Sage Reference Manual: Category Theory

Release 6.8

The Sage Development Team

ELEMENTS, PARENTS, AND CATEGORIES IN SAGE: A (DRAFT OF) PRIMER

Contents

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!).

1.1.1 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).

1.1.2 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.

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, ...

1.2.1 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()
  120

  sage: GL(2,2).cardinality()
  6

  sage: A=random_matrix(ZZ,6,3,x=7)
  sage: L=LatticePolytope(A.rows())
  sage: L.npoints()                # oops!   # random
  37
  ```

- How to ensure robustness?

- How to reduce duplication?

  Example: binary powering:

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

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

We want to implement binary powering only once, as *generic* code that will apply in all cases.

1.3 A bit of help from abstract algebra

1.3.1 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*. Such a set is called a *semigroup*, and binary powering works generally for any semigroup.

Sage knows about semigroups:

```
sage: Semigroups()
Category of semigroups
```

and sure enough, binary powering is defined there:

```
sage: m._pow_.__module__
'sage.categories.semigroups'
```

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

```
sage: GL(2,ZZ) in Semigroups()
True
sage: NN in Semigroups()
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: Sets()
Category of sets
```

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

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

```
sage: Fields()
Category of fields
```

```
sage: HopfAlgebras(QQ)
Category of hopf algebras over Rational Field
```

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)
sage: G.category()
Category of groups
```

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

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

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

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

Those can be viewed graphically:

```
sage: g = Groups().category_graph()
sage: g.set_latex_options(format="dot2tex")
sage: view(g, tightpage=True)              # not tested
```

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

```
sage: g.show(vertex_shape=None, figsize=20)
```

Here is an overview of all categories in Sage:

```
sage: g = sage.categories.category.category_graph()
sage: g.set_latex_options(format="dot2tex")
sage: view(g, tightpage=True)              # not tested
```

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

1.3.2 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()
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 $\mathbf{Q}^3$ to $\mathbf{Q}^3$.

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:

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

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

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

Todo

Give a concrete example, typically using `ElementWrapper`.

Category

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

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

```
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:

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

```
sage: ZZ
Integer Ring
```

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

```
sage: ZZ.categories()
[Join of Category of euclidean domains and Category of infinite enumerated sets,
 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 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, tightpage=True)                    # not tested
```

1.4 A bit of help from computer science

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

Applying an operation is generally done by *calling a method*:

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

sage: x = var('x')
sage: p = 6*x^2 + 12*x + 6
sage: type(p)
<type 'sage.symbolic.expression.Expression'>
sage: p.factor()
6*(x + 1)^2

sage: R.<x> = PolynomialRing(QQ, sparse=True)
sage: pQ = R ( p )
sage: type(pQ)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field'>
sage: pQ.factor()
(6) * (x + 1)^2

sage: pZ = ZZ['x'] ( p )
sage: type(pZ)
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>
sage: pZ.factor()
2 * 3 * (x + 1)^2
```

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

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

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

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

```
sage: i._pow_.__module__
'sage.categories.semigroups'
```

```
sage: pQ._mul_.__module__
'sage.rings.polynomial.polynomial_element_generic'
sage: pQ._pow_.__module__
'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 **Z** of integers, and the set **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 matrices:

```
sage: G = GL(2,ZZ)
sage: m = G.an_element()
sage: for cls in m.__class__.mro(): print cls
<class 'sage.groups.matrix_gps.group_element.LinearMatrixGroup_gap_with_category.element_class'>
<class 'sage.groups.matrix_gps.group_element.MatrixGroupElement_gap'>
...
<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__)
sage: g.set_latex_options(format="dot2tex")
sage: view(g, tightpage=True)                  # not tested
```

1.4.2 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)
sage: type(G)
<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
<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.semigroups.Semigroups.parent_class'>
...
```

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

```
sage: g = class_graph(m.__class__)
sage: g.relabel(lambda x: x.replace("_","\_"))
sage: g.set_latex_options(format="dot2tex")
sage: view(g, tightpage=True)                    # not tested
```

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".

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`, ... ?

1.5.1 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, SemigroupElement, SemigroupMorphism, MagmaMorphism, ...

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

We could implement the above hierarchies as usual:

```
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*:

```
class Groups(Category):

    def super_categories(self):
        return [Monoids(), ...]

    class ParentMethods:
        # generic methods that apply to all groups

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

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

    class SubcategoryMethods:
        # generic methods that apply to all subcategories of Groups()
```

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

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

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

```
sage: Groups().element_class
<class 'sage.categories.groups.Groups.element_class'>

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

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

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

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

Another advantage of building the hierarchy of classes dynamically is that, for parametrized categories, the hierarchy may depend on the parameters. For example an algebra over $\mathbf{Q}$ is a $\mathbf{Q}$-vector space, but an algebra over $\mathbf{Z}$ is not (it is just a $\mathbf{Z}$-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`.

1.5.2 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 `Cs()` is a *super category* of a category `Ds()` if Sage considers any object of `Ds()` to be an object of `Cs()`, up to an implicit application of a canonical functor from `Ds()` to `Cs()`. This functor is normally an inclusion of categories or a forgetful functor. Reciprocally, `Ds()` is said to be a *subcategory* of `Cs()`.

> **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 trac ticket #16183 for more details. The name *supercategory* is also used with a different meaning in certain areas of mathematics.

1.5.3 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:

```
sage: Groups().example()
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:

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

Its source code can be obtained by introspection:

```
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:

```
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:

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

1.5.4 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:

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

Documentation about those methods can be obtained with:

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

See also the `abstract_method()` decorator.

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

1.5.5 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?):

```
sage: S = FiniteSemigroups().example(alphabet=('a', 'b'))
sage: TestSuite(S).run(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

Tests can be run individually:

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

Here is how to access the code of this test:

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

Here is how to run the test on all elements:

```
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:

```
sage: S.product = lambda x, y: S("("+x.value +y.value+")")
```

And rerun the test:

```
sage: S._test_associativity(elements=L)
Traceback (most recent call last):
...
  File ".../sage/categories/semigroups.py", line ..., in _test_associativity
    tester.assert_((x * y) * z == x * (y * z))
...
AssertionError: False is not true
```

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):

```
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_assoc:
-> tester.assert_((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))'
```

1.5.6 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.

1.6 Case study

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

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

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

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

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

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

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

`__pow__` is a generic method for all finite semigroups:

```
sage: x.__pow__.__module__
'sage.categories.semigroups'
```

`__mul__` is a default implementation from the [Magmas](#) category (a *magma* is a set with an inner law ∗, not necessarily associative):

```
sage: x.__mul__.__module__
'sage.categories.magmas'
```

It delegates the work to the parent (following the advice: if you do not know what to do, ask your parent):

```
sage: x.__mul__??                                # not tested
```

`product` is a mathematical method implemented by the parent:

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

`cayley_graph` is a generic method on the parent, provided by the [FiniteSemigroups](#) category:

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

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

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

Consider now the implementation of the semigroup:

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

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

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

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

`product` does not appear in the list because a default implementation is provided in term of the method `_mul_` on elements. Of course, at least one of them should be implemented. On the other hand, a default implementation for `__contains__` is provided by `Parent`.

Documentation about those methods can be obtained with:

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

See also the `abstract_method()` decorator.

Here is the code for the finite semigroups category:

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

1.7 Specifying the category of a parent

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

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

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

```
sage: P = PermutationGroup([[(1,2,3)]], category=PermutationGroups().Finite().Commutative()); P
Permutation Group with generators [(1,2,3)]
sage: P.category()
Category of finite commutative permutation groups
```

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())
sage: P.foo()
foo
sage: p = P.an_element()
sage: p.bar()
bar
```

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

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

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.

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.

1.8.1 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.algebras()`.

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")
sage: B = HopfAlgebrasWithBasis(QQ).example(); B.rename("B")
sage: C = cartesian_product([A, B, B]); C
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()
True
```

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

```
sage: C.one()
B[(0, word: )] + B[(1, ())] + B[(2, ())]
sage: cartesian_product([A.one(), B.one(), B.one()])
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__
'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.nested_class_test`.

Let us now look at the categories of `C`:

```
sage: C.categories()
[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.

1.8.2 Axioms

First examples

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

```
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:

```
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:

```
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
```

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

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

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

Axioms and categories with axioms

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

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

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

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

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

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

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

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

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

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

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

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

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

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, `*`, `/`, `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 trac ticket #15741 to unclutter the global name space from, for example, the many variants of the category of algebras like:

```
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: 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 $*$:

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

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

sage: C.Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().Unital()
Category of semirings

sage: C.Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().AdditiveInverse()
Category of rings

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

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

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

or for more advanced categories:

```
sage: g = HopfAlgebras(QQ).WithBasis().Graded().Connected().category_graph()
sage: g.set_latex_options(format="dot2tex")
sage: view(g, tightpage=True)                # not tested
```

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 `Cs.A()` which is also in `Ds`? A finite dimensional module which is also an algebra is a finite dimensional algebra:

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

On the other hand, a graded module *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*:

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

The relevant difference between `FiniteDimensional` and `Graded` is that `FiniteDimensional` is a statement about the properties of `O` seen as a module (and thus does not depend on the given category), whereas `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 `Cs` of `Ds`, `Ds.A()` is the intersection of the categories `Ds` and `Cs.A()`:

```
sage: As = Algebras(QQ).FiniteDimensional(); As
Category of finite dimensional algebras over Rational Field
sage: Bs = Algebras(QQ) & Modules(QQ).FiniteDimensional(); As
Category of finite dimensional algebras over Rational Field
sage: As is Bs
True
```

An immediate consequence is that, as we have already noticed, axioms commute:

```
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:

```
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 `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, `As` is the category of graded

algebras, which are further endowed with some basis; that basis need not respect the grading. In fact `As` is really a join category:

```
sage: type(As)
<class 'sage.categories.category.JoinCategory_with_category'>
sage: As._repr_(as_join=True)
'Join of Category of algebras with basis over Rational Field and Category of graded algebras over Rat
```

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.

1.8.3 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.

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`.

1.9.1 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:

```
Cs().is_subcategory( Category.join(Cs().super_categories()) )
```

```
Cs()._cmp_key  >  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 `Magmas.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.

1.9.2 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[]
sage: P.prod([x,y,2])
2*x*y
sage: P.prod.__module__
'sage.categories.monoids'
sage: P.prod.__func__ is Monoids().ParentMethods.prod.__func__
True
```

We recommend to study the code of one example:

```
sage: C = CommutativeAdditiveMonoids()
sage: C??                                       # not tested
```

1.9.3 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 trac ticket #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 trac ticket #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()`.

1.9.4 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
```

IMPLEMENTING A NEW PARENT: A (DRAFT OF) 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:

```
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:

```
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 occurence of FiniteSemigroups().example(...) in the documentation by LeftRegularBand. This will be equivalent to:

```
sage: from sage.categories.examples.finite_semigroups import LeftRegularBand
```

Now, try:

```
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 FiniteSemigroups.

Rename the class to 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 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.

TODO: the tutorial should explain there how to reuse the enumerated set of subsets, and endow it with more structure.

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:

```
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:

```
sage: V = VectorSpace(RationalField(), 3)
sage: V.category()
Category of vector spaces with basis over quotient fields
sage: G = SymmetricGroup(9)
sage: G.category()
Join of Category of finite permutation groups and Category of finite weyl groups
sage: P = PerfectMatchings(3)
sage: P.category()
Category of finite enumerated sets
```

Let's check some memberships:

```
sage: V in VectorSpaces(QQ)
True
sage: V in VectorSpaces(FiniteField(11))
False
sage: G in Monoids()
True
```

```
sage: P in Rings()
False
```

For parametrized categories one can use the following shorthand:

```
sage: V in VectorSpaces
True
sage: G in VectorSpaces
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 compatibilty this is not yet enforced:

```
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 an may be deprecated):

```
sage: V = VectorSpace(RationalField(), 3)
sage: v = V.gen(1)
sage: v.category()
Category of elements of Vector space of dimension 3 over Rational Field
```

---

class sage.categories.category.**Category**(*s=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation,
sage.structure.sage_object.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 $\mathbf{Q}[x]$) 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 $\mathbf{Q}[x]$.

- Generic operations on elements of such parents (e. g., the Euclidean algorithm for computing gcds).

- Generic operations on morphisms of this category.

This is achieved as follows:

```
sage: from sage.categories.all 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:

- `As()`: the category of sets

- `Bs()`: the category of additive groups

- `Cs()`: the category of multiplicative monoids

- `Ds()`: the category of rings

```
sage: from sage.categories.all import Category
sage: from sage.misc.lazy_attribute import lazy_attribute
sage: class As (Category):
....:     def super_categories(self):
....:         return []
....:
....:     class ParentMethods:
....:         def fA(self):
....:             return "A"
....:         f = fA
```

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

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

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

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

We check this before proceeding:

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

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

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

sage: C = Cs().parent_class()
sage: [ C.fA(), C.fC() ]
['A', 'C']
sage: C.f()
'C'
```

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

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

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'>, <type 'object'>]

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'>, <type 'object'>)

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

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

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

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

```
sage: Algebras(GF(5)).parent_class is Algebras(GF(7)).parent_class
True
sage: Coalgebras(QQ).parent_class is Coalgebras(FractionField(QQ['x'])).parent_class
True
```

We now construct a parent in the usual way:

```
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'>, <class '__main__.Ds.parent_class'>)
sage: D.__class__.mro()
[<class '__main__.myparent_with_category'>,
<class '__main__.myparent'>,
<type 'sage.structure.parent.Parent'>,
<type 'sage.structure.category_object.CategoryObject'>,
<type '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'>,
<type 'object'>]
sage: D.fA()
'A'
sage: D.fB()
'B'
sage: D.fC()
'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 at ...>,
<class '__main__.Ds.element_class'>,
<class '__main__.Cs.element_class'>,
<class '__main__.Bs.element_class'>,
<class '__main__.As.element_class'>,
<type 'object'>]
```

TESTS:
```
sage: import __main__
sage: __main__.myparent = myparent
sage: __main__.As = As
sage: __main__.Bs = Bs
sage: __main__.Cs = Cs
sage: __main__.Ds = Ds
sage: loads(dumps(Ds)) is Ds
True
sage: loads(dumps(Ds())) is Ds()
True
sage: loads(dumps(Ds().element_class)) is Ds().element_class
True
```

_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 trac ticket #11943, the order of super categories is determined by Python's method resolution order
    C3 algorithm.

    See also:

    `all_super_categories()`

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

Note: this sets `_super_categories_for_classes()` as a side effect

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

_all_super_categories_proper()
    All the proper super categories of this category.

    Since trac ticket #11943, the order of super categories is determined by Python's method resolution order
    C3 algorithm.

    See also:

---

all_super_categories()

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

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

_set_of_super_categories()
> The frozen set of all proper super categories of this category.

Note: this is used for speeding up category containment tests.

See also:

all_super_categories()

EXAMPLES:
```
sage: Groups()._set_of_super_categories
frozenset({Category of inverse unital magmas,
           Category of unital magmas,
           Category of magmas,
           Category of monoids,
           Category of objects,
           Category of semigroups,
           Category of sets with partial maps,
           Category of sets})
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]
```

TESTS:
```
sage: C = HopfAlgebrasWithBasis(GF(7))
sage: C._set_of_super_categories == frozenset(C._all_super_categories_proper)
True
```

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

> INPUT:

> - name – a string; the name of the class as an attribute of self. E.g. "parent_class"

> - method_provider – a string; the name of an attribute of self that provides methods for the new class (in addition to those coming from the super categories). E.g. "ParentMethods"

> - cache – a boolean or ignore_reduction (default: False) (passed down to dynamic_class; for internal use only)

> - picklable – a boolean (default: True)

ASSUMPTION:

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

OUTPUT:

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

Note:

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

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

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

See also:

`CategoryWithParameters._make_named_class()`

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

Note that, by default, the result is not cached:
```
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:
```
sage: Rings().parent_class
<class 'sage.categories.rings.Rings.parent_class'>
```

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

TESTS:
```
sage: class A: pass
sage: class BrokenCategory(Category):
....:     def super_categories(self): return []
....:     ParentMethods = 1
....:     class ElementMethods(A):
```

```
....:          pass
....:      class MorphismMethods(object):
....:          pass
sage: C = BrokenCategory()
sage: C._make_named_class("parent_class",    "ParentMethods")
Traceback (most recent call last):
...
AssertionError: BrokenCategory.ParentMethods should be a class
sage: C._make_named_class("element_class",  "ElementMethods")
doctest:...: UserWarning: BrokenCategory.ElementMethods should not have a super class
<class '__main__.BrokenCategory.element_class'>
sage: C._make_named_class("morphism_class", "MorphismMethods")
<class '__main__.BrokenCategory.morphism_class'>
```

repr()

Return the print representation of this category.

EXAMPLES:

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

_repr_object_names()

Return the name of the objects of this category.

EXAMPLES:

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

_test_category(***options*)

Run generic tests on this category

See also:

TestSuite.

EXAMPLES:

```
sage: Sets()._test_category()
```

Let us now write a couple broken categories:

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

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

sage: class MyObjects(Category):
```

```
....:        def super_categories(self):
....:            return []
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 `self` satisfying the given `axiom`.

   INPUT:

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

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

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: Algebras(QQ).WithBasis().Commutative() is Algebras(QQ).Commutative().WithBasis()
True
```

   When `axiom` is not defined for `self`, `self` is returned:
```
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:
```
sage: Sets()._with_axiom_as_tuple('Finite')
(Category of finite sets,)
sage: Magmas()._with_axiom_as_tuple('Finite')
(Category of magmas, Category of finite sets)
sage: Rings().Division()._with_axiom_as_tuple('Finite')
(Category of division rings,
 Category of finite monoids,
 Category of commutative magmas)
sage: HopfAlgebras(QQ)._with_axiom_as_tuple('FiniteDimensional')
(Category of hopf algebras over Rational Field,
 Category of finite dimensional modules over Rational Field)
```

_without_axioms(*named=False*)

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

INPUT:

•`named` – a boolean (default: `False`)

___

Todo

Improve this explanation.

___

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

EXAMPLES:

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

static **_sort**(*categories*)

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

See also:

`_cmp_key()`

INPUT:

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

OUTPUT:

A sorted tuple of categories, possibly with repeats.

___

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

___

EXAMPLES:

```
sage: Category._sort([Sets(), Objects(), Coalgebras(QQ), Monoids(), Sets().Finite()])
(Category of monoids,
 Category of coalgebras over Rational Field,
 Category of finite sets,
 Category of sets,
 Category of objects)
sage: Category._sort([Sets().Finite(), Semigroups().Finite(), Sets().Facade(),Magmas().Commu
(Category of finite semigroups,
 Category of commutative magmas,
 Category of finite sets,
 Category of facade sets)
sage: Category._sort(Category._flatten_categories([Sets().Finite(), Algebras(QQ).WithBasis()
(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)
```

_sort_uniq(*categories*)

Return the categories after sorting them and removing redundant categories.

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

INPUT:

- `categories` – a list (or iterable) of categories

OUTPUT: a sorted tuple of mutually incomparable categories

EXAMPLES:

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

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

static __classcall__(**args*, ***options*)

Input mangling for unique representation.

Let `C = Cs(...)` be a category. Since [trac ticket #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: 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*)

Initializes this category.

EXAMPLES:

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

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

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 magm
> ```

> The category of realizations of a given algebra:
> ```
> sage: A = Sets().WithRealizations().example(); A
> The subset algebra of {1, 2, 3} over Rational Field
> sage: A.Realizations()
> 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 real
> sage: C.super_categories()
> [Category of graded hopf algebras with basis over Rational Field, Category of realizations o
> 
> sage: TestSuite(C).run()
> ```

> **See also:**

> > •`Sets().WithRealizations`

> > •`ClasscallMetaclass`

> ---

> **Todo**

> Add an optional argument to allow for:
> ```
> sage: Realizations(A, category = Blahs()) # todo: not implemented
> ```

> ---

WithRealizations()
> Returns 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.

