

The base category for collections of elements with = (equality).

This is also the category whose objects are all parents.

EXAMPLES:

```sage
sage: Sets()
Category of sets
sage: Sets().all_super_categories()
[Category of sets, Category of sets with partial maps, Category of objects]
Let us consider an example of set:
```
sage: P = Sets().example("inherits")
sage: P
Set of prime numbers

See P?? for the code.

P is in the category of sets:

sage: P.category()
Category of sets

and therefore gets its methods from the following classes:

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.misc.fast_methods.WithEqualityById'>
<class 'sage.structure.parent.Parent'>
<class 'sage.structure.category_object.CategoryObject'>
<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'>
<... 'object'>

We run some generic checks on P:

sage: TestSuite(P).run(verbos=True)

--- optional - 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
running _test_pickling() ... pass
running _test_some_elements() ... pass

Now, we manipulate some elements of P:
They get their methods from the following classes:

```python
sage: 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.categories.examples.sets_cat.PrimeNumbers_Inherits.Element'>
<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 ...

```python
class Algebras(category, *args)
    Bases: AlgebrasCategory

class ParentMethods
    Bases: object

collection()
    Return the functorial construction of self.

    EXAMPLES:

    sage: A = GroupAlgebra(KleinFourGroup(), QQ)  # optional - sage.groups sage.modules
    sage: F, arg = A.construction(); F, arg  # optional - sage.groups sage.modules
    (GroupAlgebraFunctor, Rational Field)
    sage: F(arg) is A  # optional - sage.groups sage.modules
    True
```
This also works for structures such as monoid algebras (see github issue #27937):

```python
sage: A = FreeAbelianMonoid('x,y').algebra(QQ)  # optional - sage.groups sage.modules
sage: F, arg = A.construction(); F, arg  # optional - sage.groups sage.modules
(The algebra functorial construction,
Free abelian monoid on 2 generators (x, y))
sage: F(arg) is A  # optional - sage.groups sage.modules
True
```

```
extra_super_categories()
```

EXEMPLARY:

```python
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()
[Category of set algebras over Integer Ring,
Category of modules with basis over Integer Ring,
...]
```

```python
class CartesianProducts(category, *args)
```

Bases: `CartesianProductsCategory`

EXEMPLARY:

```python
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
```

```python
class ElementMethods
```

Bases: object

```
cartesian_factors()
```

Return the Cartesian factors of self.

EXEMPLARY:
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.rename("F")  # optional - sage.modules
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.rename("G")  # optional - sage.modules
sage: H = CombinatorialFreeModule(ZZ, [4,7]); H.rename("H")  # optional - sage.modules
sage: S = cartesian_product([F, G, H])  # optional - sage.modules
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()  # optional - sage.modules

sage: [s.parent() for s in x.cartesian_factors()]  # optional - sage.modules
[F, G, H]
sage: S.zero().cartesian_factors()  # optional - sage.modules
(0, 0, 0)
sage: [s.parent() for s in S.zero().cartesian_factors()]  # optional - sage.modules
[F, G, H]

```
cartesian_projection(i)
```

Return the projection of \texttt{self} onto the \(i\)-th factor of the Cartesian product.

INPUT:

- \(i\) -- the index of a factor of the Cartesian product

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)

cardinality()
Return the cardinality of self.
EXAMPLES:

sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E, SymmetricGroup(4)])
# optional - sage.groups
sage: C.cardinality()  # optional - 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()  # optional - sage.rings.finite_rings sage.combinat
1814400

sage: cartesian_product([GF(71)**20]).cardinality() == 71**20  # optional - sage.rings.finite_rings
True

cartesian_factors()
Return the Cartesian factors of self.
EXAMPLES:

sage: 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:

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\})

(continues on next page)
construction()
The construction functor and the list of Cartesian factors.

EXAMPLES:

```python
sage: C = cartesian_product([QQ, ZZ, ZZ])
sage: 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
sage: cartesian_product([S1, S2, S1]).is_empty()
True
```

is_finite()
Return whether this set is finite.

EXAMPLES:

```python
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E, SymmetricGroup(4)])
# optional - sage.groups
sage: C.is_finite()
# optional - 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:

```python
sage: C = cartesian_product([Permutations(10)]*5)
# optional - sage.combinat
```

(continues on next page)
example()

EXAMPLES:

```python
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})
```

extra_super_categories()

A Cartesian product of sets is a set.

EXAMPLES:

```python
sage: Sets().CartesianProducts().extra_super_categories()
[Category of sets]
sage: Sets().CartesianProducts().super_categories()
[Category of sets]
```

class ElementMethods

Bases: object
cartesian_product(*elements)

Return the Cartesian product of its arguments, as an element of the Cartesian product of the parents of those elements.

EXAMPLES:

```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()  # optional - sage.combinat sage.modules
sage: a, b, c = A.algebra_generators()  # optional - sage.combinat sage.modules
sage: a.cartesian_product(b, c)  # optional - sage.combinat sage.modules
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
```

**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
```

**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()
(continues on next page)
[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: P = Partitions(6)           # optional - sage.combinat
  sage: H = Hom(P, ZZ)              # optional - sage.combinat
  sage: f = H(ZZ.sum)               # optional - sage.combinat
  sage: X = f.image()               # optional - sage.combinat
  sage: list(X)                     # optional - sage.combinat
  [6]
  ```

  **is_injective**( )
  Return whether this map is injective.
  
  EXAMPLES:

  ```python
  sage: f = ZZ.hom(GF(3)); f          # optional - sage.rings.finite_rings
  Natural morphism:
  From: Integer Ring
  To:    Finite Field of size 3
  sage: f.is_injective()            # optional - sage.rings.finite_rings
  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 \( \text{None} \)

This returns the space of formal linear combinations of elements of \( S \) with coefficients in \( K \), endowed with whatever structure can be induced from that of \( S \). See the documentation of sage.categories.algebra_functor for details.

EXAMPLES:

If \( S \) is a \textit{group}, the result is its group algebra \( KS \):

\begin{verbatim}
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)
\end{verbatim}

This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication of \( G \) to a multiplication on \( RG \):

\begin{verbatim}
sage: a * a
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)
\end{verbatim}

If \( S \) is a \textit{monoid}, the result is its monoid algebra \( KS \):

\begin{verbatim}
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
Algebra of
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
over Rational Field
sage: A.category()
Category of monoid algebras over Rational Field
\end{verbatim}

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 \( GF_3 \):

\begin{verbatim}
4.143. Sets 787
\end{verbatim}
```
sage: from sage.categories.additive_groups import AdditiveGroups
sage: S = GF(7)  # optional - sage.rings.finite_rings
sage: A = S.algebra(QQ, category=AdditiveGroups()); A  # optional - sage.rings.finite_rings sage.modules
Algebra of Finite Field of size 7 over Rational Field
sage: A.category()  # optional - sage.rings.finite_rings sage.modules
Category of finite dimensional additive group algebras over Rational Field
sage: a = A(S(1))  # optional - sage.rings.finite_rings sage.modules
sage: a
1
sage: 1 + a * a * a  # optional - sage.rings.finite_rings sage.modules
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 `EmptySetError`.

This default implementation calls `_an_element_()` and caches the result. Any parent should implement either `an_element()` or `_an_element_()`.

**EXAMPLES:**

```
sage: CDF.an_element()
1.0*I
sage: ZZ[['t']].an_element()
t
```

### cartesian_product(*parents, **kwargs)

Return the Cartesian product of the parents.

**INPUT:**

- `parents` – a list (or other iterable) of parents.
- `category` – (default: None) the category the Cartesian product belongs to. If None is passed, then `category_from_parents()` is used to determine the category.
- `extra_category` – (default: None) a category that is added to the Cartesian product in addition to the categories obtained from the parents.
- other keyword arguments will passed on to the class used for this Cartesian product (see also `CartesianProduct`).

**OUTPUT:**

The Cartesian product.

**EXAMPLES:**
```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A") # optional - sage.combinat sage.modules
sage: A.cartesian_product(A, A) # optional - sage.combinat sage.modules
A (+) A (+) A
sage: ZZ.cartesian_product(GF(2), FiniteEnumeratedSet([1,2,3])) # optional - sage.rings.finite_rings
The Cartesian product of (Integer Ring,
            Finite Field of size 2, \{1, 2, 3\})

sage: C = ZZ.cartesian_product(A); C # optional - sage.combinat sage.modules
The Cartesian product of (Integer Ring, A)
```

**construction()**

Return a pair (functor, parent) such that functor(parent) returns self. If self does not have a functorial construction, return None.

**EXAMPLES:**

```python
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
```

**is_parent_of(element)**

Return whether self is the parent of element.

**INPUT:**

- element – any object

**EXAMPLES:**

```python
sage: S = ZZ
sage: S.is_parent_of(1)
True
sage: S.is_parent_of(2/1)
False
```

This method differs from `__contains__()` because it does not attempt any coercion:

```python
sage: 2/1 in S, S.is_parent_of(2/1)
(True, False)
sage: int(1) in S, S.is_parent_of(int(1))
(True, False)
```

**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,
  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
The subset algebra of {1, 2, 3} over Rational Field
sage: In = A.In(); In
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: In.realization_of()
```

(continues on next page)
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 *Sets.ParentMethods.algebra()* for the precise meaning in Sage of the *algebra of an object*.

    **EXAMPLES:**

    ```python
    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
    ```

    **See also:**
    
    • *algebra_functor.AlgebrasCategory*
    • *CovariantFunctorialConstruction*

    **CartesianProducts**()
    
    Return the full subcategory of the objects of *self* constructed as Cartesian products.

    **See also:**
    
    • *cartesian_product.CartesianProductFunctor*
    • *RegressiveCovariantFunctorialConstruction*

    **EXAMPLES:**

    ```python
    sage: Sets().CartesianProducts()
    Category of Cartesian products of sets
    sage: Semigroups().CartesianProducts()
    Category of Cartesian products of semigroups
    sage: EuclideanDomains().CartesianProducts()
    Category of Cartesian products of commutative rings
    ```

    **Enumerated**()
    
    Return the full subcategory of the enumerated objects of *self*.

    An enumerated object can be iterated to get its elements.

    **EXAMPLES:**
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
```

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])  
{4, 6, 9}  
sage: Sets().Facade().example()  
An example of facade set: the monoid of positive integers
```

or the union of several parents:

```
sage: Sets().Facade().example("union")  
An example of a facade set: the integers completed by +-infinity
```

or endowing an existing parent with more (or less!) structure:
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:

```python
sage: P = Poset((divisors(12), attrcall("divides")), facade=True)
```

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
```

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
```

On the other hand, with option 2. and once constructed, the elements know unambiguously how to compare themselves:

```python
sage: Q = Poset((divisors(12), attrcall("divides")), facade=False)
sage: Q(2) < Q(3)
False
sage: Q(2) < Q(6)
True
```

Beware that $P(2)$ is still the integer $2$. Therefore $P(2) < P(3)$ still compares $2$ and $3$ as integers!:

```python
sage: P(2) < P(3)
True
```

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:
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:

```
sage: Sets().Finite()
Category of finite sets
sage: Rings().Finite()
Category of finite rings
```

Infinite()

Return the full subcategory of the infinite objects of `self`.

EXAMPLES:

```
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 `As()` (i.e. a subcategory of `Sets()`), `As().IsomorphicObjects()` returns the category of objects of `As()` endowed with a distinguished description as the image of some other object of `As()` 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 \( B = \{1, 2, 3\} \):

```
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`:
The isomorphism from $B$ to $A$ is available as:

\begin{verbatim}
  sage: A.retract(3)
  9
\end{verbatim}

and its inverse as:

\begin{verbatim}
  sage: A.lift(9)
  3
\end{verbatim}

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:

\begin{verbatim}
  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]
\end{verbatim}

Unless something specific about isomorphic objects is implemented for this category, one actually get an optimized super category:

\begin{verbatim}
  sage: C = Semigroups().IsomorphicObjects(); C
  Join of Category of quotients of semigroups
  and Category of isomorphic objects of sets
\end{verbatim}

See also:

- `Subquotients()` for background
- `isomorphic_objects.IsomorphicObjectsCategory`
- `RegressiveCovariantFunctorialConstruction`

**Metric()**

Return the subcategory of the metric objects of `self`.

**Quotients()**

Return the full subcategory of the objects of `self` constructed as quotients.

Given a concrete category `As()` (i.e. a subcategory of `Sets()`), `As().Quotients()` returns the category of objects of `As()` endowed with a distinguished description as quotient (in fact homomorphic image) of some other object of `As()`.

Implementing an object of `As().Quotients()` is done in the same way as for `As().Subquotients()`; namely by providing an ambient space and a lift and a retract map. See `Subquotients()` for detailed instructions.

**See also:**

- `Subquotients()` for background
- `quotients.QuotientsCategory`
- `RegressiveCovariantFunctorialConstruction`

**EXAMPLES:**

```sage
sage: C = Semigroups().Quotients(); C
Category of quotients of semigroups
sage: C.super_categories()
[Category of subquotients of semigroups, Category of quotients of sets]
sage: 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
sage: 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:**

```python
sage: C = Sets().Subobjects(); C
Category of subobjects of sets

sage: C.super_categories()
[Category of subquotients of sets]

sage: C.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]
```

Unless something specific about subobjects is implemented for this category, one actually gets an optimized super category:

```python
sage: 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:**

```python
sage: 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:

$$l : A \rightarrow B' \text{ and } r : B' \rightarrow A$$

called respectively the *lifting map* and *retract map* such that $rol$ is the identity of $A$ and $r$ is a morphism in `As()`.

**Todo:** Draw the typical commutative diagram.
It follows that, for each operation $op$ of the category, we have some property like:

$$op_A(e) = r(op_B(l(e)))$$

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 `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 `Subobjects()`;

**Note:**

- The usual definition of “subquotient” ([Wikipedia article Subquotient](https://en.wikipedia.org/wiki/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 $As()$, $As().Subquotients()$ is a subcategory of $As()$.

  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 $As()$ is a subcategory of $Bs()$, then $As().Subquotients()$ is a subcategory of $Bs().Subquotients()$.

  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 $As().Subquotients()$, 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 $As()$ does not specify anything about its subquotients, then its subquotient category looks like this:

```
sage: 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 $a.lift()$:

```
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
```

(continues on next page)
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.

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)

class ParentMethods
    Bases: object
    ambient()
    Return the ambient space for self.
    EXAMPLES:

    sage: Semigroups().Subquotients().example().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:

    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)

    See also:
\textbf{retract}(x)
Retract \(x\) to \texttt{self}.

\textbf{INPUT:}
\begin{itemize}
\item \(x\) – an element of the ambient space for \texttt{self}
\end{itemize}

\textbf{See also:}
\begin{itemize}
\item \texttt{Sets.SubcategoryMethods.Subquotients} for the specifications, \texttt{ambient()}, \texttt{retract()},
and also \texttt{Sets.Subquotients.ElementMethods.retract()}.
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
sage: S = Semigroups().Subquotients().example()
sage: s = S.ambient().an_element()
sage: 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)
\end{verbatim}

\textbf{Topological}
alias of \texttt{TopologicalSpaces}

\textbf{class \texttt{WithRealizations}(category, *args)}
Bases: \texttt{WithRealizationsCategory}

\textbf{class \texttt{ParentMethods}}
Bases: \texttt{object}

\textbf{class \texttt{Realizations}(parent_with_realization)}
Bases: \texttt{Category_realization_of_parent}

\textbf{super_categories()}
\begin{verbatim}
sage: A = Sets().WithRealizations().example(); A
#...
optional - sage.combinat sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.Realizations().super_categories()
#...
optional - sage.combinat sage.modules
[Category of realizations of sets]
\end{verbatim}

\textbf{a_realization()}
Return a realization of \texttt{self}.

\textbf{EXAMPLES:}
\begin{verbatim}
sage: A = Sets().WithRealizations().example(); A
#...
optional - sage.combinat sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.a_realization()
#...
optional - sage.combinat sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field
in the Fundamental basis
\end{verbatim}
facade_for()

Return the parents self is a facade for, that is the realizations of self

EXAMPLES:

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

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:

```
sage: S = SymmetricFunctions(ZZ); S
Symmetric Functions over Integer Ring
sage: s = S.s(); s
Symmetric Functions over Integer Ring in the Schur basis
sage: e = S.e(); e
Symmetric Functions over Integer Ring in the elementary basis
```

This method automates the process:
Sometimes, like for symmetric functions, one can request for all shorthands to be defined, including less common ones:

```
sage: S.inject_shorthands("all")
# optional - sage.combinat
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
Defining sp as shorthand for
    Symmetric Functions over Integer Ring in the symplectic basis
```

(continues on next page)
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

The messages can be silenced by setting verbose=False:

```
sage: Q = QuasiSymmetricFunctions(ZZ)  # optional - sage.combinat
sage: Q.inject_shorthands()          # optional - sage.combinat
```

```
```

One can also just import a subset of the shorthands:

```
sage: SQ = SymmetricFunctions(QQ)  # optional - sage.combinat
sage: SQ.inject_shorthands(['p', 's'], verbose=False)  # optional - sage.combinat
```

```
sage: p  # optional - sage.combinat
Symmetric Functions over Rational Field in the powersum basis
sage: s  # optional - sage.combinat
Symmetric Functions over Rational Field in the Schur basis
```

Note that e is left unchanged:

```
sage: e  # optional - sage.combinat
```

```
sage: A = Sets().WithRealizations().example(); A  # optional - sage.combinat sage.modules
```

realizations()
Return all the realizations of self that self is aware of.

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A  # optional - sage.combinat sage.modules
```

(continues on next page)
The subset algebra of \{1, 2, 3\} over Rational Field

sage: A.realizations()  # optional - sage.combinat 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)

example(base_ring=None, set=None)

Return an example of set with multiple realizations, as per Category.example().

EXAMPLES:

sage: Sets().WithRealizations().example()  # optional - sage.combinat sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field

sage: Sets().WithRealizations().example(ZZ, Set([1,2]))  # optional - sage.combinat sage.modules
The subset algebra of \{1, 2\} over Integer Ring

extra_super_categories()

A set with multiple realizations is a facade parent.

EXAMPLES:

sage: Sets().WithRealizations().extra_super_categories()
[Category of facade sets]

sage: Sets().WithRealizations().super_categories()
[Category of facade sets]

example(choice=None)

Return examples of objects of Sets(), as per Category.example().

EXAMPLES:

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)
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]
```

```
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
```

4.144 Sets With a Grading

class sage.categories.sets_with_grading.SetsWithGrading(s=None)

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):

\[
S = \biguplus_{i \in I} S_i
\]

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 \( \text{graded\_component()} \) or \( \text{subset()} \). If only \( \text{subset()} \) is implemented, the first argument must be the grading for compatibility with \( \text{graded\_component()} \). Additionally either the parent must implement \( \text{grading()} \) or its elements must implement a method \( \text{grade()} \). See the example \sage.categories.examples.sets_with_grading.NonNegativeIntegers.\n
Finally, if the graded set is enumerated (see \text{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:
The grading function is given by \texttt{N.grading}: 

\begin{verbatim}
sage: N.grading(4)
4
\end{verbatim}

The graded component $N_i$ is the set with one element $i$: 

\begin{verbatim}
sage: N.graded_component(grade=5)
\{5\}
sage: N.graded_component(grade=42)
\{42\}
\end{verbatim}

Here are some information about this category: 

\begin{verbatim}
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]
\end{verbatim}

Todo:

- This should be moved to \texttt{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()
```

(continues on next page)
\[ \frac{1}{-z + 1} \]

```
sage: Permutations().generating_series()
˓→# optional - 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}
```

**grading** *(elt)*

Return the grading of the element *elt* of *self*.

This default implementation calls *elt*.grade().

**EXAMPLES:**

```
sage: N = SetsWithGrading().example(); N
Non negative integers
sage: N.grading(4)
4
```

**grading_set()**

Return the set *self* is graded by. By default, this is the set of non-negative integers.

**EXAMPLES:**

```
sage: 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:**

```
sage: W = WeightedIntegerVectors([3,2,1]); W
˓→# optional - sage.combinat
Integer vectors weighted by [3, 2, 1]
Sage: W.subset(4)
˓→# ...
```

(continues on next page)
Integer vectors of 4 weighted by \([3, 2, 1]\)

**EXAMPLES:**

```python
sage: SetsWithGrading().super_categories()

[Category of sets]
```

## 4.145 SetsWithPartialMaps

**class** `sage.categories.sets_with_partial_maps.SetsWithPartialMaps(s=None)`

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 \(\{\text{error}\}\), 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 \(\text{error}\) if it does.

**EXAMPLES:**

```python
sage: SetsWithPartialMaps()
Category of sets with partial maps

sage: SetsWithPartialMaps().super_categories()

[Category of objects]
```

**super_categories()**

**EXAMPLES:**

```python
sage: SetsWithPartialMaps().super_categories()

[Category of objects]
```

## 4.146 Shephard Groups

**class** `sage.categories.shephard_groups.ShephardGroups(s=None)`

Bases: `Category_singleton`

The category of Shephard groups.

**EXAMPLES:**

```python
sage: from sage.categories.shephard_groups import ShephardGroups
sage: C = ShephardGroups(); C
Category of shephard groups
```
super_categories()
EXAMPLES:

```python
sage: from sage.categories.shephard_groups import ShephardGroups
sage: ShephardGroups().super_categories()
[Category of finite generalized coxeter groups]
```

### 4.147 Simplicial Complexes

class sage.categories.simplicial_complexes.SimplicialComplexes(s=None)

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:

```python
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: C = SimplicialComplexes(); C
Category of simplicial complexes
```

class Connected(base_category)

Bases: CategoryWithAxiom

The category of connected simplicial complexes.

EXAMPLES:

```python
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: C = SimplicialComplexes().Connected(); C
Category of simplicial complexes
```

class Finite(base_category)

Bases: CategoryWithAxiom

Category of finite simplicial complexes.

#### class ParentMethods

Bases: object

```python
dimension()
```

Return the dimension of self.

EXAMPLES:
class ParentMethods

Bases: object

faces()

Return the faces of self.

EXAMPLES:

```
sage: S = SimplicialComplex([[1,3,4], [1,2],[2,5],[4,5]])  #
  #optional - sage.graphs
sage: S.dimension()  #
  #optional - sage.graphs
2
```

facets()

Return the facets of self.

EXAMPLES:

```
sage: S = SimplicialComplex([[1,3,4], [1,2],[2,5],[4,5]])  #
  #optional - sage.graphs
sage: S.faces()  #
  #optional - 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)}}
```

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
```

super_categories()

Return the super categories of self.

EXAMPLES:

```
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
sage: SimplicialComplexes().super_categories()
[Category of sets]
```
### 4.148 Simplicial Sets

**class** `sage.categories.simplicial_sets.SimplicialSets(s=None)`

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

\[
d_i : X_n \to X_{n-1}, \quad 0 \leq i \leq n \quad \text{(face maps)}
\]

\[
s_j : X_n \to X_{n+1}, \quad 0 \leq j \leq n \quad \text{(degeneracy maps)}
\]

satisfying the *simplicial identities*:

\[
d_id_j = d_{j-1}d_i \quad \text{if } i < j
\]

\[
d_is_j = s_{j-1}d_i \quad \text{if } i < j
\]

\[
d_js_j = 1 = d_{j+1}s_j
\]

\[
d_is_j = s_jd_{i-1} \quad \text{if } i > j + 1
\]

\[
s_is_j = s_{j+1}s_i \quad \text{if } i \leq j
\]

Morphisms are sequences of maps \( f_i : X_i \to Y_i \) which commute with the face and degeneracy maps.

**EXAMPLES:**

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: 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 \( \text{Hom}(S, S) \).

**EXAMPLES:**

```python
sage: T = simplicial_sets.Torus()
sage: Hom(T, T).identity()  #optional - 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()  # optional - sage.graphs
True
sage: C5 = groups.misc.MultiplicativeAbelian([5])  # optional - sage.graphs sage.groups
sage: simplicial_sets.ClassifyingSpace(C5).is_finite()  # optional - sage.graphs sage.groups
False
```

is_pointed()

Return True if this simplicial set is pointed, i.e., has a base point.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet  # optional - sage.graphs
sage: v = AbstractSimplex(0)  # optional - sage.graphs
sage: w = AbstractSimplex(0)  # optional - sage.graphs
sage: e = AbstractSimplex(1)  # optional - sage.graphs
sage: X = SimplicialSet({e: (v, w)})  # optional - sage.graphs
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)  # optional - sage.graphs
sage: X.is_pointed()  # optional - sage.graphs
False
sage: Y.is_pointed()  # optional - sage.graphs
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:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet  # optional - sage.graphs
sage: v = AbstractSimplex(0, name='v_0')  # optional - sage.graphs
sage: w = AbstractSimplex(0, name='w_0')  # optional - sage.graphs
sage: e = AbstractSimplex(1)  # optional - sage.graphs
```

(continues on next page)
sage: X = SimplicialSet({e: (v, w)})  # optional - sage.graphs
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)  # optional - sage.graphs
sage: Y.base_point()  # optional - sage.graphs
w_0
sage: X_star = X.set_base_point(w)  # optional - sage.graphs
sage: X_star.base_point()  # optional - sage.graphs
w_0
sage: Y_star = Y.set_base_point(v)  # optional - sage.graphs
sage: Y_star.base_point()  # optional - sage.graphs
v_0

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: S1 = simplicial_sets.Sphere(1)  # optional - sage.graphs
sage: S1.fat_wedge(0)  # optional - sage.graphs
Point
sage: S1.fat_wedge(1)  # optional - sage.graphs
S^1
sage: S1.fat_wedge(2).fundamental_group()  # optional - sage.graphs sage.groups
Finitely presented group < e0, e1 | >

smash_product(*others)

Return the smash product of this simplicial set with others.
INPUT:
• others – one or several simplicial sets

EXAMPLES:

```python
sage: S1 = simplicial_sets.Sphere(1)  #
    optional - sage.graphs
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)  #
    optional - sage.graphs sage.groups
sage: X = S1.smash_product(RP2)  #
    optional - sage.graphs sage.groups
sage: X.homology(base_ring=GF(2))  #
    optional - sage.graphs sage.modules sage.rings.finite_rings
{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}
```

```python
sage: T = S1.product(S1)  #
    optional - sage.graphs
sage: X = T.smash_product(S1)  #
    optional - sage.graphs
sage: X.homology(reduced=False)  #
    optional - sage.graphs sage.modules
{0: Z, 1: 0, 2: Z x Z, 3: Z}
```

```python
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  # optional - sage.graphs
sage: v = AbstractSimplex(0, name='v_0')  #
    optional - sage.graphs
sage: w = AbstractSimplex(0, name='w_0')  #
    optional - sage.graphs
sage: e = AbstractSimplex(1)  #
    optional - sage.graphs
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)  #
    optional - sage.graphs
sage: Y.is_pointed()  #
    optional - sage.graphs
True
sage: Y.base_point()  #
    optional - sage.graphs
w_0
sage: Z = Y.unset_base_point()  #
    optional - sage.graphs
sage: Z.is_pointed()  #
    optional - sage.graphs
False
```

```python
class ParentMethods
    Bases: object
```

4.148. Simplicial Sets
base_point()

Return this simplicial set's base point.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet  # optional - sage.graphs
sage: v = AbstractSimplex(0, name='*')  # optional - sage.graphs
sage: e = AbstractSimplex(1)  # optional - sage.graphs
sage: S1 = SimplicialSet({e: (v, v)}, base_point=v)  # optional - sage.graphs
sage: S1.is_pointed()  # optional - sage.graphs
True
sage: S1.base_point()  # optional - sage.graphs
*
```

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:

```
sage: T = simplicial_sets.Torus()  # optional - sage.graphs
sage: f = T.base_point_map(); f  # optional - sage.graphs
Simplicial set morphism:
   From: Point
   To:   Torus
   Defn: Constant map at (v_0, v_0)

sage: S3 = simplicial_sets.Sphere(3)  # optional - sage.graphs
sage: g = S3.base_point_map()  # optional - sage.graphs
sage: f.domain() == g.domain()  # optional - sage.graphs
True
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)  # optional - sage.graphs sage.groups
sage: temp = simplicial_sets.Simplex(0)  # optional - sage.graphs sage.groups
sage: pt = temp.set_base_point(temp.n_cells(0)[0])  # optional - sage.graphs sage.groups
sage: h = RP3.base_point_map(domain=pt)  # optional - sage.graphs sage.groups
sage: f.domain() == h.domain()  # optional - sage.graphs sage.groups
```

(continues on next page)
connectivity(max_dim=None)

Return the connectivity of this pointed simplicial set.