- `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 `Sets.Facade`) 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
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()
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: A.Out()
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: A.In()
The subset algebra of {1, 2, 3} over Rational Field in the In basis

sage: A.an_element()
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

If B and B' are two bases, then the change of basis from B to B' is implemented by a canonical coercion between P_B and $P_{B'}$:
```
sage: F = A.F(); In = A.In(); Out = A.Out()
sage: i = In.an_element(); i
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
sage: F(i)
7*F[{}] + 3*F[{1}] + 4*F[{2}] + F[{1, 2}]
sage: F.coerce_map_from(Out)
Generic morphism:
  From: The subset algebra of {1, 2, 3} over Rational Field in the Out basis
  To:   The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
```

allowing for mixed arithmetic:
```
sage: (1 + Out.from_set(1)) * In.from_set(2,3)
Out[{}] + 2*Out[{1}] + 2*Out[{2}] + 2*Out[{3}] + 2*Out[{1, 2}] + 2*Out[{1, 3}] + 4*Out[{2, 3
```

In our example, there are three realizations:
```
sage: A.realizations()
[The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis,
```

```
The subset algebra of {1, 2, 3} over Rational Field in the In basis,
The subset algebra of {1, 2, 3} over Rational Field in the Out basis]
```

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

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

The various parent realizing A belong to this category:

```
sage: A.F() in A.Realizations()
True
```

A itself is in the category of algebras with realizations:

```
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, `FiniteDimensionalVectorSpaces(QQ).WithRealizations()` will eventually be in charge, whenever a coercion $\phi : A \mapsto B$ is registered, to register ϕ^{-1} as coercion $B \mapsto A$ 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):

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

and in the category of algebras (regressiveness):

```
sage: A in Algebras(QQ)
True
```

Note: For `C` a category, `C.WithRealizations()` in fact calls `sage.categories.with_realizations.Realizations(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:

```
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.

TESTS:

```
sage: Semigroups().WithRealizations()
Join of Category of semigroups and Category of sets with realizations
sage: C = GradedHopfAlgebrasWithBasis(QQ).WithRealizations(); C
Category of graded hopf algebras with basis over Rational Field with realizations
sage: C.super_categories()
[Join of Category of hopf algebras over Rational Field
     and Category of graded algebras over Rational Field]
sage: TestSuite(Semigroups().WithRealizations()).run()
```

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: CoxeterGroups().additional_structure()     # distinguished generators
Category of coxeter groups
sage: Crystals().additional_structure()          # crystal operators
Category of crystals
```

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

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

> Most *categories with axiom* don't define additional structure:

```
sage: Sets().Finite().additional_structure()
sage: Rings().Commutative().additional_structure()
sage: Modules(QQ).FiniteDimensional().additional_structure()
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(QQ).Unital().additional_structure()
```

As of Sage 6.4, the only exceptions are the category of unital magmas or the category of unital additive magmas (both define a unit which shall be preserved by morphisms):

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

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

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

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

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

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

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

This might change, as we are lacking enough data points to guarantee that this was the correct design decision.

Note: In some cases a category defines additional structure, where the structure can be useful to manipulate morphisms but where, in most use cases, we don't want the morphisms to necessarily preserve it. For example, in the context of finite dimensional vector spaces, having a distinguished basis allows for representing morphisms by matrices; yet considering only morphisms that preserve that distinguished basis would be boring.

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

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

- an euclidean domain morphism preserve euclidean division?

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

- an enumerated set morphism preserve the distinguished enumeration?

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

- a module with basis morphism preserve the distinguished basis?

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

See also:

This method together with the methods overloading it provide the basic data to determine, for a given category, the super categories that define some structure (see `structure()`), and to test whether a category is a full subcategory of some other category (see `is_full_subcategory()`).

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

all_super_categories(*proper=False*)
Returns the list of all super categories of this category.

INPUT:

- `proper` – a boolean (default: `False`); whether to exclude this category.

Since trac ticket #11943, the order of super categories is determined by Python's method resolution order C3 algorithm.

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

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

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

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

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 precisi
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({'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()
> sage: G.is_directed_acyclic()
> True
> sage: G.girth()
> 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:
> ```
> sage: C = Algebras(QQ).element_class; C
> <class 'sage.categories.algebras.Algebras.element_class'>
> sage: type(C)
> <class 'sage.structure.dynamic_class.DynamicMetaclass'>
> ```

By [trac ticket #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:

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

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 insted.

　　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):

```
sage: Monoids().super_categories()
[Category of semigroups, Category of unital magmas]
sage: 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 can't be taught that the category of groups is a full subcategory of the category of semigroups:

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

`is_abelian``()`

Returns 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:

```
sage: Modules(ZZ).is_abelian()
True
sage: FreeModules(ZZ).is_abelian()
```

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

is_full_subcategory(*other*)

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

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

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

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

See also:

- `is_subcategory()`

- `full_super_categories()`

EXAMPLES:
```
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
```

Here are two typical examples of false negatives:
```
sage: Groups().is_full_subcategory(Semigroups())
False
sage: Groups().is_full_subcategory(Semigroups())    # todo: not implemented
True
sage: Fields().is_full_subcategory(Rings())
False
sage: Fields().is_full_subcategory(Rings())          # todo: not implemented
True
```

Todo

The latter is a consequence of `EuclideanDomains` currently being a structure category. Is this what we want?
```
sage: EuclideanDomains().is_full_subcategory(Rings())
False
```

is_subcategory(*c*)

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

    EXAMPLES:

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

    The `is_subcategory` function takes into account the base.

```
sage: M3 = VectorSpaces(FiniteField(3))
sage: M9 = VectorSpaces(FiniteField(9, 'a'))
sage: M3.is_subcategory(M9)
False
```

    Join categories are properly handled:

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

```
sage: V3 = VectorSpaces(FiniteField(3))
sage: POSet = PartiallyOrderedSets()
sage: PoV3 = Category.join((V3, POSet))
sage: A3 = AlgebrasWithBasis(FiniteField(3))
sage: PoA3 = Category.join((A3, POSet))
sage: PoA3.is_subcategory(PoV3)
True
sage: PoV3.is_subcategory(PoV3)
True
sage: PoV3.is_subcategory(PoA3)
False
```

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

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

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

    INPUT:

        •`categories` – a list (or iterable) of categories

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

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

    See also:

    `__and__()` for a shortcut

    EXAMPLES:

```
sage: J = Category.join((Groups(), CommutativeAdditiveMonoids())); J
Join of Category of groups and Category of commutative additive monoids
sage: J.super_categories()
[Category of groups, Category of commutative additive monoids]
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 fields, Category of modules over Integer Ring]
sage: Category.join([], as_list=True)
[]
sage: Category.join([Groups()], as_list=True)
[Category of groups]
sage: Category.join([Groups() & Posets()], as_list=True)
[Category of groups, Category of posets]
```

Support for axiom categories (TODO: put here meaningfull 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
```

TESTS:
```
sage: Magmas().Unital().Commutative().Finite() is Magmas().Finite().Commutative().Unital()
True
sage: from sage.categories.category_with_axiom import TestObjects
sage: T = TestObjects()
sage: TCF = T.Commutative().Facade(); TCF
Category of facade commutative test objects
sage: TCF is T.Facade().Commutative()
True
sage: TCF is (T.Facade() & T.Commutative())
True
sage: TCF.axioms()
frozenset({'Commutative', 'Facade'})
sage: type(TCF)
<class 'sage.categories.category_with_axiom.TestObjects.Commutative.Facade_with_category'>

sage: TCF = T.Commutative().FiniteDimensional()
sage: TCF is T.FiniteDimensional().Commutative()
True
sage: TCF is T.Commutative() & T.FiniteDimensional()
True
sage: TCF is T.FiniteDimensional() & T.Commutative()
True
sage: type(TCF)
<class 'sage.categories.category_with_axiom.TestObjects.Commutative.FiniteDimensional_with_c

sage: TCU = T.Commutative().Unital()
sage: TCU is T.Unital().Commutative()
True
sage: TCU is T.Commutative() & T.Unital()
True
sage: TCU is T.Unital() & T.Commutative()
True

sage: TUCF = T.Unital().Commutative().FiniteDimensional(); TUCF
Category of finite dimensional commutative unital test objects
sage: type(TUCF)
<class 'sage.categories.category_with_axiom.TestObjects.FiniteDimensional.Unital.Commutative

sage: TFFC = T.Facade().FiniteDimensional().Commutative(); TFFC
Category of facade finite dimensional commutative test objects
sage: type(TFFC)
<class 'sage.categories.category.JoinCategory_with_category'>
```

```
sage: TFFC.super_categories()
[Category of facade commutative test objects,
 Category of finite dimensional commutative test objects]
```

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 morphims 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(None)
Category of monoids
```

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):
...
AssertionError: Subcategory of 'Category of enumerated sets' required; got 'Category of mono
```

Otherwise, the two categories are joined together:

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

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:
>
> ```
> sage: C = Algebras(QQ).parent_class; C
> <class 'sage.categories.algebras.Algebras.parent_class'>
> sage: type(C)
> <class 'sage.structure.dynamic_class.DynamicMetaclass'>
> ```
>
> By trac ticket #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:
>
> ```
> sage: Algebras(GF(7)).parent_class is Algebras(GF(5)).parent_class
> True
> sage: Algebras(QQ).parent_class is Algebras(ZZ).parent_class
> False
> sage: Algebras(ZZ['t']).parent_class is Algebras(ZZ['t','x']).parent_class
> True
> ```
>
> 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:
```
sage: Algebras(QQ).required_methods()
{'element': {'optional': ['_add_', '_mul_'], 'required': ['__nonzero__']},
 '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:
> ```
> sage: Objects().structure()
> frozenset()
>
> sage: def structure(C):
> ....:     return Category._sort(C.structure())
>
> sage: structure(Sets())
> (Category of sets, Category of sets with partial maps)
> sage: structure(Magmas())
> (Category of magmas, Category of sets, Category of sets with partial maps)
> ```
>
> In the following example, we only list the smallest structure categories to get a more readable output:
> ```
> sage: def structure(C):
> ....:     return Category._sort_uniq(C.structure())
>
> 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:

> - •trac ticket #12895
>
> - •`parent_class()`
>
> - •`element_class()`
>
> - •`_make_named_class()`

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

TESTS:
```
sage: cls = Algebras(QQ).subcategory_class; cls
<class 'sage.categories.algebras.Algebras.subcategory_class'>
sage: type(cls)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
```

super_categories()

> Return the *immediate* super categories of `self`.

> OUTPUT:

> > - •a duplicate-free list of categories.

> Every category should implement this method.

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

> ---
> **Note:** Since trac ticket #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: `sage.categories.category.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

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

_make_named_class(*name*, *method_provider*, *cache=False*, ***options*)

Return the parent/element/... class of `self`.

INPUT:

- `name` – a string; the name of the class as an attribute of `self`

- `method_provider` – a string; the name of an attribute of `self` that provides methods for the new class (in addition to what comes from the super categories)

- `**options` – other named options to pass down to `Category._make_named_class()`.

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 that has the corresponding named classes of the super categories of `self` as bases and contains the methods provided by `getattr(self, method_provider)`.

Note: This method overrides `Category._make_named_class()` so that the returned class *only* depends on the corresponding named classes of the super categories and on the provided methods. This allows for sharing the named classes across closely related categories providing the same code to their parents, elements and so on.

EXAMPLES:

The categories of bimodules over the fields `CC` or `RR` provide the same methods to their parents and elements:
```
sage: Bimodules(ZZ,RR).parent_class is Bimodules(ZZ,RDF).parent_class #indirect doctest
True
sage: Bimodules(CC,ZZ).element_class is Bimodules(RR,ZZ).element_class
True
```

On the other hand, modules over a field have more methods than modules over a ring:
```
sage: Modules(GF(3)).parent_class is Modules(ZZ).parent_class
False
sage: Modules(GF(3)).element_class is Modules(ZZ).element_class
False
```

For a more subtle example, one could possibly share the classes for GF (3) and GF (2^3, 'x'), but this is not currently the case:

```
sage: Modules(GF(3)).parent_class is Modules(GF(2^3,'x')).parent_class
False
```

This is because those two fields do not have the exact same category:

```
sage: GF(3).category()
Join of Category of finite fields and Category of subquotients of monoids and Category of qu
sage: GF(2^3,'x').category()
Category of finite fields
```

Similarly for QQ and RR:

```
sage: QQ.category()
Category of quotient fields
sage: RR.category()
Category of fields
sage: Modules(QQ).parent_class is Modules(RR).parent_class
False
```

Some other cases where one could potentially share those classes:

```
sage: Modules(GF(3),dispatch=False).parent_class  is Modules(ZZ).parent_class
False
sage: Modules(GF(3),dispatch=False).element_class is Modules(ZZ).element_class
False
```

TESTS:

```
sage: PC = Algebras(QQ).parent_class; PC   # indirect doctest
<class 'sage.categories.algebras.Algebras.parent_class'>
sage: type(PC)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
sage: PC.__bases__
(<class 'sage.categories.rings.Rings.parent_class'>,
 <class 'sage.categories.associative_algebras.AssociativeAlgebras.parent_class'>,
 <class 'sage.categories.unital_algebras.UnitalAlgebras.parent_class'>)
sage: loads(dumps(PC)) is PC
True
```

class sage.categories.category.**JoinCategory**(*super_categories*, ***kwds*)

Bases: sage.categories.category.CategoryWithParameters

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

EXAMPLES:

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

By trac ticket #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:

```
sage: G = SymmetricGroup(10)
sage: A3 = G.algebra(GF(3))
sage: A5 = G.algebra(GF(5))
sage: type(A3.category())
<class 'sage.categories.category.JoinCategory_with_category'>
sage: type(A3) is type(A5)
True
```

_repr_object_names()

    Return the name of the objects of this category.

    See also:

    Category._repr_object_names(), _repr_(), _without_axioms()

    EXAMPLES:

```
sage: Groups().Finite().Commutative()._repr_(as_join=True)
'Join of Category of finite groups and Category of commutative groups'
sage: Groups().Finite().Commutative()._repr_object_names()
'finite commutative groups'
```

    This uses _without_axioms() which may fail if this category is not obtained by adjoining axioms to some super categories:

```
sage: Category.join((Groups(), CommutativeAdditiveMonoids()))._repr_object_names()
Traceback (most recent call last):
...
ValueError: This join category isn't built by adding axioms to a single category
```

repr(*as_join=False*)

    Print representation.

    INPUT:

        •as_join – a boolean (default: False)

    EXAMPLES:

```
sage: Category.join((Groups(), CommutativeAdditiveMonoids())) #indirect doctest
Join of Category of groups and Category of commutative additive monoids
```

    By default, when a join category is built from category by adjoining axioms, a nice name is printed out:

```
sage: Groups().Facade().Finite()
Category of facade finite groups
```

    But this is in fact really a join category:

```
sage: Groups().Facade().Finite()._repr_(as_join = True)
'Join of Category of finite groups and Category of facade sets'
```

    The rationale is to make it more readable, and hide the technical details of how this category is constructed internally, especially since this construction is likely to change over time when new axiom categories are implemented.

    This join category may possibly be obtained by adding axioms to different categories; so the result is not guaranteed to be unique; when this is not the case the first found is used.

    See also:

```
Category._repr_(),_repr_object_names()
```

TESTS:
```
sage: Category.join((Sets().Facade(), Groups()))
Category of facade groups
```

_without_axioms(*named=False*)

When adjoining axioms to a category, one often gets a join category; this method tries to recover the original category from this join category.

INPUT:

- named – a boolean (default: `False`)

See `Category._without_axioms()` for the description of the `named` parameter.

EXAMPLES:
```
sage: C = Category.join([Monoids(), Posets()]).Finite()
sage: C._repr_(as_join=True)
'Join of Category of finite monoids and Category of finite posets'
sage: C._without_axioms()
Traceback (most recent call last):
...
ValueError: This join category isn't built by adding axioms to a single category
sage: C = Monoids().Infinite()
sage: C._repr_(as_join=True)
'Join of Category of monoids and Category of infinite sets'
sage: C._without_axioms()
Category of magmas
sage: C._without_axioms(named=True)
Category of monoids
```

TESTS:

`C` is in fact a join category:
```
sage: from sage.categories.category import JoinCategory
sage: isinstance(C, JoinCategory)
True
```

additional_structure()

Return `None`.

Indeed, a join category defines no additional structure.

See also:

`Category.additional_structure()`

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

is_subcategory(*C*)

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

EXAMPLES:
```
sage: Category.join([Rings(),Modules(QQ)]).is_subcategory(Category.join([Rngs(),Bimodules(QQ
True
```

super_categories()

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

>   EXAMPLES:
>   ```
>   sage: from sage.categories.category import JoinCategory
>   sage: JoinCategory((Semigroups(), FiniteEnumeratedSets())).super_categories()
>   [Category of semigroups, Category of finite enumerated sets]
>   ```

sage.categories.category.**category_graph**(*categories=None*)

>   Return the graph of the categories in Sage.

>   INPUT:

>   •`categories` – a list (or iterable) of categories

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

>   For readability, the names of the category are shortened.

>   ---

>   **Todo**

>   Further remove the base ring (see also trac ticket #15801).

>   ---

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

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:
>   ```
>   sage: from sage.categories.category import category_sample
>   sage: sorted(category_sample(), key=str)
>   [Category of G-sets for Symmetric group of order 8! as a permutation group,
>    Category of Hecke modules over Rational Field,
>    Category of additive magmas, ...,
>    Category of fields, ...,
>    Category of graded hopf algebras with basis over Rational Field, ...,
>    Category of modular abelian varieties over Rational Field, ...,
>    Category of simplicial complexes, ...,
>    Category of vector spaces over Rational Field, ...,
>    Category of weyl groups,...
>   ```

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

>   Returns True if x is a category.

>   EXAMPLES:

```
sage: sage.categories.category.is_Category(CommutativeAdditiveSemigroups())
True
sage: sage.categories.category.is_Category(ZZ)
False
```

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: [sage.categories.category.Category](#)

Initializes this category.

EXAMPLES:

```
sage: class SemiprimitiveRings(Category):
....:     def super_categories(self):
....:         return [Rings()]
....:
....:     class ParentMethods:
....:         def jacobson_radical(self):
....:             return self.ideal(0)
....:
sage: C = SemiprimitiveRings()
sage: C
Category of semiprimitive rings
sage: C.__class__
<class '__main__.SemiprimitiveRings_with_category'>
```

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.

---

is_abelian()

class sage.categories.category_types.**Category_ideal**(*ambient*, *name=None*)

Bases: [sage.categories.category_types.Category_in_ambient](#)

classmethod an_instance()

Returns an instance of this class

EXAMPLES:

```
sage: AlgebraIdeals.an_instance()
Category of algebra ideals in Univariate Polynomial Ring in x over Rational Field
```

ring()

class sage.categories.category_types.**Category_in_ambient**(*ambient*, *name=None*)

Bases: [sage.categories.category.Category](#)

ambient()

Return the ambient object in which objects of this category are embedded.

---

class sage.categories.category_types.**Category_module**(*base*, *name=None*)

    Bases: sage.categories.category_types.AbelianCategory,
sage.categories.category_types.Category_over_base_ring

    Initialize self.

    EXAMPLES:

```
sage: C = Algebras(GF(2)); C
Category of algebras over Finite Field of size 2
sage: TestSuite(C).run()
```

class sage.categories.category_types.**Category_over_base**(*base*, *name=None*)

    Bases: sage.categories.category.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 trac ticket #11935 for details.