INPUT:
• 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 is_simply_connected() method.

The connectivity of a contractible space is +Infinity.

EXAMPLES:

sage: simplicial_sets.Sphere(3).connectivity()  # optional - sage.graphs sage.groups
2
sage: simplicial_sets.Sphere(0).connectivity()  # optional - sage.graphs sage.groups
-1
sage: K = simplicial_sets.Simplex(4)  # optional - sage.graphs
sage: K = K.set_base_point(K.n_cells(0)[0])  # optional - sage.graphs
sage: K.connectivity()  # optional - sage.graphs sage.groups
+Infinity
sage: X = simplicial_sets.Torus().suspension(2)  # optional - sage.graphs
sage: X.connectivity()  # optional - sage.graphs sage.groups
2
sage: C2 = groups.misc.MultiplicativeAbelian([2])  # optional - sage.graphs sage.groups
sage: BC2 = simplicial_sets.ClassifyingSpace(C2)  # optional - sage.graphs sage.groups
(continues on next page)
optional - sage.graphs sage.groups
sage: BC2.connectivity()  #_
optional - sage.graphs sage.groups
@

\textbf{cover(\texttt{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 nondegenerate 1-dimensional cell. It should correspond to an epimorphism from the fundamental group.

**INPUT:**
- \texttt{character} – a dictionary

**EXAMPLES:**

```
sage: S1 = simplicial_sets.Sphere(1)  #_
optional - sage.graphs
sage: W = S1.wedge(S1)  #_
optional - sage.graphs
sage: G = CyclicPermutationGroup(3)  #_
optional - sage.graphs
sage: (a, b) = W.n_cells(1)  #_
optional - sage.graphs
sage: C = W.cover({a : G.gen(0), b : G.gen(0)^2})  #_
optional - sage.graphs sage.groups
sage: C.face_data()  #_
optional - sage.graphs sage.groups
{(*, ()): 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)): ((*, ()), (*, (1,2,3))),
 (sigma_1, (1,3,2)): ((*, ()), (*, (1,3,2))),
 (sigma_1, (1,3,2)): ((*, (1,2,3)), (*, (1,3,2)))}
```

```
sage: C.homology(1)  #_
optional - sage.graphs sage.groups sage.modules
Z x Z x Z x Z
sage: C.fundamental_group()  #_
optional - sage.graphs sage.groups
Finitely presented group < e0, e1, e2, e3 | >
```

optional - sage.graphs sage.groups
sage: BC2.connectivity()  #_
optional - sage.graphs sage.groups
@

\textbf{covering_map(\texttt{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 nondegenerate 1-dimensional cell. It should correspond to an epimorphism from the fundamental group.

**INPUT:**
- \texttt{character} – a dictionary

**EXAMPLES:**

```
sage: S1 = simplicial_sets.Sphere(1)  #_
optional - sage.graphs
sage: W = S1.wedge(S1)  #_
optional - sage.graphs
sage: G = CyclicPermutationGroup(3)  #_
optional - sage.graphs
sage: (a, b) = W.n_cells(1)  #_
optional - sage.graphs
sage: C = W.cover({a : G.gen(0), b : G.gen(0)^2})  #_
optional - sage.graphs sage.groups
sage: C.face_data()  #_
optional - sage.graphs sage.groups
{(*, ()): 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)): ((*, ()), (*, (1,2,3))),
 (sigma_1, (1,3,2)): ((*, ()), (*, (1,3,2))),
 (sigma_1, (1,3,2)): ((*, (1,2,3)), (*, (1,3,2)))}
```
```
sage: S1 = simplicial_sets.Sphere(1)  # optional - sage.graphs
sage: W = S1.wedge(S1)  # optional - sage.graphs
sage: G = CyclicPermutationGroup(3)  # optional - sage.groups
sage: a, b = W.n_cells(1)  # optional - sage.graphs
sage: C = W.covering_map({a : G.gen(0), b : G.one()}); C  # optional - sage.graphs sage.groups
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, (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()  # optional - sage.graphs sage.groups
Simplicial set with 9 non-degenerate simplices
sage: C.domain().face_data()  # optional - sage.graphs sage.groups
{(*, ()): 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)))}
```

**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 $e_i = 1$ if $e_i$ is degenerate.

**EXAMPLES:**

```
sage: S1 = simplicial_sets.Sphere(1)  # optional - sage.graphs
sage: eight = S1.wedge(S1)  # optional - sage.graphs
sage: eight.fundamental_group() # free group on 2 generators  # optional - sage.graphs sage.groups
```

(continues on next page)
The fundamental group of a disjoint union of course depends on the choice of base point:

```
sage: T = simplicial_sets.Torus()
    # optional - sage.graphs
sage: K = simplicial_sets.KleinBottle()
    # optional - sage.graphs
sage: X = T.disjoint_union(K)
    # optional - sage.graphs
sage: X_0 = X.set_base_point(X.n_cells(0)[0])
    # optional - sage.graphs
sage: X_0.fundamental_group().is_abelian()
    # optional - sage.graphs sage.groups
True
sage: X_1 = X.set_base_point(X.n_cells(0)[1])
    # optional - sage.graphs
sage: X_1.fundamental_group().is_abelian()
    # optional - sage.graphs sage.groups
False
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
    # optional - sage.graphs sage.groups
sage: RP3.fundamental_group()
    # optional - sage.graphs sage.groups
Finitely presented group < e | e^2 >
```

Compute the fundamental group of some classifying spaces:

```
sage: C5 = groups.misc.MultiplicativeAbelian([5])
    # optional - sage.graphs sage.groups
sage: BC5 = C5.nerve()
    # optional - sage.graphs sage.groups
sage: BC5.fundamental_group()
    # optional - sage.graphs sage.groups
Finitely presented group < e0 | e0^5 >
```

```
sage: Sigma3 = groups.permutation.Symmetric(3)
    # optional - sage.graphs sage.groups
sage: BSigma3 = Sigma3.nerve()
    # optional - sage.graphs sage.groups
sage: pi = BSigma3.fundamental_group(); pi
    # optional - sage.graphs sage.groups
Finitely presented group < e1, e2 | e2^2, e1^3, (e2*e1)^2 >
sage: pi.order()
    # optional - sage.graphs sage.groups
6
sage: pi.is_abelian()
    # optional - sage.graphs sage.groups
False
```

The sphere has a trivial fundamental group:
sage: S2 = simplicial_sets.Sphere(2)  # optional - sage.graphs
sage: S2.fundamental_group()  # optional - 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:**

```python
sage: T = simplicial_sets.Torus()  # optional - sage.graphs
sage: T.is_simply_connected()  # optional - sage.graphs sage.groups
False
sage: T.suspension().is_simply_connected()  # optional - sage.graphs sage.groups
True
sage: simplicial_sets.KleinBottle().is_simply_connected()  # optional - sage.graphs sage.groups
False
sage: S2 = simplicial_sets.Sphere(2)  # optional - sage.graphs
sage: S3 = simplicial_sets.Sphere(3)  # optional - sage.graphs
sage: (S2.wedge(S3)).is_simply_connected()  # optional - sage.graphs sage.groups
True
sage: X = S2.disjoint_union(S3)
```

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: RP3 = simplicial_sets.RealProjectiveSpace(3)  #optional - sage.groups
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 (1, 1), (f, 1), (f * f, 1)),
 (f * f * f, e): ((f * f, 1), s_0 (f, 1), (f, e), (f * f, e))}

sage: C.fundamental_group()
Finitely presented group < | >

universal_cover_map()

Return the universal covering map of the simplicial set.
It requires the fundamental group to be finite.

**EXAMPLES:**

```python
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)  #optional - 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()
{(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:**
super_categories()

EXAMPLES:

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: 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
```

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.

tensor(*parents, **kwargs)

Return the tensor product of the parents.

EXAMPLES:

```python
sage: A.<x,y,z> = ExteriorAlgebra(ZZ); A.rename("A")
optional - sage.combinat sage.modules
sage: T = A.tensor(A,A); T
optional - sage.combinat sage.modules
```

(continues on next page)
A # A # A

```python
sage: T in Algebras(ZZ).Graded().SignedTensorProducts()  # ~
  ...optional - sage.combinat sage.modules
True
sage: T in Algebras(ZZ).Graded().TensorProducts()  # ~
  ...optional - sage.combinat sage.modules
False
sage: A.rename(None)  # ~
  ...optional - sage.combinat sage.modules
```

This also works when the other elements do not have a signed tensor product (github issue #31266):

```python
sage: a = SteenrodAlgebra(3).an_element()  # ~
  ...optional - sage.modules
sage: M = CombinatorialFreeModule(GF(3), ['s', 't', 'u'])  # ~
  ...optional - sage.modules sage.rings.finite_rings
sage: s = M.basis()['s']  # ~
  ...optional - sage.modules sage.rings.finite_rings
sage: tensor([a, s])  # ~
2*Q_1 Q_3 P(2,1) # B['s']
```

class `SignedTensorProducts`(category, *args)

Bases: `SignedTensorProductsCategory`

def `extra_super_categories`()

EXAMPLES:

```python
sage: Coalgebras(QQ).Graded().SignedTensorProducts().extra_super_categories()
[Category of graded coalgebras over Rational Field]
```

Meaning: a signed tensor product of coalgebras is a coalgebra

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:

```python
sage: Algebras(ZZ).Super().Supercommutative()  
Category of supercommutative algebras over Integer Ring
sage: Algebras(ZZ).Super().WithBasis().Supercommutative()  
Category of supercommutative algebras with basis over Integer Ring
```
Supercommutative
    alias of SupercommutativeAlgebras

element_extra_super_categories()
    EXAMPLES:

```
sage: Algebras(ZZ).Super().super_categories() # indirect doctest
[Category of graded algebras over Integer Ring,
 Category of super modules over Integer Ring]
```

## 4.150 Super algebras with basis

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
```

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: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
    (optional - sage.modules)
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
    (optional - sage.combinat sage.modules)
sage: a = x*y - z
    (optional - sage.combinat sage.modules)
sage: b = x - y + y*z
    (optional - sage.combinat sage.modules)
sage: a.supercommutator(b)
    (optional - sage.combinat sage.modules)
-5*x*y + 8*x*z - 2*y*z - 6*x + 12*y - 5*z
sage: a.supercommutator(Cl.one())
    (optional - sage.combinat sage.modules)
0
sage: Cl.one().supercommutator(a)
    (optional - sage.combinat sage.modules)
0
```

(continues on next page)
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)

sage: all(b1.supercommutator(b2) == 0
....: for b1 in E.basis() for b2 in E.basis())
True
```

### 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:**

```python
sage: W.<x,y> = algebras.DifferentialWeyl(QQ)

sage: W.graded_algebra()

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:

```python
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]
```

Meaning: a signed tensor product of super algebras is a super algebra

extra_super_categories()

EXAMPLES:

```python
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]
```

4.151 Super Hopf algebras with basis

```python
class sage.categories.super_hopf_algebras_with_basis.SuperHopfAlgebrasWithBasis(base_category)

Bases: SuperModulesCategory

The category of super Hopf algebras with a distinguished basis.

EXAMPLES:

```python
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 algebras with basis over Integer Ring,
 Category of super coalgebras with basis over Integer Ring,
 Category of super hopf algebras over Integer Ring]
```

```python
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:

```python
sage: A = SteenrodAlgebra(7)
  # optional - sage.combinat sage.modules
sage: a = A.an_element()
  # optional - sage.combinat sage.modules
sage: a, A.antipode(a)
  # optional - sage.combinat sage.modules
(6 Q_1 Q_3 P(2,1), Q_1 Q_3 P(2,1))
```

4.151. Super Hopf algebras with basis 827
4.152 Super Lie Conformal Algebras

AUTHORS:


class sage.categories.super_lie_conformal_algebras.SuperLieConformalAlgebras(base_category)

    Bases: SuperModulesCategory

    The category of super Lie conformal algebras.

    EXAMPLES:

    sage: LieConformalAlgebras(AA).Super()  # optional - 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:

    sage: bosondict = {('a','a'): {1:{('K',0):1}}}
    sage: R = LieConformalAlgebra(QQ, bosondict, names=('a',))  # optional - sage.combinat sage.modules
    ....:     central_elements=('K',), super=True)
    sage: [g.is_even_odd() for g in R.gens()]  # optional - sage.combinat sage.modules
    [0, 0]

class ElementMethods

    Bases: object

    is_even_odd()

    Return 0 if this element is even and 1 if it is odd.

    EXAMPLES:

    sage: R = lie_conformal_algebras.NeveuSchwarz(QQ)  # optional - sage.combinat sage.modules
    sage: R.inject_variables()  # optional - sage.combinat sage.modules
    Defining L, G, C
    sage: G.is_even_odd()  # optional - sage.combinat sage.modules
    1

class Graded(base_category)

    Bases: GradedModulesCategory

    The category of H-graded super Lie conformal algebras.

    EXAMPLES:

    sage: LieConformalAlgebras(AA).Super().Graded()  # optional - sage.rings.number_field
    Category of H-graded super Lie conformal algebras over Algebraic Real Field

    class ParentMethods

        Bases: object
example()

An example parent in this category.

EXAMPLES:

```sage
sage: LieConformalAlgebras(QQ).Super().example()
# optional - sage.combinat sage.modules
The Neveu-Schwarz super Lie conformal algebra over Rational Field
```

eextra_super_categories()

The extra super categories of self.

EXAMPLES:

```sage
sage: LieConformalAlgebras(QQ).Super().super_categories()
[Category of super modules over Rational Field,
 Category of Lambda bracket algebras over Rational Field]
```

## 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
sage: Modules(ZZ).Super()
Category of super modules over Integer Ring
sage: 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
sage: 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
sage: cat = Algebras(QQ).WithBasis().Super()
sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)  # optional - sage.combinat sage.modules
sage: C.de]:
```
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
cat = Algebras(QQ).WithBasis().Super()
C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
C.degree_on_basis = sum
C.basis()[2,2,1].is_even_odd()
C.basis()[2,2].is_even_odd()
```

is_odd()

Return if self is an odd element.

**EXAMPLES:**

```sage
cat = Algebras(QQ).WithBasis().Super()
C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
C.degree_on_basis = sum
C.basis()[2,2,1].is_odd()
C.basis()[2,2].is_odd()
```

class ParentMethods

   Bases: object

   **extra_super_categories()**

   Adds *VectorSpaces* to the super categories of self if the base ring is a field.

   **EXAMPLES:**
This makes sure that `Modules(QQ).Super()` returns an instance of `SuperModules` and not a join category of an instance of this class and of `VectorSpaces(QQ)`:

```
sage: type(Modules(QQ).Super())
<class 'sage.categories.super_modules.SuperModules_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.
• category – a category \textit{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:

\begin{verbatim}
sage: HopfAlgebras(ZZ).WithBasis().FiniteDimensional().Super()  # indirect
˓→doctest
Category of finite dimensional super hopf algebras with basis over Integer Ring
\end{verbatim}

\section{4.154 Super modules with basis}

\begin{verbatim}
class sage.categories.super_modules_with_basis.SuperModulesWithBasis(base_category)

Bases: SuperModulesCategory

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:

\begin{verbatim}
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_modules_with_basis(QQ).Graded()
True
\end{verbatim}

\end{verbatim}

\begin{verbatim}
class ElementMethods

Bases: object

\begin{verbatim}
even_component()

Return the even component of self.

EXAMPLES:

\begin{verbatim}
sage: Q = QuadraticForm(QQ, 2, [1,2,3])          #˓→optional - sage.modules
˓→optional - sage.combinat
sage: C.<x,y> = CliffordAlgebra(Q)                 #˓→optional - sage.combinat sage.modules
˓→optional - sage.combinat sage.modules
sage: a = x*y + x - 3*y + 4                         #˓→optional - sage.combinat sage.modules
˓→optional - sage.combinat sage.modules
sage: a.even_component()                           #˓→optional - sage.combinat sage.modules
x^2 + 4
\end{verbatim}
\end{verbatim}

\end{verbatim}
is_even_odd()  
Return $0$ if `self` is an even element and $1$ if `self` is an odd element.

EXAMPLES:

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])  #˓→optional - sage.modules
sage: C.<x,y> = CliffordAlgebra(Q)  #˓→optional - sage.combinat sage.modules
sage: a = x + y  #˓→optional - sage.combinat sage.modules
sage: a.is_even_odd()  #˓→optional - sage.combinat sage.modules
1
sage: a = x*y + 4  #˓→optional - sage.combinat sage.modules
sage: a.is_even_odd()  #˓→optional - sage.combinat sage.modules
0
sage: a = x + 4  #˓→optional - sage.combinat sage.modules
sage: a.is_even_odd()  #˓→optional - sage.combinat sage.modules
Traceback (most recent call last):
  ...  
ValueError: element is not homogeneous

sage: E.<x,y> = ExteriorAlgebra(QQ)  #˓→optional - sage.combinat sage.modules
sage: (x*y).is_even_odd()  #˓→optional - sage.combinat sage.modules
0
```

is_super_homogeneous()  
Return whether this element is homogeneous, in the sense of a super module (i.e., is even or odd).

EXAMPLES:

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])  #˓→optional - sage.modules
sage: C.<x,y> = CliffordAlgebra(Q)  #˓→optional - sage.combinat sage.modules
sage: a = x + y  #˓→optional - sage.combinat sage.modules
sage: a.is_super_homogeneous()  #˓→optional - sage.combinat sage.modules
True
sage: a = x*y + 4  #˓→optional - sage.combinat sage.modules
sage: a.is_super_homogeneous()  #˓→optional - sage.combinat sage.modules
True
sage: a = x*y + x - 3*y + 4  #˓→optional - sage.combinat sage.modules
```

(continues on next page)
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:

```python
sage: E.<x,y> = ExteriorAlgebra(QQ)
# optional - sage.combinat sage.modules
sage: a = x*y + 4
# optional - sage.combinat sage.modules
sage: a.is_super_homogeneous()  # optional - sage.combinat sage.modules
True
sage: a.is_homogeneous()  # optional - sage.combinat sage.modules
False
```

### odd_component()

Return the odd component of self.

**EXAMPLES:**

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
# optional - sage.modules
sage: C.<x,y> = CliffordAlgebra(Q)
# optional - sage.combinat sage.modules
sage: a = x*y + x - 3*y + 4
# optional - sage.combinat sage.modules
sage: a.odd_component()  # optional - sage.combinat sage.modules
x - 3*y
```

### 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:

\[
x_0 x'_0 = x'_0 x_0, \quad x_1 x'_1 = -x'_1 x_1, \quad x_0 x_1 = x_1 x_0,
\]

for all \(x_0, x'_0 \in A_0\) and \(x_1, x'_1 \in A_1\).

**EXAMPLES:**
sage: Algebras(ZZ).Supercommutative()
Category of supercommutative algebras over Integer Ring

class SignedTensorProducts(category, *args)
    Bases: SignedTensorProductsCategory
    
    extra_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(s=None)
    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]] True
    [[-2, -2, -2], [-1, 2], [1]] False
    [[-2, -2, 2], [-1, -1], [1]] False
    [[-2, -2, 2], [-1, 2], [2]]
    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', [1,1]], shape=[3,2,1])
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, 2], [-1, 2], [2]] False
    [[-1, 1, 2], [1, 2], [2]] True
```

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)]
```

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]]]]

c 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,)

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)]

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)

genuine_highest_weight_vectors()

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,))

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: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_lowest_weight_vectors()
((3,))

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]])

highest_weight_vectors()

Return the highest weight vectors of self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,2]])
sage: B.highest_weight_vectors()
((-2,))

sage: T = B.tensor(B)
sage: T.highest_weight_vectors()
(([-2, -2], [-2, -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]]])

sage: 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,)
sage: T = B.tensor(B)
sage: sorted(T.lowest_weight_vectors())
[[3, 2], [3, 3]]
```

We give an example from [BKK2000] that has fake lowest weight vectors:

```python
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: sorted(B.lowest_weight_vectors())
[[[-2, 1, 2], [-1, 2], [1]],
 [-2, 1, 2], [-1, 2], [2]],
 [[-1, 1, 2], [1, 2], [2]]
```

```python
class ParentMethods
Bases: object

tensor(*crystals, **options)

Return the tensor product of self with the crystals B.

EXAMPLES:
```
```
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

    extra_super_categories()
        Implement the fact that a (finite) Cartesian product of topological spaces is a topological space.

    EXAMPLES:

    sage: from sage.categories.topological_spaces import TopologicalSpaces
    sage: C = TopologicalSpaces().CartesianProducts()
    sage: C.extra_super_categories()
    [Category of topological spaces]
    sage: C.super_categories()
    [Category of Cartesian products of sets, Category of topological spaces]
    sage: C.axioms()
    frozenset()

class Compact(base_category)
    Bases: CategoryWithAxiom
    The category of compact topological spaces.

    EXAMPLES:

    sage: from sage.categories.topological_spaces import TopologicalSpaces
    sage: C = TopologicalSpaces().CartesianProducts()
    sage: C.extra_super_categories()
    [Category of topological spaces]
    sage: C.super_categories()
    [Category of Cartesian products of sets, Category of topological spaces]
    sage: C.axioms()
    frozenset()
```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
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
class SubcategoryMethods
    Bases: object

    Compact()

    Return the subcategory of the compact objects of self.

    EXAMPLES:
```
```
```
```
```python
Connected()

    Return the full subcategory of the connected objects of self.

    EXAMPLES:
```
```
```
```
```python
class sage.categories.topological_spaces.TopologicalSpacesCategory(category, *args)
    Bases: RegressiveCovariantConstructionCategory

4.157. Topological Spaces
4.158 Kac-Moody Algebras With Triangular Decomposition Basis

AUTHORS:

• Travis Scrimshaw (07-15-2017): Initial implementation

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.

class 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: L = LieAlgebra(QQ, cartan_type="F4")
˓→optional - sage.combinat sage.modules
sage: L.inject_variables()  #→
˓→optional - sage.combinat sage.modules
Defining e1, e2, e3, e4, f1, f2, f3, f4, h1, h2, h3, h4
sage: e1.part()  #→
˓→optional - sage.combinat sage.modules
1
sage: f4.part()  #→
˓→optional - sage.combinat sage.modules
-1
sage: (h2 + h3).part()  #→
˓→optional - sage.combinat sage.modules
0
sage: (f1.bracket(f2) + 4*f4).part()  #→
˓→optional - sage.combinat sage.modules
-1
sage: (e1 + f1).part()  #→
˓→optional - sage.combinat sage.modules
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)  # optional - sage.combinat sage.modules
sage: L.e()  # optional - sage.combinat sage.modules
Finite family {1: E[alpha[1]], 2: E[alpha[2]]}
sage: L.e(1)  # optional - sage.combinat sage.modules
E[alpha[1]]

\textbf{f}(i=\text{None})

Return the generators \(f\) of \(\text{self}\).

\textbf{INPUT:}
- \(i\) – (optional) if specified, return just the generator \(f_i\)

\textbf{EXAMPLES:}

sage: L = lie_algebras.so(QQ, 5)  # optional - sage.combinat sage.modules
sage: L.f()  # optional - sage.combinat sage.modules
Finite family {1: E[-alpha[1]], 2: E[-alpha[2]]}
sage: L.f(1)  # optional - sage.combinat sage.modules
E[-alpha[1]]

\textbf{verma_module}(\text{la}, \text{basis_key}=\text{None}, **\text{kwds})

Return the Verma module with highest weight \(\text{la}\) over \(\text{self}\).

\textbf{INPUT:}
- \(\text{basis_key}\) – (optional) a key function for the indexing set of the basis elements of \(\text{self}\)

\textbf{EXAMPLES:}

sage: L = lie_algebras.sl(QQ, 3)  # optional - sage.combinat sage.modules
sage: P = L.cartan_type().root_system().weight_lattice()  # optional - sage.combinat sage.modules
sage: La = P.fundamental_weights()  # optional - sage.combinat sage.modules
sage: M = L.verma_module(La[1] + La[2])  # optional - sage.combinat sage.modules
sage: M  # optional - sage.combinat sage.modules
Verma module with highest weight \(\text{Lambda}[1] + \text{Lambda}[2]\) of \(\text{Lie algebra of } ['A', 2]\) in the Chevalley basis

\textbf{super_categories}()

\textbf{EXAMPLES:}

sage: from sage.categories.triangular_kac_moody_algebras import TriangleKacMooodyAlgebras
sage: TriangleKacMooodyAlgebras(QQ).super_categories()
[Join of Category of graded lie algebras with basis over Rational Field and Category of kac moody algebras over Rational Field]

4.158. Kac-Moody Algebras With Triangular Decomposition Basis 843
4.159 Unique factorization domains

class sage.categories.unique_factorization_domains.UniqueFactorizationDomains(s=None)
    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:

    sage: UniqueFactorizationDomains()
    Category of unique factorization domains

    sage: UniqueFactorizationDomains().super_categories()
    [Category of gcd domains]

class ElementMethods
    Bases: object

    radical(*args, **kwds)
        Return the radical of this element, i.e. the product of its irreducible factors.

        This default implementation calls squarefree_decomposition if available, and factor otherwise.

        See also:

        squarefree_part()

    EXAMPLES:

    sage: Pol.<x> = QQ[]
    sage: (x^2*(x-1)^3).radical()
    x^2 - x

    sage: pol = 37 * (x-1)^3 * (x-2)^2 * (x-1/3)^7 * (x-3/7)
    sage: pol.radical()
    37*x^4 - 2923/21*x^3 + 1147/7*x^2 - 1517/21*x + 74/7

    sage: Integer(10).radical()
    10

    sage: Integer(-100).radical()
    10

    sage: 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 = 2^10000-1; n / radical(n, limit=100000)
    125

squarefree_part()
    Return the square-free part of this element, i.e. the product of its irreducible factors appearing with odd multiplicity.
This default implementation calls `squarefree_decomposition`.

See also:

`radical()`

**EXAMPLES:**

```python
sage: Pol.<x> = QQ[]
sage: (x^2*(x-1)^3).squarefree_part()
x - 1
sage: pol = 37 * (x-1)^3 * (x-2)^2 * (x-1/3)^7 * (x-3/7)
sage: pol.squarefree_part()
37*x^3 - 1369/21*x^2 + 703/21*x - 37/7
```

**class** `ParentMethods`

**Bases:** object

**is_unique_factorization_domain**(proof=True)

Return True, since this in an object of the category of unique factorization domains.

**EXAMPLES:**

```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.

**EXAMPLES:**

```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 gh{2}ub 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:

```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()
```

```python
sage: from sage.categories.unital_algebras import UnitalAlgebras
sage: C = UnitalAlgebras(ZZ); C
Category of unital algebras over Integer Ring
```

```python
class WithBasis(base_category)
Bases: CategoryWithAxiom_over_base_ring

class ParentMethods
Bases: object

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
```

```python
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.from_base_ring(1)
B[word: ]
```

```python
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.from_base_ring_from_one_basis(3)
3*B[word: ]
sage: A(3)
3*B[word: ]
```
**one()**

Return the multiplicative unit element.

**EXAMPLES:**

```python
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: A.one()  # optional - sage.combinat sage.modules
word: 
sage: A.one()  # optional - sage.combinat sage.modules
B[word: ]
```

**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:**

```python
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: A.one_basis()  # optional - sage.combinat sage.modules
word: 
sage: A.one()  # optional - sage.combinat sage.modules
B[word: ]
sage: A.from_base_ring(4)  # optional - sage.combinat sage.modules
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:**

```python
sage: A = AlgebrasWithBasis(QQ).example()  # optional - sage.combinat sage.modules
sage: A.one_from_one_basis()  # optional - sage.combinat sage.modules
word: 
sage: A.one()  # optional - sage.combinat sage.modules
B[word: ]
```

Even if called in the wrong order, they should returns their respective one:

```python
sage: Bone().parent() is B  # optional - sage.combinat sage.modules
```

(continues on next page)
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`

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]
```

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
```
Smooth()

Return the subcategory of the smooth objects of self.

EXAMPLES:

```
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
```

base_space()

Return the base space of this category.

EXAMPLES:

```
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
```

super_categories()

EXAMPLES:

```
sage: M = Manifold(2, 'M')
sage: from sage.categories.vector_bundles import VectorBundles
sage: 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]
```

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

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]

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

class Graded(base_category)
    Bases: GradedModulesCategory
    Category of graded vector spaces.
class ParentMethods
Bases: object
dimension()

Return the dimension of this vector space.

EXAMPLES:

```
sage: M = FreeModule(FiniteField(19), 100)  # optional - sage.modules sage.rings.finite_rings
    # W = M.submodule([M.gen(50)])  # optional - sage.modules sage.rings.finite_rings
sage: W.dimension()  # optional - sage.modules sage.rings.finite_rings
1
```

```
sage: M = FiniteRankFreeModule(QQ, 3)  # optional - sage.modules
sage: M.dimension()  # optional - sage.modules
3
sage: M.tensor_module(1, 2).dimension()  # optional - sage.modules
27
```

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
```

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:

```
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.

example(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()  # optional - 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 Rational Field]
sage: VectorSpaces(QQ).WithBasis().FiniteDimensional().TensorProducts().FiniteDimensional()
Category of tensor products of finite dimensional vector spaces with basis over Rational Field
```

class Graded(base_category)

Bases: GradedModulesCategory

Category of graded vector spaces with basis.

example(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()  # optional - 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
```
Category Framework, Release 10.1

Field

```
sage: C in C.TensorProducts().super_categories()
True
```

**is_abelian()**

Return whether this category is abelian.

This is always True since the base ring is a field.

EXAMPLES:

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

**base_field()**

Returns the base field over which the vector spaces of this category are all defined.

EXAMPLES:

```
sage: VectorSpaces(QQ).base_field()
Rational Field
```

**super_categories()**

EXAMPLES:

```
sage: VectorSpaces(QQ).super_categories()
[Category of modules over Rational Field]```
4.163 Weyl Groups

class sage.categories.weyl_groups.WeylGroups(s=None)
    Bases: Category_singleton
    The category of Weyl groups
    See the Wikipedia page of Weyl Groups.

    EXAMPLES:
    sage: WeylGroups()
    Category of weyl groups
    sage: WeylGroups().super_categories()
    [Category of coxeter groups]

    Here are some examples:
    sage: WeylGroups().example()  # todo: not implemented
    sage: FiniteWeylGroups().example()
    The symmetric group on {0, ..., 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)

    This one will eventually be also in this category:
    sage: SymmetricGroup(4)
    Symmetric group of order 4! as a permutation group

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 self given as roots.

The side parameter determines on which side to compute the inversions.

EXAMPLES:

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)

The identity element gives the empty arrangement:

sage: W = WeylGroup(['A',3])
sage: W.one().inversion_arrangement()
Empty hyperplane arrangement of dimension 3

inversions(side='right', inversion_type='reflections')

Return the set of inversions of self.

INPUT:
• side – ‘right’ (default) or ‘left’
• inversion_type – ‘reflections’ (default), ‘roots’, or ‘coroots’

OUTPUT:
For reflections, the set of reflections r in the Weyl group such that self r < self. For (co)roots, the set of positive (co)roots that are sent by self to negative (co)roots; their associated reflections are described above.

If 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]]

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 = W.from_reduced_word([3,2,5]).is_pieri_factor()
True
sage: w = W.from_reduced_word([3,2,4,5]).is_pieri_factor()
False

sage: W = WeylGroup(['C',4,1])
sage: w = W.from_reduced_word([0,2,1]).is_pieri_factor()
True
sage: w = W.from_reduced_word([0,2,1,0]).is_pieri_factor()
False

sage: W = WeylGroup(['B',3])
sage: w = W.from_reduced_word([2,3]).is_pieri_factor()
False
sage: w = W.from_reduced_word([2,1,2]).is_pieri_factor()
True

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:

```python
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, 1, 0]],
([0], [3, 2, 1, 0]),
([3], [2, 0, 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
```

The `quantum_bruhat_successors` function returns 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 \((w, \alpha)\) where \(w\) is a successor and \(\alpha\) is the positive root associated with the successor relation
- **quantum_only** – (default: False) if True, returns only the quantum successors

**EXAMPLES:**

```python
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()
[1, s2*s3]
sage: w.quantum_bruhat_successors(quantum_only = True)
[s3*s1]
sage: w = W.from_reduced_word([2,3])
sage: w.quantum_bruhat_successors([1,3])
Traceback (most recent call last):
  ...
ValueError: s2*s3 is not of minimum length in its coset of the parabolic subgroup generated by the reflections (1, 3)
```

**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:**

```python
```
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()
8*m[1, 1, 1, 1, 1, 1, 1] + 4*m[2, 1, 1, 1, 1, 1] + 2*m[2, 2, 2, 1, 1] + m[2, 2, 2, 2, 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, 2, 1, 1] + m[2, 2, 2, 2, 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 = AffinePermutationGroup(['A',4,1])
sage: a = A([-2,0,1,4,12])
sage: a.stanley_symmetric_function()
6*m[1, 1, 1, 1, 1, 1, 1, 1, 1] + 5*m[2, 1, 1, 1, 1, 1, 1, 1, 1]
```
One more example (github issue #14095):

```python
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_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[i].stanley_symm_poly_weight()`.

EXAMPLES:

```python
sage: W = WeylGroup(['A', 3, 1])
sage: W.from_reduced_word([]).stanley_symmetric_function_as_polynomial()
1
sage: W.from_reduced_word([1]).stanley_symmetric_function_as_polynomial()
x1
sage: W.from_reduced_word([1,2]).stanley_symmetric_function_as_polynomial()
x1^2
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
```
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 $x \prec a$ for all atoms $a$ 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:**

```
sage: W = WeylGroup(["A",2])
sage: x = W.from_reduced_word([1])
sage: y = W.w0
sage: W.bruhat_cone(x, y)
A 2-dimensional polyhedron in QQ^3
defined as the convex hull of 1 vertex and 2 rays

sage: W = WeylGroup(["E",6])
```
```python
sage: x = W.one()
sage: y = W.w0
sage: 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]
```

```python
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
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
```
```python
sage: W = WeylGroup(['B',3])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF])
[]
[1],
[1, 2],
[1, 2, 1],
[1, 2, 3],
[1, 2, 3, 1],
[1, 2, 3, 2],
[1, 2, 3, 2, 1],
[2],
[2, 1],
```

(continues on next page)
quantum_bruhat_graph(index_set=())

Return the quantum Bruhat graph of the quotient of the Weyl group by a parabolic subgroup \( W_J \).

**INPUT:**
- **index_set** – (default: \((\))\) a tuple \( J \) of nodes of the Dynkin diagram

By default, the value for **index_set** indicates that the subgroup is trivial and the quotient is the full Weyl group.

**EXAMPLES:**

```python
sage: W = WeylGroup(['C',4,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([3,2,0]) in PF
True
```

### additional_structure()

Return None.

Indeed, the category of Weyl groups defines no additional structure: Weyl groups are a special class of Coxeter groups.
See also:

Category.additionastructure()  

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

```
sage: WeylGroups().additionastructure()
```

super_categories()  

EXAMPLES:

```
sage: 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 github issue #13801 is corrected:

```
sage: class A(Parent):
....:    def __init__(self):
....:        Parent.__init__(self, category=Sets(), facade=True)
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
```
**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:

```python
sage: -1 in S, S.is_parent_of(-1)
(False, True)
```

Furthermore, there is no coercion attempted:

```python
sage: int(1) in S, S.is_parent_of(int(1))
(True, False)
```

**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
```
5.1 Covariant Functorial Constructions

A functorial construction is a collection of functors \((F_{\text{Cat}})_{\text{Cat}}\) (indexed by a collection of categories) which associate to a sequence of parents \((A, B, ...)\) in a category \(\text{Cat}\) a parent \(F_{\text{Cat}}(A, B, ...)\). Typical examples of functorial constructions are \text{cartesian_product} and \text{tensor_product}.

The category of \(F_{\text{Cat}}(A, B, ...)\), which only depends on \(\text{Cat}\), is called the (functorial) construction category.

A functorial construction is \((\text{category})\)-covariant if for every categories \(\text{Cat}\) and \(\text{SuperCat}\), the category of \(F_{\text{Cat}}(A, B, ...)\) is a subcategory of the category of \(F_{\text{SuperCat}}(A, B, ...)\) whenever \(\text{Cat}\) is a subcategory of \(\text{SuperCat}\).

A functorial construction is \((\text{category})\)-recessive if the category of \(F_{\text{Cat}}(A, B, ...)\) is a subcategory of \(\text{Cat}\).

The goal of this module is to provide generic support for covariant functorial constructions. In particular, given some parents \(A, B, ...\), in respective categories \(\text{Cat}_A, \text{Cat}_B, ...\), it provides tools for calculating the best known category for the parent \(F(A, B, ...)\). 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 \text{CovariantFunctorialConstruction}, \text{CovariantConstructionCategory} and \text{RegressiveCovariantConstructionCategory} for more details.
classmethod default_super_categories(category, *args)

Return the default super categories of $F_{Cat}(A, B, \ldots)$ for $A, B, \ldots$ parents in $Cat$.

INPUT:

- cls – the category class for the functor $F$
- category – a category $Cat$
- *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:

```python
sage: 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:

```python
sage: 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:

```python
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 $Q$:

```python
sage: FiniteMonoids().super_categories()
[Category of monoids, Category of finite semigroups]
```

```python
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スーパー_categories()` was called internally:
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:

```python
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:

```python
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'
```

class sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction

An abstract class for construction functors $F$ (eg $F =$ Cartesian product, tensor product, $Q$-algebra, ...) such that:

- Each category $Cat$ (eg $Cat =$ Groups()) can provide a category $F_{Cat}$ for parents constructed via this functor (e.g. $F_{Cat} =$ CartesianProductsOf(Groups())).
- For every category $Cat$, $F_{Cat}$ is a subcategory of $F_{SuperCat}$ for every super category $SuperCat$ of $Cat$ (the functorial construction is (category)-covariant).
- For parents $A, B, ...$, respectively in the categories $Cat_A, Cat_B, ...$, the category of $F(A, B, ...)$ is $F_{Cat}$ where $Cat$ is the meet of the categories $Cat_A, Cat_B, ...$.

This covers two slightly different use cases:

- In the first use case, one uses directly the construction functor to create new parents:

  ```python
  sage: tensor()  # todo: not implemented (add an example)
  ```

  or even new elements, which indirectly constructs the corresponding parent:
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, `CartesianProductsOf(Groups())` is set automatically as a subcategory of `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` - an 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 supports 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.

```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
```

The function `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: cartesian_product.category_from_categories((Cat1, Cat1))
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:

```
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:

```
sage: E = CombinatorialFreeModule(QQ, ["a", "b", "c"])
# optional - sage.modules
sage: tensor.category_from_parents((E, E, E))
# optional - sage.modules
Category of tensor products of finite dimensional vector spaces with basis over Rational Field
```

class sage.categories.covariant_functorial_construction.FunctorialConstructionCategory(category, *args)

Bases: Category

Abstract class for categories $F_{Cat}$ obtained through a functorial construction

base_category()

Return the base category of the category self.

For any category $B = F_{Cat}$ obtained through a functorial construction $F$, the call $B.base_category()$ returns the category $Cat$.

EXAMPLES:

```
sage: Semigroups().Quotients().base_category()
Category of semigroups
```

category_of = <bound method category_of of <class 'sage.categories.covariant_functorial_construction.FunctorialConstructionCategory'>>

eextra_super_categories()

Return the extra super categories of a construction category.

Default implementation which returns [].

EXAMPLES:
super_categories()

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]
```

class sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory(category, *args)

Bases: `CovariantConstructionCategory`

Abstract class for categories $F\mathcal{C}at$ obtained through a regressive covariant functorial construction

classmethod default_super_categories(category, *args)

Return the default super categories of $F\mathcal{C}at(A, B, \ldots)$ for $A, B, \ldots$ parents in $\mathcal{C}at$.

INPUT:

- `cls` – the category class for the functor $F$
- `category` – a category $\mathcal{C}at$
- `*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 refactorization

class sage.categories.cartesian_product.CartesianProductFunctor(category=None)

Bases: `CovariantFunctorialConstruction`, `MultivariateConstructionFunctor`

The Cartesian product functor.

EXAMPLES:
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
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)
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
```

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):

```python
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 unital magmas, Category of magmas, Category of Cartesian products of sets, Category of sets, ...]
```

(continues on next page)
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`.

```python
def symbol():
    return '+'
```

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:**

    ```python
    sage: 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 $\text{Algebras}(\mathbb{Q})$, it is automatically endowed with its natural algebra structure, thanks to the category $\text{Algebras}(\mathbb{Q}).\text{TensorProducts}()$ 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 $A.\text{TensorProducts}()$ is a subcategory of $B.\text{TensorProducts}()$ (see also $\text{CovariantFunctorialConstruction}$). Hence, the role of $\text{Algebras}(\mathbb{Q}).\text{TensorProducts}()$ is solely to provide mathematical information and algorithms which are relevant to tensor product of algebras.

Those are implemented in the nested class $\text{TensorProducts}$ of $\text{Algebras}(\mathbb{Q})$. This nested class is itself a subclass of $\text{TensorProductsCategory}$.

```python
symbol = ' # '  
unicode_symbol = ' ⊗ '
```

**class** `sage.categories.tensor.TensorProductsCategory(category, *args)`

Bases: `CovariantConstructionCategory`

An abstract base class for all $\text{TensorProducts}$'s categories

**TensorProducts()**

Returns the category of tensor products of objects of $\text{self}$

By associativity of tensor products, this is $\text{self}$ (a tensor product of tensor products of $\text{Cat}$'s is a tensor product of $\text{Cat}$'s)

**base()**

The base of a tensor product is the base (usually a ring) of the underlying category.

**sage: tensor**

The tensor functorial construction

See $\text{TensorProductFunctor}$ for more information

**EXAMPLES:**

```
sage: ModulesWithBasis(QQ).TensorProducts().TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
```

```
sage: ModulesWithBasis(ZZ).TensorProducts().base()
Integer Ring
```

```
sage: tensor
The tensor functorial construction
```

5.3. Tensor Product 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

symbol = '
unicode_symbol = '
```

```python
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:
        sage: 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:
        sage: 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 = '^\ast'
```

```python
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
```

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)
```

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:

```python
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 github 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')
sage: A = S.algebra(QQ); A
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
over Rational Field
sage: 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')
sage: 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

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
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: 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
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:

```python
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,
 ...]
```

### 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
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

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
```
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:

```sage
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: $\Delta(g) = g \otimes g$:

```sage
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:

```sage
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 $\chi(g) = g^{-1}$:

```sage
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
sage: SymmetricGroup(5).algebra(QQ) in Algebras(QQ).Semisimple()
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: A = DihedralGroup(2).algebra(ZZ)
sage: B = SymmetricGroup(4).algebra(QQ)
sage: a = A.an_element(); a
() + 2*(3,4) + 3*(1,2) + (1,2)(3,4)
sage: b = B.an_element(); b
() + (2,3,4) + 2*(1,3)(2,4) + 3*(1,4)(2,3)
sage: B(a)
() + 2*(3,4) + 3*(1,2) + (1,2)(3,4)
sage: a * b  # a is automatically converted to an element of B
() + 2*(3,4) + 2*(2,3) + (2,3,4) + 3*(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)
sage: 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)
```

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*(())
```

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)
```

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:
class sage.categories.algebra_functor.AlgebraFunctor(base_ring)
Bases: CovariantFunctorialConstruction
For a fixed ring, a functor sending a group/... to the corresponding group/... algebra.

EXAMPLES:

```python
sage: from sage.categories.algebra_functor import AlgebraFunctor
sage: F = AlgebraFunctor(QQ); F
The algebra functorial construction
sage: F(DihedralGroup(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:

```python
sage: from sage.categories.algebra_functor import AlgebraFunctor
sage: AlgebraFunctor(QQ).base_ring()
Rational Field
```

class sage.categories.algebra_functor.AlgebrasCategory(category, *args)
Bases: CovariantConstructionCategory, Category_over_base_ring
An abstract base class for categories of monoid algebras, groups algebras, and the like.

See also:

- `Sets.ParentMethods.algebra()`
- `Sets.SubcategoryMethods.Algebras()`
- `CovariantFunctorialConstruction`

INPUT:

- `base_ring` – a ring

EXAMPLES:

```python
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'
```
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
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
[1, 1, 1, 1]
sage: A.coproduct_on_basis(g)
B[[1, 1, 1, 1]] # B[[1, 1, 1, 1]]
sage: a = A.an_element(); a
2*B[[1, 1, 1, 1]] + 2*B[[1, 1, 1, 2]] + 3*B[[1, 1, 1, 3]]
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]]
```

class sage.categories.algebra_functor.GroupAlgebraFunctor\(\text{\texttt{\textit{group}}}\)

Bases: ConstructionFunctor

For a fixed group, a functor sending a commutative ring to the corresponding group algebra.

INPUT:

- \text{\texttt{\textit{group}}} – the group associated to each group algebra under consideration

EXAMPLES:

```sage
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 Field

group()

Return the group which is associated to this functor.

EXAMPLES:

```sage
sage: from sage.categories.algebra_functor import GroupAlgebraFunctor
class sage.categories.algebra_functor.GroupAlgebraFunctor\(\text{\texttt{\textit{group}}}\)
```
5.7 Subquotient Functorial Construction

AUTHORS:

• Nicolas M. Thiery (2010): initial revision

class sage.categories.subquotients.SubquotientsCategory(category, *args)
    Bases: RegressiveCovariantConstructionCategory

5.8 Quotients Functorial Construction

AUTHORS:

• Nicolas M. Thiery (2010): initial revision

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 \( \text{Cat} \)
    OUTPUT: a (join) category

In practice, this returns \( \text{category.Subquotients()} \), joined together with the result of the method
\( \text{RegressiveCovariantConstructionCategory.default_super_categories()} \) (that is the join of
\( \text{category and cat.Quotients()} \) for each \( \text{cat} \) in the super categories of \( \text{category} \)).

EXAMPLES:

Consider \( \text{category=Groups()} \), which has \( \text{cat=Monoids()} \) 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 \( G \):

\[
\text{sage: Groups().Quotients().super_categories()}
\]

\[
[\text{Category of groups, Category of subquotients of monoids, Category of quotients of semigroups}]
\]

Mind the last item above: there is indeed currently nothing implemented about quotient monoids.

This resulted from the following call:

\[
\text{sage: sage.categories.quotients.QuotientsCategory.default_super_categories(Groups())}
\]

Join of Category of groups and Category of subquotients of monoids and Category of quotients of semigroups
5.9 Subobjects Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

```python
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:
    • cls – the class SubobjectsCategory
    • 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.Subobjects() 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 submonoid of \( G \):

```
sage: Groups().Subobjects().super_categories()
[Category of groups, Category of subquotients of monoids, Category of...
→subobjects of sets]
```

Mind the last item above: there is indeed currently nothing implemented about submonoids.

This resulted from the following call:

```
sage: sage.categories.subobjects.SubobjectsCategory.default_super_categories(Groups())
Join of Category of groups and Category of subquotients of monoids and 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 group \( G' \) by a group isomorphism is simultaneously a subgroup of \( G \), a subquotient of \( G \), a group by itself, and the image of \( G \) by a monoid isomorphism:

```
sage: Groups().IsomorphicObjects().super_categories()
[Category of groups,
 Category of subquotients of monoids,
 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).
github 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
class Endset(base_category):
    Bases: CategoryWithAxiom

    The category of all endomorphism sets.

    This category serves too purposes: making sure that the Endset axiom is implemented in the category where it's defined, namely `Homsets`, and specifying that `Endsets` are monoids.

    EXAMPLES:

    ```python
    sage: from sage.categories.homsets import Homsets
    sage: Homsets().Endset()
    Category of endsets
    ```

    class ParentMethods
        Bases: object

        is_endomorphism_set()
            Return True as self is in the category of Endsets.

            EXAMPLES:

            ```python
            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:
```
class SubcategoryMethods

    Bases: object

    Endset()

        Return the subcategory of the homsets of self that are endomorphism sets.

        EXAMPLES:

        sage: Sets().Homsets().Endset()
       Category of endsets of sets
        sage: Posets().Homsets().Endset()
       Category of endsets of posets

super_categories()

    Return the super categories of self.

    EXAMPLES:

    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:

        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 $Cat$

    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:

```python
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:

```python
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:

```python
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
sage: AdditiveMagmas.Homsets
<class 'sage.categories.additive_magmas.AdditiveMagmas.Homsets'>
sage: HomsetsCategory.default_super_categories(AdditiveMagmas())
Category of homsets
```

which gives one of the super categories of `category.Homsets()`:

```python
sage: AdditiveMagmas().Homsets().super_categories()
[Category of additive magmas, Category of homsets]
sage: AdditiveMagmas().AdditiveUnital().Homsets().super_categories()
[Category of additive unital additive magmas, Category of homsets]
```

the other coming from `category.Homsets().extra_super_categories()`:

```python
sage: 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
```

(continues on next page)
sage: H = HomsetsCategory.default_super_categories(Posets()); H
Category of homsets of posets
sage: type(H)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
sage: Posets().Homsets()
Category of homsets of posets

class sage.categories.homsets.HomsetsOf(category, *args)
    Bases: HomsetsCategory

    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'>
    sage: C.super_categories()
    [Category of homsets]

5.12 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.
• \texttt{sage.categories.examples.with_realizations} for an example.

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

    INPUT:
    • parent_with_realization – a parent
See also:

```
Sets().WithRealizations
```

**EXAMPLES:**

```
sage: A = Sets().WithRealizations().example(); A
#optional - sage.combinat 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
#optional - sage.combinat 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
```

The category of realizations of a given algebra:

```
sage: A = Sets().WithRealizations().example(); A
#optional - sage.combinat sage.modules
The subset algebra of \{1, 2, 3\} over Rational Field
sage: A.Realizations()
#optional - sage.combinat 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()
```

---

5.12. Realizations Covariant Functorial Construction 891
See also:

- ` Sets().WithRealizations`
- `ClasscallMetaclass`

**Todo:** Add an optional argument to allow for:

```python
sage: Realizations(A, category=Blahs()) # todo: not implemented
```

```python
class sage.categories.realizations.RealizationsCategory(category, *args)

Bases: RegressiveCovariantConstructionCategory

An abstract base class for all categories of realizations category

Relization are implemented as `RegressiveCovariantConstructionCategory`. See there for the document-
tation of how the various bindings such as `Sets().Realizations()` and `P.Realizations()`, where P is a
parent, work.