    EXAMPLES:

```
sage: Algebras(GF(2)).element_class is Algebras(GF(3)).element_class
True
```

```
sage: C = GF(2).category()
sage: Algebras(GF(2)).parent_class is Algebras(C).parent_class
True
```

```
sage: C = ZZ.category()
sage: Algebras(ZZ).element_class is Algebras(C).element_class
True
```

    classmethod an_instance()

        Returns an instance of this class

        EXAMPLES:

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

    base()

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

class sage.categories.category_types.**Category_over_base_ring**(*base*, *name=None*)

    Bases: sage.categories.category_types.Category_over_base

    Initialize self.

    EXAMPLES:

```
sage: C = Algebras(GF(2)); C
Category of algebras over Finite Field of size 2
sage: TestSuite(C).run()
```

    base_ring()

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

        EXAMPLES:

```
sage: C = Algebras(GF(2))
sage: C.base_ring()
Finite Field of size 2
```

class sage.categories.category_types.**ChainComplexes**(*base*, *name=None*)

   Bases: `sage.categories.category_types.Category_module`

   The category of all chain complexes over a base ring.

   EXAMPLES:

```
sage: ChainComplexes(RationalField())
Category of chain complexes over Rational Field

sage: ChainComplexes(Integers(9))
Category of chain complexes over Ring of integers modulo 9
```

   TESTS::

```
sage: TestSuite(ChainComplexes(RationalField())).run()
```

   super_categories()

      EXAMPLES:

```
sage: ChainComplexes(Integers(9)).super_categories()
[Category of modules with basis over Ring of integers modulo 9]
```

class sage.categories.category_types.**Elements**(*object*)

   Bases: `sage.categories.category.Category`

   The category of all elements of a given parent.

   EXAMPLES:

```
sage: a = IntegerRing()(5)
sage: C = a.category(); C
Category of elements of Integer Ring
sage: a in C
True
sage: 2/3 in C
False
sage: loads(C.dumps()) == C
True
```

   classmethod an_instance()

      Returns an instance of this class

      EXAMPLES:

```
sage: Elements(ZZ)
Category of elements of Integer Ring
```

   object()

   super_categories()

      EXAMPLES:

```
sage: Elements(ZZ).super_categories()
[Category of objects]
```

      TODO:

check that this is what we want.

class sage.categories.category_types.**SimplicialComplexes**(*s=None*)

Bases: `sage.categories.category.Category`

The category of simplicial complexes.

EXAMPLES:
```
sage: SimplicialComplexes()
Category of simplicial complexes
```

TESTS:
```
sage: TestSuite(SimplicialComplexes()).run()
```

super_categories()

EXAMPLES:
```
sage: SimplicialComplexes().super_categories()
[Category of objects]
```

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:

```
sage: ZZ in Sets()
True
```

class sage.categories.category_singleton.**Category_singleton**(*s=None*)

    Bases: sage.categories.category.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:

    sage: QQ in Fields() True sage: 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:

```
sage: from sage.categories.category_singleton import Category_singleton
sage: class MyRings(Category):
....:     def super_categories(self): return Rings().super_categories()
sage: 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: R = QQ['x,y']
sage: R1 = Parent(category = MyRings())
sage: R2 = Parent(category = MyRingsSingleton())
sage: R in MyRings()
False
sage: R1 in MyRings()
True
sage: R1 in MyRingsSingleton()
```

```
False
sage: R2 in MyRings()
False
sage: R2 in MyRingsSingleton()
True
```

One sees that containment tests for the singleton class is a lot faster than for a usual class:

```
sage: timeit("R in MyRings()", number=10000)                    # not tested
10000 loops, best of 3: 7.12 µs per loop
sage: timeit("R1 in MyRings()", number=10000)                   # not tested
10000 loops, best of 3: 6.98 µs per loop
sage: timeit("R in MyRingsSingleton()", number=10000)           # not tested
10000 loops, best of 3: 3.08 µs per loop
sage: timeit("R2 in MyRingsSingleton()", number=10000)          # not tested
10000 loops, best of 3: 2.99 µs per loop
```

So this is an improvement, but not yet competitive with a pure Cython method:

```
sage: timeit("R.is_ring()", number=10000)                       # not tested
10000 loops, best of 3: 383 ns per loop
```

However, it is competitive with a Python method. Actually it is faster, if one stores the category in a variable:

```
sage: _Rings = Rings()
sage: R3 = Parent(category = _Rings)
sage: R3.is_ring.__module__
'sage.categories.rings'
sage: timeit("R3.is_ring()", number=10000)                      # not tested
10000 loops, best of 3: 2.64 µs per loop
sage: timeit("R3 in Rings()", number=10000)                     # not tested
10000 loops, best of 3: 3.01 µs per loop
sage: timeit("R3 in _Rings", number=10000)                      # not tested
10000 loops, best of 3: 652 ns per loop
```

This might not be easy to further optimize, since the time is consumed in many different spots:

```
sage: timeit("MyRingsSingleton.__classcall__()", number=10000)# not tested
10000 loops, best of 3: 306 ns per loop

sage: X = MyRingsSingleton()
sage: timeit("R in X  ", number=10000)                          # not tested
10000 loops, best of 3: 699 ns per loop

sage: c = MyRingsSingleton().__contains__
sage: timeit("c(R)", number = 10000)                            # not tested
10000 loops, best of 3: 661 ns per loop
```

> **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:
>
> ```
> sage: class MyRingsSingleton(Category_singleton):
> ....:     def super_categories(self): return Rings().super_categories()
> sage: class Disaster(MyRingsSingleton): pass
> sage: Disaster()
> Traceback (most recent call last):
> ...
> AssertionError: <class '__main__.Disaster'> is not a direct subclass of <class 'sage.categories
> ```
>
> 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:
>
> ```
> sage: from sage.categories.category_singleton import Category_singleton
> sage: class R(Category_singleton):
> ....:     def super_categories(self): return [Sets()]
> sage: R()
> Category of r
> sage: R().__class__
> <class '__main__.R_with_category'>
> 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'>,
>  <type 'sage.misc.fast_methods.WithEqualityById'>,
>  <type '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'>,
>  <class 'sage.categories.objects.Objects.subcategory_class'>,
>  <type 'object'>]
> sage: R() is R()
> True
> 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__()`

TESTS:

```
sage: import __main__
sage: __main__.MyRings = MyRings
sage: __main__.MyRingsSingleton = MyRingsSingleton
sage: TestSuite(MyRingsSingleton()).run(skip=["_test_category"])
```

> **Note:** The `_test_category` test is failing because `MyRingsSingleton()` is not a subcategory of the join of its super categories:

```
sage: C = MyRingsSingleton()
sage: C.super_categories()
[Category of rngs, Category of semirings]
sage: Rngs() & Semirings()
Category of rings
sage: C.is_subcategory(Rings())
False
```

Oh well; it's not really relevant for those tests.

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*.

6.1 Implementing axioms

6.1.1 Simple case involving a single predefined axiom

Suppose that one wants to provide code (and documentation, tests, ...) for the objects of some existing category `Cs()` that satisfy some predefined axiom `A`.

The first step is to open the hood and check whether there already exists a class implementing the category `Cs().A()`. For example, taking `Cs=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:

```
sage: Semigroups().Infinite().super_categories()
[Category of semigroups, Category of infinite sets]
```

In this case, one needs to create a new class to implement the axiom `Infinite` for this category. This boils down to adding a nested class `Semigroups.Infinite` inheriting from `CategoryWithAxiom`.

In the following example, we implement a category `Cs`, with a subcategory for the objects satisfying the `Finite` axiom defined in the super category `Sets` (we will see later on how to *define* new axioms):

```
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"

sage: Cs().Finite()
Category of finite cs
sage: Cs().Finite().super_categories()
[Category of finite sets, Category of cs]
sage: Cs().Finite().all_super_categories()
[Category of finite cs, Category of finite sets,
 Category of cs, Category of sets, ...]
sage: Cs().Finite().axioms()
frozenset({'Finite'})
```

Now a parent declared in the category `Cs().Finite()` inherits from all the methods of finite sets and of finite C's, as desired:

```
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
```

Note:

- This follows the same idiom as for *Covariant Functorial Constructions*.

- From an object oriented point of view, any subcategory `Cs()` of `Sets` inherits a `Finite` method. Usually `Cs` could complement this method by overriding it with a method `Cs.Finite` which would make a super call to `Sets.Finite` and then do extra stuff.

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

- By defining the axiom `Finite`, `Sets` fixes the semantic of `Cs.Finite()` for all its subcategories `Cs`: namely "the category of Cs which are finite as sets". Hence, for example, `Modules.Free.Finite` cannot be used to model the category of free modules of finite rank, even though their traditional name "finite free modules" might suggest it.

- It may come as a surprise that we can actually use the same name `Finite` for the mixin class and for the method defining the axiom; indeed, by default a class does not have a binding behavior and would completely override the method. See the section *Defining a new axiom* for details and the rationale behind it.

    An alternative would have been to give another name to the mixin class, like `FiniteCategory`. However this would have resulted in more namespace pollution, whereas using `Finite` is already clear, explicit, and easier to remember.

- Under the hood, the category `Cs().Finite()` is aware that it has been constructed from the category `Cs()` by adding the axiom `Finite`:

    ```
    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`:

```
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 trac ticket #15648), one should pass the option `as_name` to `LazyImport`:

```
Finite = LazyImport('sage.categories.finite_groups', 'FiniteGroups', as_name='Finite')
```

in order to prevent `Groups.Finite` to keep on reimporting `FiniteGroups`.

Given that passing this option introduces some redundancy and is error prone, the axiom infrastructure includes a little workaround which makes the `as_name` unnecessary in this case.

Making the category with axiom directly callable

If desired, a category with axiom can be constructed directly through its class rather than through its base category:

```
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')
sage: FiniteCs._base_category_class_and_axiom_origin
'set by __classget__'
```

But calling `FiniteCs()` right after defining the class would have failed (try it!). In general, one needs to set the attribute explicitly:

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

Having to set explicitly this link back from `FiniteCs` to `Cs` introduces redundancy in the code. It would therefore be desirable to have the infrastructure set the link automatically instead (a difficulty is to achieve this while supporting lazy imported categories with axiom).

As a first step, the link is set automatically upon accessing the class from the base category class:

```
sage: Algebras.WithBasis._base_category_class_and_axiom
(<class 'sage.categories.algebras.Algebras'>, 'WithBasis')
sage: Algebras.WithBasis._base_category_class_and_axiom_origin
'set by __classget__'
```

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

```
True
sage: FiniteGroups._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 trac ticket #15741 and the *related discussion in the primer*).

6.1.2 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_with_category.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
sage: Cs().Green.__doc__
'<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.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: Magmas() & AdditiveMagmas() is MagmasAndAdditiveMagmas()
False
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.

6.1.3 Handling multiple axioms, arborescence structure of the code

Prelude

Let us consider the category of magmas, together with two of its axioms, namely `Associative` and `Unital`. An associative magma is a *semigroup* and a unital semigroup is a *monoid*. We have also seen that axioms commute:

```
sage: Magmas().Unital()
Category of unital magmas
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Associative().Unital()
Category of monoids
sage: 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'>
sage: Magmas.Associative
<class 'sage.categories.semigroups.Semigroups'>
sage: 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 `Cs()` 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 `Cs`, 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 `Cs()` satisfying the axioms in S. For example, if `Cs()` is the category of magmas, the pairs (S, D_S) would include:

```
{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 $(D_S)_{S \in \mathcal{S}, S \subset T}$, and build from them a category for the objects of `Cs` satisfying the axioms in T, together with its hierarchy of super categories. If T is in the indexing set $\mathcal{S}$, then the class of the resulting category is directly D_T:

```
sage: C = Magmas().Unital().Inverse().Associative(); C
Category of groups
sage: type(C)
<class 'sage.categories.groups.Groups_with_category'>
```

Otherwise, we get a join category:

```
sage: C = Magmas().Infinite().Unital().Associative(); C
Category of infinite monoids
sage: type(C)
<class 'sage.categories.category.JoinCategory_with_category'>
sage: C.super_categories()
[Category of monoids, Category of infinite sets]
```

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 `Cs`, 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 `Magmas` and `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 *Description of the algorithmic*). 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: 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: 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.AdditiveAssocia`

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.AdditiveAssocia`

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 maintainted 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 trac ticket #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:

```
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:

```
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()`:

```
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()`:

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

```
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()`:

```
sage: DivisionRings().Finite()
Category of finite fields
sage: DivisionRings().Finite() is Fields().Finite()
True
```

Therefore we cannot create a class `DivisionRings.Finite` to hold the desired `extra_super_categories` method, because there is already a class for this category with axiom, namely `Fields.Finite`.

A natural idiom would be to have `DivisionRings.Finite` be a link to `Fields.Finite` (locally introducing an undirected cycle in the arborescence of nested classes). It would be a bit tricky to implement though, since one would need to detect, upon constructing `DivisionRings().Finite()`, that `DivisionRings.Finite` is actually `Fields.Finite`, in order to construct appropriately `Fields().Finite()`; and reciprocally, upon computing the super categories of `Fields().Finite()`, to not try to add `DivisionRings().Finite()` as a super category.

Instead the current idiom is to have a method `DivisionRings.Finite_extra_super_categories` which mimicks the behavior of the would-be `DivisionRings.Finite.extra_super_categories`:

```
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`:

```
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 occurences already implemented in upcoming tickets. 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 trac ticket #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 `axiom()` for conciseness (don't do it for real axioms; they deserve a full documentation!):

```
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:

```
sage: class A1s(Category_singleton):
....:     def super_categories(self):
....:         return [As()]
....:
....:     class B(CategoryWithAxiom):
....:         def C_extra_super_categories(self):
....:             return [As().E(), As().F()]
....:
....:     class E(CategoryWithAxiom):
....:         def F_extra_super_categories(self):
....:             return [As().B(), As().C()]

sage: A1s().B().C()
Category of e f a1s
```

The axioms `B` and `C` do not show up in the name of the obtained category because, for concision, the printing uses some heuristics to not show axioms that are implied by others. But they are satisfied:

```
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):
```

```
....:                def extra_super_categories(self):
....:                    return [As().E(), As().F()]
....:
....:        class E(CategoryWithAxiom):
....:            def F_extra_super_categories(self):
....:                return [As().B(), As().C()]


sage: A2s().B().C()
Category of e f a2s
sage: sorted(A2s().B().C().axioms())
['B', 'C', 'E', 'F']
sage: type(A2s().B().C())
<class '__main__.A2s.B.C_with_category'>
```

As desired, William's theorem and its consequences hold:

```
sage: A2s().B().C() is A2s().E().F()
True
sage: C =  A2s().E().F().D().B().C()
sage: C is A2s().B().C().E().F().D()   # commutativity
True
sage: C is A2s().E().F().E().F().D()   # William's theorem
True
sage: C is A2s().E().E().F().F().D()   # commutativity
True
sage: C is A2s().E().F().D()           # idempotency
True
sage: C is A2s().D().E().F()
True
```

Finally, we "accidentally" implement the category of `b c a1s`, both in `A3s.B.C` and `A3s.E.F`:

```
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:

```
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).

6.2 Specifications

After fixing some vocabulary, we summarize here some specifications about categories and axioms.

6.2.1 The lattice of constructible categories

A mathematical category C is *implemented* if there is a class in Sage modelling it; it is *constructible* if it is either implemented, or is the intersection of *implemented* categories; in the latter case it is modelled by a `JoinCategory`. The comparison of two constructible categories with the `Category.is_subcategory()` method is supposed to model the comparison of the corresponding mathematical categories for inclusion of the objects (see *On the category hierarchy: subcategories and super categories* for details). For example:

```
sage: Fields().is_subcategory(Rings())
True
```

However this modelling may be incomplete. It can happen that a mathematical fact implying that a category A is a subcategory of a category B is not implemented. Still, the comparison should endow the set of constructible categories with a poset structure and in fact a lattice structure.

In this lattice, the join of two categories (`Category.join()`) is supposed to model their intersection. Given that we compare categories for inclusion, it would be more natural to call this operation the *meet*; blames go to me (Nicolas) for originally comparing categories by *amount of structure* rather than by *inclusion*. In practice, the join of two categories may be a strict super category of their intersection; first because this intersection might not be constructible; second because Sage might miss some mathematical information to recover the smallest constructible super category of the intersection.

6.2.2 Axioms

We say that an axiom `A` is *defined by* a category `Cs()` if `Cs` defines an appropriate method `Cs.SubcategoryMethods.A`, with the semantic of the axiom specified in the documentation; for any subcategory `Ds()`, `Ds().A()` models the subcategory of the objects of `Ds()` satisfying `A`. In this case, we say that the axiom `A` is *defined for* the category `Ds()`. Furthermore, `Ds` *implements the axiom* `A` if `Ds` has a category with axiom as nested class `Ds.A`. The category `Ds()` *satisfies* the axiom if `Ds()` is a subcategory of `Cs().A()` (meaning that all the objects of `Ds()` are known to satisfy the axiom `A`).

A digression on the structure of fibers when adding an axiom

Consider the application ϕ_A which maps a category to its category of objects satisfying A. Equivalently, ϕ_A is computing the intersection with the defining category with axiom of A. It follows immediately from the latter that ϕ_A is a regressive endomorphism of the lattice of categories. It restricts to a regressive endomorphism `Cs() |->` `Cs().A()` on the lattice of constructible categories.

This endomorphism may have non trivial fibers, as in our favorite example: `DivisionRings()` and `Fields()` are in the same fiber for the axiom `Finite`:

```
sage: DivisionRings().Finite() is Fields().Finite()
True
```

Consider the intersection S of such a fiber of ϕ_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.

6.2.3 Specifications

- Any constructible category `C` should admit a finite number of larger constructible categories.

- The methods `super_categories`, `extra_super_categories`, and friends should always return strict supercategories.

  For example, to specify that a finite division ring is a finite field, `DivisionRings.Finite_extra_super_categories` should not return `Fields().Finite()`! It could possibly return `Fields()`; but it's preferable to return the largest category that contains the relevant information, in this case `Magmas().Commutative()`, and to let the infrastructure apply the derivations.

- The base category of a `CategoryWithAxiom` should be an implemented category (i.e. not a `JoinCategory`). This is checked by `CategoryWithAxiom._test_category_with_axiom()`.

- Arborescent structure: Let `Cs()` be a category, and S be some set of axioms defined in some super categories of `Cs()` but not satisfied by `Cs()`. Suppose we want to provide a category with axiom for the elements of `Cs()` satisfying the axioms in S. Then, there should be a single enumeration `A1, A2, ..., Ak` without repetition of axioms in S such that `Cs.A1.A2....Ak` is an implemented category. Furthermore, every intermediate step `Cs.A1.A2....Ai` with $i \leq k$ should be a category with axiom having `Ai` as axiom and `Cs.A1.A2....Ai-1` as base category class; this base category class should not satisfy `Ai`. In particular, when some axioms of S can be deduced from previous ones by deduction rules, they should not appear in the enumeration `A1, A2, ..., Ak`.

- In particular, if `Cs()` is a category that satisfies some axiom `A` (e.g. from one of its super categories), then it should not implement that axiom. For example, a category class `Cs` can never have a nested class `Cs.A.A`. Similarly, applying the specification recursively, a category satisfying `A` cannot have a nested class `Cs.A1.A2.A3.A` where `A1, A2, A3` are axioms.

- A category can only implement an axiom if this axiom is defined by some super category. The code has not been systematically checked to support having two super categories defining the same axiom (which should of course have the same semantic). You are welcome to try, at your own risk. :-)

- When a category defines an axiom or functorial construction `A`, this fixes the semantic of `A` for all the subcategories. In particular, if two categories define `A`, then these categories should be independent, and either the semantic of `A` should be the same, or there should be no natural intersection between the two hierarchies of subcategories.

- Any super category of a `CategoryWithParameters` should either be a `CategoryWithParameters` or a `Category_singleton`.

- 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()`.

6.3 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 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:

```
sage: Groups().Finite()
Category of finite groups
sage: Monoids().Finite().Inverse()
Category of finite groups
sage: Sets().Finite() & Monoids().Inverse()
Category of finite groups
```

- Concise idioms for the users (adding axioms, ...)

- Concise idioms and well highlighted hierarchy of bookshelves for the developer (especially with code folding)

- Introspection friendly (listing the axioms, recovering the mixins)

Note: The constructor for instances of this class takes as input the base category. Hence, they should in principle be constructed as:

```
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__()`.