See also:

  `Sets().WithRealizations`
```

## 5.13 With 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 con-
structions.

```python
sage.categories.with_realizations.WithRealizations(self)

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 real-
izations.
- 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:

```
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 SetsFacade) admitting multiple concrete realizations where its elements are represented. Consider for example an algebra $A$ which admits several natural bases:

```
sage: A = Sets().WithRealizations().example(); A
# optional - sage.combinat sage.modules
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 $B$:

```
sage: A.F()
# optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: A.Out()
# optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: A.In()
# optional - sage.combinat sage.modules
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: A.an_element()
# optional - sage.combinat sage.modules
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 $P_{B'}$:

```
sage: F = A.F(); In = A.In(); Out = A.Out()
# optional - sage.combinat sage.modules
sage: i = In.an_element(); i
# optional - sage.combinat sage.modules
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
sage: F(i)
# optional - sage.combinat sage.modules
```

(continues on next page)
\[ 7 \text{F}\{\}\} + 3 \text{F}\{1\} + 4 \text{F}\{2\} + \text{F}\{1, 2\} \]

\textbf{sage:} \text{F.coerce_map_from(Out)}  
\textit{\#optional - sage.combinat sage.modules}

\text{Generic morphism:}
\begin{itemize}
  \item From: The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis
  \item To: The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
\end{itemize}

allowing for mixed arithmetic:

\textbf{sage:} \( (1 + \text{Out.from_set(1)}) \ast \text{In.from_set(2,3)} \)  
\textit{\#optional - sage.combinat sage.modules}

\text{Out}\{\}\} + 2\text{Out}\{1\} + 2\text{Out}\{2\} + 2\text{Out}\{3\} + 2\text{Out}\{1, 2\}  
+ 2\text{Out}\{1, 3\} + 4\text{Out}\{2, 3\} + 4\text{Out}\{1, 2, 3\} 

In our example, there are three realizations:

\textbf{sage:} \text{A.realizations()}  
\textit{\#optional - sage.combinat 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, In, and Out, as above one can just do:

\textbf{sage:} \text{A.inject_shorthands()}  
\textit{\#optional - sage.combinat sage.modules}

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

\textbf{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 \( A \):

\textbf{sage:} \text{P = A['x']; P}  
\textit{\#optional - sage.combinat sage.modules}

Univariate Polynomial Ring in x over  
The subset algebra of \{1, 2, 3\} over Rational Field  
\textbf{sage:} \text{x = P.gen()}  
\textit{\#optional - sage.combinat sage.modules}
In the following examples, the coefficients turn out to be all represented in the $F$ basis:

```python
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:

```python
sage: p = P([1, In[1]], Out[2])); p
Out[2]*x^2 + In[1]*x + F[]
sage: p^2
  - 4*In[1, 2] - 2*In[1, 3] - 4*In[2, 3])\*x^3
  + (2*F[]) + 2*F[1])\*x
  + F[]
```

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)
  - 2*In[1, 2] - In[1, 3] - 2*In[2, 3] + In[1, 2, 3])\*x^2
  + In[1]\*x + In[]
```

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

```python
sage: In[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'
```
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()
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()
True
```

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

```python
sage: A in Algebras(QQ).WithRealizations()
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 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()
True
```

and in the category of algebras (regressiveness):

```python
sage: A in Algebras(QQ)
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 regressive construction. That is `self.WithRealizations()` is a subcategory of `self` by default:
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.

The role of this base class is to implement some technical goodies, such as the name for that category.

```
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]
```
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:
        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.algebra_generators()
        Family (B[a], B[b], B[c])

    one_basis()
        Returns the empty word, which index the one of this algebra, as per AlgebrasWithBasis.ParentMethods.one_basis().
        EXAMPLES:
        sage: A = AlgebrasWithBasis(QQ).example()
        sage: A.one()
        B[]

    product_on_basis(w1, w2)
        Product of basis elements, as per AlgebrasWithBasis.ParentMethods.product_on_basis().
        EXAMPLES:
        sage: A = AlgebrasWithBasis(QQ).example()
        sage: Words = A.basis().keys()
        sage: A.product_on_basis(Words("acb"), Words("cba"))
        B[acbba]
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(alphabet=('a', 'b', 'c', 'd'))

Bases: FreeCommutativeAdditiveSemigroup

An example of a commutative additive monoid: the free commutative monoid
This class illustrates a minimal implementation of a commutative monoid.

EXAMPLES:

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

This is the free semigroup generated by:

sage: S.additive_semigroup_generators()
Family (a, b, c, d)

with product rule given by $a \times b = a$ for all $a, b$:

sage: (a,b,c,d) = S.additive_semigroup_generators()

We conclude by running systematic tests on this commutative monoid:

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_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . 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

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(alphabet=(
    'b',
    'c',
    'd'))

    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:

    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:
with product rule given by $a \times b = a$ for all $a, b$:

```
sage: (a,b,c,d) = S.additive_semigroup_generators()
```

We conclude by running systematic tests on this commutative monoid:

```
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
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
```

```python
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')
```

Internally, elements are represented as dense dictionaries which associate to each generator of the monoid its multiplicity. In order to get an element, we wrap the dictionary into an element via `ElementWrapper`:

```
sage: x.value
{'a': 2, 'b': 0, 'c': 1, 'd': 5}
```
additive_semigroup_generators()

Returns the generators of the semigroup.

EXAMPLES:

```
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.additive_semigroup_generators()
Family (a, b, c, d)
```

an_element()

Returns an element of the semigroup.

EXAMPLES:

```
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 CommutativeAdditiveSemigroups.
ParentMethods.summation().

EXAMPLES:

```
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\}:
The crystal operators themselves correspond to the elementary transpositions:

```python
sage: b = C.module_generators[0]
sage: b.f(1)
2
sage: b.f(1).e(1) == b
True
```

Only the following basic operations are implemented:

- `cartan_type()` or an attribute `_cartan_type`
- An attribute `module_generators`
- `Element.e()`
- `Element.f()`

All the other usual crystal operations are inherited from the categories; for example:

```python
sage: C.cardinality()
4
```

### class `Element`

```
Bases: `ElementWrapper`

```

### e(i)
Returns the action of $e_i$ on `self`.

**EXAMPLES:**

```python
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:**

```python
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]]
```

### 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:

```python
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:

```python
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 [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:

```python
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 [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

```python
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:

```python
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:

```python
sage: TestSuite(X).run()
```
class **Element**(parent, dim, name)

Bases: **Element**

A cell in a CW complex.

dimension()

Return the dimension of self.

EXAMPLES:

```python
 sage: from sage.categories.cw_complexes import CWComplexes
 sage: X = CWComplexes().example()
 sage: f = X.an_element()
 sage: 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 $+ - \infty$

This class illustrates a minimal implementation of a facade parent that models the union of several other parents.

EXAMPLES:

```python
 sage: S = Sets().Facade().example("union"); S
An example of a facade set: the integers completed by $+\infty$
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

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()

This group is generated by two simple reflections $s_1$ and $s_2$ subject to the relation $(s_1s_2)^n = 1$:

sage: G.simple_reflections()
Finite family {1: (1,), 2: (2,)}

sage: s1, s2 = G.simple_reflections()
sage: (s1*s2)^5 == G.one()
True

An element is represented by its reduced word (a tuple of elements of self.index_set()):

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, 1, 2, 1),
 (1, 2, 1, 2, 1)]

This reduced word is unique, except for the longest element where the chosen reduced word is $(1, 2, 1, 2 \ldots)$:
class Element
    Bases: ElementWrapper

    apply_simple_reflection_right(i)
    Implements CoxeterGroups.ElementMethods.apply_simple_reflection().

    EXAMPLES:
    sage: D5 = FiniteCoxeterGroups().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:
    sage: D6 = FiniteCoxeterGroups().example(6)
    sage: s = D6.simple_reflections()
    sage: s[1].has_descent(1)
    True
    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
    sage: D6.long_element().has_descent(1)
    True
    sage: D6.long_element().has_descent(2)
    True

    wrapped_class
    alias of tuple

coxeter_matrix()
    Return the Coxeter matrix of self.

    EXAMPLES:
    sage: FiniteCoxeterGroups().example(6).coxeter_matrix()
    [1 6]
    [6 1]

degrees()
    Return the degrees of self.

    EXAMPLES:
sage: FiniteCoxeterGroups().example(6).degrees()
(2, 6)

index_set()

Implements \texttt{CoxeterGroups.ParentMethods.index\_set()}.

EXAMPLES:

sage: D4 = FiniteCoxeterGroups().example(4)
sage: D4.index_set()
(1, 2)

one()

Implements \texttt{Monoids.ParentMethods.one()}.

EXAMPLES:

sage: D6 = FiniteCoxeterGroups().example(6)
sage: D6.one()

sage.categories.examples.finite_coxeter_groups.Example
alias of \texttt{DihedralGroup}

### 6.9 Example of a finite dimensional algebra with basis

sage.categories.examples.finite_dimensional_algebras_with_basis.Example
alias of \texttt{KroneckerQuiverPathAlgebra}

class sage.categories.examples.finite_dimensional_algebras_with_basis.KroneckerQuiverPathAlgebra(base_ring)
Bases: CombinatorialFreeModule

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 \texttt{sage.quivers.algebra.PathAlgebra} for a full-featured implementation of path algebras.

algebra\_generators()

Return algebra generators for this algebra.

See also:

\texttt{Algebras.ParentMethods.algebra\_generators()}.

EXAMPLES:

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 \(a:x\rightarrow y\) and \(b:x\rightarrow y\) over \texttt{Rational Field})
sage: A.algebra\_generators()
Finite family \{'x': x, 'y': y, 'a': a, 'b': b\}
one()

Return the unit of this algebra.

See also:

AlgebrasWithBasis.ParentMethods.one_basis()

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
sage: 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

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
- \( \text{ambient} \) – (optional) a Lie algebra; if this is set, then the resulting Lie algebra is declared a Lie subalgebra of \( \text{ambient} \)

**OUTPUT:**

The abelian Lie algebra on \( M \).

**class** **Element**

**Bases:** **Element**

**lift()**

Return the lift of \( \text{self} \) to the universal enveloping algebra.

**EXAMPLES:**

```python
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()
2*b0 + 2*b1 + 3*b2
```

**monomial_coefficients**(\( \text{copy=\text{True}} \))

Return the monomial coefficients of \( \text{self} \).

**EXAMPLES:**

```python
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}
```

**to_vector**(\( \text{order=\text{None}} \), \( \text{sparse=\text{False}} \))

Return \( \text{self} \) as a vector in \( \text{self.parent().module()} \).

See the docstring of the latter method for the meaning of this.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
```

(continues on next page)
ambient()

Return the ambient Lie algebra of self.

EXAMPLES:

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> a, b, c = L.lie_algebra_generators()
>>> S = L.subalgebra([2*a+b, b + c])
>>> S.ambient() == L
True

basis()

Return the basis of self.

EXAMPLES:

>>> 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:

>>> 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:

>>> L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
>>> u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
>>> parent(u) is L
True

gens()

Return the generators of self.

EXAMPLES:
\begin{align*}
\texttt{sage: } & L = \text{LieAlgebras}(\mathbb{Q}).\text{FiniteDimensional()}.\text{WithBasis()}.\text{example()} \\
\texttt{sage: } & L.\text{gens()}
\end{align*}

\begin{align*}
(1, 0, 0), (0, 1, 0), (0, 0, 1)
\end{align*}

**ideal** (*gens*)

Return the Lie subalgebra of \texttt{self} generated by the elements of the iterable \texttt{gens}.

This currently requires the ground ring \( R \) to be a field.

**EXAMPLES:**

\begin{align*}
\texttt{sage: } & L = \text{LieAlgebras}(\mathbb{Q}).\text{FiniteDimensional()}.\text{WithBasis()}.\text{example()} \\
\texttt{sage: } & a, b, c = L.\text{lie_algebra_generators()} \\
\texttt{sage: } & L.\text{subalgebra}([2^*a+b, b + c]) \\
\text{An example of a finite dimensional Lie algebra with basis:} \\
\text{the 2-dimensional abelian Lie algebra over Rational Field with} \\
\text{basis matrix:} \\
\begin{bmatrix}
1 & 0 & -1/2 \\
0 & 1 & 1
\end{bmatrix}
\end{align*}

**is_ideal** (*A*)

Return if \texttt{self} is an ideal of the ambient space \texttt{A}.

**EXAMPLES:**

\begin{align*}
\texttt{sage: } & L = \text{LieAlgebras}(\mathbb{Q}).\text{FiniteDimensional()}.\text{WithBasis()}.\text{example()} \\
\texttt{sage: } & a, b, c = L.\text{lie_algebra_generators()} \\
\texttt{sage: } & L.\text{is_ideal}(L) \\
\texttt{True} \\
\texttt{sage: } & S1 = L.\text{subalgebra}([2^*a+b, b + c]) \\
\texttt{sage: } & S1.\text{is_ideal}(L) \\
\texttt{True} \\
\texttt{sage: } & S2 = L.\text{subalgebra}([2^*a+b]) \\
\texttt{sage: } & S2.\text{is_ideal}(S1) \\
\texttt{True} \\
\texttt{sage: } & S1.\text{is_ideal}(S2) \\
\texttt{False}
\end{align*}

**lie_algebra_generators**()

Return the basis of \texttt{self}.

**EXAMPLES:**

\begin{align*}
\texttt{sage: } & L = \text{LieAlgebras}(\mathbb{Q}).\text{FiniteDimensional()}.\text{WithBasis()}.\text{example()} \\
\texttt{sage: } & L.\text{basis()}
\end{align*}

Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}

**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}.

**EXAMPLES:**
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]

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]

zero()

Return the zero element.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.zero()
(0, 0, 0)

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 FiniteEnumeratedSet for a full featured implementation.

EXAMPLES:
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:

sage: TestSuite(C).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
  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

class sage.categories.examples.finiteEnumeratedSets.IsomorphicObjectOfFiniteEnumeratedSet(ambient=An example of a finite enumerated set: \{1, 2, 3\})

Bases: UniqueRepresentation, Parent

6.11. Examples of finite enumerated sets
ambient()
Returns the ambient space for self, as per \texttt{Sets.Subquotients.ParentMethods.ambient()}.

EXAMPLES:

\begin{verbatim}
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\}
\end{verbatim}

lift(x)
INPUT:
• x – an element of self
Lifts x to the ambient space for self, as per \texttt{Sets.Subquotients.ParentMethods.lift()}.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

retract(x)
INPUT:
• x – an element of the ambient space for self
Retracts x from the ambient space to self, as per \texttt{Sets.Subquotients.ParentMethods.retract()}.

EXAMPLES:

\begin{verbatim}
sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: \{1,2,3\}
sage: C.retract(3)
9
\end{verbatim}

6.12 Examples of finite monoids

sage.categories.examples.finite_monoids.Example
alias of \texttt{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
\end{verbatim}
We conclude by running systematic tests on this monoid:

```python
sage: TestSuite(S).run(verboses = 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
```

class Element

   Bases: ElementWrapper

   wrapped_class
      alias of Integer

an_element()

   Returns an element of the monoid, as per Sets.ParentMethods.an_element().

   EXAMPLES:

```python
sage: M = FiniteMonoids().example()
sage: M.an_element()
6
```

one()

   Return the one of the monoid, as per Monoids.ParentMethods.one().

   EXAMPLES:

```python
sage: M = FiniteMonoids().example()
sage: M.one()
1
```
product\((x, y)\)

Return the product of two elements \(x\) and \(y\) of the monoid, as per \texttt{Semigroups.ParentMethods.product()}.

**EXAMPLES:**

```
sage: M = FiniteMonoids().example()
sage: M.product(M(3), M(5))
3
```

semigroup_generators()

Returns a set of generators for \texttt{self}, as per \texttt{Semigroups.ParentMethods.semigroup_generators()}. Currently this returns all integers mod \(n\), which is of course far from optimal!

**EXAMPLES:**

```
sage: M = FiniteMonoids().example()
sage: M.semigroup_generators()
Family (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

### 6.13 Examples of finite semigroups

\texttt{sage.categories.examples.finite_semigroups.Example} alias of \texttt{LeftRegularBand}

\texttt{class sage.categories.examples.finite_semigroups.LeftRegularBand(alphabet=('a', 'b', 'c', 'd'))}

Bases: \texttt{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')
```

This is the semigroup generated by:

```
sage: 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'
```

It follows that the elements of \(S\) are strings without repetitions over the alphabet \(a, b, c, d\):
sage: sorted(S.list())

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:

sage: S = FiniteSemigroups().example(alphabet = ('a', 'b', 'c'))
sage: S.cayley_graph(side="left", simple=True).plot()         # optional - sage.graphs
Graphics object consisting of 60 graphics primitives
sage: S.j_transversal_of_idempotents()      # random (arbitrary choice)   # optional - sage.graphs
['acb', 'ac', 'ab', 'bc', 'a', 'c', 'b']

We conclude by running systematic tests on this semigroup:

sage: 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
(continues on next page)
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

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'

    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'

    semigroup_generators()
        Returns the generators of the semigroup.

        EXAMPLES:

        sage: S = FiniteSemigroups().example(alphabet=('x', 'y'))
        sage: S.semigroup_generators()
        Family ('x', 'y')
6.14 Examples of finite Weyl groups

sage.categories.examples.finite_weyl_groups.Example
alias of SymmetricGroup

class sage.categories.examples.finite_weyl_groups.SymmetricGroup(n=4)
    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 SymmetricGroup for a full featured and optimized implementation.

EXAMPLES:

```python
sage: S = FiniteWeylGroups().example()
sage: S
The symmetric group on {0, ..., 3}
sage: S.category()
Category of finite irreducible weyl groups
```

The elements of this group are permutations of the set \{0, ..., 3\}:

```python
sage: S.one()
(0, 1, 2, 3)
sage: S.an_element()
(1, 2, 3, 0)
```

The group itself is generated by the elementary transpositions:

```python
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:

```python
sage: S.cardinality()
24
sage: S.long_element()
(3, 2, 1, 0)
sage: S.cayley_graph(side="left").plot()  #optional - 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 CoxeterGroups.ElementMethods.has_right_descent().

        EXAMPLES:

        sage: S = FiniteWeylGroups().example()
        sage: s = S.simple_reflections()
        sage: (s[1] * s[2]).has_descent(2)
        True
        sage: S._test_has_descent()

cartan_type()

        Return the Cartan type of self.

        EXAMPLES:

        sage: FiniteWeylGroups().example().cartan_type()  # optional - sage.combinat
        ['A', 3] relabelled by {1: 0, 2: 1, 3: 2}

degrees()

        Return the degrees of self.

        EXAMPLES:

        sage: W = FiniteWeylGroups().example()
        sage: W.degrees()
        (2, 3, 4)

index_set()

        Implements CoxeterGroups.ParentMethods.index_set().

        EXAMPLES:

        sage: FiniteWeylGroups().example().index_set()
        [0, 1, 2]

one()

        Implements Monoids.ParentMethods.one().

        EXAMPLES:

        sage: FiniteWeylGroups().example().one()
        (0, 1, 2, 3)

product(x, y)

        Implements Semigroups.ParentMethods.product().

        EXAMPLES:

        sage: s = FiniteWeylGroups().example().simple_reflections()
        (0, 2, 3, 1)
simple_reflection(\(i\))
Implement CoxeterGroups.ParentMethods.simple_reflection() by returning the transposition \((i, i + 1)\).

**EXAMPLES:**
```python
sage: FiniteWeylGroups().example().simple_reflection(2)
(0, 1, 3, 2)
```

### 6.15 Examples of graded connected Hopf algebras with basis

```python
sage.categories.examples.graded_connected_hopf_algebras_with_basis.Example
alias of GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator
```

class sage.categories.examples.graded_connected_hopf_algebras_with_basis.GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator
Bases: CombinatorialFreeModule

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

\[(\mu \otimes \mu) \circ (\text{id} \otimes \tau \otimes \text{id}) \circ (\Delta \otimes \Delta) = \Delta \circ \mu\]

where \(\tau(x \otimes y) = y \otimes x\).

coprodut_on_basis(\(i\))
The coproduct of a basis element.

\[\Delta(P_i) = \sum_{j=0}^{i} P_{i-j} \otimes P_j\]

**INPUT:**
- \(i\) – a non-negative integer

**OUTPUT:**
- an element of the tensor square of self

degree_on_basis(\(i\))
The degree of a non-negative integer is itself

**INPUT:**
- \(i\) – a non-negative integer

**OUTPUT:**
- a non-negative integer

one_basis()
Returns 0, which index the unit of the Hopf algebra.

**OUTPUT:**
- the non-negative integer 0
EXAMPLES:

```python
sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
sage: H.one_basis()
0
sage: H.one()
P0
```

**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:**

- i, j – non-negative integers

**OUTPUT:**

- a basis element indexed by i+j

### 6.16 Examples of graded modules with basis

```python
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.

```python
sage: A = GradedModulesWithBasis(QQ).example()
```

- 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.

```python
sage: A.basis(2)
Lazy family (Term map from Partitions to An example of a graded module with ...)_{i in Partitions of the integer 2}
sage: A.basis(6)[Partition([3,2,1])]
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)]
[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.

sage: A.degree_on_basis(Partition([4,3]))
7
sage: A._repr_term(Partition([4,3]))
'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.

sage: p = A.monomial(Partition([3,2,1])); p
P[3, 2, 1]
sage: p.is_homogeneous()
True
sage: p.degree()
6

`basis(d=None)`

Return the basis for (the d-th homogeneous component of) `self`.

INPUT:

- d – (optional, 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}

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.

```python
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:

```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'}}
```

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
def 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: A = GradedModulesWithBasis(QQ).example()
sage: A.degree_on_basis(Partition((2,1)))
3
sage: A.degree_on_basis(Partition((4,2,1,1,1,1)))
10
sage: type(A.degree_on_basis(Partition((1,1))))
<class 'sage.rings.integer.Integer'>
```

### 6.17 Examples of graphs

```python
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:
```
```python
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
```

We conclude by running systematic tests on this graph:

```python
sage: TestSuite(C).run()
```

### class `Element`

Bases: `ElementWrapper`

#### `dimension()`

Return the dimension of `self`.

**EXAMPLES:**

```python
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
```

#### `an_element()`

Return an element of the graph, as per `Sets.ParentMethods.an_element()`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.an_element()
0
```

#### `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)]
```

#### `vertices()`

Return the vertices of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
```

(continues on next page)
6.18 Examples of Hopf algebras with basis

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)]}

antipode_on_basis(g)
   Antipode, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.antipode_on_basis().
   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)]

coproduct_on_basis(g)
   Coproduct, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.coproduct_on_basis().
   The basis elements are group like: \( \Delta(g) = g \otimes g \).
   EXAMPLES:
   sage: A = HopfAlgebrasWithBasis(QQ).example()
   sage: (a, b) = A._group.gens()
   sage: A.coprodct_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
```

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()
()
sage: A.one()
B[()]
```

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:

```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: a*b
(1,2)
sage: A.product_on_basis(a, b)
B[(1,2)]
```

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:

```python
sage: NN = InfiniteEnumeratedSets().example()
sage: NN
An example of an infinite enumerated set: the non negative integers
```
This checks that the different methods of $\mathbb{N}$ return consistent results:

```
sage: TestSuite(NN).run( verbose = True )
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . 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
```

`Element` alias of `Integer`
an_element()  
EXAMPLES:

```
sage: InfiniteEnumeratedSets().example().an_element()  
42
```

next(o)  
EXAMPLES:

```
sage: NN = InfiniteEnumeratedSets().example()  
sage: NN.next(3)  
4
```

### 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 $\mathfrak{sl}_2$ 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
```
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()
    sage: L.lie_algebra_generators()
    Family ([2, 1, 3], [2, 3, 1])

zero()
    Return the element 0.