6.4 Upcoming features

6.5 Description of the algorithmic

6.5.1 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. Futhermore, 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 `C1`, `C1.A()` specifies a category `C2` as super category such that `C2.A()` specifies `C3` as super category such that ...; this would clearly cause an infinite execution. Note that this situation violates the specifications since `C1.A()` is supposed to be a subcategory of `C2.A()`, ... so we would have an infinite increasing chain of constructible categories.

It's reasonnable 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.

6.5.2 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 `Cs()` implementing it

In this case `Cs().A()` simply constructs an instance D of `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:

- `Cs()`

- `Bs().A()` for every super category `Bs()` of `Cs()`

- the categories appearing in `D.extra_super_categories()`

This by itself may not be sufficient, due in particular to deduction rules. On may for example discover a new axiom `A1` satisfied by D, imposing to add `A1` 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 `Cs().A()`! To avoid this, `Category.join()` is given an optional argument to specify that the axiom `A` should *not* be applied to `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.

> **Theorem**
>
> Consider the construction of a category C by adding an axiom to a category (or computing of a join). Let H be the hierarchy of implemented categories above C. Let n and m be respectively the number of categories and the number of inheritance edges in H.
> Assuming that the specifications are satisfied, the construction of C involves constructing the categories in H exactly once (and no other category), and at most n join calculations. In particular, the time complexity should be, roughly speaking, bounded by n^2. In particular, it's finite.

> **Remark**
>
> It's actually to be expected that the complexity is more of the order of magnitude of $na + m$, where a is the number of axioms satisfied by C. But this is to be checked in detail, in particular due to the many category inclusion tests involved.

The key argument is that `Category.join` cannot call itself recursively without going through the construction of some implemented category. In turn, the construction of some implemented category C only involves constructing strictly smaller categories, and possibly a direct join calculation whose result is strictly smaller than C. This statement is obvious if C implements the `super_categories` method directly, and easy to check for functorial construction categories. It requires a proof for categories with axioms since there is a recursive join involved.

> **Lemma**
>
> Let C be a category implementing an axiom A. Recall that the construction of `C.A()` involves a single direct join calculation for computing the super categories. No other direct join calculation occur, and the calculation involves only implemented categories that are strictly smaller than `C.A()`.

> **Proof**
>
> Let D be a category involved in the join calculation for the super categories of `C.A()`, and assume by induction that D is strictly smaller than `C.A()`. A category E newly constructed from D can come from:
>
> - `D.(extra_)super_categories()`
>   In this case, the specifications impose that E should be strictly smaller than D and therefore strictly smaller than C.
> - `D.with_axiom_as_tuple('B')` or `D.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.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.

6.6 Conclusion

This is the end of the axioms documentation. Congratulations on having read that far!

6.7 Tests

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!

```
sage: Magmas()
Category of magmas
sage: Magmas().Finite()
Category of finite magmas

sage: Magmas().Unital()
Category of unital magmas
sage: Magmas().Commutative().Unital()
Category of commutative unital magmas
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Associative() & Magmas().Unital().Inverse() & Sets().Finite()
Category of finite groups
sage: _ is Groups().Finite()
True

sage: from sage.categories.semigroups import Semigroups
sage: Semigroups()
Category of semigroups
sage: Semigroups().Finite()
Category of finite semigroups

sage: from sage.categories.modules_with_basis import ModulesWithBasis
sage: ModulesWithBasis(QQ) is Modules(QQ).WithBasis()
True
sage: ModulesWithBasis(ZZ) is Modules(ZZ).WithBasis()
True

sage: Semigroups().Unital()
Category of monoids
sage: Semigroups().Unital().Commutative()
Category of commutative monoids
sage: Semigroups().Commutative()
Category of commutative semigroups
sage: Semigroups().Commutative().Unital()
Category of commutative monoids
sage: Semigroups().Commutative().Unital().super_categories()
[Category of monoids, Category of commutative magmas]

sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative()
Category of commutative additive semigroups

sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: C = CommutativeAdditiveMonoids() & Monoids() & MagmasAndAdditiveMagmas().Distributive(); C
Category of semirings
sage: C is (CommutativeAdditiveMonoids() & Monoids()).Distributive()
True
sage: C.AdditiveInverse()
Category of rings
sage: Rings().axioms()
frozenset({'AdditiveAssociative',
           'AdditiveCommutative',
           'AdditiveInverse',
           'AdditiveUnital',
           'Associative',
           'Distributive',
           'Unital'})
sage: sorted(Rings().axioms())
```

```
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse',
 'AdditiveUnital', 'Associative', 'Distributive', 'Unital']

sage: Domains().Commutative()
Category of integral domains

sage: DivisionRings().Finite() # Wedderburn's theorem
Category of finite fields

sage: FiniteMonoids().Algebras(QQ)
Join of Category of monoid algebras over Rational Field
    and Category of finite dimensional algebras with basis over Rational Field
    and Category of finite set algebras over Rational Field
sage: FiniteGroups().Algebras(QQ)
Category of finite group algebras over Rational Field
```

class sage.categories.category_with_axiom.**Bars**(*s=None*)

    Bases: `sage.categories.category_singleton.Category_singleton`

    A toy singleton category, for testing purposes.

    See also:

    `Blahs`

    Unital_extra_super_categories()

        Return extraneous super categories for the unital objects of `self`.

        This method specifies that a unital bar is a test object. Thus, the categories of unital bars and of unital test objects coincide.

        EXAMPLES:
```
sage: from sage.categories.category_with_axiom import Bars, TestObjects
sage: Bars().Unital_extra_super_categories()
[Category of test objects]
sage: Bars().Unital()
Category of unital test objects
sage: TestObjects().Unital().all_super_categories()
[Category of unital test objects,
 Category of unital blahs,
 Category of test objects,
 Category of bars,
 Category of blahs,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

    super_categories()

        TESTS:
```
sage: from sage.categories.category_with_axiom import Bars
sage: Bars().super_categories()
[Category of blahs]
sage: TestSuite(Bars()).run()
```

class sage.categories.category_with_axiom.**Blahs**(*s=None*)

    Bases: `sage.categories.category_singleton.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.

TESTS:
```
sage: from sage.categories.category_with_axiom import Blahs, TestObjects, Bars
sage: Blahs().Blue_extra_super_categories()
[Category of unital blahs]
sage: Blahs().Blue()                            # todo: not implemented
Category of blue unital blahs
```

class Commutative(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class `Blahs`.**Connected**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
```

```
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class Blahs.**FiniteDimensional**(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class Blahs.**Flying**(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

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`.

TESTS:
```
sage: from sage.categories.category_with_axiom import Blahs, TestObjects, Bars
sage: Blahs().Flying().extra_super_categories()
[Category of unital blahs]
sage: Blahs().Flying()
Category of flying unital blahs
```

class Blahs.**SubcategoryMethods**

Blue()

Commutative()

Connected()

FiniteDimensional()

Flying()

Unital()

class Blahs.**Unital**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class **Blue**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

Blahs.**super_categories**()

TESTS:
```
sage: from sage.categories.category_with_axiom import Blahs
sage: Blahs().super_categories()
[Category of sets]
sage: TestSuite(Blahs()).run()
```

class sage.categories.category_with_axiom.**CategoryWithAxiom**(*base_category*)

Bases: `sage.categories.category.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
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:
```
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
```

We check that this also works when the class is implemented in a separate file, and lazy imported:
```
sage: Sets().Finite
Cached version of <function Finite at ...>
```

There is no binding behavior when accessing `Finite` or `Infinite` from the class of the category instead of the category itself:
```
sage: Sets.Finite
<class 'sage.categories.finite_sets.FiniteSets'>
sage: Sets.Infinite
<class 'sage.categories.sets_cat.Sets.Infinite'>
```

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

__init__ (*base_category*)

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

_repr_object_names ()

The names of the objects of this category, as used by _repr_.

See also:

Category._repr_object_names()

EXAMPLES:

```
sage: FiniteSets()._repr_object_names()
'finite sets'
sage: AlgebrasWithBasis(QQ).FiniteDimensional()._repr_object_names()
'finite dimensional algebras with basis over Rational Field'
sage: Monoids()._repr_object_names()
'monoids'
sage: Semigroups().Unital().Finite()._repr_object_names()
'finite monoids'
sage: Algebras(QQ).Commutative()._repr_object_names()
'commutative algebras over Rational Field'
```

> **Note:** This is implemented by taking _repr_object_names from self._without_axioms(named=True), and adding the names of the relevant axioms in appropriate order.

static **_repr_object_names_static**(*category*, *axioms*)
INPUT:

- base_category – a category

- axioms – a list or iterable of strings

EXAMPLES:
```
sage: from sage.categories.category_with_axiom import CategoryWithAxiom
sage: CategoryWithAxiom._repr_object_names_static(Semigroups(), ["Flying", "Blue"])
'flying blue semigroups'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ), ["Flying", "WithBasis", "Blu
'flying blue algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ), ["WithBasis"])
'algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object_names_static(Sets().Finite().Subquotients(), ["Finite"]
'subquotients of finite sets'
sage: CategoryWithAxiom._repr_object_names_static(Monoids(), ["Unital"])
'monoids'
sage: CategoryWithAxiom._repr_object_names_static(Algebras(QQ['x']['y']), ["Flying", "WithBa
'flying blue algebras with basis over Univariate Polynomial Ring in y over Univariate Polyno
```

If the axioms is a set or frozen set, then they are first sorted using canonicalize_axioms():
```
sage: CategoryWithAxiom._repr_object_names_static(Semigroups(), set(["Finite", "Commutative"
'facade finite commutative semigroups'
```

> **See also:**
>
> _repr_object_names()

> **Note:** The logic here is shared between _repr_object_names() and category.JoinCategory._repr_object_names()

TESTS:
```
sage: from sage.categories.homsets import Homsets
sage: CategoryWithAxiom._repr_object_names_static(Homsets(), ["Endset"])
'endsets'
sage: CategoryWithAxiom._repr_object_names_static(PermutationGroups(), ["FinitelyGeneratedAs
'finitely generated permutation groups'
sage: CategoryWithAxiom._repr_object_names_static(Rings(), ["FinitelyGeneratedAsMagma"])
'finitely generated as magma rings'
```

_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 similarwise for :class'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()

TESTS:
```
sage: Sets().Finite().additional_structure.__module__
'sage.categories.category_with_axiom'
```

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
```

TESTS:
```
sage: from sage.categories.category_with_axiom import TestObjects, CategoryWithAxiom
sage: C = TestObjects().Commutative().Facade()
sage: assert isinstance(C, CategoryWithAxiom)
```

```
sage: C._without_axioms()
Category of test objects
```

extra_super_categories()

Return the extra super categories of a category with axiom.

Default implementation which returns `[]`.

EXAMPLES:
```
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:
```
sage: Magmas().Finite().super_categories()
[Category of magmas, Category of finite sets]
```

Variants:
```
sage: Sets().Finite().super_categories()
[Category of sets]

sage: Monoids().Finite().super_categories()
[Category of monoids, Category of finite semigroups]
```

EXAMPLES:

TESTS:
```
sage: from sage.categories.category_with_axiom import TestObjects
sage: C = TestObjects().FiniteDimensional().Unital().Commutative().Finite()
sage: sorted(C.super_categories(), key=str)
[Category of finite commutative test objects,
 Category of finite dimensional commutative unital test objects,
 Category of finite finite dimensional test objects]
```

class sage.categories.category_with_axiom.**CategoryWithAxiom_over_base_ring**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`, `sage.categories.category_types.Category_over_base_ring`

TESTS:
```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class sage.categories.category_with_axiom.**CategoryWithAxiom_singleton**(*base_category*)

Bases: `sage.categories.category_singleton.Category_singleton`, `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class sage.categories.category_with_axiom.**TestObjects**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

A toy singleton category, for testing purposes.

See also:

`Blahs`

class Commutative(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class Facade(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class TestObjects.Commutative.**Finite**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>
```

```
sage: TestSuite(C).run()
```

class TestObjects.Commutative.**FiniteDimensional**(*base_category*)

> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`
>
> TESTS:
> ```
> sage: C = Sets.Finite(); C
> Category of finite sets
> sage: type(C)
> <class 'sage.categories.finite_sets.FiniteSets_with_category'>
> sage: type(C).__base__.__base__
> <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>
>
> sage: TestSuite(C).run()
> ```

class TestObjects.**FiniteDimensional**(*base_category*)

> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`
>
> TESTS:
> ```
> sage: C = Sets.Finite(); C
> Category of finite sets
> sage: type(C)
> <class 'sage.categories.finite_sets.FiniteSets_with_category'>
> sage: type(C).__base__.__base__
> <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>
>
> sage: TestSuite(C).run()
> ```

class Finite(*base_category*)

> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`
>
> TESTS:
> ```
> sage: C = Sets.Finite(); C
> Category of finite sets
> sage: type(C)
> <class 'sage.categories.finite_sets.FiniteSets_with_category'>
> sage: type(C).__base__.__base__
> <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>
>
> sage: TestSuite(C).run()
> ```

class TestObjects.FiniteDimensional.**Unital**(*base_category*)

> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`
>
> TESTS:
> ```
> sage: C = Sets.Finite(); C
> Category of finite sets
> sage: type(C)
> <class 'sage.categories.finite_sets.FiniteSets_with_category'>
> sage: type(C).__base__.__base__
> <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>
>
> sage: TestSuite(C).run()
> ```

> **class Commutative**(*base_category*)
>
> > Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`
> >
> > TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class TestObjects.**Unital**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

TestObjects.**super_categories**()

TESTS:

```
sage: from sage.categories.category_with_axiom import TestObjects
sage: TestObjects().super_categories()
[Category of bars]
sage: TestSuite(TestObjects()).run()
```

class sage.categories.category_with_axiom.**TestObjectsOverBaseRing**(*base, name=None*)

Bases: `sage.categories.category_types.Category_over_base_ring`

A toy singleton category, for testing purposes.

See also:

`Blahs`

class Commutative(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class Facade(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
```

```
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class TestObjectsOverBaseRing.Commutative.**Finite**(*base_category*)

    Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring](#)

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class TestObjectsOverBaseRing.Commutative.**FiniteDimensional**(*base_category*)

    Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring](#)

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class TestObjectsOverBaseRing.**FiniteDimensional**(*base_category*)

    Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring](#)

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class **Finite**(*base_category*)

    Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring](#)

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class `TestObjectsOverBaseRing.FiniteDimensional.`**`Unital`**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

    class **`Commutative`**(*base_category*)

        Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

        TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class `TestObjectsOverBaseRing.`**`Unital`**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

`TestObjectsOverBaseRing.`**`super_categories`**()

    TESTS:

```
sage: from sage.categories.category_with_axiom import TestObjectsOverBaseRing
sage: TestObjectsOverBaseRing(QQ).super_categories()
[Category of test objects]
sage: TestObjectsOverBaseRing.Unital.an_instance()
Category of unital test objects over base ring over Rational Field
sage: TestObjectsOverBaseRing.FiniteDimensional.Unital.an_instance()
Category of finite dimensional unital test objects over base ring over Rational Field
sage: TestSuite(TestObjectsOverBaseRing(QQ).FiniteDimensional().Unital().Commutative()).run(
```

`sage.categories.category_with_axiom.`**`axiom`**(*axiom*)

    Return a function/method `self -> self._with_axiom(axiom)`.

    This can used as a shorthand to define axioms, in particular in the tests below. Usually one will want to attach documentation to an axiom, so the need for such a shorthand in real life might not be that clear, unless we start creating lots of axioms.

    In the long run maybe this could evolve into an `@axiom` decorator.

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

Upon assigning the result to a class this becomes a method:
```
sage: class As:
....:     def _with_axiom(self, axiom): return self, axiom
....:     Finite = axiom("Finite")
sage: As().Finite()
(<__main__.As instance at ...>, 'Finite')
```

sage.categories.category_with_axiom.**axiom_of_nested_class**(*cls*, *nested_cls*)

Given a class and a nested axiom class, return the axiom.

EXAMPLES:

This uses some heuristics like checking if the nested_cls carries the name of the axiom, or is built by appending or prepending the name of the axiom to that of the class:
```
sage: from sage.categories.category_with_axiom import TestObjects, axiom_of_nested_class
sage: axiom_of_nested_class(TestObjects, TestObjects.FiniteDimensional)
'FiniteDimensional'
sage: axiom_of_nested_class(TestObjects.FiniteDimensional, TestObjects.FiniteDimensional.Finite)
'Finite'
sage: axiom_of_nested_class(Sets, FiniteSets)
'Finite'
sage: axiom_of_nested_class(Algebras, AlgebrasWithBasis)
'WithBasis'
```

In all other cases, the nested class should provide an attribute _base_category_class_and_axiom:
```
sage: Semigroups._base_category_class_and_axiom
(<class 'sage.categories.magmas.Magmas'>, 'Associative')
sage: axiom_of_nested_class(Magmas, Semigroups)
'Associative'
```

sage.categories.category_with_axiom.**base_category_class_and_axiom**(*cls*)

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:
```
sage: from sage.categories.category_with_axiom import base_category_class_and_axiom, CategoryWit
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:

```
sage: class FacadeSemigroups(CategoryWithAxiom):
....:     pass
sage: base_category_class_and_axiom(FacadeSemigroups)
Traceback (most recent call last):
...
AssertionError: Missing (lazy import) link for <class 'sage.categories.semigroups.Semigroups'> t

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 [trac ticket #9107](#) is fixed. Anyway this feature has not been needed so far:

```
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
...
```

---

sage.categories.category_with_axiom.**uncamelcase**(*s*, *separator=' '*)
    EXAMPLES:
```
sage: sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras")
'finite dimensional algebras'
sage: sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras", "_")
'finite_dimensional_algebras'
```

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 trac ticket #16291.

class sage.categories.map.**FormalCompositeMap**

Bases: `sage.categories.map.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.

EXAMPLE:

```
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: from sage.categories.morphism import SetMorphism
sage: f = SetMorphism(Hom(R, S, Rings()), lambda p: p[0]*a^p.degree())
sage: g = S.hom([2*x])
sage: f*g
Composite map:
  From: Univariate Polynomial Ring in a over Rational Field
  To:   Univariate Polynomial Ring in a over Rational Field
  Defn:   Ring morphism:
          From: Univariate Polynomial Ring in a over Rational Field
          To:   Univariate Polynomial Ring in x over Rational Field
          Defn: a |--> 2*x
        then
          Generic morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Univariate Polynomial Ring in a over Rational Field
sage: g*f
Composite map:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field
  Defn:   Generic morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Univariate Polynomial Ring in a over Rational Field
        then
          Ring morphism:
```

```
            From: Univariate Polynomial Ring in a 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)
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()`.

EXAMPLE:
```
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: fg = f * g
sage: fg.first() == g
True
sage: fg == fg.then() * fg.first()
True
```

is_injective()

Tell whether `self` is injective.