    EXAMPLES:
    sage: L = LieAlgebras(QQ).example()
    sage: L.zero()

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:
        sage: L = LieAlgebras(QQ).WithBasis().example()
        sage: elt = L.an_element()
        sage: elt.lift()

    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
```

**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
sage.categories.examples.lie_algebras_with_basis.Example
alias of AbelianLieAlgebra
```

**class** sage.categories.examples.lie_algebras_with_basis.IndexedPolynomialRing(R, indices, **kwds)**

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:**

```python
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:**

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
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:**

6.21. Examples of a Lie algebra with basis
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')

    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

class Element
    Bases: ElementWrapper
    The class for elements of the free magma.
    wrapped_class
        alias of str
    an_element()
        Return an element of the magma.
        EXAMPLES:

        sage: F = Magmas().example()
        sage: F.an_element()
        '(((a*b)*c)*d)'
magma_generators()

    Return the generators of the magma.

    EXAMPLES:

    sage: F = Magmas().example()
sage: F.magma_generators()
Family ('a', 'b', 'c', 'd')

product(x, y)

    Return the product of x and y in the magma, as per Magmas.ParentMethods.product().

    EXAMPLES:

    sage: F = Magmas().example()
sage: F('a') * F.an_element()
'(a*(((a*b)*c)*d))'

6.23 Examples of manifolds

sage.categories.examples.manifolds.Example
alias of Plane
class sage.categories.examples.manifolds.Plane(n=3, base_ring=\text{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:

```
sage: from sage.categories.manifolds import Manifolds
sage: M = Manifolds(QQ).example()
sage: M.dimension()
3
```

### 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:

```
sage: S = Monoids().example(); S
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
sage: S.category()
Category of monoids
```

This is the free semigroup generated by:

```
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

with product rule given by concatenation of words:

```
sage: S('dab') * S('acb')
dabacb
```

and unit given by the empty word:

```
sage: S.one()
'
```

We conclude by running systematic tests on this monoid:

```
sage: TestSuite(S).run(underscore=True)
running _test_an_element() . . . pass
running _test_associativity() . . . pass
running _test_cardinality() . . . pass
running _test_category() . . . pass
running _test_construction() . . . pass
```

(continues on next page)
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
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'

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()
    ''
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

   We conclude by running systematic tests on this poset:

   sage: TestSuite(P).run(quiet = 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
   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

class Element
   Bases: ElementWrapper

   wrapped_class
      alias of Set_objectEnumerated

   an_element()
      Returns an element of this poset
      EXAMPLES:

      sage: B = Posets().example()
      sage: B.an_element()
      {1, 4, 6}
```

le(x, y)

Returns whether x is a subset of y

EXAMPLES:

```python
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
```

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:

```python
sage: P = Posets().example("facade"); P
An example of a facade poset: the positive integers ordered by divisibility

sage: P(5)
5
sage: P(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
```

class element_class

Bases: Set_object_enumerated, parent_class

A finite enumerated set.

le(x, y)

Returns whether x is divisible by y

EXAMPLES:

```python
sage: P = Posets().example("facade")
sage: P.le(3, 6)
True
sage: P.le(3, 3)
True
sage: P.le(3, 7)
False
```
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 of a semigroup.

    EXAMPLES:

    sage: 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')

    and with product given by concatenation:

    sage: 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'

    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'

    semigroup_generators()
    Returns the generators of the semigroup.

    EXAMPLES:
sage: F = Semigroups().example('free')
sage: 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

class Element
    Bases: ElementWrapper

    ambient()
    Returns the ambient semigroup.
    EXAMPLES:

    sage: S = Semigroups().Subquotients().example()
sage: 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:

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

    This is the semigroup that contains all sorts of objects:

    sage: 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

class Element
    Bases: ElementWrapper

6.26. Examples of semigroups
is_idempotent()

Trivial implementation of Semigroups.Element.is_idempotent since all elements of this semi-
group are idempotent!

EXAMPLES:

```python
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:

```python
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:

```python
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:

```python
sage: Semigroups().example().some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

class sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup(category=None)

Bases: UniqueRepresentation, Parent

Example of a quotient semigroup

EXAMPLES:

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

This is the quotient of:

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

obtained by setting $x = 42$ for any $x \geq 42$:
The product is inherited from the ambient semigroup:

```python
sage: S(1)*S(2) == S(1)
True
```

### class Element

Bases: `ElementWrapper`

#### ambient()

Returns the ambient semigroup.

**EXAMPLES:**

```python
sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
```

#### an_element()

Returns an element of the semigroup.

**EXAMPLES:**

```python
sage: S = Semigroups().Subquotients().example()
sage: S.an_element()
```

#### 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
sage: S.lift(x)
42
sage: S.lift(x) in S.ambient()
True
sage: y = S.ambient()(100); y
100
sage: 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 (self.ambient()).

OUTPUT:

• an element of self.

EXAMPLES:

```
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(171))
42
```

some_elements()
Returns a list of some elements of the semigroup.

EXAMPLES:

```
sage: S = Semigroups().Subquotients().example()
sage: S.some_elements()
[1, 2, 3, 8, 42, 42]
```

the_answer()
Returns the Answer to Life, the Universe, and Everything as an element of this semigroup.

EXAMPLES:

```
sage: S = Semigroups().Subquotients().example()
sage: S.the_answer()
42
```

6.27 Examples of semigroups in cython

class sage.categories.examples.semigroups_cython.IdempotentSemigroups($s=None$)
Bases: Category
class ElementMethods
    Bases: object
    is_idempotent()
    EXAMPLES:

```
sage: from sage.categories.examples.semigroups_cython import...
sage: S = LeftZeroSemigroup()
sage: S(2).is_idempotent()
True
```

super_categories()

EXAMPLES:

```python
sage: from sage.categories.examples.semigroups_cython import...
˓→IdempotentSemigroups
sage: IdempotentSemigroups().super_categories()
[Category of semigroups]
```

class sage.categories.examples.semigroups_cython.LeftZeroSemigroup

Bases: LeftZeroSemigroup

An example of 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:

```python
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup(); S
An example of a semigroup: the left zero semigroup

This is the semigroup which contains all sort of objects:

```python
sage: 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)
3
sage: S(3)^12312321312321
3
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
```

(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_pickling() . . . pass
running ._test_some_elements() . . . pass

That’s really the only method which is obtained from the category ...

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

**Element**

alias of *LeftZeroSemigroupElement*

class sage.categories.examples.semigroups_cython.LeftZeroSemigroupElement
   Bases: Element

   EXAMPLES:

sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: x = S(3)
sage: TestSuite(x).run()
6.28 Examples of sets

```python
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 \( \mathbb{Z} \) (facade implementation).

This is a minimal implementations. For more advanced examples of implementations, see also:
```
sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: 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()
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: TestSuite(P).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
```
(continues on next page)
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

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")

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

next()
    Return the next prime number.

    EXAMPLES:

    sage: P = Sets().example("inherits")
    sage: p = P.an_element(); p
    47
    sage: 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()

Implements Sets.ParentMethods.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:

```
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
```
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
```

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
```

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'
```

unlike in the other implementations:

```python
sage: pw.next()
53
sage: pi.next()
53
```

`element_class`
alias of `Integer`

```python
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:

```python
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.parent()  
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 the test suite of self.an_element() . . . pass  
Running the test suite of self.category() . . . pass  
Running the test suite of self.construction() . . . pass  
Running the test suite of self.elements() . . .  
Running the test suite of self.elements() . . . pass  
Running the test suite of self.eq() . . . pass  
Running the test suite of self.new() . . . pass  
Running the test suite of self_not_implemented_methods() . . . pass  
Running the test suite of self_pickling() . . . pass  
Running the test suite of self_elements_eq_reflexive() . . . pass  
Running the test suite of self_elements_eq_symmetric() . . . pass  
Running the test suite of self_elements_eq_transitive() . . . pass  
Running the test suite of self_neq() . . . pass

(continues on next page)
running ._test_eq() . . . passunning ._test_new() . . . passunning ._test_not_implemented_methods() . . . passunning ._test_pickling() . . . passunning ._test_some_elements() . . . pass

See also:

```python
sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: P = Sets().example("wrapper")
```

class `Element`

```
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)
Traceback (most recent call last):
  ... ValueError: 12 is not a prime number
sage: a = P.an_element()
sage: a.parent()
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'>
```
class Element
    Bases: ElementWrapper, Element

ElementWrapper
    alias of ElementWrapper

6.29 Example of a set with grading

sage.categories.examples.sets_with_grading.Example
    alias of NonNegativeIntegers

class sage.categories.examples.sets_with_grading.NonNegativeIntegers
    Bases: UniqueRepresentation, Parent
    Non negative integers graded by themselves.

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

generating_series(var='z')
    Return 1/(1 - z).

EXAMPLES:

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:

```
sage: N = SetsWithGrading().example()
sage: N.graded_component(65)
{65}
```

grading(elt)
Return the grade of elt.

EXAMPLES:

```
sage: N = SetsWithGrading().example()
sage: N.grading(10)
10
```

6.30 Examples of parents endowed with multiple realizations

class sage.categories.examples.with_realizations.SubsetAlgebra(R, S)
Bases: 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: $F$, In, and Out. In the first basis, the product is given by the union of the indexing sets. That is, for any $s, t \subset S$

$$F_s F_t = F_{s \cup t}$$

The In basis and Out basis are defined respectively by:

$$In_s = \sum_{t \subset s} F_t \quad \text{and} \quad F_s = \sum_{t \supset s} Out_t$$

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:

• Sets().WithRealizations
• the Implementing Algebraic Structures thematic tutorial.

EXAMPLES:

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

The three bases of $A$: 
Category Framework, Release 10.1

```python
sage: F = A.F() ; F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: In = A.In() ; In
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: Out = A.Out(); Out
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
```

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
```

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}]]
```

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}]
```

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}]
```

Some conversions:

```python
sage: F(In[2,3])
F[{}] + F[{2}] + F[{3}] + F[{2, 3}]
sage: In(F[3])
In[{}] - In[{2}] - In[{3}] + In[{2, 3}]
sage: Out(F[3])
Out[{3}] + Out[{1, 3}] + Out[{2, 3}] + Out[{1, 2, 3}]
sage: F(Out[3])
F[{3}] - F[{1, 3}] - F[{2, 3}] + F[{1, 2, 3}]
sage: Out(In[2,3])
Out[{}] + 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}]
```

We can now mix expressions:
class Bases(parent_with_realization)

Bases: Category_realization_of_parent

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:

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}]

As a shorthand, one can construct elements using the following notation:

sage: In[2,3]
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[{1}]
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]
F
  alias of Fundamental
class Fundamental(A)
  Bases: CombinatorialFreeModule, BindableClass
  The Subset algebra, in the fundamental basis
  INPUT:
  • A – a parent with realization in SubsetAlgebra
  EXAMPLES:
  
  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

one()
  Return the multiplicative unit element.
  EXAMPLES:
  
  sage: A = AlgebrasWithBasis(QQ).example()
  # optional - sage.combinat sage.modules
  sage: A.one_basis()
  # optional - sage.combinat sage.modules
  word:
sage: A.one()
  # optional - sage.combinat sage.modules
  B[word: ]

one_basis()
  Returns the index of the basis element which is equal to ‘1’.
  EXAMPLES:
  
  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[{}]
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}]

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

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

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

base_set()

EXAMPLES:

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

The objects that index the basis elements of this algebra.

EXAMPLES:

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

```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\}]
```

supsets(set)

Returns all the subsets of \(S\) containing set

INPUT:

• set – a subset of the base set \(S\) of self

EXAMPLES:

```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\}]
```
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).

class sage.categories.category_types.AbelianCategory(s=None):
    Bases: Category
    is_abelian()
    Return True as self is an abelian category.
    EXAMPLES:
    sage: 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

    ring()  
    Return the ambient ring used to describe objects self.
    EXAMPLES:
    sage: C = Ideals(IntegerRing())
    sage: C.ring()
    Integer Ring

class sage.categories.category_types.Category_in_ambient(ambient, name=None):
    Bases: Category
    Initialize self.
    EXAMPLES:
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

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 github issue #11935 for details.

EXAMPLES:

```python
sage: Algebras(GF(2)).element_class is Algebras(GF(3)).element_class
#optional - sage.rings.finite_rings
True
sage: C = GF(2).category()
#optional - sage.rings.finite_rings
sage: Algebras(GF(2)).parent_class is Algebras(C).parent_class
#optional - sage.rings.finite_rings
True
sage: C = ZZ.category()
sage: 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
```

base()

Return the base over which elements of this category are defined.

EXAMPLES:
```python
sage: C = Algebras(QQ)
sage: C.base()
Rational Field
```

class sage.categories.category_types.Category_over_base_ring(base, name=None)

Bases: Category_over_base

Initialize self.

EXAMPLES:

```python
sage: C = Algebras(GF(2)); C
# optional - sage.rings.finite_rings
Category of algebras over Finite Field of size 2
sage: TestSuite(C).run()
# optional - sage.rings.finite_rings
```

base_ring()

Return the base ring over which elements of this category are defined.

EXAMPLES:

```python
sage: C = Algebras(GF(2))
# optional - sage.rings.finite_rings
sage: C.base_ring()
# optional - sage.rings.finite_rings
Finite Field of size 2
```

class sage.categories.category_types.Elements(object)

Bases: Category

The category of all elements of a given parent.

EXAMPLES:

```python
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
```

classmethod an_instance()

Returns an instance of this class

EXAMPLES:

```python
sage: Elements.an_instance()
Category of elements of Rational Field
```

object()

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 `True` if and only if `x` has a category which is a subcategory of this one.

**EXAMPLES:**

```python
testsage: ZZ in Sets()
True```

**class** `sage.categories.category_singleton.Category_singleton(s=None)`

Bases: `Category`

A base class for implementing singleton category

A *singleton* category is a category whose class takes no parameters like `Fields()` or `Rings()`. See also the Singleton design pattern.

This is a subclass of `Category`, with a couple optimizations for singleton categories.

The main purpose is to make the idioms:

```python
testsage: QQ in Fields()
True
testsage: ZZ in Fields()
False```

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:**

```python
testsage: from sage.categories.category_singleton import Category_singleton
testsage: class MyRings(Category):
    ....:    def super_categories(self):
    ....:        return Rings().super_categories()
testsage: class MyRingsSingleton(Category_singleton):
    ....:    def super_categories(self):
    ....:        return Rings().super_categories()```
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”:

```sage
category: R = QQ['x,y']
category: R1 = Parent(category = MyRings())
category: R2 = Parent(category = MyRingsSingleton())
category: R in MyRings()
False
category: R1 in MyRings()
True
category: R1 in MyRingsSingleton()
False
category: R2 in MyRings()
False
category: R2 in MyRingsSingleton()
True
```

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
timeit("R1 in MyRings()", number=10000)
10000 loops, best of 3: 6.98 µs per loop
timeit("R in MyRingsSingleton()", number=10000)
10000 loops, best of 3: 3.08 µs per loop
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
# not tested
timeit("R.is_ring()", number=10000)
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()
category: R3 = Parent(category = _Rings)
category: R3.is_ring.__module__
'sage.categories.rings'
timeit("R3.is_ring()", number=10000)
10000 loops, best of 3: 2.64 µs per loop
timeit("R3 in Rings()", number=10000)
10000 loops, best of 3: 3.01 µs per loop
timeit("R3 in _Rings", number=10000)
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
timeit("R in X ", number=10000)# not tested
```

(continues on next page)
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 `Category_singleton` are supported:

```python
sage: class MyRingsSingleton(Category_singleton):
    ....: def super_categories(self): return Rings().super_categories()
```

```python
sage: class Disaster(MyRingsSingleton):
    pass
```

```python
sage: Disaster()
Traceback (most recent call last):
  ...
AssertionError: <class '__main__.Disaster'> is not a direct subclass of <class '__main__.Category_singleton.Category_singleton'>
```

However, it is acceptable for a direct subclass $R$ of `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:

```python
sage: from sage.categories.category_singleton import Category_singleton
sage: class R(Category_singleton):
    ....: def super_categories(self): return [Sets()]
```

```python
sage: R()
Category of r
```

```python
sage: R().__class__
<class '__main__.R_with_category'>
```

```python
sage: R().__class__.mro()
[<class '__main__.R_with_category'>,
 <class '__main__.R'>,
 <class 'sage.categories.category_singleton.Category_singleton'>,
 <class 'sage.categories.category.Category'>,
 <class 'sage.structure.unique_representation.UniqueRepresentation'>,
 <class 'sage.structure.unique_representation.CachedRepresentation'>,
 <class 'sage.misc.fast_methods.WithEqualityById'>,
 <class 'sage.structure.sage_object.SageObject'>,
 <class '__main__.R.subcategory_class'>,
 <class 'sage.categories.sets_cat.Sets.subcategory_class'>,
 <class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.subcategory_class'>,
 <... 'object'>]
```

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

```python
sage: R() is R().__class__()
True
```

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 MyRingsSingleton() is not a subcategory of the join of its super categories:

```python
sage: C = MyRingsSingleton()
sage: C.super_categories()
[Category of rngs, Category of semirings]
sage: Rings() & Semirings()
Category of rings
sage: 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)

```python
class sage.categories.category_cy_helper.AxiomContainer
    Bases: dict
    A fast container for axioms.
```

This is derived from 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 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
```

**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
```

To avoid side effects, we remove the added axiom:
```
sage: del all_axioms['Awesome']
```

```
sage.categories.category_cy_helper.canonize_axioms(all_axioms, axioms)
Canonize 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:
```
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')
```

```
sage.categories.category_cy_helper.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.

EXAMPLES:
```
sage: from sage.categories.category_cy_helper import category_sort_key
sage: category_sort_key(Rings()) is Rings()._cmp_key
True
```

```
sage.categories.category_cy_helper.get_axiom_index(all_axioms, axiom)
Helper function: Return the rank of an axiom.

INPUT:

- all_axioms – the axiom collection
- axiom – string, name of an axiom

EXAMPLES:
```
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: from sage.categories.category_cy_helper import get_axiom_index
```
Helper for :func:`join()`.

INPUT:

- **categories** – tuple of categories to be joined,
- **axioms** – tuple of strings; the names of some supplementary axioms.
- **ignore_axioms** – tuple of pairs (cat, axiom), such that axiom will not be applied to cat, should cat
  occur in the algorithm.

EXAMPLES:

```python
sage: T = (Coalgebras(QQ), Sets().Finite(), Algebras(ZZ), SimplicialComplexes())

sage: from sage.categories.category_cy_helper import join_as_tuple
sage: join_as_tuple(T,(),())
(Category of algebras over Integer Ring,
 Category of finite monoids,
 Category of finite additive groups,
 Category of coalgebras over Rational Field,
 Category of finite simplicial complexes)

sage: join_as_tuple(T,('WithBasis',),())
(Category of algebras with basis over Integer Ring,
 Category of finite monoids,
 Category of coalgebras with basis over Rational Field,
 Category of finite additive groups,
 Category of finite simplicial complexes)

sage: join_as_tuple(T,(),(Monoids(),'Finite',))
(Category of algebras over Integer Ring,
 Category of finite additive groups,
 Category of coalgebras over Rational Field,
 Category of finite simplicial complexes)
```
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=None, 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:
    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:
    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:
    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
    codomain()