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

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

If both constituents are injective, the composition is injective:
```
sage: from sage.categories.map import FormalCompositeMap
sage: c1 = FormalCompositeMap(Hom(QQ^1, V2, phi1.category_for()), phi1, phi2)
sage: c1.is_injective()
True
```

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

```
sage: psi1 = V2.hom(Matrix([[1, 2], [3, 4], [5, 6]]), V1)
sage: c2 = FormalCompositeMap(Hom(V1, V1, phi2.category_for()), phi2, psi1)
sage: c2.is_injective()
Traceback (most recent call last):
...
NotImplementedError: Not enough information to deduce injectivity.
```

If the first map is surjective and the second map is not injective, then the composition is not injective:

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

is_surjective()

Tell whether `self` is surjective.

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

EXAMPLE:

```
sage: from sage.categories.map import FormalCompositeMap
sage: V3 = QQ^3
sage: V2 = QQ^2
sage: V1 = QQ^1
```

If both maps are surjective, the composition is surjective:

```
sage: phi32 = V3.hom(Matrix([[1, 2], [3, 4], [5, 6]]), V2)
sage: phi21 = V2.hom(Matrix([[1], [1]]), V1)
sage: c_phi = FormalCompositeMap(Hom(V3, V1, phi32.category_for()), phi32, phi21)
sage: c_phi.is_surjective()
True
```

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

```
sage: FormalCompositeMap(Hom(V3, V1, phi32.category_for()), phi32, V2.hom(Matrix([[0], [0]])
False
```

If the second map is an isomorphism and the first map is not surjective, then the composition is not surjective:

```
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()), V2.hom(Matrix([[0], [0]]), V1),
False
```

Otherwise, surjectivity of the composition cannot be determined:

```
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()),
...       V2.hom(Matrix([[1, 1], [1, 1]]), V2),
...       V2.hom(Matrix([[1], [1]]), V1)).is_surjective()
Traceback (most recent call last):
...
NotImplementedError: Not enough information to deduce surjectivity.
```

second(*args*, **kwds*)

Deprecated: Use `then()` instead. See trac ticket #16291 for details.

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()`.

EXAMPLE:

```
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).then() == f
True
```

class `sage.categories.map.`**`Map`**

    Bases: `sage.structure.element.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)
x^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:

```
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
sage: Y = ZZ
sage: phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])
sage: phi.category_for()
Category of rings
sage: phi.category()
Category of homsets of unital magmas and additive unital additive magmas
sage: R.<x,y> = QQ[]
sage: f = R.hom([x+y, x-y], R)
sage: f.category_for()
Join of Category of unique factorization domains and Category of commutative algebras over q
sage: f.category()
Category of endsets of unital magmas and right modules over quotient fields and left modules
```

        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:

```
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 ϕ in Hom(Y, Z)

    OUTPUT:

    An element of Hom(X, Z) obtained by composing self with ϕ. If no canonical ϕ exists, a TypeError is raised.

    EXAMPLES:

```
sage: mor = QQ.coerce_map_from(ZZ)
sage: mor.extend_codomain(RDF)
Composite map:
  From: Integer Ring
  To:   Real Double Field
  Defn:   Natural morphism:
          From: Integer Ring
          To:   Rational Field
        then
          Native morphism:
          From: Rational Field
          To:   Real Double Field
sage: mor.extend_codomain(GF(7))
Traceback (most recent call last):
...
TypeError: No coercion from Rational Field to Finite Field of size 7
```

extend_domain(*new_domain*)

    INPUT:

> - `self` – a member of Hom(Y, Z)
>
> - `new_codomain` – an object X such that there is a canonical coercion ϕ in Hom(X, Y)

    OUTPUT:

    An element of Hom(X, Z) obtained by composing self with ϕ. If no canonical ϕ 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
```

is_injective()

Tells whether the map is injective (not implemented in the base class).

TEST:
```
sage: from sage.categories.map import Map
sage: f = Map(Hom(QQ, ZZ, Rings()))
sage: f.is_injective()
Traceback (most recent call last):
...
NotImplementedError: <type 'sage.categories.map.Map'>
```

is_surjective()

Tells whether the map is surjective (not implemented in the base class).

TEST:
```
sage: from sage.categories.map import Map
sage: f = Map(Hom(QQ, ZZ, Rings()))
sage: f.is_surjective()
Traceback (most recent call last):
...
NotImplementedError: <type 'sage.categories.map.Map'>
```

parent()

Return the homset containing this map.

Note: The method _make_weak_references(), that is used for the maps found by the coercion system, needs to remove the usual strong reference from the coercion map to the homset containing it. As long as the user keeps strong references to domain and codomain of the map, we will be able to reconstruct the homset. However, a strong reference to the coercion map does not prevent the domain from garbage collection!

EXAMPLES:
```
sage: Q = QuadraticField(-5)
sage: phi = CDF._internal_convert_map_from(Q)
sage: print phi.parent()
Set of field embeddings from Number Field in a with defining polynomial x^2 + 5 to Complex D
```

We now demonstrate that the reference to the coercion map ϕ does not prevent Q from being garbage collected:

```
sage: import gc
sage: del Q
sage: _ = gc.collect()
sage: phi.parent()
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:

```
sage: Q = QuadraticField(-5)
sage: phi = CDF.convert_map_from(Q)
sage: print phi.parent()
Set of field embeddings from Number Field in a with defining polynomial x^2 + 5 to Complex D
sage: import gc
sage: del Q
sage: _ = gc.collect()
sage: phi.parent()
Set of field embeddings from Number Field in a with defining polynomial x^2 + 5 to Complex D
```

post_compose(*left*)

> INPUT:
>
> > •`self` – a Map in some `Hom(X, Y, category_right)`
> >
> > •`left` – a Map in some `Hom(Y, Z, category_left)`
>
> Returns the composition of `self` followed by `right` as a morphism in `Hom(X, Z, category)` where `category` is the meet of `category_left` and `category_right`.
>
> Caveat: see the current restrictions on `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_xy.post_compose(phi_yz); phi_xz
> Composite map:
>   From: Univariate Polynomial Ring in x over Integer Ring
>   To:   Rational Field
>   Defn:   Generic morphism:
>           From: Univariate Polynomial Ring in x over Integer Ring
>           To:   Integer Ring
>         then
>           Generic morphism:
>           From: Integer Ring
>           To:   Rational Field
> sage: phi_xz.category_for()
> Category of monoids
> ```

pre_compose(*right*)

> INPUT:
>
> > •`self` – a Map in some `Hom(Y, Z, category_left)`
> >
> > •`left` – a Map in some `Hom(X, Y, category_right)`

Returns the composition of `right` followed by `self` as a morphism in `Hom(X, Z, category)` where `category` is the meet of `category_left` and `category_right`.

EXAMPLES:

```
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
```

section()

Return a section of self.

NOTE:

> By default, it returns `None`. You may override it in subclasses.

TEST:

```
sage: R.<x,y> = QQ[]
sage: f = R.hom([x+y, x-y], R)
sage: print f.section()
None

sage: f = QQ.coerce_map_from(ZZ); f
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: ff = f.section(); ff
Generic map:
  From: Rational Field
  To:   Integer Ring
sage: ff(4/2)
2
sage: parent(ff(4/2)) is ZZ
True
sage: ff(1/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

class `sage.categories.map.`**`Section`**

Bases: `sage.categories.map.Map`

A formal section of a map.

NOTE:

> Call methods are not implemented for the base class `Section`.

EXAMPLE:

```
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: <type 'sage.categories.map.Section'>
```

sage.categories.map.**is_Map**(*x*)

> Auxiliary function: Is the argument a map?

> EXAMPLE:

```
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.

> TEST:

```
sage: R.<x,y> = QQ[]
sage: f = R.hom([x+y, x-y], R)
sage: f == loads(dumps(f))  # indirect doctest
True
```

HOMSETS

The class `Hom` is the base class used to represent sets of morphisms between objects of a given category. `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 are not stongly ref'ed anymore.

EXAMPLES:

In the following, the `Hom` object is indeed cached:

```
sage: K = GF(17)
sage: H = Hom(ZZ, K)
sage: H
Set of Homomorphisms from Integer Ring to Finite Field of size 17
sage: H is Hom(ZZ, K)
True
```

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

```
sage: for p in prime_range(200):
...       K = GF(p)
...       H = Hom(ZZ, K)
...
sage: import gc
sage: _ = gc.collect()
sage: from sage.rings.finite_rings.finite_field_prime_modn import FiniteField_prime_modn as FF
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]
sage: len(L)
2
sage: L
[Finite Field of size 2, Finite Field of size 199]
```

AUTHORS:

- David Kohel and William Stein

- David Joyner (2005-12-17): added examples

- William Stein (2006-01-14): Changed from Homspace to Homset.

- Nicolas M. Thiery (2008-12-): Updated for the new category framework

- Simon King (2011-12): Use a weak cache for homsets

- Simon King (2013-02): added examples

sage.categories.homset.**End**(*X*, *category=None*)
    Create the set of endomorphisms of `X` in the category category.

    INPUT:

- X – anything

- `category` – (optional) category in which to coerce X

OUTPUT:

A set of endomorphisms in category

EXAMPLES:
```
sage: V = VectorSpace(QQ, 3)
sage: End(V)
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)
sage: S = End(G); S
Set of Morphisms from Alternating group of order 3!/2 as a permutation group to Alternating grou
sage: from sage.categories.homset import is_Endset
sage: is_Endset(S)
True
sage: S.domain()
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)
sage: G.category()
Category of finite groups
sage: H = Hom(G,G)
sage: H.homset_category()
Category of groups
sage: H.category()
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)
sage: Hom(V, V)
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)
sage: Hom(G, G)
Set of Morphisms from Alternating group of order 3!/2 as a permutation group to Alternating grou
sage: Hom(ZZ, QQ, Sets())
Set of Morphisms from Integer Ring to Rational Field in Category of sets

sage: Hom(FreeModule(ZZ,1), FreeModule(QQ,1))
Set of Morphisms from Ambient free module of rank 1 over the principal ideal domain Integer Ring
sage: Hom(FreeModule(QQ,1), FreeModule(ZZ,1))
Set of Morphisms from Vector space of dimension 1 over Rational Field to Ambient free module of
```

Here, we test against a memory leak that has been fixed at trac ticket #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
sage: L = [x for x in gc.get_objects() if isinstance(x, FF)]
sage: len(L), L[0], L[len(L)-1]
(2, Finite Field of size 2, 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
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)
Set of Morphisms from Integer Ring
 to Symmetric group of order 3! as a permutation group
 in Join of Category of monoids and Category of enumerated sets
```

Otherwise, if `category` is specified, then `category` is used, after checking that X and Y are indeed in `category`:
```
sage: Hom(X, Y, Magmas())
Set of Morphisms from Integer Ring to Symmetric group of order 3! as a permutation group in Cate

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

TESTS:

Homset are unique parents:
```
sage: k = GF(5)
sage: H1 = Hom(k,k)
sage: H2 = Hom(k,k)
sage: H1 is H2
True
```

Moreover, if no category is provided, then the result is identical with the result for the meet of the categories of the domain and the codomain:
```
sage: Hom(QQ, ZZ) is Hom(QQ,ZZ, Category.meet([QQ.category(), ZZ.category()]))
True
```

Some doc tests in sage.rings (need to) break the unique parent assumption. But if domain or codomain are not unique parents, then the homset will not fit. That is to say, the hom set found in the cache will have a (co)domain that is equal to, but not identical with, the given (co)domain.

By trac ticket #9138, we abandon the uniqueness of homsets, if the domain or codomain break uniqueness:
```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: Q.<x,y,z>=MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P == Q
True
sage: P is Q
False
```

Hence, P and Q are not unique parents. By consequence, the following homsets aren't either:
```
sage: H1 = Hom(QQ,P)
sage: H2 = Hom(QQ,Q)
sage: H1 == H2
True
sage: H1 is H2
False
```

It is always the most recently constructed homset that remains in the cache:
```
sage: H2 is Hom(QQ,Q)
True
```

Variation on the theme:
```
sage: U1 = FreeModule(ZZ,2)
sage: U2 = FreeModule(ZZ,2,inner_product_matrix=matrix([[1,0],[0,-1]]))
sage: U1 == U2, U1 is U2
(True, False)
sage: V = ZZ^3
```

```
sage: H1 = Hom(U1, V); H2 = Hom(U2, V)
sage: H1 == H2, H1 is H2
(True, False)
sage: H1 = Hom(V, U1); H2 = Hom(V, U2)
sage: H1 == H2, H1 is H2
(True, False)
```

Since trac ticket #11900, the meet of the categories of the given arguments is used to determine the default category of the homset. This can also be a join category, as in the following example:

```
sage: PA = Parent(category=Algebras(QQ))
sage: PJ = Parent(category=Rings() & Modules(QQ))
sage: Hom(PA,PJ)
Set of Homomorphisms from <type 'sage.structure.parent.Parent'> to <type 'sage.structure.parent.
sage: Hom(PA,PJ).category()
Category of homsets of unital magmas and right modules over Rational Field and left modules over
sage: Hom(PA,PJ, Rngs())
Set of Morphisms from <type 'sage.structure.parent.Parent'> to <type 'sage.structure.parent.Pare
```

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).

TESTS:

Facade parents over plain Python types are supported:

```
sage: R = sage.structure.parent.Set_PythonType(int)
sage: S = sage.structure.parent.Set_PythonType(float)
sage: Hom(R, S)
Set of Morphisms from Set of Python objects of type 'int' to Set of Python objects of type 'floa
```

Checks that the domain and codomain are in the specified category. Case of a non parent:

```
sage: S = SimplicialComplex([[1,2], [1,4]]); S.rename("S")
sage: Hom(S, S, SimplicialComplexes())
Set of Morphisms from S to S in Category of simplicial complexes

sage: H = Hom(Set(), S, Sets())
Traceback (most recent call last):
...
ValueError: S is not in Category of sets

sage: H = Hom(S, Set(), Sets())
Traceback (most recent call last):
...
ValueError: S is not in Category of sets

sage: H = Hom(S, S, ChainComplexes(QQ))
Traceback (most recent call last):
...
ValueError: S is not in Category of chain complexes over Rational Field
```

Those checks are done with the natural idiom `X in category`, and not

`X.category().is_subcategory(category)` as it used to be before trac ticket #16275 (see trac ticket #15801 for a real use case):

```
sage: class PermissiveCategory(Category):
....:     def super_categories(self): return [Objects()]
....:     def __contains__(self, X): return True
sage: C = PermissiveCategory(); C.rename("Permissive category")
sage: S.category().is_subcategory(C)
False
sage: S in C
True
sage: Hom(S, S, C)
Set of Morphisms from S to S in Permissive category
```

With `check=False`, unitialized parents, as can appear upon unpickling, are supported. Case of a parent:

```
sage: cls = type(Set())
sage: S = unpickle_newobj(cls, ())  # A non parent
sage: H = Hom(S, S, SimplicialComplexes(), check=False);
sage: H = Hom(S, S, Sets(),                check=False)
sage: H = Hom(S, S, ChainComplexes(QQ),    check=False)
```

Case of a non parent:

```
sage: cls = type(SimplicialComplex([[1,2], [1,4]]))
sage: S = unpickle_newobj(cls, ())
sage: H = Hom(S, S, Sets(),                check=False)
sage: H = Hom(S, S, Groups(),              check=False)
sage: H = Hom(S, S, SimplicialComplexes(), check=False)
```

Typical example where unpickling involves calling Hom on an unitialized parent:

```
sage: P.<x,y> = QQ['x,y']
sage: Q = P.quotient([x^2-1,y^2-1])
sage: q = Q.an_element()
sage: explain_pickle(dumps(Q))
pg_...
... = pg_dynamic_class('QuotientRing_generic_with_category', (pg_QuotientRing_generic, pg_getatt
si... = unpickle_newobj(..., ())
...
si... = pg_unpickle_MPolynomialRing_libsingular(..., ('x', 'y'), ...)
si... = ... pg_Hom(si..., si..., ...) ...
sage: Q == loads(dumps(Q))
True
```

class sage.categories.homset.**Homset**(*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: `sage.structure.parent.Set_generic`

The class for collections of morphisms in a category.

EXAMPLES:

```
sage: H = Hom(QQ^2, QQ^3)
sage: loads(H.dumps()) is H
True
```

Homsets of unique parents are unique as well:

```
sage: H = End(AffineSpace(2, names='x,y'))
sage: loads(dumps(AffineSpace(2, names='x,y'))) is AffineSpace(2, names='x,y')
True
sage: loads(dumps(H)) is H
```

```
True
```

Conversely, homsets of non-unique parents are non-unique:

> sage: H = End(ProjectiveSpace(2, names='x,y,z')) sage: loads(dumps(ProjectiveSpace(2, names='x,y,z'))) is ProjectiveSpace(2, names='x,y,z') False sage: loads(dumps(ProjectiveSpace(2, names='x,y,z'))) == ProjectiveSpace(2, names='x,y,z') True sage: loads(dumps(H)) is H False sage: loads(dumps(H)) == H True

codomain()

Return the codomain of this homset.

EXAMPLES:
```
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
```

coerce_map_from_c(*R*)

> **Warning:** For compatibility with old coercion model. DO NOT USE!

TESTS:
```
sage: H = Hom(ZZ^2, ZZ^3)
sage: H.coerce_map_from_c(ZZ)
```

domain()

Return the domain of this homset.

EXAMPLES:
```
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().domain()
Univariate Polynomial Ring in t over Integer Ring
sage: f.domain() is f.parent().domain()
True
```

element_class_set_morphism()

A base class for elements of this homset which are also `SetMorphism`, i.e. implemented by mean of a Python function.

This is currently plain `SetMorphism`, without inheritance from categories.

Todo

Refactor during the upcoming homset cleanup.

EXAMPLES:
```
sage: H = Hom(ZZ, ZZ)
sage: H.element_class_set_morphism
<type 'sage.categories.morphism.SetMorphism'>
```

get_action_c(*R*, *op*, *self_on_left*)

> **Warning:** For compatibility with old coercion model. DO NOT USE!

TESTS:

```
sage: H = Hom(ZZ^2, ZZ^3)
sage: H.get_action_c(ZZ, operator.add, ZZ)
```

homset_category()

Return the category that this is a Hom in, i.e., this is typically the category of the domain or codomain object.

EXAMPLES:

```
sage: H = Hom(AlternatingGroup(4), AlternatingGroup(7))
sage: H.homset_category()
Category of finite permutation groups
```

identity()

The identity map of this homset.

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

EXAMPLES:

```
sage: H = Hom(QQ,QQ)
sage: H.identity()
Identity endomorphism of Rational Field
sage: H = Hom(ZZ,QQ)
sage: H.identity()
Traceback (most recent call last):
...
TypeError: Identity map only defined for endomorphisms. Try natural_map() instead.
sage: H.natural_map()
Ring Coercion morphism:
  From: Integer Ring
  To:   Rational Field
```

is_endomorphism_set()

Return `True` if the domain and codomain of `self` are the same object.

EXAMPLES:

```
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().is_endomorphism_set()
False
sage: g = P.hom([2*t])
sage: g.parent().is_endomorphism_set()
True
```

natural_map()

Return the "natural map" of this homset.

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

EXAMPLES:

```
sage: H = Hom(ZZ['t'],QQ['t'], CommutativeAdditiveGroups())
sage: H.natural_map()
Coercion morphism:
  From: Univariate Polynomial Ring in t over Integer Ring
```

```
   To:   Univariate Polynomial Ring in t over Rational Field
sage: H = Hom(QQ['t'],GF(3)['t'])
sage: H.natural_map()
Traceback (most recent call last):
...
TypeError: Natural coercion morphism from Univariate Polynomial Ring in t over Rational Fiel
```

reversed()

> Return the corresponding homset, but with the domain and codomain reversed.

> EXAMPLES:

```
sage: H = Hom(ZZ^2, ZZ^3); H
Set of Morphisms from Ambient free module of rank 2 over
the principal ideal domain Integer Ring to Ambient free
module of rank 3 over the principal ideal domain Integer
Ring in Category of modules with basis over (euclidean
domains and infinite enumerated sets)
sage: type(H)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: H.reversed()
Set of Morphisms from Ambient free module of rank 3 over
the principal ideal domain Integer Ring to Ambient free
module of rank 2 over the principal ideal domain Integer
Ring in Category of modules with basis over (euclidean
domains and infinite enumerated sets)
sage: type(H.reversed())
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
```

class sage.categories.homset.**HomsetWithBase**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

> Bases: sage.categories.homset.Homset

> TESTS:

```
sage: X = ZZ['x']; X.rename("X")
sage: Y = ZZ['y']; Y.rename("Y")
sage: class MyHomset(HomsetWithBase):
...         def my_function(self, x):
...             return Y(x[0])
...         def _an_element_(self):
...             return sage.categories.morphism.SetMorphism(self, self.my_function)
...
sage: import __main__; __main__.MyHomset = MyHomset # fakes MyHomset being defined in a Python m
sage: H = MyHomset(X, Y, category=Monoids())
sage: H
Set of Morphisms from X to Y in Category of monoids
sage: H.base()
Integer Ring
sage: TestSuite(H).run()
```

sage.categories.homset.**end**(*X*, *f*)

> Return End(X)(f), where f is data that defines an element of End(X).

> EXAMPLES:

```
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 `Hom(X,Y)(f)`, where `f` is data that defines an element of `Hom(X,Y)`.

EXAMPLES:

```
sage: phi = hom(QQ['x'], 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
```

MORPHISMS

AUTHORS:

- William Stein: initial version

- David Joyner (12-17-2005): added examples

- Robert Bradshaw (2007-06-25) Pyrexification

class sage.categories.morphism.**CallMorphism**
    Bases: sage.categories.morphism.Morphism

class sage.categories.morphism.**FormalCoercionMorphism**
    Bases: sage.categories.morphism.Morphism

class sage.categories.morphism.**IdentityMorphism**
    Bases: sage.categories.morphism.Morphism

class sage.categories.morphism.**Morphism**
    Bases: sage.categories.map.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

    sage: K = CyclotomicField(12)
    sage: L = CyclotomicField(132)
    sage: phi = L._internal_coerce_map_from(K)
    sage: phi.category()
    Category of homsets of unital magmas and additive unital additive magmas
    ```

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)
    sage: L = CyclotomicField(132)
    sage: phi = L._internal_coerce_map_from(K)
    ```

```
sage: phi.is_endomorphism()
False
```

is_identity()
> Return `True` if this morphism is the identity morphism.

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

> EXAMPLES:
> ```
> sage: R.<t> = ZZ[]
> sage: f = R.hom([t])
> sage: f.is_identity()
> True
> sage: g = R.hom([t+1])
> sage: g.is_identity()
> False
> ```

> A morphism between two different spaces cannot be the identity:
> ```
> sage: R2.<t2> = QQ[]
> sage: h = R.hom([t2])
> sage: h.is_identity()
> False
> ```

> AUTHOR:

> > •Xavier Caruso (2012-06-29)

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[]
> sage: Y.<y> = ZZ[]
> sage: x^2 + y
> Traceback (most recent call last):
> ...
> TypeError: unsupported operand parent(s) for '+': 'Univariate Polynomial Ring in x over Inte
> ```

> Let us declare a coercion from $\mathbf{Z}[x]$ to $\mathbf{Z}[z]$:
> ```
> sage: Z.<z> = ZZ[]
> sage: phi = Hom(X, Z)(z)
> sage: phi(x^2+1)
> z^2 + 1
> sage: phi.register_as_coercion()
> ```

> Now we can add elements from $\mathbf{Z}[x]$ and $\mathbf{Z}[z]$, because the elements of the former are allowed to be implicitly coerced into the later:
> ```
> sage: x^2 + z
> z^2 + z
> ```

> Caveat: the registration of the coercion must be done before any other coercion is registered or discovered:

```
sage: phi = Hom(X, Y)(y)
sage: phi.register_as_coercion()
Traceback (most recent call last):
...
AssertionError: coercion from Univariate Polynomial Ring in x over Integer Ring to Univariat
```

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 **Z** through the sign map:

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

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

Bases: `sage.categories.morphism.Morphism`

INPUT:

- `parent` – a Homset

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

EXAMPLES:

```
sage: from sage.categories.morphism import SetMorphism
sage: f = SetMorphism(Hom(QQ, ZZ, Sets()), numerator)
sage: f.parent()
Set of Morphisms from Rational Field to Integer Ring in Category of sets
sage: f.domain()
Rational Field
sage: f.codomain()
Integer Ring
sage: TestSuite(f).run()
```

sage.categories.morphism.**is_Morphism**(*x*)

sage.categories.morphism.**make_morphism**(*_class*, *parent*, *_dict*, *_slots*)

FUNCTORS

AUTHORS:

- David Kohel and William Stein

- David Joyner (2005-12-17): examples

- Robert Bradshaw (2007-06-23): Pyrexify

- 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 loosing 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:

```
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:

```
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:

```
sage: ForgetfulFunctor(abgrps, abgrps) == IdentityFunctor(abgrps)
True
```

class sage.categories.functor.**ForgetfulFunctor_generic**

Bases: sage.categories.functor.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 fields to Category of fields
sage: F(GF(3))
Finite Field of size 3
```

class `sage.categories.functor.`**`Functor`**

    Bases: `sage.structure.sage_object.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
sage: I.domain()
Category of commutative additive groups
sage: is_Functor(I)
True
```

Note that by default, an instance of the class Functor is coercion from the domain into the codomain. The above subclasses overloaded this behaviour. Here we illustrate the default:
```
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))
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
  Defn: Induced from base ring by
        Ring morphism:
          From: Univariate Polynomial Ring in x over Integer Ring
          To:   Multivariate Polynomial Ring in a, b over Rational Field
          Defn: x |--> a + b
sage: p = R1['t']('(-x^2 + x)*t^2 + (x^2 - x)*t - 4*x^2 - x + 1')
sage: F(f)(p)
(-a^2 - 2*a*b - b^2 + a + b)*t^2 + (a^2 + 2*a*b + b^2 - a - b)*t - 4*a^2 - 8*a*b - 4*b^2 - a - b
```

codomain()

The codomain of self

EXAMPLE:

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

domain()

The domain of self

EXAMPLE:

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

sage.categories.functor.**IdentityFunctor**(*C*)

> Construct the identity functor of the given category.

> INPUT:

> A category, `C`.

> OUTPUT:

> The identity functor in `C`.

> EXAPLES:

> ```
> sage: rings = Rings()
> sage: F = IdentityFunctor(rings)
> sage: F(ZZ['x','y']) is ZZ['x','y']
> True
> ```

class sage.categories.functor.**IdentityFunctor_generic**(*C*)

> Bases: `sage.categories.functor.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
> ```

> TESTS:

> ```
> sage: R = IdentityFunctor(Rings())
> sage: P, _ = QQ['t'].construction()
> sage: R == P
> False
> sage: P == R
> False
> sage: R == QQ
> False
> ```

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:

```
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
```

COERCION VIA CONSTRUCTION FUNCTORS

class sage.categories.pushout.**AlgebraicClosureFunctor**

Bases: sage.categories.pushout.ConstructionFunctor

Algebraic Closure.

EXAMPLE:
```
sage: F = CDF.construction()[0]
sage: F(QQ)
Algebraic Field
sage: F(RR)
Complex Field with 53 bits of precision
sage: F(F(QQ)) is F(QQ)
True
```

merge (*other*)

Mathematically, Algebraic Closure subsumes Algebraic Extension. However, it seems that people do want to work with algebraic extensions of RR. Therefore, we do not merge with algebraic extension.

TEST:
```
sage: K.<a>=NumberField(x^3+x^2+1)
sage: CDF.construction()[0].merge(K.construction()[0]) is None
True
sage: CDF.construction()[0].merge(CDF.construction()[0])
AlgebraicClosureFunctor
```

class sage.categories.pushout.**AlgebraicExtensionFunctor** (*polys*, *names*, *embeddings*, *cyclotomic=None*, ***kwds*)

Bases: sage.categories.pushout.ConstructionFunctor

Algebraic extension (univariate polynomial ring modulo principal ideal).

EXAMPLE:
```
sage: K.<a> = NumberField(x^3+x^2+1)
sage: F = K.construction()[0]
sage: F(ZZ['t'])
Univariate Quotient Polynomial Ring in a over Univariate Polynomial Ring in t over Integer Ring
```

Note that, even if a field is algebraically closed, the algebraic extension will be constructed as the quotient of a univariate polynomial ring:
```
sage: F(CC)
Univariate Quotient Polynomial Ring in a over Complex Field with 53 bits of precision with modul
sage: F(RR)
Univariate Quotient Polynomial Ring in a over Real Field with 53 bits of precision with modulus
```

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: F(ZZ)
Order in Number Field in a with defining polynomial x^3 + x^2 + 1
```

This also holds for non-absolute number fields:

```
sage: K.<a,b> = NumberField([x^3+x^2+1,x^2+x+1])
sage: F = K.construction()[0]
sage: O = F(ZZ); O
Relative Order in Number Field in a with defining polynomial x^3 + x^2 + 1 over its base field
```

Unfortunately, the relative number field is not a unique parent:

```
sage: O.ambient() is K
False
sage: O.ambient() == K
True
```

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)
sage: L.<b> = K.extension(x^2+a)
sage: F,R = L.construction()
sage: prod(F.expand())(R) == L
True
sage: K = NumberField([x^2-2, x^2-3],'a')
sage: F, R = K.construction()
sage: F
AlgebraicExtensionFunctor
sage: L = F.expand(); L
[AlgebraicExtensionFunctor, AlgebraicExtensionFunctor]
sage: L[-1](QQ)
Number Field in a1 with defining polynomial x^2 - 3
```

merge(*other*)

>   Merging with another `AlgebraicExtensionFunctor`.

>   INPUT:

>   `other` – Construction Functor.

>   OUTPUT:

>   •If `self==other`, `self` is returned.

>   •If `self` and `other` are simple extensions and both provide an embedding, then it is tested whether one of the number fields provided by the functors coerces into the other; the functor associated with the target of the coercion is returned. Otherwise, the construction functor associated with the pushout of the codomains of the two embeddings is returned, provided that it is a number field.

>   •If these two extensions are defined by Conway polynomials over finite fields, merges them into a single extension of degree the lcm of the two degrees.

>   •Otherwise, None is returned.

>   REMARK:

Algebraic extension with embeddings currently only works when applied to the rational field. This is why we use the admittedly strange rule above for merging.

EXAMPLES:

The following demonstrate coercions for finite fields using Conway or pseudo-Conway polynomials:

```
sage: k = GF(3^2, conway=True, prefix='z'); a = k.gen()
sage: l = GF(3^3, conway=True, prefix='z'); b = l.gen()
sage: a + b # indirect doctest
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
```

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

```
sage: m = GF(3^5, conway=True, prefix='z'); c = m.gen()
sage: (a+b)+c == a+(b+c) # indirect doctest
True
sage: from sage.categories.pushout import pushout
sage: n = pushout(k, l)
sage: o = pushout(l, m)
sage: q = pushout(n, o)
sage: q(o(b)) == q(n(b)) # indirect doctest
True
```

Coercion is also available for number fields:

```
sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^8-x^4+1, embedding=CDF.0)
sage: M1.<c1> = NumberField(x^2+x+1, embedding=b^4-1)
sage: M2.<c2> = NumberField(x^2+1, embedding=-b^6)
sage: M1.coerce_map_from(M2)
sage: M2.coerce_map_from(M1)
sage: c1+c2; parent(c1+c2)    #indirect doctest
-b^6 + b^4 - 1
Number Field in b with defining polynomial x^8 - x^4 + 1
sage: pushout(M1['x'],M2['x'])
Univariate Polynomial Ring in x over Number Field in b with defining polynomial x^8 - x^4 +
```

In the previous example, the number field `L` becomes the pushout of `M1` and `M2` since both are provided with an embedding into `L`, *and* since `L` is a number field. If two number fields are embedded into a field that is not a numberfield, no merging occurs:

```
sage: K.<a> = NumberField(x^3-2, embedding=CDF(1/2*I*2^(1/3)*sqrt(3) - 1/2*2^(1/3)))
sage: L.<b> = NumberField(x^6-2, embedding=1.1)
sage: L.coerce_map_from(K)
sage: K.coerce_map_from(L)
sage: pushout(K,L)
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', Number Field in a with defining polynomial x
```

class sage.categories.pushout.**BlackBoxConstructionFunctor**(*box*)

   Bases: `sage.categories.pushout.ConstructionFunctor`

   Construction functor obtained from any callable object.

   EXAMPLES:

```
sage: from sage.categories.pushout import BlackBoxConstructionFunctor
sage: FG = BlackBoxConstructionFunctor(gap)
sage: FS = BlackBoxConstructionFunctor(singular)
sage: FG
```

```
      BlackBoxConstructionFunctor
sage: FG(ZZ)
      Integers
sage: FG(ZZ).parent()
      Gap
sage: FS(QQ['t'])
//   characteristic : 0
//   number of vars : 1
//        block   1 : ordering lp
//                  : names     t
//        block   2 : ordering C
sage: FG == FS
      False
sage: FG == loads(dumps(FG))
      True
```

class sage.categories.pushout.**CompletionFunctor**(*p*, *prec*, *extras=None*)

    Bases: sage.categories.pushout.ConstructionFunctor

    Completion of a ring with respect to a given prime (including infinity).

    EXAMPLES:

```
sage: R = Zp(5)
sage: R
      5-adic Ring with capped relative precision 20
sage: F1 = R.construction()[0]
sage: F1
      Completion[5]
sage: F1(ZZ) is R
      True
sage: F1(QQ)
      5-adic Field with capped relative precision 20
sage: F2 = RR.construction()[0]
sage: F2
      Completion[+Infinity]
sage: F2(QQ) is RR
      True
sage: P.<x> = ZZ[]
sage: Px = P.completion(x) # currently the only implemented completion of P
sage: Px
      Power Series Ring in x over Integer Ring
sage: F3 = Px.construction()[0]
sage: F3(GF(3)['x'])
      Power Series Ring in x over Finite Field of size 3
```

    TEST:

```
sage: R1.<a> = Zp(5,prec=20)[]
sage: R2 = Qp(5,prec=40)
sage: R2(1) + a
      (1 + O(5^20))*a + (1 + O(5^40))
sage: 1/2 + a
      (1 + O(5^20))*a + (3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9 + 2*5
```

    commutes(*other*)

        Completion commutes with fraction fields.

        EXAMPLE:

```
sage: F1 = Qp(5).construction()[0]
sage: F2 = QQ.construction()[0]
sage: F1.commutes(F2)
True
```

TEST:

The fraction field `R` in the example below has no completion method. But completion commutes with the fraction field functor, and so it is tried internally whether applying the construction functors in opposite order works. It does:

```
sage: P.<x> = ZZ[]
sage: C = P.completion(x).construction()[0]
sage: R = FractionField(P)
sage: hasattr(R,'completion')
False
sage: C(R) is Frac(C(P))
True
sage: F = R.construction()[0]
sage: (C*F)(ZZ['x']) is (F*C)(ZZ['x'])
True
```

The following was fixed in trac ticket #15329 (it used to result in an infinite recursion):

```
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7),RLF)
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', 7-adic Field with capped relative precision
```

merge(*other*)

> Two Completion functors are merged, if they are equal. If the precisions of both functors coincide, then a Completion functor is returned that results from updating the `extras` dictionary of `self` by `other.extras`. Otherwise, if the completion is at infinity then merging does not increase the set precision, and if the completion is at a finite prime, merging does not decrease the capped precision.

> EXAMPLE:

```
sage: R1.<a> = Zp(5,prec=20)[]
sage: R2 = Qp(5,prec=40)
sage: R2(1)+a          # indirect doctest
(1 + O(5^20))*a + (1 + O(5^40))
sage: R3 = RealField(30)
sage: R4 = RealField(50)
sage: R3(1) + R4(1)   # indirect doctest
2.0000000
sage: (R3(1) + R4(1)).parent()
Real Field with 30 bits of precision
```

> TESTS:

> We check that #12353 has been resolved:

```
sage: RealIntervalField(53)(-1) > RR(1)
False
sage: RealIntervalField(54)(-1) > RR(1)
False
sage: RealIntervalField(54)(1) > RR(-1)
True
sage: RealIntervalField(53)(1) > RR(-1)
```

```
True
```

We check that various pushouts work:

```
sage: R0 = RealIntervalField(30)
sage: R1 = RealIntervalField(30, sci_not=True)
sage: R2 = RealIntervalField(53)
sage: R3 = RealIntervalField(53, sci_not = True)
sage: R4 = RealIntervalField(90)
sage: R5 = RealIntervalField(90, sci_not = True)
sage: R6 = RealField(30)
sage: R7 = RealField(30, sci_not=True)
sage: R8 = RealField(53, rnd = 'RNDD')
sage: R9 = RealField(53, sci_not = True, rnd = 'RNDZ')
sage: R10 = RealField(53, sci_not = True)
sage: R11 = RealField(90, sci_not = True, rnd = 'RNDZ')
sage: Rlist = [R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11]
sage: from sage.categories.pushout import pushout
sage: pushouts = [R0,R0,R0,R1,R0,R1,R0,R1,R0,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R0
sage: all([R is S for R, S in zip(pushouts, [pushout(a, b) for a in Rlist for b in Rlist])])
True
```

```
sage: P0 = ZpFM(5, 10)
sage: P1 = ZpFM(5, 20)
sage: P2 = ZpCR(5, 10)
sage: P3 = ZpCR(5, 20)
sage: P4 = ZpCA(5, 10)
sage: P5 = ZpCA(5, 20)
sage: P6 = Qp(5, 10)
sage: P7 = Qp(5, 20)
sage: Plist = [P2,P3,P4,P5,P6,P7]
sage: from sage.categories.pushout import pushout
sage: pushouts = [P2,P3,P4,P5,P6,P7,P3,P3,P5,P5,P7,P7,P4,P5,P4,P5,P6,P7,P5,P5,P5,P5,P7,P7,P6
sage: all([P is Q for P, Q in zip(pushouts, [pushout(a, b) for a in Plist for b in Plist])])
True
```

class sage.categories.pushout.**CompositeConstructionFunctor**(*args*)

Bases: `sage.categories.pushout.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.constru
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'])
Univariate Polynomial Ring in y over Fraction Field of Univariate Polynomial Ring in x over Frac
```

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

class sage.categories.pushout.**ConstructionFunctor**
   Bases: `sage.categories.functor.Functor`

Base class for construction functors.

A construction functor is a functorial algebraic construction, such as the construction of a matrix ring over a given ring or the fraction field of a given ring.

In addition to the class `Functor`, construction functors provide rules for combining and merging constructions. This is an important part of Sage's coercion model, namely the pushout of two constructions: When a polynomial `p` in a variable `x` with integer coefficients is added to a rational number `q`, then Sage finds that the parents `ZZ['x']` and `QQ` are obtained from `ZZ` by applying a polynomial ring construction respectively the fraction field construction. Each construction functor has an attribute `rank`, and the rank of the polynomial ring construction is higher than the rank of the fraction field construction. This means that the pushout of `QQ` and `ZZ['x']`, and thus a common parent in which `p` and `q` can be added, is `QQ['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
sage: from sage.categories.pushout import pushout
sage: P.<x> = ZZ[]
sage: pushout(QQ,P)
Univariate Polynomial Ring in x over Rational Field
sage: ((x+1) + 1/2).parent()
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)[]
sage: f = A.hom([a+b,a-b],A)
sage: F(A)
Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Finite Field
sage: F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a
  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
```

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)
> False
> sage: F.commutes(F)
> False
> ```

expand()

> Decompose `self` into a list of construction functors.
>
> NOTE:
>
> The default is to return the list only containing `self`.
>
> EXAMPLE:
>
> ```
> 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:

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

TESTS:

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

class sage.categories.pushout.**FractionField**

Bases: `sage.categories.pushout.ConstructionFunctor`

Construction functor for fraction fields.