    Returns the codomain of self

    EXAMPLES:
    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+1, domain=(1,2,3), codomain=(2,3,4)).domain()
(1, 2, 3)
```
INDICES AND TABLES

• Index
• Module Index
• Search Page
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.categories.action</td>
<td>169</td>
</tr>
<tr>
<td>sage.categories.additive_groups</td>
<td>172</td>
</tr>
<tr>
<td>sage.categories.additive_magmas</td>
<td>174</td>
</tr>
<tr>
<td>sage.categories.additive_monoids</td>
<td>186</td>
</tr>
<tr>
<td>sage.categories.additive_semigroups</td>
<td>187</td>
</tr>
<tr>
<td>sage.categories.affine_weyl_groups</td>
<td>190</td>
</tr>
<tr>
<td>sage.categories.algebra_functor</td>
<td>877</td>
</tr>
<tr>
<td>sage.categories.algebra_ideals</td>
<td>193</td>
</tr>
<tr>
<td>sage.categories.algebra_modules</td>
<td>194</td>
</tr>
<tr>
<td>sage.categories.algebras</td>
<td>195</td>
</tr>
<tr>
<td>sage.categories.algebras_with_basis</td>
<td>198</td>
</tr>
<tr>
<td>sage.categories.aperiodic_semigroups</td>
<td>203</td>
</tr>
<tr>
<td>sage.categories.associative_algebras</td>
<td>204</td>
</tr>
<tr>
<td>sage.categories.bialgebras</td>
<td>206</td>
</tr>
<tr>
<td>sage.categories.bialgebras_with_basis</td>
<td>206</td>
</tr>
<tr>
<td>sage.categories.cartesian_product</td>
<td>872</td>
</tr>
<tr>
<td>sage.categories.category</td>
<td>31</td>
</tr>
<tr>
<td>sage.categories.category_cy_helper</td>
<td>967</td>
</tr>
<tr>
<td>sage.categories.category_singleton</td>
<td>964</td>
</tr>
<tr>
<td>sage.categories.category_types</td>
<td>961</td>
</tr>
<tr>
<td>sage.categories.category_with_axiom</td>
<td>69</td>
</tr>
<tr>
<td>sage.categories.classical_crystals</td>
<td>212</td>
</tr>
<tr>
<td>sage.categories.coalgebras</td>
<td>216</td>
</tr>
<tr>
<td>sage.categories.coalgebras_with_basis</td>
<td>221</td>
</tr>
<tr>
<td>sage.categories.coercion_methods</td>
<td>970</td>
</tr>
<tr>
<td>sage.categories.commutative_additive_groups</td>
<td>224</td>
</tr>
<tr>
<td>sage.categories.commutative_additive_monoids</td>
<td>225</td>
</tr>
<tr>
<td>sage.categories.commutative_additive_semigroups</td>
<td>226</td>
</tr>
<tr>
<td>sage.categories.commutative_algebra_ideals</td>
<td>226</td>
</tr>
<tr>
<td>sage.categories.commutative_algebras</td>
<td>227</td>
</tr>
<tr>
<td>sage.categories.commutative_ring_ideals</td>
<td>227</td>
</tr>
<tr>
<td>sage.categories.commutative_rings</td>
<td>228</td>
</tr>
<tr>
<td>sage.categories.complete_discrete_valuation</td>
<td>234</td>
</tr>
<tr>
<td>sage.categories.complex_reflection_groups</td>
<td>238</td>
</tr>
<tr>
<td>sage.categories.complex_reflection_or_generalized_coxeter_groups</td>
<td>240</td>
</tr>
<tr>
<td>sage.categories.covariant_functorial_construction</td>
<td>867</td>
</tr>
<tr>
<td>sage.categories.coxeter_group_algebras</td>
<td>258</td>
</tr>
<tr>
<td>sage.categories.coxeter_groups</td>
<td>261</td>
</tr>
<tr>
<td>sage.categories.crystals</td>
<td>301</td>
</tr>
<tr>
<td>sage.categories.discrete_valuation</td>
<td>327</td>
</tr>
<tr>
<td>sage.categories.distributive_magmas_and_additive_magmas</td>
<td>330</td>
</tr>
<tr>
<td>sage.categories.division_rings</td>
<td>331</td>
</tr>
<tr>
<td>sage.categories.domains</td>
<td>332</td>
</tr>
<tr>
<td>sage.categories.dual</td>
<td>877</td>
</tr>
<tr>
<td>sage.categories.enumerated_sets</td>
<td>333</td>
</tr>
<tr>
<td>sage.categories.euclidean_domains</td>
<td>340</td>
</tr>
<tr>
<td>sage.categories.examples.algebras_with_basis</td>
<td>899</td>
</tr>
<tr>
<td>sage.categories.examples.commutative_additive_monoids</td>
<td>900</td>
</tr>
<tr>
<td>sage.categories.examples.commutative_additive_semigroups</td>
<td>901</td>
</tr>
<tr>
<td>sage.categories.examples.coxeter_groups</td>
<td>903</td>
</tr>
<tr>
<td>sage.categories.examples.crystals</td>
<td>903</td>
</tr>
<tr>
<td>sage.categories.examples.cw_complexes</td>
<td>905</td>
</tr>
<tr>
<td>sage.categories.examples.facade_sets</td>
<td>906</td>
</tr>
<tr>
<td>sage.categories.examples.finite_coxeter_groups</td>
<td>907</td>
</tr>
<tr>
<td>sage.categories.examples.finite_dimensional_algebras_with_basis</td>
<td>909</td>
</tr>
<tr>
<td>sage.categories.examples.finite_dimensional_lie_algebras_with_basis</td>
<td>910</td>
</tr>
<tr>
<td>sage.categories.examples.finite enumerated_sets</td>
<td>914</td>
</tr>
<tr>
<td>sage.categories.examples.finite_monoids</td>
<td>916</td>
</tr>
<tr>
<td>sage.categories.examples.finite_semigroups</td>
<td>918</td>
</tr>
<tr>
<td>sage.categories.examples.finite_weyl_groups</td>
<td>921</td>
</tr>
<tr>
<td>sage.categories.examples.commutative_ring_ideals</td>
<td>923</td>
</tr>
<tr>
<td>Python Module Index</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td>sage.categories.examples.graded_modules_with_basis</td>
<td>categories.finite_groups, 475</td>
</tr>
<tr>
<td>sage.categories.examples.graphs</td>
<td>categories.finite_lattice_posets, 477</td>
</tr>
<tr>
<td>sage.categories.examples.hopf_algebras_with_basis</td>
<td>categories.finite_monoids, 481</td>
</tr>
<tr>
<td>sage.categories.examples.infinite_enumerated_sets</td>
<td>categories.finite_permutation_groups, 485</td>
</tr>
<tr>
<td>sage.categories.examples.lie_algebras</td>
<td>categories.finite_posets, 489</td>
</tr>
<tr>
<td>sage.categories.examples.lie_algebras_with_basis</td>
<td>categories.finite_semigroups, 511</td>
</tr>
<tr>
<td>sage.categories.examples.magmas</td>
<td>categories.finite_sets, 513</td>
</tr>
<tr>
<td>sage.categories.examples.manifolds</td>
<td>categories.finite_weyl_groups, 514</td>
</tr>
<tr>
<td>sage.categories.examples.monoids</td>
<td>categories.finitely_generated_lambda_bracket_algebras, 515</td>
</tr>
<tr>
<td>sage.categories.examples.posets</td>
<td>categories.finitely_generated_magmas, 518</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups</td>
<td>categories.finitely_generated_semigroups, 518</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups_cython</td>
<td>categories.function_fields, 521</td>
</tr>
<tr>
<td>sage.categories.examples.sets_cat</td>
<td>categories.g_sets, 521</td>
</tr>
<tr>
<td>sage.categories.examples.sets_with_grading</td>
<td>categories.gcd_domains, 522</td>
</tr>
<tr>
<td>sage.categories.facade_sets</td>
<td>categories.generalized_coxeter_groups, 523</td>
</tr>
<tr>
<td>sage.categories.fields</td>
<td>categories.graded_algebras, 524</td>
</tr>
<tr>
<td>sage.categories.filtered_algebras</td>
<td>categories.graded_algebras_with_basis, 525</td>
</tr>
<tr>
<td>sage.categories.filtered_algebras_with_basis</td>
<td>categories.graded_bialgebras, 528</td>
</tr>
<tr>
<td>sage.categories.filtered_modules</td>
<td>categories.graded_bialgebras_with_basis, 528</td>
</tr>
<tr>
<td>sage.categories.filtered_modules_with_basis</td>
<td>categories.graded_coalgebras, 529</td>
</tr>
<tr>
<td>sage.categories.finite_complex_reflection_groups</td>
<td>categories.graded_coalgebras_with_basis, 529</td>
</tr>
<tr>
<td>sage.categories.finite_coxeter_groups</td>
<td>categories.graded_hopf_algebras, 530</td>
</tr>
<tr>
<td>sage.categories.finite_crystals</td>
<td>categories.graded_hopf_algebras_with_basis, 530</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>categories.graded_lie_algebras, 532</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_bialgebras</td>
<td>categories.graded_lie_algebras_with_basis, 533</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_coalgebras</td>
<td>categories.graded_lie_algebras_with_basis, 533</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_graded_lie_conformal_algebras</td>
<td>categories.graded_lie_conformal_algebras, 534</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_graded_lie_conformal_algebras_with_basis</td>
<td>categories.graded_lie_conformal_algebras, 534</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_graded_lie_conformal_algebras_with_basis</td>
<td>categories.highest_weight_crystals, 560</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_graded_lie_conformal_algebras_with_basis</td>
<td>categories.homsets, 886</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_semisimple_algebras_with_basis</td>
<td>categories.hopf_algebras, 567</td>
</tr>
<tr>
<td>sage.categories.finite_fields</td>
<td>categories.hopf_algebras_with_basis, 570</td>
</tr>
</tbody>
</table>
sage.categories.infinite_enumerated_sets, 575
sage.categories.integral_domains, 576
sage.categories.isomorphic_objects, 885
sage.categories.j_trivial_semigroups, 577
sage.categories.kac_moody_algebras, 577
sage.categories.l_trivial_semigroups, 619
sage.categories.lambda_bracket_algebras, 578
sage.categories.lambda_bracket_algebras_with_basis, 581
sage.categories.lattice_posets, 583
sage.categories.left_modules, 584
sage.categories.lie_algebras, 584
sage.categories.lie_algebras_with_basis, 597
sage.categories.lie_conformal_algebras, 600
sage.categories.lie_conformal_algebras_with_basis, 603
sage.categories.lie_groups, 605
sage.categories.loop_crystals, 606
sage.categories.magmas, 619
sage.categories.magmas_and_additive_magmas, 635
sage.categories.magmatic_algebras, 637
sage.categories.manifolds, 641
sage.categories.map, 109
sage.categories.matrix_algebras, 645
sage.categories.metric_spaces, 645
sage.categories.modular_abelian_varieties, 650
sage.categories.modules, 650
sage.categories.modules_with_basis, 664
sage.categories.monoid_algebras, 698
sage.categories.monoids, 698
sage.categories.morphism, 128
sage.categories.number_fields, 706
sage.categories.objects, 708
sage.categories.partially_ordered_monoids, 710
sage.categories.permutation_groups, 710
sage.categories.pointed_sets, 711
sage.categories.polyhedra, 711
sage.categories.poor_man_map, 970
sage.categories.posets, 712
sage.categories.primer, 3
sage.categories.principal_ideal_domains, 721
sage.categories.pushout, 133
sage.categories.quantum_group_representations, 729
sage.categories.quotient_fields, 722
sage.categories.quotients, 884
sage.categories.r_trivial_semigroups, 761
sage.categories.realizations, 890
sage.categories.regular_crystals, 737
sage.categories.regular_supercrystals, 745
sage.categories.right_modules, 747
sage.categories.ring_ideals, 747
sage.categories.rings, 748
sage.categories.rngs, 760
sage.categories.schemes, 761
sage.categories.semisimple_algebras, 776
sage.categories.semisimple_algebras_with_basics, 777
sage.categories.sets_cat, 777
sage.categories.sets_with_grading, 806
sage.categories.sets_with_partial_maps, 809
sage.categories.shephard_groups, 809
sage.categories.graded, 876
sage.categories.simplicial_complexes, 810
sage.categories.simplicial_sets, 812
sage.categories.subobjects, 885
sage.categories.subquotients, 884
sage.categories.super_algebras, 823
sage.categories.super_algebras_with_basis, 825
sage.categories.super_hopf_algebras_with_basis, 827
sage.categories.super_lie_conformal_algebras, 828
sage.categories.super_modules, 829
sage.categories.super_modules_with_basis, 832
sage.categories.supercrystals, 835
sage.categories.tensor, 874
sage.categories.topological_spaces, 840
sage.categories.triangular_kac_moody_algebras, 842
sage.categories.unique_factorization_domains, 844
sage.categories.unital_algebras, 845
sage.categories.vector_bundles, 848
sage.categories.vector_spaces, 849
sage.categories.weyl_groups, 854
sage.categories.with_realizations, 892
Symbols

___classcall__() (sage.categories.category.Category static method), 44
___classcall__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 94
___classget__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 95
__init__() (sage.categories.category.Category method), 44
__init__() (sage.categories.category_with_axiom.CategoryWithAxiom method), 95
_all_super_categories() (sage.categories.category.Category method), 37
_all_super_categories_proper() (sage.categories.category.Category method), 38
_make_named_class() (sage.categories.category.Category method), 39
_make_named_class() (sage.categories.category.CategoryWithParameters.Category method), 65
__repr__() (sage.categories.category.Category method), 40
__repr__() (sage.categories.category.JoinCategory.Category method), 67
__repr_object_names() (sage.categories.category.Category method), 40
__repr_object_names() (sage.categories.category.JoinCategory.Category method), 66
__repr_object_names() (sage.categories.category_with_axiom.CategoryWithAxiom method), 95
__repr_object_names_static() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 96
_set_of_super_categories() (sage.categories.category.Category method), 38
__sort__() (sage.categories.category.Category static method), 42
__sort_uniq__() (sage.categories.category.Category static method), 43
__super_categories() (sage.categories.category_with_axiom.CategoryWithAxiom method), 37
__super_categories_for_classes() (sage.categories.category.Category method), 57
__test_category() (sage.categories.category.Category method), 40
__test_category_with_axiom() (sage.categories.category_with_axiom.CategoryWithAxiom method), 96
__with_axiom() (sage.categories.category.Category method), 41
__with_axiom_as_tuple() (sage.categories.category.Category method), 41
__without_axioms() (sage.categories.category.Category method), 42
__without_axioms() (sage.categories.category.JoinCategory.Category method), 67
__without_axioms() (sage.categories.category_with_axiom.CategoryWithAxiom method), 97

A

a_realization() (sage.categories.examples.with_realizations.SubsetAlgebra method), 958
a_realization() (sage.categories.sets_cat.Sets.WithRealizations.ParentMethods method), 801

AbelianCategory (class in sage.categories.category_types), 961
AbelianLieAlgebra (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 910
AbelianLieAlgebra (class in sage.categories.examples.lie_algebras_with_basis), 932
AbelianLieAlgebra.Element (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis),
AbelianLieAlgebra.Element (class in sage.categories.examples.lie_algebras_with_basis), (sage.categories.covariant_functorial_construction.CovariantConstructionCategory method), 240

abs() (sage.categories.metric_spaces.MetricSpaces.ElementMethods method), 867

absolute_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 262

absolute_le() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 262

absolute_length() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 263

absolute_order_ideal() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 522

absolute_poset() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 380

act() (sage.categories.action.Action method), 170

add() (sage.categories.category_cy_helper.AxiomContainer method), 967

addition_table() (sage.categories.additive_magmas.AdditiveMagmas.ParentMethods method), 181

additional_structure() (sage.categories.additive_magmas.AdditiveUnital method), 179

additional_structure() (sage.categories.affine_weyl_groups.AffineWeylGroups method), 192

additional_structure() (sage.categories.bialgebras.Bialgebras method), 205

additional_structure() (sage.categories.bimodules.Bimodules method), 211

additional_structure() (sage.categories.category.Category method), 50

additional_structure() (sage.categories.category.JoinCategory method), 67

additional_structure() (sage.categories.category_with_axiom.CategoryWithAxiom method), 97

additional_structure() (sage.categories.classical_crystals.ClassicalCrystals method), 215

additional_structure() (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 240

additional_structure() (sage.categories.covariant_functorial_construction.CovariantConstructionCategory method), 867

additional_structure() (sage.categories.coxeter_groups.CoxeterGroups method), 300

additional_structure() (sage.categories.enumerated_sets.EnumeratedSets method), 340

additional_structure() (sage.categories.gcd_domains.GcdDomains method), 522

additional_structure() (sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups method), 523

additional_structure() (sage.categories.highest_weight_crystals.HighestWeightCrystals method), 567

additional_structure() (sage.categories.lie_groups.LieGroups method), 605

additional_structure() (sage.categories.magmas.Magmas.Unital method), 634

additional_structure() (sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas method), 636

additional_structure() (sage.categories.manifolds.Manifolds method), 644

additional_structure() (sage.categories.modules.Modules method), 663

additional_structure() (sage.categories.objects.Objects method), 709

additional_structure() (sage.categories.principal_ideal_domains.PrincipalIdealDomains method), 722

additional_structure() (sage.categories.regular_crystals.RegularCrystals method), 745

additional_structure() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains method), 845

additional_structure() (sage.categories.vector_spaces.VectorSpaces method), 853

additional_structure() (sage.categories.weyl_groups.WeylGroups method), 856
sage.categories.additive_magmas), 181
AdditiveMagmas.SubcategoryMethods (class in algebra_generators()
sage.categories.additive_magmas), 185
AdditiveMonoids (class in algebra_generators()
sage.categories.additive_monoids), 186
AdditiveMonoids.Homsets (class in algebra_generators()
sage.categories.additive_monoids), 186
AdditiveMonoids.ParentMethods (class in algebra_generators()
sage.categories.additive_monoids), 187
AdditiveSemigroups (class in algebra_generators()
sage.categories.additive_semigroups), 187
AdditiveSemigroups.Algebras (class in algebra_generators()
sage.categories.additive_semigroups), 188
AdditiveSemigroups.Algebras.ParentMethods (class in algebra_generators()
sage.categories.additive_semigroups), 188
AdditiveSemigroups.CartesianProducts (class in algebra_generators()
sage.categories.additive_semigroups), 189
AdditiveSemigroups.Homsets (class in algebra_generators()
sage.categories.additive_semigroups), 189
AdditiveSemigroups.ParentMethods (class in algebra_generators()
sage.categories.additive_semigroups), 189
AdditiveUnital (sage.categories.additive_semigroups.AdditiveSemigroups.attribute), 188
AdditiveUnital () (sage.categories.additive_magmas.AdditiveMagmas.SubcategoryMethods method), 185
algebra_generators()
adjoint_matrix() (sage.categories.additive_semigroups.AdjacentMatrix美感, 432
affine_grading() (sage.categories.additive_monoids.AffineGrading美感, 612
affine_grassmannian_elements_of_given_length() (sage.categories.additive_monoids.AffineGrassmannian美感, 192
affine_grassmannian_to_core() (sage.categories.additive_monoids.AffineGrassmannian美感, 190
affine_grassmannian_to_partition() (sage.categories.additive_monoids.AffineGrassmannian美感, 191
AffineWeylGroups (class in AlgebraicExtensionFunctor美感, 133
AffineWeylGroups.ElementMethods (class in AlgebraicExtensionFunctor美感, 133
AffineWeylGroups.ParentMethods (class in AlgebraicExtensionFunctor美感, 192
affinization() (sage.categories.additive_monoids.Affinization美感, 608
algebra() (sage.categories.additive_monoids.Algebra美感, 193
algebra() (sage.categories.additive_monoids.Algebra美感, 194
algebra() (sage.categories.additive_monoids.Algebra美感, 226
algebra() (sage.categories.additive_monoids.Algebra美感, 549
algebra() (sage.categories.additive_monoids.Algebra美感, 791
algebra() (sage.categories.additive_monoids.Algebra美感, 195
algebra() (sage.categories.additive_monoids.Algebra美感, 196
algebra() (sage.categories.additive_monoids.Algebra美感, 258
algebra() (sage.categories.additive_monoids.Algebra美感, 786
AdditiveMonoids (sage.categories.additive_monoids.AdditiveMonoids method), 184
AdditiveMonoids.Homsets (sage.categories.additive_monoids.AdditiveMonoids method), 188
AdditiveMonoids.ParentMethods (sage.categories.additive_monoids.AdditiveMonoids method), 187
AdditiveSemigroups (sage.categories.additive_monoids.AdditiveSemigroups method), 187
AdditiveSemigroups.Algebras (sage.categories.additive_monoids.AdditiveSemigroups method), 188
AdditiveSemigroups.CartesianProducts (sage.categories.additive_monoids.AdditiveSemigroups method), 189
AdditiveSemigroups.Homsets (sage.categories.additive_monoids.AdditiveSemigroups method), 189
AdditiveSemigroups.ParentMethods (sage.categories.additive_monoids.AdditiveSemigroups method), 189
AdditiveUnital (sage.categories.additive_monoids.AdditiveUnital method), 188
AdditiveUnital () (sage.categories.additive_monoids.AdditiveUnital method), 185
algebra_generators()
adjoint_matrix() (sage.categories.additive_monoids.AdjacentMatrix美感, 432
affine_grading() (sage.categories.additive_monoids.AffineGrading美感, 612
affine_grassmannian_elements_of_given_length() (sage.categories.additive_monoids.AffineGrassmannian美感, 192
affine_grassmannian_to_core() (sage.categories.additive_monoids.AffineGrassmannian美感, 190
affine_grassmannian_to_partition() (sage.categories.additive_monoids.AffineGrassmannian美感, 191
AffineWeylGroups (class in AlgebraicExtensionFunctor美感, 133
AffineWeylGroups.ElementMethods (class in AlgebraicExtensionFunctor美感, 133
AffineWeylGroups.ParentMethods (class in AlgebraicExtensionFunctor美感, 192
affinization() (sage.categories.additive_monoids.Affinization美感, 608
algebra() (sage.categories.additive_monoids.Algebra美感, 193
algebra() (sage.categories.additive_monoids.Algebra美感, 194
algebra() (sage.categories.additive_monoids.Algebra美感, 226
algebra() (sage.categories.additive_monoids.Algebra美感, 549
algebra() (sage.categories.additive_monoids.Algebra美感, 791
algebra() (sage.categories.additive_monoids.Algebra美感, 195
algebra() (sage.categories.additive_monoids.Algebra美感, 196
algebra() (sage.categories.additive_monoids.Algebra美感, 258
algebra() (sage.categories.additive_monoids.Algebra美感, 786
<table>
<thead>
<tr>
<th>Category, Method, Class, Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algebras.ElementMethods</td>
</tr>
<tr>
<td>Algebras.Quotients</td>
</tr>
<tr>
<td>Algebras.Quotients.ParentMethods</td>
</tr>
<tr>
<td>Algebras.SubcategoryMethods</td>
</tr>
<tr>
<td>Algebras.TensorProducts</td>
</tr>
<tr>
<td>Algebras.TensorProducts.ElementMethods</td>
</tr>
<tr>
<td>Algebras.TensorProducts.ParentMethods</td>
</tr>
<tr>
<td>AlgebrasCategory</td>
</tr>
<tr>
<td>AlgebrasCategory.ParentMethods</td>
</tr>
<tr>
<td>AlgebrasWithBasis</td>
</tr>
<tr>
<td>AlgebrasWithBasis.CartesianProducts</td>
</tr>
<tr>
<td>AlgebrasWithBasis.CartesianProducts.ParentMethods</td>
</tr>
<tr>
<td>AlgebrasWithBasis.ElementMethods</td>
</tr>
<tr>
<td>AlgebrasWithBasis.ParentMethods</td>
</tr>
<tr>
<td>AlgebrasWithBasis.TensorProducts</td>
</tr>
<tr>
<td>AlgebrasWithBasis.TensorProducts.ElementMethods</td>
</tr>
<tr>
<td>AlgebrasWithBasis.TensorProducts.ParentMethods</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>an_element()</td>
</tr>
<tr>
<td>an_element()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
<tr>
<td>ambient()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>983</td>
</tr>
</tbody>
</table>
Category Framework, Release 10.1

method), 522
an_instance() (sage.categories.groupoid.Groupoid class method), 549
apply_simple_reflection()

Analytic() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 643
apply_simple_reflection_left()

annihilator() (sage.categories.finite_dimensional_modules_with_basis.ParentMethods method), 457
apply_simple_reflection_right()

annihilator_basis() (sage.categories.finite_dimensional_modules_with_basis.ParentMethods method), 489
apply_binary_relations()

antichains() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 244
apply_baker_campbell_hausdorff()

annihilator() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ElementMethods method), 245
apply_baker_campbell_hausdorff()

antipode() (sage.categories.hopf_algebras.HopfAlgebras.ElementMethods method), 568
apply_baker_campbell_hausdorff()

antipode() (sage.categories.hopf_algebras.HopfAlgebras.ElementMethods method), 572
apply_baker_campbell_hausdorff()

antipode() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.ElementMethods method), 827
apply_baker_campbell_hausdorff()

apply_baker_campbell_hausdorff()

antipode_on_basis() (sage.categories.examples.hopf_algebras_with_basis.ElementMethods method), 914
apply_baker_campbell_hausdorff()

antipode_on_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.ParentMethods method), 100
apply_baker_campbell_hausdorff()

antipode_on_basis() (sage.categories.group_algebras.GroupAlgebras.ElementMethods method), 545
apply_baker_campbell_hausdorff()

antipode_on_basis() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ElementMethods method), 573
apply_baker_campbell_hausdorff()

Aperiodic (sage.categories.semigroups.Semigroups attribute), 765
axiom() (in module sage.categories.category_with_axiom), 100
AxiomContainer (class in sage.categories.category_with_axiom), 101
axioms() (sage.categories.category.CCategory method), 53
axioms() (sage.categories.category_with_axiom.CategoryWithAxiomMethods method), 97

AperiodicSemigroups (class in sage.categories.aperiodic_semigroups), 203
apply_conjugation_by_simple_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ElementMethods method), 242
apply_baker_campbell_hausdorff()

apply_demazure_product() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 263
apply_baker_campbell_hausdorff()

apply_multilinear_morphism() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 695
apply_baker_campbell_hausdorff()

apply_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ElementMethods method), 242
apply_baker_campbell_hausdorff()

apply_simple_projection() (sage.categories.tensor.TensorProductsCategory method), 875
Index 985
method), 218
coproduct_iterated()
coproduct_on_basis()
coproduct_iterated()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on_basis()
coproduct_on基礎

cover() (method), 818
cover_reflections() (method), 270

Index 991
Index

cw_complexes (class in sage.categories.cw_complexes), 324

cycle_index() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ElementMethods), 927
cycle_index() (sage.categories.commutative_rings.CommutativeRings.Finite.ElementMethods), 229
cycle_index() (sage.categories.commutative_rings.CommutativeRings.Finite.ParentMethods), 485
cycle_index() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 885
cycle_index() (sage.categories.finite_coxeter_groups.DihedralGroup.ElementMethods), 400
cycle_index() (sage.categories.finite_coxeter_groups.DihedralGroup.ParentMethods), 400
cycle_index() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ElementMethods), 922
cycle_index() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods), 922
cycle_index() (sage.categories.classical_crystals.ClassicalCrystals.ElementMethods), 214

demazure_character() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods), 214
demazure_lusztig_operators() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.Finite.ElementMethods), 259
demazure_lusztig_operators() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.Finite.ParentMethods), 259
demazure_operator() (sage.categories.regular_crystals.RegularCrystals.ParentMethods), 307
Index 995

Category Framework, Release 10.1

Element (sage.categories.examples.infinite_enumerated_sets.Element, 930)

Element (sage.categories.examples.manifolds.Plane attribute), 935

Endset (sage.categories.examples.infinite_enumerated_sets.Element method), 746

EmptySetError, 777

End (module sage.categories.homset, 120)

End (module sage.categories.homset, 127)

Endset (sage.categories.homset.Homset method), 888

Endsets (sage.categories.objects.Objects.SubcategoryMethods, 791)

Enumerated (sage.categories.sets_cat.Sets attribute), 791

EnumeratedSets (sage.categories.enumerated_sets, 333)

EnumeratedSets.CartesianProducts (sage.categories.enumerated_sets, 333)

EnumeratedSets.CartesianProducts.ParentMethodsExample (sage.categories.enumerated_sets, 334)

EnumeratedSets.ElementMethods (sage.categories.enumerated_sets, 334)

EnumeratedSets.ParentMethods (sage.categories.enumerated_sets, 334)

Epsilon (sage.categories.crystals.Crystals.ElementMethods, 307)

epsilon (sage.categories.crystals.Crystals.ElementMethods, 308)

epsilon (sage.categories.regular_crystals.RegularCrystals.ElementMethods, 739)
Category Framework, Release 10.1

Example (in module sage.categories.examples.manifolds), example() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis method), 599
Example (in module sage.categories.examples.monoids), example() (sage.categories.lie_conformal_algebras.LieConformalAlgebras method), 602
Example (in module sage.categories.examples.sets_with_grading), example() (sage.categories.loop_crystals.LoopCrystals method), 618
element() (sage.categories.algebras_with_basis.AlgebrasWithBasis method), 203
element() (sage.categories.category.Category method), 54
element() (sage.categories.classical_crystals.ClassicalCrystals method), 215
element() (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 240
element() (sage.categories.crystals.Crystals method), 324
element() (sage.categories.facade_sets.FacadeSets method), 865
element() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups method), 395
element() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated method), 395
element() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated.Irreducible method), 392

Extra super categories():
element() (sage.categories.additive_groups.AdditiveGroups.Finite.Algebras method), 471
element() (sage.categories.additive_groups.AdditiveGroups.ElementMethods method), 585

Extra super classes: 
element() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis method), 599
extra_super_categories()
(sage.categories.additive_magmas.AdditiveMagmas.AdditiveCommutative.Algebras
method), 175

extra_super_categories()
(sage.categories.additive_magmas.AdditiveMagmas.AdditiveCommutative.CartesianProducts