EXAMPLE:

```
sage: F = QQ.construction()[0]
sage: F
FractionField
sage: F.domain()
Category of integral domains
sage: F.codomain()
Category of fields
sage: F(GF(5)) is GF(5)
True
sage: F(ZZ['t'])
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
sage: P.<x,y> = QQ[]
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
```

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

> Bases: `sage.categories.pushout.ConstructionFunctor`

> A construction functor that is the identity functor.

> TESTS:
```
sage: from sage.categories.pushout import IdentityConstructionFunctor
sage: I = IdentityConstructionFunctor()
sage: I(RR) is RR
True
sage: I == loads(dumps(I))
True
```

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

> Bases: `sage.categories.pushout.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:
```
sage: A.<a,b> = InfinitePolynomialRing(ZZ['t'])
sage: A.construction()
[InfPoly{[a,b], "lex", "dense"},
 Univariate Polynomial Ring in t over Integer Ring]
sage: type(_[0])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
sage: B.<x,y,a_3,a_1> = PolynomialRing(QQ, order='lex')
sage: B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
sage: A.construction()[0]*B.construction()[0]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

> Apparently the variables a_1, a_3 of the polynomial ring are merged with the variables $a_0, a_1, a_2, ...$ 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: A.construction()[0]*PolynomialRing(QQ,names=['x','y','a_3','a_1']).construction()[0]
Traceback (most recent call last):
```

```
...
CoercionException: Incompatible term orders lex, degrevlex
```

In an infinite polynomial ring with generator a_*, the variable a_3 will always be greater than the variable a_1. Hence, the orders are incompatible in the next example as well:

```
sage: A.construction()[0]*PolynomialRing(QQ,names=['x','y','a_1','a_3'], order='lex').constructi
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: A.construction()[0]*PolynomialRing(QQ,names=['a_3','a_1','x','y'], order='lex').constructi
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
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: F = InfinitePolynomialRing(QQ, ['x','y'],order='degrevlex').construction()[0]; F
> InfPoly{[x,y], "degrevlex", "dense"}
> sage: F.expand()
> [InfPoly{[y], "degrevlex", "dense"}, InfPoly{[x], "degrevlex", "dense"}]
> sage: F = InfinitePolynomialRing(QQ, ['x','y','z'],order='degrevlex').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]
```

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

> Bases: `sage.categories.pushout.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 Intege
sage: P.<x,y> = ZZ[]
sage: f = P.hom([x+2*y,3*x-y],P)
sage: F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t over Multivariate Polynomial Ring i
  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.

> EXAMPLE:

```
sage: from sage.categories.pushout import LaurentPolynomialFunctor
sage: F1 = LaurentPolynomialFunctor('t')
sage: F2 = LaurentPolynomialFunctor('t', multi_variate=True)
sage: F1.merge(F2)
LaurentPolynomialFunctor
sage: F1.merge(F2)(LaurentPolynomialRing(GF(2),'a'))
```

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

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

    Bases: `sage.categories.pushout.ConstructionFunctor`

    A construction functor for matrices over rings.

    EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ,2)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: P.<x,y> = QQ[]
sage: R = F(P); R
Full MatrixSpace of 2 by 2 dense matrices over Multivariate Polynomial Ring in x, y over Rationa
sage: f = P.hom([x+y,x-y],P); F(f)
Ring endomorphism of Full MatrixSpace of 2 by 2 dense matrices over Multivariate Polynomial Ring
  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])
sage: F(f)(M)
[    x + y      x - y]
[x^2 - y^2       2*x]
```

    merge(*other*)

        Merging is only happening if both functors are matrix functors of the same dimension. The result is sparse
        if and only if both given functors are sparse.

        EXAMPLE:

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

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

    Bases: `sage.categories.pushout.ConstructionFunctor`

    A constructor for multivariate polynomial rings.

    EXAMPLES:

```
sage: P.<x,y> = ZZ[]
sage: F = P.construction()[0]; F
MPoly[x,y]
sage: A.<a,b> = GF(5)[]
sage: F(A)
Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Finite Field
sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Ring endomorphism of Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a
  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
```

expand()

> Decompose `self` into a list of construction functors.

> EXAMPLES:
> ```
> 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:
> ```
> sage: R.<x,y,z> = ZZ[]
> sage: S.<z,t> = QQ[]
> sage: x+t
> x + t
> sage: parent(x+t)
> Multivariate Polynomial Ring in x, y, z, t over Rational Field
> sage: T.<y,s> = QQ[]
> sage: x + s
> Traceback (most recent call last):
> ...
> TypeError: unsupported operand parent(s) for '+': 'Multivariate Polynomial Ring in x, y, z o
> sage: R = PolynomialRing(ZZ, 'x', 500)
> sage: S = PolynomialRing(GF(5), 'x', 200)
> sage: R.gen(0) + S.gen(0)
> 2*x0
> ```

merge(*other*)

> Merge `self` with another construction functor, or return None.

> EXAMPLES:
> ```
> sage: F = sage.categories.pushout.MultiPolynomialFunctor(['x','y'], None)
> sage: G = sage.categories.pushout.MultiPolynomialFunctor(['t'], None)
> sage: F.merge(G) is None
> True
> sage: F.merge(F)
> MPoly[x,y]
> ```

class sage.categories.pushout.**PermutationGroupFunctor**(*gens*, *domain*)

> Bases: `sage.categories.pushout.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)]])
sage: PF, P = P1.construction()
sage: PF.gens()
[(1,2)]
```

merge(*other*)

    Merge `self` with another construction functor, or return None.

    EXAMPLES:

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

class sage.categories.pushout.**PolynomialFunctor**(*var*, *multi_variate=False*, *sparse=False*)

    Bases: [sage.categories.pushout.ConstructionFunctor](#)

    Construction functor for univariate polynomial rings.

    EXAMPLE:

```
sage: P = ZZ['t'].construction()[0]
sage: P(GF(3))
Univariate Polynomial Ring in t over Finite Field of size 3
sage: P == loads(dumps(P))
True
sage: R.<x,y> = GF(5)[]
sage: f = R.hom([x+2*y,3*x-y],R)
sage: P(f)((x+y)*P(R).0)
(-x + y)*t
```

    By trac ticket #9944, the construction functor distinguishes sparse and dense polynomial rings. Before, the following example failed:

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

merge(*other*)

> Merge `self` with another construction functor, or return None.

> NOTE:

> Internally, the merging is delegated to the merging of multipolynomial construction functors. But in effect, this does the same as the default implementation, that returns `None` unless the to-be-merged functors coincide.

> EXAMPLE:
> ```
> sage: P = ZZ['x'].construction()[0]
> sage: Q = ZZ['y','x'].construction()[0]
> sage: P.merge(Q)
> sage: P.merge(P) is P
> True
> ```

class sage.categories.pushout.**QuotientFunctor**(*I*, *names=None*, *as_field=False*)

> Bases: sage.categories.pushout.ConstructionFunctor

> Construction functor for quotient rings.

> NOTE:

> The functor keeps track of variable names.

> EXAMPLE:
> ```
> sage: P.<x,y> = ZZ[]
> sage: Q = P.quo([x^2+y^2]*P)
> sage: F = Q.construction()[0]
> sage: F(QQ['x','y'])
> Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
> sage: F(QQ['x','y']) == QQ['x','y'].quo([x^2+y^2]*QQ['x','y'])
> True
> sage: F(QQ['x','y','z'])
> Traceback (most recent call last):
> ...
> CoercionException: Can not apply this quotient functor to Multivariate Polynomial Ring in x, y,
> sage: F(QQ['y','z'])
> 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.

> > EXAMPLE:
> > ```
> > sage: P.<x> = QQ[]
> > sage: Q1 = P.quo([(x^2+1)^2*(x^2-3)])
> > sage: Q2 = P.quo([(x^2+1)^2*(x^5+3)])
> > 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 trac ticket #8800:
> > ```
> > sage: pushout(GF(5), Integers(5))
> > Finite Field of size 5
> > ```

class sage.categories.pushout.**SubspaceFunctor**(*basis*)

> Bases: sage.categories.pushout.ConstructionFunctor

Constructing a subspace of an ambient free module, given by a basis.

NOTE:

This construction functor keeps track of the basis. It can only be applied to free modules into which this basis coerces.

EXAMPLES:

```
sage: M = ZZ^3
sage: S = M.submodule([(1,2,3),(4,5,6)]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: F = S.construction()[0]
sage: F(GF(2)^3)
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

merge(*other*)

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

EXAMPLE:

```
sage: M = GF(5)^3
sage: S1 = M.submodule([(1,2,3),(4,5,6)])
sage: S2 = M.submodule([(2,2,3)])
sage: F1 = S1.construction()[0]
sage: F2 = S2.construction()[0]
sage: F1.merge(F2)
SubspaceFunctor
sage: F1.merge(F2)(GF(5)^3) == S1+S2
True
sage: F1.merge(F2)(GF(5)['t']^3)
Free module of degree 3 and rank 3 over Univariate Polynomial Ring in t over Finite Field of
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

TEST:

```
sage: P.<t> = ZZ[]
sage: S1 = (ZZ^3).submodule([(1,2,3),(4,5,6)])
sage: S2 = (Frac(P)^3).submodule([(t,t^2,t^3+1),(4*t,0,1)])
sage: v = S1([0,3,6]) + S2([2,0,1/(2*t)]); v   # indirect doctest
(2, 3, (-12*t - 1)/(-2*t))
sage: v.parent()
Vector space of degree 3 and dimension 3 over Fraction Field of Univariate Polynomial Ring i
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

class sage.categories.pushout.**VectorFunctor**(*n*, *is_sparse=False*, *inner_product_matrix=None*)

Bases: `sage.categories.pushout.ConstructionFunctor`

A construction functor for free modules over commutative rings.

EXAMPLE:

```
sage: F = (ZZ^3).construction()[0]
sage: F
VectorFunctor
sage: F(GF(2)['t'])
Ambient free module of rank 3 over the principal ideal domain Univariate Polynomial Ring in t ov
```

merge(*other*)

Two constructors of free modules merge, if the module ranks coincide. If both have explicitly given inner product matrices, they must coincide as well.

EXAMPLE:

Two modules without explicitly given inner product allow coercion:

```
sage: M1 = QQ^3
sage: P.<t> = ZZ[]
sage: M2 = FreeModule(P,3)
sage: M1([1,1/2,1/3]) + M2([t,t^2+t,3])      # indirect doctest
(t + 1, t^2 + t + 1/2, 10/3)
```

If only one summand has an explicit inner product, the result will be provided with it:

```
sage: M3 = FreeModule(P,3, inner_product_matrix = Matrix(3,3,range(9)))
sage: M1([1,1/2,1/3]) + M3([t,t^2+t,3])
(t + 1, t^2 + t + 1/2, 10/3)
sage: (M1([1,1/2,1/3]) + M3([t,t^2+t,3])).parent().inner_product_matrix()
[0 1 2]
[3 4 5]
[6 7 8]
```

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 M1 and M4 in the following example is the fact that the default inner product was *explicitly* requested for M4. It is therefore not possible to coerce with a different inner product:

```
sage: M4 = FreeModule(QQ,3, inner_product_matrix = Matrix(3,3,1))
sage: M4 == M1
True
sage: M4.inner_product_matrix() == M1.inner_product_matrix()
True
sage: M4([1,1/2,1/3]) + M3([t,t^2+t,3])      # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Ambient quadratic space of dimension 3 ov
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 Po
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

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

An auxiliary function that is used in `pushout()` and `pushout_lattice()`.

INPUT:

An object

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 `None` and the given object.

EXAMPLE:

```
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 R
```

`sage.categories.pushout.`**`expand_tower`**(*tower*)

An auxiliary function that is used in `pushout()`.

INPUT:

A construction tower as returned by `construction_tower()`.

OUTPUT:

A new construction tower with all the construction functors expanded.

EXAMPLE:

```
sage: from sage.categories.pushout import construction_tower, expand_tower
sage: construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[x,y,z], Rational Field),
 (FractionField, Integer Ring)]
sage: expand_tower(construction_tower(QQ['x,y,z']))
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[z], Univariate Polynomial Ring in y over Univariate Polynomial Ring in x over Rational F
 (MPoly[y], Univariate Polynomial Ring in x over Rational Field),
 (MPoly[x], Rational Field),
 (FractionField, Integer Ring)]
```

`sage.categories.pushout.`**`pushout`**(*R*, *S*)

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

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 $R = Complete_7(Frac(\mathbf{Z}))$ and $Frac(Poly_x(\mathbf{Z}))$, which give us $Frac(Poly_x(Complete_7(Frac(\mathbf{Z}))))$:

```
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7), Frac(ZZ['x']))
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped relative precisi
```

Note we get the same thing with

```
sage: pushout(Zp(7), Frac(QQ['x']))
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped relative precisi
sage: pushout(Zp(7)['x'], Frac(QQ['x']))
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped relative precisi
```

Note that polynomial variable ordering must be unambiguously determined.

```
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 Int
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: pushout(Zp(7)['y'], Frac(QQ['t'])['x,y,z'])
Multivariate Polynomial Ring in x, y, z over Fraction Field of Univariate Polynomial Ring in t o
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
sage: pushout(MatrixSpace(RDF, 2, 2), Frac(ZZ['x']))
Full MatrixSpace of 2 by 2 dense matrices over Fraction Field of Univariate Polynomial Ring in x
sage: pushout(ZZ, MatrixSpace(ZZ[['x']], 3, 3))
Full MatrixSpace of 3 by 3 dense matrices over Power Series Ring in x over Integer Ring
sage: pushout(QQ['x,y'], ZZ[['x']])
Univariate Polynomial Ring in y over Power Series Ring in x over Rational Field
sage: pushout(Frac(ZZ['x']), QQ[['x']])
Laurent Series Ring in x over Rational Field
```

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

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

sage: V = (QQ^3).span([[1, 2, 3/4]])
sage: A = ZZ^3
sage: pushout(A, V)
Vector space of dimension 3 over Rational Field
sage: B = A.span([[0, 0, 2/3]])
sage: pushout(B, V)
```

```
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

Some more tests with `coercion_reversed = True`:

```
sage: from sage.categories.pushout import ConstructionFunctor
sage: class EvenPolynomialRing(type(QQ['x'])):
....:     def __init__(self, base, var):
....:         super(EvenPolynomialRing, self).__init__(base, var)
....:         self.register_embedding(base[var])
....:     def __repr__(self):
....:         return "Even Power " + super(EvenPolynomialRing, self).__repr__()
....:     def construction(self):
....:         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
....:     def _coerce_map_from_(self, R):
....:         return self.base().has_coerce_map_from(R)
....:
sage: class EvenPolynomialFunctor(ConstructionFunctor):
....:     rank = 10
....:     coercion_reversed = True
....:     def __init__(self):
....:         ConstructionFunctor.__init__(self, Rings(), Rings())
....:     def __call__(self, R):
....:         return EvenPolynomialRing(R.base(), R.variable_name())
....:
sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR)
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of precision

sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR['x'])
Univariate Polynomial Ring in x over Real Field with 53 bits of precision

sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(QQ, 'x'))
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), EvenPolynomialRing(RR, 'x'))
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of precision

sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR^2)
Ambient free module of rank 2 over the principal ideal domain Even Power Univariate Polynomial R
sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR['x']^2)
Ambient free module of rank 2 over the principal ideal domain Univariate Polynomial Ring in x ov
```

AUTHORS:

– Robert Bradshaw

`sage.categories.pushout.`**`pushout_lattice`**`(R, S)`

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

ALGORITHM:

This is based on the model that arose from much discussion at Sage Days 4. Going up the tower of constructions of R and S (e.g. the reals come from the rationals come from the integers), try to find a common parent, and then try to fill in a lattice with these two towers as sides with the top as the common ancestor and the bottom will be the desired ring.

See the code for a specific worked-out example.

EXAMPLES:

```
sage: from sage.categories.pushout import pushout_lattice
sage: A, B = pushout_lattice(Qp(7), Frac(ZZ['x']))
sage: A.codomain()
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped relative precisi
sage: A.codomain() is B.codomain()
True
sage: A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], 3, 3))
sage: B
Identity endomorphism of Full MatrixSpace of 3 by 3 dense matrices over Power Series Ring in x c
```

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

TEST:

```
sage: from sage.categories.pushout import type_to_parent
sage: type_to_parent(int)
Integer Ring
sage: type_to_parent(float)
Real Double Field
sage: type_to_parent(complex)
Complex Double Field
sage: type_to_parent(list)
Traceback (most recent call last):
...
TypeError: Not a scalar type.
```

FUNCTORIAL CONSTRUCTIONS

12.1 Covariant Functorial Constructions

A *functorial construction* is a collection of functors $(F_{Cat})_{Cat}$ (indexed by a collection of categories) which associate to a sequence of parents $(A, B, ...)$ in a category Cat a parent $F_{Cat}(A, B, ...)$. Typical examples of functorial constructions are `cartesian_product` and `tensor_product`.

The category of $F_{Cat}(A, B, ...)$, which only depends on Cat, is called the (functorial) construction category.

A functorial construction is *(category)-covariant* if for every categories Cat and $SuperCat$, the category of $F_{Cat}(A, B, ...)$ is a subcategory of the category of $F_{SuperCat}(A, B, ...)$ whenever Cat is a subcategory of $SuperCat$. A functorial construction is *(category)-regressive* if the category of $F_{Cat}(A, B, ...)$ is a subcategory of Cat.

The goal of this module is to provide generic support for covariant functorial constructions. In particular, given some parents A, B, ..., in respective categories Cat_A, 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 `CovariantFunctorialConstruction`, `CovariantConstructionCategory` and `RegressiveCovariantConstructionCategory` for more details.

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

class sage.categories.covariant_functorial_construction.**CovariantConstructionCategory**(*category*, **args*)

    Bases: `sage.categories.covariant_functorial_construction.FunctorialConstructionCategory`

    Abstract class for categories F_{Cat} obtained through a covariant functorial construction

    additional_structure()

        Return the additional structure defined by `self`.

        By default, a functorial construction category `A.F()` defines additional structure if and only if A is the category defining F. The rationale is that, for a subcategory B of A, the fact that $B.F()$ morphisms shall preserve the F-specific structure is already imposed by $A.F()$.

        See also:

            •`Category.additional_structure()`.

            •`is_construction_defined_by_base()`.

        EXAMPLES:

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

TESTS:
```
sage: Modules(ZZ).Graded().additional_structure.__module__
'sage.categories.covariant_functorial_construction'
```

classmethod **default_super_categories**(*category*, **args*)
> Return the default super categories of $F_{Cat}(A, B, ...)$ for $A, B, ...$ 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 $F(A, B, ...)$ for $A, B, ...$ in turn in each of the super categories of `category`.

> This is implemented as a class method, in order to be able to reconstruct the functorial category associated to each of the super categories of `category`.