method), 175

extra_super_categories()
(sage.categories.additive_magmas.AdditiveMagmas.AdditiveCommutative.Homsets
method), 177

extra_super_categories()
method), 177

extra_super_categories()
(sage.categories.additive_monoids.AdditiveMonoids.Homsets
method), 186

extra_super_categories()
(sage.categories.additive_semigroups.AdditiveSemigroups.Algebras
method), 189

extra_super_categories()
(sage.categories.additive_semigroups.AdditiveSemigroups.CartesianProducts
method), 189

extra_super_categories()
(sage.categories.additive_semigroups.AdditiveSemigroups.Homsets
method), 186

extra_super_categories()
method), 195

extra_super_categories()
method), 195

extra_super_categories()
(sage.categories.algebras.Algebras.TensorProducts
method), 197

extra_super_categories()
(sage.categories.algebras_with_basis.Algebras.AdditiveMonoids.Homsets
method), 200

extra_super_categories()
(sage.categories.algebras_with_basis.Algebras.AdditiveSemigroups.Homsets
method), 203
extra_super_categories()
(sage.categories.finite_crystals.FiniteCrystals method), 408
extra_super_categories()
(sage.categories.finite_crystals.FiniteCrystals.TensorProducts method), 409
extra_super_categories()
(sage.categories.finite_crystals.FiniteCrystals.TensorProducts method), 407
extra_super_categories()
(sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.TensorProducts method), 413
extra_super_categories()
(sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.TensorProducts method), 463
extra_super_categories()
(sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.CartesianProducts method), 471
extra_super_categories()
(sage.categories.finite_fields.FiniteFields method), 474
extra_super_categories()
(sage.categories.finite_groups.FiniteGroups.Algebras method), 475
extra_super_categories()
(sage.categories.finite_permutation_groups.FinitePermutationGroups method), 489
extra_super_categories()
(sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups method), 520
extra_super_categories()
(sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups.Finite method), 523
extra_super_categories()
(sage.categories.graded_algebras.GradedAlgebras.SignedTensorProducts method), 524
extra_super_categories()
(sage.categories.graded_algebras.GradedAlgebras.SignedTensorProducts method), 528
extra_super_categories()
(sage.categories.graded_coalgebras.GradedCoalgebras.SignedTensorProducts method), 530
extra_super_categories()
(sage.categories.graded_lie_algebras.GradedLieAlgebras.Stratified.FiniteDimensional method), 532
extra_super_categories()
(sage.categories.graphs.Graphs.Connected method), 541
extra_super_categories()
(sage.categories.highest_weight_crystals.HighestWeightCrystals.TensorProducts method), 566
extra_super_categories()
(sage.categories.hopf_algebras.HopfAlgebras.TensorProducts method), 569
extra_super_categories()
(sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.TensorProducts method), 573
extra_super_categories()
(sage.categories.j_trivial_semigroups.JTrivialSemigroups method), 577
extra_super_categories()
(sage.categories.l_trivial_semigroups.LTrivialSemigroups method), 619
extra_super_categories()
(sage.categories.lie_algebras.LieAlgebras.FiniteDimensional method), 587
extra_super_categories()
(sage.categories.magmas.Magmas.Algebras method), 620
extra_super_categories()
(sage.categories.magmas.Magmas.Algebras method), 622
extra_super_categories()
(sage.categories.magmas.Magmas.Commutative.Algebras method), 622
extra_super_categories()
(sage.categories.magmas.Magmas.Commutative.CartesianProducts method), 622
extra_super_categories()
(sage.categories.magmas.Magmas.Unital.Algebras method), 632
extra_super_categories()
<table>
<thead>
<tr>
<th>Category Framework, Release 10.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>extra_super_categories()</td>
</tr>
<tr>
<td>(sage.categories.magas_and_additive_magmas.MagmasAndAdditiveMagmas.CartesianProducts method), 635</td>
</tr>
</tbody>
</table>

extra_super_categories()
(sage.categories.manifolds.Manifolds.AlmostComplex method), 641

extra_super_categories()
(sage.categories.manifolds.Manifolds.Analytic method), 642

extra_super_categories()
(sage.categories.manifolds.Manifolds.Smooth method), 643

extra_super_categories()
(sage.categories.metric_spaces.MetricSpaces.CartesianProducts method), 646

extra_super_categories()

extra_super_categories()
(sage.categories.quantum_group_representations.QuantumGroupRepresentations.TensorProducts method), 731

extra_super_categories()
(sage.categories.quantum_group_representations.QuantumGroupRepresentations.RTrivialSemigroups method), 737

extra_super_categories()
(sage.categories.r_trivial_semigroups.RTrivialSemigroups method), 761

extra_super_categories()
(sage.categories.regular_crystals.RegularCrystals.TensorProducts method), 744

extra_super_categories()
(sage.categories.regular_crystals.RegularCrystals.SignedTensorProducts method), 746

extra_super_categories()
(sage.categories.regular_supercrystals.RegularSuperCrystals.TensorProducts method), 746

extra_super_categories()
(sage.categories.r_trivial_semigroups.RTrivialSemigroups.Semigroups.Algebras method), 765

extra_super_categories()
(sage.categories.modules.Modules.CartesianProducts method), 652

extra_super_categories()
(sage.categories.modules.Modules.FiniteDimensional method), 653

extra_super_categories()
(sage.categories.modules.Modules.FinitelyPresented method), 653

extra_super_categories()
(sage.categories.modules.Modules.Homsets method), 655

extra_super_categories()

extra_super_categories()
(sage.categories.modules.Modules.TensorProducts method), 663

extra_super_categories()
(sage.categories.modules_with_basis.ModulesWithBasis.tensor_products method), 666

extra_super_categories()
(sage.categories.modules_with_basis.ModulesWithBasis.tensor_products method), 666

extra_super_categories()
(sage.categories.modules_with_basis.ModulesWithBasis.tensor_products method), 666

extra_super_categories()
(sage.categories.quantum_group_representations.QuantumGroupRepresentations.tensor_products method), 731

extra_super_categories()
(sage.categories.quantum_group_representations.QuantumGroupRepresentations.tensor_products method), 737

extra_super_categories()
(sage.categories.r_trivial_semigroups.RTrivialSemigroups.tensor_products method), 761

extra_super_categories()
(sage.categories.regular_crystals.RegularCrystals.tensor_products method), 744

extra_super_categories()
(sage.categories.regular_supercrystals.RegularSuperCrystals.tensor_products method), 746

extra_super_categories()
(sage.categories.semigroups.Semigroups.Algebras method), 765

extra_super_categories()
(sage.categories.semigroups.Semigroups.CartesianProducts method), 765

extra_super_categories()
(sage.categories.sets_cat.Sets.Algebras method), 780

extra_super_categories()
(sage.categories.sets_cat.Sets.CartesianProducts method), 784

extra_super_categories()
(sage.categories.sets_cat.Sets.WithRealizations method), 805

extra_super_categories()
(sage.categories.super_algebras.SuperAlgebras.SignedTensorProducts method), 824

extra_super_categories()
(sage.categories.super_algebras.SuperAlgebras.tensor_products method), 827

extra_super_categories()
(sage.categories.super_lie_conformal_algebras.SuperLieConformalAlgebras.SignedTensorProducts method), 829

extra_super_categories()
(sage.categories.super_algebras.SuperAlgebras.tensor_products method), 827

extra_super_categories()
(sage.categories.super_algebras.SuperAlgebras.tensor_products method), 827

extra_super_categories()
(sage.categories.supermodule_with_basis.tensor_products method), 826

extra_super_categories()
(sage.categories.supermodule_with_basis.tensor_products method), 826

extra_super_categories()
(sage.categories.tensor.tensor_product_with_basis.tensor_products method), 829

extra_super_categories()
(sage.categories.tensor.tensor_product_with_basis.tensor_products method), 829

extra_super_categories()
(sage.categories.tensor.tensor_product_with_basis.tensor_products method), 830

extra_super_categories()
(sage.categories.tensor.tensor_product_with_basis.tensor_products method), 830

extra_super_categories()
(sage.categories.tensor.tensor_product_with_basis.tensor_products method), 835

extra_super_categories()
Category Framework, Release 10.1

FiniteDimensionalAlgebrasWithBasis

FiniteDimensionalHopfAlgebrasWithBasis

FiniteDimensionalGradedLieAlgebrasWithBasis

FiniteDimensionalCoalgebrasWithBasis

FiniteDimensionalSemisimpleAlgebrasWithBasis

FiniteDimensionalGradedLieAlgebrasWithBasis.Commutative

FiniteDimensionalHopfAlgebrasWithBasis

FiniteEnumeratedSets
FiniteEnumeratedSets.CartesianProducts (class in sage.categories.finite.enumerated_sets), 468
FiniteEnumeratedSets.CartesianProducts.ParentMethods (class in sage.categories.finite.enumerated_sets), 468
FiniteEnumeratedSets.IsomorphicObjects (class in sage.categories.finite.enumerated_sets), 471
FiniteEnumeratedSets.IsomorphicObjects.ParentMethods (class in sage.categories.finite.enumerated_sets), 471
FiniteEnumeratedSets.ParentMethods (class in sage.categories.finite.enumerated_sets), 472
FiniteFields (class in sage.categories.finite_fields), 474
FiniteFields.ElementMethods (class in sage.categories.finite_fields), 474
FiniteFields.ParentMethods (class in sage.categories.finite_fields), 474
FiniteGroups (class in sage.categories.finite.groups), 475
FiniteGroups.Algebras (class in sage.categories.finite.groups), 475
FiniteGroups.Algebras.ParentMethods (class in sage.categories.finite.groups), 475
FiniteGroups.ElementMethods (class in sage.categories.finite.groups), 475
FiniteGroups.ParentMethods (class in sage.categories.finite.groups), 475
FiniteLatticePosets (class in sage.categories.finite.lattice.posets), 477
FiniteLatticePosets.ParentMethods (class in sage.categories.finite.lattice.posets), 477
FiniteMonoids (class in sage.categories.finite.monoids), 481
FiniteMonoids.ElementMethods (class in sage.categories.finite.monoids), 481
FiniteMonoids.ParentMethods (class in sage.categories.finite.monoids), 481
FinitePresented() (sage.categories.modules.Modules.SubcategoryMethods method), 517
FinitelyGeneratedAsLambdaBracketAlgebras (class in sage.categories.finitely_generated.lambda_bracket_algebras), 515
FinitelyGeneratedLambdaBracketAlgebras (class in sage.categories.lambda_bracket_algebras), 580
FinitelyGeneratedLambdaBracketAlgebras.Graded (class in sage.categories.lambda_bracket_algebras), 515
FinitelyGeneratedLambdaBracketAlgebras.ParentMethods (class in sage.categories.lambda_bracket_algebras), 515
FinitelyGeneratedLieConformalAlgebras (class in sage.categories.finitely_generated.lie_conformal.algebras), 516
FinitelyGeneratedLieConformalAlgebras.Graded (class in sage.categories.finitely_generated.lie_conformal.algebras), 516
FinitelyGeneratedLieConformalAlgebras.ParentMethods (class in sage.categories.finitely_generated.lie_conformal.algebras), 517
FinitelyGeneratedMagma (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 630
FinitelyGeneratedMagma (sage.categories.magas.Magmas.SubcategoryMethods method), 630
FinitelyGeneratedMagma () (sage.categories.magas.Magmas.SubcategoryMethods method), 630
FinitelyGeneratedLambdaBracketAlgebras (class in sage.categories.finitely_generated.lambda_bracket_algebras), 515
FinitelyGeneratedLieConformalAlgebras (class in sage.categories.finitely_generated.lie_conformal.algebras), 516
FinitelyGeneratedMagma (class in sage.categories.magas.Magmas.SubcategoryMethods method), 518
FinitePresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitePresented() (sage.categories.magas.Magmas.SubcategoryMethods method), 629
FinitePresented() (sage.categories.magas.Magmas.SubcategoryMethods method), 629
FinitelyGenerated() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyGenerated() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyGenerated() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyGenerated() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyGenerated() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyPresented() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyPresented() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyGenerated() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyPresented() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgebras.SubcategoryMethods method), 580
FinitelyPresented() (sage.categories.gamma.bracket.algebras.GammaBracketAlgebras.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgeas.SubcategoryMethods method), 580
FinitelyPresented() (sage.categories.gamma.bracket.algebras.GammaBracketAlgeas.SubcategoryMethods method), 518
FinitelyPresented() (sage.categories.lambda_bracket_algebras.LambdaBracketAlgeas.SubcategoryMethods method), 580
Index 1003
FinitePermutationGroups (class in sage.categories.finite_permutation_groups), 485
FinitePermutationGroups.ElementMethods (class in sage.categories.finite_permutation_groups), 485
FinitePermutationGroups.ParentMethods (class in sage.categories.finite_permutation_groups), 485
FinitePosets (class in sage.categories.finite_posets), 489
FinitePosets.ParentMethods (class in sage.categories.finite_posets), 489
FiniteSemigroups (class in sage.categories.finite_semigroups), 489
FiniteSemigroups.ElementMethods (class in sage.categories.finite_semigroups), 489
FiniteSemigroups.ParentMethods (class in sage.categories.finite_semigroups), 489
FiniteSets (class in sage.categories.finite_sets), 513
FiniteSets.ElementMethods (class in sage.categories.finite_sets), 513
FiniteSets.ParentMethods (class in sage.categories.finite_sets), 513
FiniteSets.Subquotients (class in sage.categories.finite_sets), 514
FiniteSets.OrderedByInclusion (class in sage.categories.finite_sets), 938
FiniteSets.OrderedByInclusion.Element (class in sage.categories.finite_sets), 938
FiniteWeylGroups (class in sage.categories.finite_weyl_groups), 514
FiniteWeylGroups.ElementMethods (class in sage.categories.finite_weyl_groups), 514
FiniteWeylGroups.ParentMethods (class in sage.categories.finite_weyl_groups), 514
first() (sage.categories.enumerated_setsEnumeratedSets.ElementMethods (class in sage.categories.enumerated_sets), 334
first() (sage.categories.enumerated_setsEnumeratedSets.ElementMethods (class in sage.categories.enumerated_sets), 334
first() (sage.categories.mapFormalCompositeMap (class in sage.categories.map), 110
first_descent() (sage.categories.coxeter_groupsCoxeterGroups.ElementMethods (class in sage.categories.coxeter_groups), 273
Flying() (sage.categories.category_with_axiomBlahs.SubcategoryMethods (class in sage.categories.category_with_axiom), 94
ForgetfulFunctor () (module from sage.categories.monoidsMonoids.ElementMethods (class in sage.categories.monoids), 937
ForgetfulFunctor_generic (class in sage.categories.monoids), 103
formal_series_ring() (sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis.ParentMethods (class in sage.categories.graded_algebras_with_basis), 526
FormalCoercionMorphism (class in sage.categories.morphism), 129
FormalCompositeMap (class in sage.categories.map), 349
<table>
<thead>
<tr>
<th>Function</th>
<th>Class</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_graded_conversion()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_graded_conversion()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_reduced_word()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_set()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_vector()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_vector()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>from_vector()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>full_super_categories()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
<tr>
<td>gens()</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
<td>GcdDomains.ElementMethods (class in sage.categories.graded_algebras)</td>
</tr>
</tbody>
</table>

**Index**

1005
GradedLieConformalAlgebrasCategory (class in sage.categories.graded_lie_conformal_algebras), 534

GradedModules (class in sage.categories.graded_modules), 534

GradedModules.ElementMethods (class in sage.categories.graded_modules), 535

GradedModules.ParentMethods (class in sage.categories.graded_modules), 535

GradedModulesCategory (class in sage.categories.graded_modules), 535

GradedModulesWithBasis (class in sage.categories.graded_modules_with_basis), 536

GradedModulesWithBasis.ElementMethods (class in sage.categories.graded_modules_with_basis), 536

GradedModulesWithBasis.ParentMethods (class in sage.categories.graded_modules_with_basis), 536

GradedModulesWithBasis.Quotients (class in sage.categories.graded_modules_with_basis), 540

GradedModulesWithBasis.Quotients.ElementMethods (class in sage.categories.graded_modules_with_basis), 540

GradedModulesWithBasis.Quotients.ParentMethods (class in sage.categories.graded_modules_with_basis), 540

GradedPartitionModule (class in sage.categories.examples.graded_partition_modules), 924

group_generators() (sage.categories.grading, 250)
group_generators() (sage.categories.grading, 549)
group_generators() (sage.categories.groups.Groups.CartesianProducts, 549)
group_generators() (sage.categories.groups.Groups.ParentMethods, 557)

GroupAlgebraFunctor (class in sage.categories.algebra_functor, 883)

GroupAlgebras (class in sage.categories.algebra_functor, 543)

GroupAlgebras.ElementMethods (class in sage.categories.algebra_functor, 544)

GroupAlgebras.ParentMethods (class in sage.categories.algebra_functor, 545)

Groupoid (class in sage.categories.groupoid, 548)

Groups (class in sage.categories.groups, 549)

Groups.CartesianProducts (class in sage.categories.groups, 549)

Groups.CartesianProducts.ParentMethods (class in sage.categories.groups, 549)

Groups.Commutative (class in sage.categories.groups, 551)

Groups.ElementMethods (class in sage.categories.groups, 551)

Groups.ParentMethods (class in sage.categories.groups, 552)

Groups.Topological (class in sage.categories.groups, 558)

GradedModulesWithBasis.Quotients.ElementMethods (class in sage.categories.graded_modules_with_basis), 540

Graphs (class in sage.categories.graphs), 541

Graphs.Connected (class in sage.categories.graphs), 541

graph() (sage.categories.algebra_functor, 883)

group() (sage.categories.additive_groups, 173)

group() (sage.categories.additive_groups.additive_groups, 547)

group() (sage.categories.algebra_functor, 883)

group() (sage.categories.additive_groups.additive_groups, 547)

HeckeModules (class in sage.categories.hecke_modules, 559)

HeckeModules.Homsets (class in sage.categories.hecke_modules, 559)

HeckeModules.Homsets.ParentMethods (class in sage.categories.hecke_modules, 559)
HeckeModules.ParentMethods (class in sage.categories.hecke_modules), 560
highest_weight_vector() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods, method), 563
highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods, method), 563
highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods, method), 566
highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods, method), 566
highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods, method), 566
HighestWeightCrystalHomset (class in sage.categories.highest_weight_crystals), 560
HighestWeightCrystalMorphism (class in sage.categories.highest_weight_crystals), 560
HighestWeightCrystalOfTypeA (class in sage.categories.examples.crystals), 903
HighestWeightCrystalOfTypeA.Element (class in sage.categories.examples.crystals), 904
HighestWeightCrystals (class in sage.categories.highest_weight_crystals), 561
HighestWeightCrystals.ElementMethods (class in sage.categories.highest_weight_crystals), 561
HighestWeightCrystals.ParentMethods (class in sage.categories.highest_weight_crystals), 561
HighestWeightCrystals.TensorProducts (class in sage.categories.highest_weight_crystals), 566
HighestWeightCrystals.TensorProducts.ParentMethods (class in sage.categories.highest_weight_crystals), 566
holomorph() (sage.categories.groups.Groups.ParentMethods, method), 557
Hom() (in module sage.categories.homset), 122
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 361
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 361
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 368
homogeneous_component_as_submodule() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ParentMethods, method), 201
hochschild_complex() (sage.categories.algebras_with_basis.AlgebrasWithBasis.Morphism, method), 848
homogeneous_component_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 430
homogeneous_degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 441
homology() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ParentMethods, method), 441
homogeneous_component_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 430
homogeneous_degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 441
homology() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ParentMethods, method), 441
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 361
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 361
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods, method), 368
homogeneous_component_as_submodule() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ParentMethods, method), 201
Category Framework, Release 10.1

InfiniteEnumeratedSets

InfiniteEnumeratedSets.ParentMethods

InfinitePolynomialFunctor

inject_shorthands()
inner_derivations_basis()
invariant_module()
Inverse
inverse()
inversion_arrangement()

Irreducible()
irreducible_component_index_sets()
irreducible_components()
iminversions_as_reflections()

is_abelian()
Category Framework, Release 10.1

LambdaBracketAlgebras.ElementMethods
LambdaBracketAlgebras.ParentMethods
LambdaBracketAlgebras.SubcategoryMethods
LambdaBracketAlgebrasWithBasis.ElementMethods
LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra.Graded
LatticePosets
LaurentPolynomialFunctor

leading_item() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods), 669
leading_monomial() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods), 669
leading_support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods), 670
leading_term() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods), 670
left_base_ring() (sage.categories.bimodules.Bimodules.ElementMethods), 211
left_domain() (sage.categories.action.Action method), 170
left_inversions_as_reflections()
LambdaBracketAlgebrasWithBasis.FinitelyGeneratedAsLambdaBracketAlgebra.Graded
LeftModules (class in sage.categories.left_modules), 584
LeftRegularBand.Element
LeftZeroSemigroup.Element

last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ElementMethods), 469
last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ElementMethods), 472
latex() (sage.categories.crystals.Crystals.ParentMethods), 918
latex_file() (sage.categories.crystals.Crystals.ParentMethods), 920
LatticePosets (class in sage.categories.lattice_posets), 583
LatticePosets.ParentMethods (class in sage.categories.lattice_posets), 583
LaurentPolynomialFunctor (class in sage.categories.pushout), 148

1cm() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods), 941
1cm() (sage.categories.fields.Fields.ElementMethods), 329
1cm() (sage.categories.fields.Fields.ElementMethods), 345
1cm() (sage.categories.fields.Fields.ElementMethods), 724

1e() (sage.categories.examples.posets.FiniteSetsOrderedByInclusion.ElementMethods), 671
1e() (sage.categories.examples.posets.FiniteSetsOrderedByInclusion.ElementMethods), 610
1e() (sage.categories.examples.posets.FiniteSetsOrderedByInclusion.ElementMethods), 552

leading_coefficient() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods), 668
lie_algebra_generators() (sage.categories.groups.Groups method), 552
lie_algebra_generators()
MetricSpaces.Complete (class in sage.categories.metric_spaces), 646
MetricSpaces.Complete.CartesianProducts (class in sage.categories.metric_spaces), 646
MetricSpaces.ElementMethods (class in sage.categories.metric_spaces), 647
MetricSpaces.Homsets (class in sage.categories.metric_spaces), 647
MetricSpaces.Homsets.ElementMethods (class in sage.categories.metric_spaces), 647
MetricSpaces.ParentMethods (class in sage.categories.metric_spaces), 648
MetricSpaces.SubcategoryMethods (class in sage.categories.metric_spaces), 648
MetricSpaces.WithRealizations (class in sage.categories.metric_spaces), 649
MetricSpaces.Category (class in sage.categories.metric_spaces), 649
min_demazure_product_greater() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 280
ModularAbelianVarieties (class in sage.categories.modular_abelian_varieties), 650
ModularAbelianVarieties.Homsets (class in sage.categories.modular_abelian_varieties), 650
ModularAbelianVarieties.Homsets.Endset (class in sage.categories.modular_abelian_varieties), 650
module
  sage.categories.action, 169
  sage.categories.additive_groups, 172
  sage.categories.additive_magmas, 174
  sage.categories.additive_monoids, 186
  sage.categories.additive_semigroups, 187
  sage.categories.affine_weyl_groups, 190
  sage.categories.algebra_functor, 877
  sage.categories.algebra_ideals, 193
  sage.categories.algebra_modules, 194
  sage.categories.algebras, 195
  sage.categories.algebras_with_basis, 198
  sage.categories.all, 1
  sage.categories.aperiodic_semigroups, 203
  sage.categories.associative_algebras, 204
  sage.categories.bialgebras, 204
  sage.categories.bialgebras_with_basis, 206
  sage.categories.bimodules, 211
  sage.categories.cartesian_product, 872
  sage.categories.category, 31
  sage.categories.category_singleton, 964
  sage.categories.category_types, 961
  sage.categories.category_with_axiom, 69
  sage.categories.classical_crystals, 212
  sage.categories.coalgebras, 216
  sage.categories.coalgebras_with_basis, 221
  sage.categories.coercion_methods, 970
  sage.categories.commutative_additive_groups, 224
  sage.categories.commutative_additive_monoids, 225
  sage.categories.commutative_additive_semigroups, 226
  sage.categories.commutative_algebra_ideals, 226
  sage.categories.commutative_algebra_ideals, 227
  sage.categories.commutative_ring_ideals, 227
  sage.categories.commutative_rings, 228
  sage.categories.complete_discretevaluation, 283
  sage.categories.complex_reflection_groups, 238
  sage.categories.complex_reflection_or_generalized_coxeter_groups, 240
  sage.categories.covariant Functorial_construction, 867
  sage.categories.coxeter_group_algebras, 258
  sage.categories.coxeter_groups, 261
  sage.categories.crystals, 301
  sage.categories.cw_complexes, 324
  sage.categories.distributive_magmas_and_additive_magmas, 330
  sage.categories.division_rings, 331
  sage.categories.domains, 332
  sage.categories.dual, 877
  sage.categories.enumerated_sets, 333
  sage.categories.euclidean_domains, 340
  sage.categories.examples.algebras_with_basis, 899
  sage.categories.examples.commutative_additive_monoids, 900
  sage.categories.examples.commutative_additive_semigroups, 901
  sage.categories.examples.coxeter_groups, 903
  sage.categories.examples.crystals, 905
  sage.categories.examples.cw_complexes, 906
  sage.categories.examples.facade_sets, 906
sage.categories.examples.finite_dimensional_algebras_with_basis
sage.categories.examples.finite_dimensional_coalgebras_with_basis
sage.categories.examples.finite_dimensional_lie_algebras_with_basis
sage.categories.examples.finite_enumerated_sets
sage.categories.examples.finite_monoids
sage.categories.examples.finite_semigroups
sage.categories.examples.finite_weyl_groups
sage.categories.examples.graded_connected_hopf_algebras_with_basis
sage.categories.examples.graded_modules_with_basis
sage.categories.examples.graphs
sage.categories.examples.hopf_algebras_with_basis
sage.categories.examples.infinite_enumerated_sets
sage.categories.examples.lie_algebras
sage.categories.examples.lie_algebras_with_basis
sage.categories.examples.magmas
sage.categories.examples.manifolds
sage.categories.examples.monoids
sage.categories.examples.posets
sage.categories.examples.semigroups
sage.categories.examples.semigroups_cython
sage.categories.examples.sets_cat
sage.categories.examples.sets_with_grading
sage.categories.examples.with_realizations
sage.categories.facade_sets
sage.categories.fields
sage.categories.filtered_algebras
sage.categories.filtered_algebras_with_basis
sage.categories.filtered_modules
sage.categories.filtered_modules_with_basis
sage.categories.finite_complex_reflection_groups
sage.categories.finite_coxeter_groups
sage.categories.finite_crystals
sage.categories.finitely_generated_lambda_bracket_algebras
sage.categories.finitely_generated_lie_conformal_algebras
sage.categories.finitely_generated_magmas
sage.categories.finitely_generated_semigroups
sage.categories.function_fields
sage.categories.gcd_domains
sage.categories.generalized_coxeter_groups
sage.categories.graded_algebras
sage.categories.graded_algebras_with_basis
sage.categories.graded_bialgebras
sage.categories.graded_bialgebras_with_basis
sage.categories.graded_coalgebras
sage.categories.graded_coalgebras_with_basis
sage.categories.graded_hopf_algebras
sage.categories.graded_hopf_algebras_with_basis
sage.categories.graded_hopf_algebras_with_basis
sage.categories.graded_lie_algebras
sage.categories.graded_lie_algebras_with_basis
sage.categories.graded_modules
sage.categories.graded_modules_with_basis
sage.categories.graded_hopf_algebras
sage.categories.graded_hopf_algebras_with_basis
sage.categories.infinite_enumerated_sets
sage.categories.infinite_sets
sage.categories.with_realizations
sage.categories.fields
sage.categories.filtered_algebras
sage.categories.filtered_algebras_with_basis
sage.categories.filtered_modules
sage.categories.filtered_modules_with_basis
sage.categories.finite_complex_reflection_groups
sage.categories.finite_coxeter_groups
sage.categories.finite_crystals
sage.categories.finitely_generated_lambda_bracket_algebras
sage.categories.finitely_generated_lie_conformal_algebras
sage.categories.finitely_generated_magmas
sage.categories.finitely_generated_semigroups
sage.categories.function_fields
sage.categories.gcd_domains
sage.categories.generalized_coxeter_groups
sage.categories.graded_algebras
sage.categories.graded_algebras_with_basis
sage.categories.graded_bialgebras
sage.categories.graded_bialgebras_with_basis
sage.categories.graded_coalgebras
sage.categories.graded_coalgebras_with_basis
sage.categories.graded_hopf_algebras
sage.categories.graded_hopf_algebras_with基礎
sage.categories.graded_lie_algebras, 532
sage.categories.graded_lie_algebras_with_basis, 533
sage.categories.graded_lie_conformal_algebras, 534
sage.categories.graded_modules, 534
sage.categories.graded_modules_with_basis, 536
sage.categories.graphs, 541
sage.categories.group_algebras, 543
sage.categories.groupoid, 548
sage.categories.groups, 549
sage.categories.h_trivial_semigroups, 574
sage.categories.hecke_modules, 559
sage.categories.highest_weight_crystals, 560
sage.categories.homset, 120
sage.categories.homsets, 886
sage.categories.hopf_algebras, 567
sage.categories.hopf_algebras_with_basis, 570
sage.categories.infinite_enumerated_sets, 575
sage.categories.integral_domains, 576
sage.categories.isomorphic_objects, 885
sage.categories.j_trivial_semigroups, 577
sage.categories.kac_moody_algebras, 577
sage.categories.l_trivial_semigroups, 619
sage.categories.lambda_bracket_algebras, 578
sage.categories.lambda_bracket_algebras_with_basis, 581
sage.categories.lattice_posets, 583
sage.categories.left_modules, 584
sage.categories.lie_algebras, 584
sage.categories.lie_algebras_with_basis, 597
sage.categories.lie_conformal_algebras, 600
sage.categories.lie_conformal_algebras_with_basis, 603
sage.categories.lie_groups, 605
sage.categories.loop_crystals, 606
sage.categories.magmas, 619
sage.categories.magmas_and_additive_magmas, 635
sage.categories.magmatic_algebras, 637
sage.categories.manifolds, 641
sage.categories.map, 109
sage.categories.matrix_algebras, 645
sage.categories.metric_spaces, 645
sage.categories.modular_abelian_varieties, 650
sage.categories.modules, 650
sage.categories.modules_with_basis, 664
sage.categories.monoid_algebras, 698
sage.categories.monoids, 698
sage.categories.morphism, 128
sage.categories.number_fields, 706
sage.categories.objects, 708
sage.categories.partially_ordered_monoids, 710
sage.categories.permutation_groups, 710
sage.categories.pointed_sets, 711
sage.categories.polyhedra, 711
sage.categories.poor_man_map, 970
sage.categories.posets, 712
sage.categories.primer, 3
sage.categories.principal_ideal_domains, 721
sage.categories.pushout, 133
sage.categories.quantum_group_representations, 729
sage.categories.quotient_fields, 722
sage.categories.quotients, 884
sage.categories.r_trivial_semigroups, 761
sage.categories.realizations, 890
sage.categories.regular_crystals, 737
sage.categories.regular_supercrystals, 745
sage.categories.right_modules, 747
sage.categories.ring_ideals, 747
sage.categories.rings, 748
sage.categories.rngs, 760
sage.categories.schemes, 761
sage.categories.semisimple_algebras, 776
sage.categories.sets_cat, 777
sage.categories.sets_with_grading, 806
sage.categories.sets_with_partial_maps, 809
sage.categories.shephard_groups, 809
sage.categories.signed_tensor, 876
sage.categories.simplicial_complexes, 810
sage.categories.simplicial_sets, 812
sage.categories.subobjects, 885
sage.categories.subquotients, 884
sage.categories.supercrystals, 823
sage.categories.super_algebras, 825
sage.categories.super_algebras_with_basis, 827
sage.categories.super_hopf_algebras, 828
sage.categories.super_modules, 829
sage.categories.super_modules_with_basis, 832
NaiveCrystal.Element (class in sage.categories.examples.crystals), 905
Morphism (class in sage.categories.morphism), 130
morphism() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods), 447
morphism_class() (sage.categories.category.Category method), 60
multiplication_table() (sage.categories.mags.Magmas.ParentMethods.multiplcative_order()), 623
multiplicative_order() (sage.categories.monoids.Monoids.CartesianProducts.method), 701
MultiPolynomialFunctor (class in sage.categories.pushout), 151
MultivariateConstructionFunctor (class in sage.categories.pushout), 152
MyGroupAlgebra (class in sage.categories.examples.hopf_algebras_with_basis), 928

N
NaiveCrystal (class in sage.categories.examples.crystals), 904

\textbf{Category Framework, Release 10.1}
PrimeNumbers_Abstract (class in sage.categories.examples.sets_cat), 947
PrimeNumbers_Abstract.Element (class in sage.categories.examples.sets_cat), 948
PrimeNumbers_Facade (class in sage.categories.examples.sets_cat), 949
PrimeNumbers_Inherits (class in sage.categories.examples.sets_cat), 950
PrimeNumbers_Inherits.Element (class in sage.categories.examples.sets_cat), 952
PrimeNumbers_Wrapper (class in sage.categories.examples.sets_cat), 953
principal_ideal() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods), 425
principal_lower_set() (sage.categories.posets.Posets.ParentMethods), 720
principal_order_filter() (sage.categories.posets.Posets.ParentMethods), 720
principal_order_ideal() (sage.categories.posets.Posets.ParentMethods), 720
principal_upper_set() (sage.categories.posets.Posets.ParentMethods), 720
PrincipalIdealDomains (class in sage.categories.principal_ideal_domains), 721
PrincipalIdealDomains.ElementMethods (class in sage.categories.principal_ideal_domains), 722
PrincipalIdealDomains.ParentMethods (class in sage.categories.principal_ideal_domains), 722
print_compare() (in module sage.categories.sets_cat), 806
product() (sage.categories.magma.Magma.ParentMethods), 621
product() (sage.categories.magma.Magma.Subquotients.ParentMethods), 626
product_from_element_class_mul() (sage.categories.magma.Magma.ParentMethods), 626
product_on_basis() (sage.categories.additive_semigroups.AdditiveSemigroups.Algebras.ParentMethods), 188
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteDimensionalFiniteGroupAlgebras.ParentMethods), 202
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteGroupAlgebras.ParentMethods), 899
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteGroupRings.ParentMethods), 910
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRings.ParentMethods), 924
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingParent.Methods), 929
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingMethods), 933
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingMethods), 957
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingMethods), 527
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingMethods), 640
product_on_basis() (sage.categories.algebras_with_basis.Algebras.UInt32Order.FiniteRingsWithBaseRingMethods), 764
product_space() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods), 448
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.FinitePermutationGroup.ElementMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.FinitePermutationGroup.ElementMethods), 487
profile() (sage.categories.finite_permutation_groups.FinitePermutationGroups.FinitePermutationGroup.ElementMethods), 487
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 917
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 920
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 922
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 935
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 940
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 942
product() (sage.categories.finite_monoids.finite_monoids.AdditiveMonoids.ElementMethods), 942
Category Framework, Release 10.1

pushout() (in module sage.categories.pushout), 160
pushout() (sage.categories.pushout.ConstructionFunctor method), 143
pushout_lattice() (in module sage.categories.pushout), 167

Q
q() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 730
q_dimension() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 611
quantum_bruhat_graph() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 863
quantum_bruhat_successors() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 857
QuantumGroupRepresentations (class in sage.categories.quantum_group_representations), 729
QuantumGroupRepresentations.ParentMethods (class in sage.categories.quantum_group_representations), 729
QuantumGroupRepresentations.TensorProducts (class in sage.categories.quantum_group_representations), 730
QuantumGroupRepresentations.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 730
QuantumGroupRepresentations.WithBasis (class in sage.categories.quantum_group_representations), 731
QuantumGroupRepresentations.WithBasis.ElementMethods (class in sage.categories.quantum_group_representations), 731
QuantumGroupRepresentations.WithBasis.ParentMethods (class in sage.categories.quantum_group_representations), 733
QuantumGroupRepresentations.WithBasis.TensorProducts (class in sage.categories.quantum_group_representations), 734
QuantumGroupRepresentations.WithBasis.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 774
r() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 611
R_matrix() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 607
radical() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 425
radical() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 449
radical() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 425
radical() (sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods method), 467
radical() (sage.categories.semisimple_algebras.SemisimpleAlgebras.ParentMethods method), 774
random_element() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 337
random_element() (sage.categories.enumerated_sets.FiniteEnumeratedSets.ParentMethods method), 329
random_element() (sage.categories.euclidean_domains.EuclideanDomains.ParentMethods method), 341

QuotientFields (class in sage.categories.quotient_fields), 722
QuotientFields.ElementMethods (class in sage.categories.quotient_fields), 722
QuotientFields.ParentMethods (class in sage.categories.quotient_fields), 729
QuotientFields.Quotients() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 796
QuotientsCategory (class in sage.categories.quotients), 884
R
quo() (sage.categories.rings.Rings.ParentMethods method), 776
quo_rem() (sage.categories.euclidean_domains.EuclideanDomains.ElementMethods method), 341
regular_representation()

RegularCrystals (class in sage.categories.regular_crystals), 737
RegularCrystals.ElementMethods (class in sage.categories.regular_crystals), 738
RegularCrystals.MorphismMethods (class in sage.categories.regular_crystals), 741
RegularCrystals.ParentMethods (class in sage.categories.regular_crystals), 742
RegularCrystals.TensorProducts (class in sage.categories.regular_crystals), 744

RegularLoopCrystals (class in sage.categories.loop_crystals), 618
RegularLoopCrystals.ElementMethods (class in sage.categories.loop_crystals), 618

RegularSuperCrystals (class in sage.categories.regular_super_crystals), 745
RegularSuperCrystals.ElementMethods (class in sage.categories.regular_super_crystals), 746

required_methods() (sage.categories.category.Category method), 62
residue_field() (sage.categories.discrete_valuation.DiscreteValuations.ParentMethods method), 328

retract() (sage.categories.examples.finite_enumerated_sets.IsomorphicObjectOfFiniteEnumeratedSet method), 916
retract() (sage.categories.examples.finite_enumerated_sets.QuotientOfFiniteEnumeratedSet method), 943

reversed() (sage.categories.homset.Homset method), 127

rowmotion() (sage.categories.finite_posets.FinitePosets.ElementMethods method), 507
rowmotion_orbit_iter() (sage.categories.finite_posets.FinitePosets.ElementMethods method), 507
rowmotion_orbits() (sage.categories.finite_posets.FinitePosets.ElementMethods method), 508
rowmotion_orbits_plots() (sage.categories.finite_posets.FinitePosets.ElementMethods method), 509

RTrivial (sage.categories.semigroups.SemigroupsAttributeName), 771
RTrivial() (sage.categories.semigroups.Semigroups.SubcategoryMethods method), 773
RTrivial_extra_super_categories() (sage.categories.l_trivial_semigroups.LTrivialSemigroups.ElementMethods method), 619

RTrivialSemigroups (class in sage.categories.l_trivial_semigroups), 761
RTrivialSupercrystals (class in sage.categories.l_trivial_supercrystals), 762

S

s() (sage.categories.crystals.Crystals.ElementMethods method), 310
s() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ElementMethods method), 611

sage.categories.algebra_functor module, 877
sage.categories.algebra_ideals module, 1028
sage.categories.additive_groups module, 172
sage.categories.additive_magmas module, 174
sage.categories.additive_monoids module, 186
sage.categories.affine_weyl_groups module, 187
sage.categories.affine_weyl_groups module, 190
sage.categories.algebra_functor module, 877
sage.categories.algebra_ideals module, 1028
<table>
<thead>
<tr>
<th>Category Framework, Release 10.1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sage.categories.algebra_modules</strong></td>
</tr>
<tr>
<td>module, 193</td>
</tr>
<tr>
<td><strong>sage.categories.algebras</strong></td>
</tr>
<tr>
<td>module, 194</td>
</tr>
<tr>
<td><strong>sage.categories.algebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 198</td>
</tr>
<tr>
<td><strong>sage.categories.all</strong></td>
</tr>
<tr>
<td>module, 1</td>
</tr>
<tr>
<td><strong>sage.categories.aperiodic_semigroups</strong></td>
</tr>
<tr>
<td>module, 203</td>
</tr>
<tr>
<td><strong>sage.categories.associative_algebras</strong></td>
</tr>
<tr>
<td>module, 204</td>
</tr>
<tr>
<td><strong>sage.categories.bialgebras</strong></td>
</tr>
<tr>
<td>module, 204</td>
</tr>
<tr>
<td><strong>sage.categories.bialgebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 206</td>
</tr>
<tr>
<td><strong>sage.categories.bimodules</strong></td>
</tr>
<tr>
<td>module, 211</td>
</tr>
<tr>
<td><strong>sage.categories.cartesian_product</strong></td>
</tr>
<tr>
<td>module, 872</td>
</tr>
<tr>
<td><strong>sage.categories.category</strong></td>
</tr>
<tr>
<td>module, 31</td>
</tr>
<tr>
<td><strong>sage.categories.category_cy_helper</strong></td>
</tr>
<tr>
<td>module, 967</td>
</tr>
<tr>
<td><strong>sage.categories.category_singleton</strong></td>
</tr>
<tr>
<td>module, 964</td>
</tr>
<tr>
<td><strong>sage.categories.category_types</strong></td>
</tr>
<tr>
<td>module, 961</td>
</tr>
<tr>
<td><strong>sage.categories.category_with_axiom</strong></td>
</tr>
<tr>
<td>module, 69</td>
</tr>
<tr>
<td><strong>sage.categories.classical_crystals</strong></td>
</tr>
<tr>
<td>module, 212</td>
</tr>
<tr>
<td><strong>sage.categories.coalgebras</strong></td>
</tr>
<tr>
<td>module, 216</td>
</tr>
<tr>
<td><strong>sage.categories.coalgebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 221</td>
</tr>
<tr>
<td><strong>sage.categories.coercion_methods</strong></td>
</tr>
<tr>
<td>module, 970</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_additive_groups</strong></td>
</tr>
<tr>
<td>module, 224</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_additive_monoids</strong></td>
</tr>
<tr>
<td>module, 225</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_additive_semigroups</strong></td>
</tr>
<tr>
<td>module, 226</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_algebra_ideals</strong></td>
</tr>
<tr>
<td>module, 226</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_algebras</strong></td>
</tr>
<tr>
<td>module, 227</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_ring_ideals</strong></td>
</tr>
<tr>
<td>module, 227</td>
</tr>
<tr>
<td><strong>sage.categories.commutative_rings</strong></td>
</tr>
<tr>
<td>module, 228</td>
</tr>
<tr>
<td><strong>sage.categories.complete_discrete_valuation</strong></td>
</tr>
<tr>
<td>module, 234</td>
</tr>
<tr>
<td><strong>sage.categories.complex_reflection_groups</strong></td>
</tr>
<tr>
<td>module, 238</td>
</tr>
<tr>
<td><strong>sage.categories.complex_reflection_or_generalized_coxeter_groups</strong></td>
</tr>
<tr>
<td>module, 240</td>
</tr>
<tr>
<td><strong>sage.categories.covariant_functorial_construction</strong></td>
</tr>
<tr>
<td>module, 867</td>
</tr>
<tr>
<td><strong>sage.categories.coxeter_group_algebras</strong></td>
</tr>
<tr>
<td>module, 258</td>
</tr>
<tr>
<td><strong>sage.categories.coxeter_groups</strong></td>
</tr>
<tr>
<td>module, 261</td>
</tr>
<tr>
<td><strong>sage.categories.crystals</strong></td>
</tr>
<tr>
<td>module, 301</td>
</tr>
<tr>
<td><strong>sage.categories.cw_complexes</strong></td>
</tr>
<tr>
<td>module, 324</td>
</tr>
<tr>
<td><strong>sage.categories.discrete_valuation</strong></td>
</tr>
<tr>
<td>module, 327</td>
</tr>
<tr>
<td><strong>sage.categories.distributive_magmas_and_additive_magmas</strong></td>
</tr>
<tr>
<td>module, 330</td>
</tr>
<tr>
<td><strong>sage.categories.division_rings</strong></td>
</tr>
<tr>
<td>module, 331</td>
</tr>
<tr>
<td><strong>sage.categories.domains</strong></td>
</tr>
<tr>
<td>module, 332</td>
</tr>
<tr>
<td><strong>sage.categories.dual</strong></td>
</tr>
<tr>
<td>module, 877</td>
</tr>
<tr>
<td><strong>sage.categories.enumerated_sets</strong></td>
</tr>
<tr>
<td>module, 333</td>
</tr>
<tr>
<td><strong>sage.categories.euclidean_domains</strong></td>
</tr>
<tr>
<td>module, 340</td>
</tr>
<tr>
<td><strong>sage.categories.examples.algebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 899</td>
</tr>
<tr>
<td><strong>sage.categories.examples.commutative_additive_monoids</strong></td>
</tr>
<tr>
<td>module, 900</td>
</tr>
<tr>
<td><strong>sage.categories.examples.commutative_additive_semigroups</strong></td>
</tr>
<tr>
<td>module, 901</td>
</tr>
<tr>
<td><strong>sage.categories.examples.coxeter_groups</strong></td>
</tr>
<tr>
<td>module, 903</td>
</tr>
<tr>
<td><strong>sage.categories.examples.crystals</strong></td>
</tr>
<tr>
<td>module, 903</td>
</tr>
<tr>
<td><strong>sage.categories.examples.cw_complexes</strong></td>
</tr>
<tr>
<td>module, 905</td>
</tr>
<tr>
<td><strong>sage.categories.examples.facade_sets</strong></td>
</tr>
<tr>
<td>module, 906</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_coxeter_groups</strong></td>
</tr>
<tr>
<td>module, 907</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_dimensional_algebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 909</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_dimensional_lie_algebras_with_basis</strong></td>
</tr>
<tr>
<td>module, 910</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_enumerated_sets</strong></td>
</tr>
<tr>
<td>module, 914</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_monoids</strong></td>
</tr>
<tr>
<td>module, 916</td>
</tr>
<tr>
<td><strong>sage.categories.examples.finite_semigroups</strong></td>
</tr>
<tr>
<td>module, 916</td>
</tr>
<tr>
<td>Module Name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>category framework, release 10.1</td>
</tr>
<tr>
<td>sage.categories.examples.finite_weyl_groups</td>
</tr>
<tr>
<td>sage.categories.examples.graded_connected_hopf_algebras_with_basis</td>
</tr>
<tr>
<td>sage.categories.examples.graded_modules_with_basis</td>
</tr>
<tr>
<td>sage.categories.examples.graphs</td>
</tr>
<tr>
<td>sage.categories.examples.hopf_algebras_with_basis</td>
</tr>
<tr>
<td>sage.categories.examples.infinite enumerated sets</td>
</tr>
<tr>
<td>sage.categories.examples.lie algebras</td>
</tr>
<tr>
<td>sage.categories.examples.lie algebras with basis</td>
</tr>
<tr>
<td>sage.categories.examples.magmas</td>
</tr>
<tr>
<td>sage.categories.examples.manifolds</td>
</tr>
<tr>
<td>sage.categories.examples.posets</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups_cython</td>
</tr>
<tr>
<td>sage.categories.examples.sets_cat</td>
</tr>
<tr>
<td>sage.categories.examples.sets_with_grading</td>
</tr>
<tr>
<td>sage.categories.examples.with_realizations</td>
</tr>
<tr>
<td>sage.categories.facade_sets</td>
</tr>
<tr>
<td>sage.categories.fields</td>
</tr>
<tr>
<td>sage.categories.filtered algebras</td>
</tr>
<tr>
<td>sage.categories.filtered algebras with basis</td>
</tr>
<tr>
<td>sage.categories.filtered modules</td>
</tr>
<tr>
<td>sage.categories.filtered modules with basis</td>
</tr>
<tr>
<td>sage.categories.finite complex reflection groups</td>
</tr>
<tr>
<td>sage.categories.finite coxeter groups</td>
</tr>
<tr>
<td>sage.categories.finite crystals</td>
</tr>
<tr>
<td>sage.categories.finite dimensional algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional bialgebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional coalgebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional graded lie algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional hopf algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional lie algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional modules with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional nilpotent lie algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite dimensional semisimple algebras with basis</td>
</tr>
<tr>
<td>sage.categories.finite enumerated sets</td>
</tr>
<tr>
<td>sage.categories.finite fields</td>
</tr>
<tr>
<td>sage.categories.finite groups</td>
</tr>
<tr>
<td>sage.categories.finite lattice posets</td>
</tr>
<tr>
<td>sage.categories.finite monoids</td>
</tr>
<tr>
<td>sage.categories.finite permutation groups</td>
</tr>
<tr>
<td>sage.categories.finitely generated lambda bracket algebras</td>
</tr>
<tr>
<td>sage.categories.finitely generated lie conformal algebras</td>
</tr>
<tr>
<td>sage.categories.finitely generated magmas</td>
</tr>
<tr>
<td>sage.categories.finitely generated semigroups</td>
</tr>
<tr>
<td>sage.categories.function fields</td>
</tr>
<tr>
<td>sage.categories.g_sets</td>
</tr>
<tr>
<td>sage.categories.gcd domains</td>
</tr>
<tr>
<td>sage.categories.generalized coxeter groups</td>
</tr>
</tbody>
</table>
SubspaceFunctor (class in sage.categories.pushout),

sum() (sage.categories.module_with_basis.ModulesWithBasis method), 532

SubcategoryFromSuperCategories (sage.categories.category_with_axiom.CategoryWithAxiomWithSuperMethods method), 228

Super (sage.categories.additive_magmas.AdditiveMagmas method), 520

Super (sage.categories.additive_monoids.AdditiveMonoids method), 100
super_categories() (sage.categories.weyl_groups.WeylGroups method), 853
SuperAlgebras (class in sage.categories.super_algebras), 823
SuperAlgebras.ParentMethods (class in sage.categories.super_algebras), 823
SuperAlgebras.SignedTensorProducts (class in sage.categories.super_algebras), 824
SuperAlgebras.SubcategoryMethods (class in sage.categories.super_algebras), 824
SuperAlgebrasWithBasis (class in sage.categories.super_algebras_with_basis), 825
SuperAlgebrasWithBasis.ElementMethods (class in sage.categories.super_algebras_with_basis), 825
SuperAlgebrasWithBasis.ParentMethods (class in sage.categories.super_algebras_with_basis), 826
SuperAlgebrasWithBasis.SignedTensorProducts (class in sage.categories.super_algebras_with_basis), 826
Supercommutative (sage.categories.super_algebras.SuperAlgebras attribute), 824
Supercommutative() (sage.categories.algebras.Algebras.SubcategoryMethods method), 197
Supercommutative() (sage.categories.super_algebras.SuperAlgebras.SubcategoryMethods method), 824
SupercommutativeAlgebras (class in sage.categories.supercommutative_algebras), 834
SupercommutativeAlgebras.SignedTensorProducts (class in sage.categories.supercommutative_algebras), 835
SupercommutativeAlgebras.WithBasis (class in sage.categories.supercommutative_algebras), 835
SupercommutativeAlgebras.WithBasis.ElementMethods (class in sage.categories.supercommutative_algebras), 835
Supercommutator() (sage.categories.super_algebras_with_basis.SuperAlgebras.ElementMethods), 825
SuperCrystals (class in sage.categories.supercrystals), 835
SuperCrystals.Finite (class in sage.categories.supercrystals), 835
SuperCrystals.Finite.ElementMethods (class in sage.categories.supercrystals), 835
SuperCrystals.Finite.ParentMethods (class in sage.categories.supercrystals), 836
SuperHopfAlgebrasWithBasis (class in sage.categories.super_hopf_algebras_with_basis), 827
SuperHopfAlgebrasWithBasis.ElementMethods (class in sage.categories.super_hopf_algebras_with_basis), 827
SuperHopfAlgebrasWithBasis.ParentMethods (class in sage.categories.super_hopf_algebras_with_basis), 827
SuperLieConformalAlgebras (class in sage.categories.super_lie_conformal_algebras), 828
SuperLieConformalAlgebras.ElementMethods (class in sage.categories.super_lie_conformal_algebras), 828
SuperLieConformalAlgebras.Graded (class in sage.categories.super_lie_conformal_algebras), 828
SuperLieConformalAlgebras.ParentMethods (class in sage.categories.super_lie_conformal_algebras), 828
SuperModules (class in sage.categories.super_modules), 829
SuperModules.ElementMethods (class in sage.categories.super_modules), 829
SuperModules.ParentMethods (class in sage.categories.super_modules), 830
SuperModules.Category (class in sage.categories.super_modules), 831
SuperModulesWithBasis (class in sage.categories.super_modules_with_basis), 832
SuperModulesWithBasis.ElementMethods (class in sage.categories.super_modules_with_basis), 832
SuperModulesWithBasis.ParentMethods (class in sage.categories.super_modules_with_basis), 834
support() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 283
support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 675
supsets() (sage.categories.examples.with_realizations.SubsetAlgebra method), 959
Surface (class in sage.categories.examples.cw_complexes), 905
Surface.Element (class in sage.categories.examples.cw_complexes), 905
symbol (sage.categories.cartesian_product.CartesianProductFunctor attribute), 874
Category Framework, Release 10.1

...
UniqueFactorizationDomains.ParentMethods (in sage.categories.unique_factorization_domains), 845
Unital (sage.categories.associahedra.Associahedra attribute), 204
Unital (sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas attribute), 330
Unital (sage.categories.magmatic_algebras.MagmaticAlgebras attribute), 637
Unital (sage.categories.rngs.Rngs attribute), 760
Unital (sage.categories.semisymmetric_algebras.SemisymmetricAlgebras attribute), 774
Unital() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 94
Unital() (sage.categories.magmas.Magmas.SubcategoryMethods method), 630
Unital_extra_super_categories() (sage.categories.category_with_axiom.Bars method), 92
UnitalAlgebras (class in sage.categories.unital_algebras), 845
UnitalAlgebras.ParentMethods (class in sage.categories.unital_algebras), 846
UnitalAlgebras.WithBasis (class in sage.categories.unital_algebras), 846
UnitalAlgebras.WithBasis.ParentMethods (class in sage.categories.unital_algebras), 846
universal_commutative_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis method), 451
universal_cover() (sage.categories.finite_enumerated_sets.Pointed finite_sets method), 821
universal_cover_map() (sage.categories.finite_enumerated_sets.Pointed finite_sets method), 822
universal_enveloping_algebra() (sage.categories.lie_algebras.LieAlgebras method), 595
universal_polynomials() (sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis method), 452
unpickle_map() (in module sage.categories.map), 119
unrank() (sage.categories.enumerated_sets.EnumeratedSets method), 338
unrank() (sage.categories.enumerated_sets.FiniteEnumeratedSets method), 470
unrank_range() (sage.categories.enumerated_sets.FiniteEnumeratedSets method), 339
unrank_range() (sage.categories.enumerated_sets.FiniteEnumeratedSets method), 473
unset_base_point() (sage.categories.finite_enumerated_sets.Pointed finite_sets method), 815
upper_covers() (sage.categories.posets.Posets method), 721
valuation() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationRings method), 238
valuation() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationFields method), 327
valuation() (sage.categories.discrete_valuation.DiscreteValuationRings method), 329
VectorSpace() (sage.categories.fields.Fields method), 347
VectorSpaces () (class in sage.categories.vector_spaces), 849
VectorSpaces.CartesianProducts (class in sage.categories.vector_spaces), 849
VectorSpaces.DualObjects (class in sage.categories.vector_spaces), 850
VectorSpaces.ElementMethods (class in sage.categories.vector_spaces), 850
VectorSpaces.Filtered (class in sage.categories.vector_spaces), 850
VectorSpaces.FiniteDimensional (class in sage.categories.vector_spaces), 850
VectorSpaces.FiniteDimensional.TensorProducts (class in sage.categories.vector_spaces), 850
VectorSpaces.Graded (class in sage.categories.vector_spaces), 850
VectorSpaces.TensorProducts (class in sage.categories.vector_spaces), 850
VectorSpaces.Wholes (class in sage.categories.vector_spaces), 850
VectorSpaces.WithBasis (class in sage.categories.vector_spaces), 851
VectorSpaces.WithBasis.CartesianProducts (class in sage.categories.vector_spaces), 851
VectorSpaces.WithBasis.Filtered (class in sage.categories.vector_spaces), 851
VectorSpaces.WithBasis.FiniteDimensional (class in sage.categories.vector_spaces), 852
VectorSpaces.WithBasis.FiniteDimensional.TensorProducts (class in sage.categories.vector_spaces), 852
zero() (sage.categories.examples.lie_algebras.LieAlgebraFromAssociative method), 932
zeta_function() (sage.categories.number_fields.NumberFields.ParentMethods method), 706