> EXAMPLES:

> Bialgebras are both algebras and coalgebras:
> ```
> sage: Bialgebras(QQ).super_categories()
> [Category of algebras over Rational Field, Category of coalgebras over Rational Field]
> ```

> Hence tensor products of bialgebras are tensor products of algebras and tensor products of coalgebras:
> ```
> sage: Bialgebras(QQ).TensorProducts().super_categories()
> [Category of tensor products of algebras over Rational Field, Category of tensor products of
> ```

> Here is how `default_super_categories()` was called internally:
> ```
> sage: sage.categories.tensor.TensorProductsCategory.default_super_categories(Bialgebras(QQ))
> Join of Category of tensor products of algebras over Rational Field and Category of tensor p
> ```

> We now show a similar example, with the `Algebra` functor which takes a parameter **Q**:
> ```
> sage: FiniteMonoids().super_categories()
> [Category of monoids, Category of finite semigroups]
> sage: sorted(FiniteMonoids().Algebras(QQ).super_categories(), key=str)
> [Category of finite dimensional algebras with basis over Rational Field,
>  Category of finite set algebras over Rational Field,
>  Category of monoid algebras over Rational Field]
> ```

> Note that neither the category of *finite* semigroup algebras nor that of monoid algebras appear in the result; this is because there is currently nothing specific implemented about them.

> Here is how `default_super_categories()` was called internally:
> ```
> sage: sage.categories.algebra_functor.AlgebrasCategory.default_super_categories(FiniteMonoid
> Join of Category of finite dimensional algebras with basis over Rational Field
>     and Category of monoid algebras over Rational Field
>     and Category of finite set algebras over Rational Field
> ```

is_construction_defined_by_base()

> Return whether the construction is defined by the base of `self`.

> EXAMPLES:

> The graded functorial construction is defined by the modules category. Hence this method returns `True` for graded modules and `False` for other graded xxx categories:

```
sage: Modules(ZZ).Graded().is_construction_defined_by_base()
True
sage: Algebras(QQ).Graded().is_construction_defined_by_base()
False
sage: Modules(ZZ).WithBasis().Graded().is_construction_defined_by_base()
False
```

> This is implemented as follows: given the base category A and the construction F of `self`, that is `self=A.F()`, check whether no super category of A has F defined.

> **Note:** Recall that, when A does not implement the construction `F`, a join category is returned. Therefore, in such cases, this method is not available:

```
sage: Coalgebras(QQ).Graded().is_construction_defined_by_base()
Traceback (most recent call last):
...
AttributeError: 'JoinCategory_with_category' object has no attribute 'is_construction_define
```

---

class sage.categories.covariant_functorial_construction.**CovariantFunctorialConstruction**

> Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.sage_object.SageObject

> 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:

```
sage: tensor()  # todo: not implemented (add an example)
```

> or even new elements, which indirectly constructs the corresponding parent:

```
sage: tensor(...) # todo: not implemented
```

> •In the second use case, one implements a parent, and then put it in the category F_{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_{Cat}? For example, for the tensor product construction, which one of the followings do we want (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.

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:
    ```
    sage: Cat1 = Rings()
    sage: Cat2 = Groups()
    sage: cartesian_product.category_from_categories((Cat1, Cat1, Cat1))
    Join of Category of rings and ...
        and Category of Cartesian products of monoids
        and Category of Cartesian products of commutative additive groups

    sage: cartesian_product.category_from_categories((Cat1, Cat2))
    Category of Cartesian products of monoids
    ```

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"])
sage: tensor.category_from_parents((E, E, E))
Category of tensor products of vector spaces with basis over Rational Field
```

class sage.categories.covariant_functorial_construction.**FunctorialConstructionCategory**(*category*, **args*)

Bases: `sage.categories.category.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
```

classmethod category_of(*category*, **args*)

Return the image category of the functor F_{Cat}.

This is the main entry point for constructing the category F_{Cat} of parents $F(A, B, ...)$ constructed from parents $A, B, ...$ in Cat.

INPUT:

- `cls` – the category class for the functorial construction F

- `category` – a category Cat

- `*args` – further arguments for the functor

EXAMPLES:
```
sage: sage.categories.tensor.TensorProductsCategory.category_of(ModulesWithBasis(QQ))
Category of tensor products of vector spaces with basis over Rational Field

sage: sage.categories.algebra_functor.AlgebrasCategory.category_of(FiniteMonoids(), QQ)
Join of Category of finite dimensional algebras with basis over Rational Field
    and Category of monoid algebras over Rational Field
    and Category of finite set algebras over Rational Field
```

extra_super_categories()

Return the extra super categories of a construction category.

Default implementation which returns `[]`.

EXAMPLES:

```
sage: Sets().Subquotients().extra_super_categories()
[]
sage: Semigroups().Quotients().extra_super_categories()
[]
```

super_categories()
Return the super categories of a construction category.

EXAMPLES:
```
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.**RegressiveCovariantConstructionCategor**

Bases: `sage.categories.covariant_functorial_construction.CovariantConstructionCategory`

Abstract class for categories F_{Cat} obtained through a regressive covariant functorial construction

classmethod default_super_categories(*category*, **args*)
Return the default super categories of $F_{Cat}(A, B, ...)$ for $A, B, ...$ 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.

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:
```
sage: Monoids().Subquotients().super_categories()
[Category of monoids, Category of subquotients of semigroups]
```

TESTS:
```
sage: C = Monoids().Subquotients()
sage: C.__class__.default_super_categories(C.base_category(), *C._args)
Category of unital subquotients of semigroups
```

12.2 Cartesian Product Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2008-2010): initial revision and refactorization

class sage.categories.cartesian_product.**CartesianProductFunctor**
Bases: `sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction`

A singleton class for the Cartesian product functor.

EXAMPLES:
```
sage: cartesian_product
The cartesian_product functorial construction
```

`cartesian_product` takes a finite collection of sets, and constructs the Cartesian product of those sets:
```
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:
```
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

sage: Monoids().CartesianProducts()
Category of Cartesian products of monoids
```

The Cartesian product functor is covariant: if `A` is a subcategory of `B`, then `A.CartesianProducts()` is a subcategory of `B.CartesianProducts()` (see also `CovariantFunctorialConstruction`):
```
sage: C.categories()
[Category of Cartesian products of monoids,
 Category of monoids,
 Category of Cartesian products of semigroups,
 Category of semigroups,
 Category of Cartesian products of unital magmas,
 Category of Cartesian products of magmas,
 Category of unital magmas,
 Category of magmas,
 Category of Cartesian products of sets,
 Category of sets, ...]

[Category of Cartesian products of monoids,
 Category of monoids,
 Category of Cartesian products of semigroups,
 Category of semigroups,
 Category of Cartesian products of magmas,
 Category of unital magmas,
 Category of magmas,
 Category of Cartesian products of sets,
 Category of sets,
```

```
Category of sets with partial maps,
Category of objects]
```

Hence, the role of `Monoids().CartesianProducts()` is solely to provide mathematical information and algorithms which are relevant to Cartesian product of monoids. For example, it specifies that the result is again a monoid, and that its multiplicative unit is the cartesian product of the units of the underlying sets:

```
sage: C.one()
('', 1, 1)
```

Those are implemented in the nested class `Monoids.CartesianProducts` of `Monoids(QQ)`. This nested class is itself a subclass of `CartesianProductsCategory`.

class sage.categories.cartesian_product.**CartesianProductsCategory**(*category*, **args*)

Bases: `sage.categories.covariant_functorial_construction.CovariantConstructionCategory`

An abstract base class for all `CartesianProducts` categories.

TESTS:

```
sage: C = Sets().CartesianProducts()
sage: C
Category of Cartesian products of sets
sage: C.base_category()
Category of sets
sage: latex(C)
\mathbf{CartesianProducts}(\mathbf{Sets})
```

CartesianProducts()

Return the category of (finite) Cartesian products of objects of `self`.

By associativity of Cartesian products, this is `self` (a Cartesian product of Cartesian products of A's is a Cartesian product of A's).

EXAMPLES:

```
sage: ModulesWithBasis(QQ).CartesianProducts().CartesianProducts()
Category of Cartesian products of vector spaces with basis over Rational Field
```

base_ring()

The base ring of a cartesian product is the base ring of the underlying category.

EXAMPLES:

```
sage: Algebras(ZZ).CartesianProducts().base_ring()
Integer Ring
```

sage.categories.cartesian_product.**cartesian_product**

The cartesian product functorial construction.

See `CartesianProductFunctor` for more information.

EXAMPLES:

```
sage: cartesian_product
The cartesian_product functorial construction
```

12.3 Tensor Product Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2008-2010): initial revision and refactorization

class sage.categories.tensor.**TensorProductFunctor**

Bases: sage.categories.covariant_functorial_construction.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 Algebras(QQ), it is automatically endowed with its natural algebra structure, thanks to the category Algebras(QQ).TensorProducts() of tensor products of algebras.

The tensor functor is covariant: if A is a subcategory of B, then A.TensorProducts() is a subcategory of B.TensorProducts() (see also CovariantFunctorialConstruction). Hence, the role of Algebras(QQ).TensorProducts() is solely to provide mathematical information and algorithms which are relevant to tensor product of algebras.

Those are implemented in the nested class TensorProducts of Algebras(QQ). This nested class is itself a subclass of TensorProductsCategory.

TESTS:
```
sage: TestSuite(tensor).run()
```

class sage.categories.tensor.**TensorProductsCategory**(*category*, **args*)

Bases: sage.categories.covariant_functorial_construction.CovariantConstructionCategory

An abstract base class for all TensorProducts's categories

TESTS:
```
sage: C = ModulesWithBasis(QQ).TensorProducts()
sage: C
Category of tensor products of vector spaces with basis over Rational Field
sage: C.base_category()
Category of vector spaces with basis over Rational Field
sage: latex(C)
\mathbf{TensorProducts}(\mathbf{WithBasis}_{\Bold{Q}})
sage: TestSuite(C).run()
```

TensorProducts()

Returns the category of tensor products of objects of self

By associativity of tensor products, this is self (a tensor product of tensor products of Cat's is a tensor product of Cat's)

EXAMPLES:
```
sage: ModulesWithBasis(QQ).TensorProducts().TensorProducts()
Category of tensor products of vector spaces with basis over Rational Field
```

base()

The base of a tensor product is the base (usually a ring) of the underlying category.

EXAMPLES:
```
sage: ModulesWithBasis(ZZ).TensorProducts().base()
Integer Ring
```

```
sage.categories.tensor.tensor
      The tensor product functorial construction
```

See `TensorProductFunctor` for more information

EXAMPLES:
```
sage: tensor
The tensor functorial construction
```

12.4 Dual functorial construction

AUTHORS:

- Nicolas M. Thiery (2009-2010): initial revision

class sage.categories.dual.**DualFunctor**
      Bases: `sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction`

A singleton class for the dual functor

class sage.categories.dual.**DualObjectsCategory**(*category*, **args*)
      Bases: `sage.categories.covariant_functorial_construction.CovariantConstructionCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:      _functor_category = "FooBars"
....:      _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

12.5 Algebra Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

class sage.categories.algebra_functor.**AlgebraFunctor**(*base_ring*)
      Bases: `sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction`

A singleton class for the algebra functor.

base_ring()
      Return the base ring for this functor.

      EXAMPLES:

```
sage: from sage.categories.algebra_functor import AlgebraFunctor
sage: AlgebraFunctor(QQ).base_ring()
Rational Field
```

class sage.categories.algebra_functor.**AlgebrasCategory**(*category*, **args*)

Bases: sage.categories.covariant_functorial_construction.CovariantConstructionCategory, sage.categories.category_types.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:

```
sage: C = Monoids().Algebras(QQ); C
Category of monoid algebras over Rational Field
sage: C = Groups().Algebras(QQ); C
Category of group algebras over Rational Field

sage: C._short_name()
'Algebras'
sage: latex(C) # todo: improve that
\mathbf{Algebras}(\mathbf{Groups})
```

12.6 Subquotient Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

class sage.categories.subquotients.**SubquotientsCategory**(*category*, **args*)

Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionC

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

12.7 Quotients Functorial Construction

AUTHORS:

• Nicolas M. Thiery (2010): initial revision

class sage.categories.quotients.**QuotientsCategory**(*category*, **args*)

Bases: `sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionC`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

classmethod default_super_categories(*category*)

Returns the default super categories of `category.Quotients()`

Mathematical meaning: if A is a quotient of B in the category C, then A is also a subquotient of B in the category C.

INPUT:

  • `cls` – the class `QuotientsCategory`

  • `category` – a category Cat

OUTPUT: a (join) category

In practice, this returns `category.Subquotients()`, joined together with the result of the method `RegressiveCovariantConstructionCategory.default_super_categories()` (that is the join of `category` and `cat.Quotients()` for each `cat` in the super categories of `category`).

EXAMPLES:

Consider `category=Groups()`, which has `cat=Monoids()` as super category. Then, a subgroup of a group G is simultaneously a subquotient of G, a group by itself, and a quotient monoid of G:

```
sage: Groups().Quotients().super_categories()
[Category of groups, Category of subquotients of monoids, Category of quotients of semigroup
```

Mind the last item above: there is indeed currently nothing implemented about quotient monoids.

This resulted from the following call:

```
sage: sage.categories.quotients.QuotientsCategory.default_super_categories(Groups())
Join of Category of groups and Category of subquotients of monoids and Category of quotients
```

12.8 Subobjects Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

class sage.categories.subobjects.**SubobjectsCategory**(*category*, **args*)

    Bases: `sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionC`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

    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 subobject
```

12.9 Isomorphic Objects Functorial Construction

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

class sage.categories.isomorphic_objects.**IsomorphicObjectsCategory**(*category*, **args*)

  Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionC

  TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

  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(
Join of Category of groups and
Category of subquotients of monoids and
Category of quotients of semigroups and
Category of isomorphic objects of sets
```

12.10 Homset categories

class sage.categories.homsets.**Homsets**(*s=None*)

Bases: `sage.categories.category_singleton.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).

trac ticket #17364: every homset category shall be a subcategory of the category of all homsets:
```
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()`.

class Endset(*base_category*)

Bases: `sage.categories.category_with_axiom.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:
```
sage: from sage.categories.homsets import Homsets
sage: Homsets().Endset()
Category of endsets
```

extra_super_categories()

Implement the fact that endsets are monoids.

See also:

`CategoryWithAxiom.extra_super_categories()`

EXAMPLES:

```
sage: from sage.categories.homsets import Homsets
sage: Homsets().Endset().extra_super_categories()
[Category of monoids]
```

class Homsets.**SubcategoryMethods**

> **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
>> ```

Homsets.**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: sage.categories.covariant_functorial_construction.FunctorialConstructionCategory
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> Category of foo bars of modules with basis over Integer Ring
> sage: C.base_category()
> Category of modules with basis over Integer Ring
> sage: latex(C)
> \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
> sage: TestSuite(C).run()
> ```
>
> **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:

```
sage: from sage.categories.homsets import HomsetsCategory
sage: from sage.categories.additive_groups import AdditiveGroups

sage: C = AdditiveGroups()
sage: C.full_super_categories()
[Category of additive inverse additive unital additive magmas,
 Category of additive monoids]
sage: H = HomsetsCategory.default_super_categories(C); H
Category of homsets of additive monoids
sage: type(H)
<class 'sage.categories.additive_monoids.AdditiveMonoids.Homsets_with_category'>
```

and, given that nothing specific is currently implemented for homsets of additive groups, `H` is directly the category thereof:

```
sage: C.Homsets()
Category of homsets of additive monoids
```

Similarly for rings: a ring homset is just a homset of unital magmas and additive magmas:

```
sage: Rings().Homsets()
Category of homsets of unital magmas and additive unital additive magmas
```

Otherwise, if `category` implements a nested class `Homsets`, this method returns the category of all homsets:

```
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()`:

```
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()`:

```
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:

```
sage: hasattr(Posets, "Homsets")
False
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
```

TESTS:
```
sage: Objects().Homsets().super_categories()
[Category of homsets]
sage: Sets().Homsets().super_categories()
[Category of homsets]
sage: (Magmas() & Posets()).Homsets().super_categories()
[Category of homsets]
```

class sage.categories.homsets.**HomsetsOf**(*category*, **args*)

Bases: `sage.categories.homsets.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'>
```

TESTS:
```
sage: TestSuite(C).run()
sage: C = Rings().Homsets()
sage: TestSuite(C).run(skip=['_test_category_graph'])
sage: TestSuite(C).run()
```

super_categories()

Return the super categories of `self`.

A stub homset category admits a single super category, namely the category of all homsets.

EXAMPLES:
```
sage: C = (Magmas() & Posets()).Homsets(); C
Category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.HomsetsOf_with_category'>
sage: C.super_categories()
[Category of homsets]
```

12.11 Realizations Covariant Functorial Construction

See also:

- `Sets().WithRealizations` for an introduction to *realizations* and *with realizations*.
- `sage.categories.covariant_functorial_construction` for an introduction to covariant functorial constructions.
- `sage.categories.examples.with_realizations` for an example.

class sage.categories.realizations.**Category_realization_of_parent**(*parent_with_realization*)

Bases: `sage.categories.category_types.Category_over_base`, `sage.misc.bindable_class.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
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
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
The subset algebra of {1, 2, 3} over Rational Field
sage: A.Realizations()
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 realizat
sage: C.super_categories()
[Category of graded hopf algebras with basis over Rational Field, Category of realizations of ho

sage: TestSuite(C).run()
```

See also:

- `Sets().WithRealizations`

- `ClasscallMetaclass`

Todo

Add an optional argument to allow for:

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

---

class sage.categories.realizations.**RealizationsCategory**(*category*, **args*)

Bases: `sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionC`

An abstract base class for all categories of realizations category

Relization are implemented as `RegressiveCovariantConstructionCategory`. See there for the documentation of how the various bindings such as `Sets().Realizations()` and `P.Realizations()`, where `P` is a parent, work.

See also:

`Sets().WithRealizations`

TESTS:

```
sage: Sets().Realizations
<bound method Sets_with_category.Realizations of Category of sets>
sage: Sets().Realizations()
Category of realizations of sets
sage: Sets().Realizations().super_categories()
[Category of sets]
sage: Groups().Realizations().super_categories()
[Category of groups, Category of realizations of unital magmas]
```

12.12 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 constructions.

sage.categories.with_realizations.**WithRealizations**(*self*)

Returns 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.

- `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 `Sets.Facade`) 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
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()
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: A.Out()
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: A.In()
The subset algebra of {1, 2, 3} over Rational Field in the In basis

sage: A.an_element()
F[{}] + 2*F[{1}] + 3*F[{2}] + F[{1, 2}]
```

If B and B' are two bases, then the change of basis from B to B' is implemented by a canonical coercion between P_B and $P_{B'}$:
```
sage: F = A.F(); In = A.In(); Out = A.Out()
sage: i = In.an_element(); i
In[{}] + 2*In[{1}] + 3*In[{2}] + In[{1, 2}]
sage: F(i)
7*F[{}] + 3*F[{1}] + 4*F[{2}] + F[{1, 2}]
sage: F.coerce_map_from(Out)
Generic morphism:
  From: The subset algebra of {1, 2, 3} over Rational Field in the Out basis
  To:   The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
```

allowing for mixed arithmetic:
```
sage: (1 + Out.from_set(1)) * In.from_set(2,3)
Out[{}] + 2*Out[{1}] + 2*Out[{2}] + 2*Out[{3}] + 2*Out[{1, 2}] + 2*Out[{1, 3}] + 4*Out[{2, 3}] +
```

In our example, there are three realizations:
```
sage: A.realizations()
[The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis,
 The subset algebra of {1, 2, 3} over Rational Field in the In basis,
 The subset algebra of {1, 2, 3} over Rational Field in the Out basis]
```

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

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

The various parent realizing A belong to this category:

```
sage: A.F() in A.Realizations()
True
```

A itself is in the category of algebras with realizations:

```
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, `FiniteDimensionalVectorSpaces(QQ).WithRealizations()` will eventually be in charge, whenever a coercion $\phi : A \mapsto B$ is registered, to register ϕ^{-1} as coercion $B \mapsto A$ 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):

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

and in the category of algebras (regressiveness):

```
sage: A in Algebras(QQ)
True
```

Note: For C a category, `C.WithRealizations()` in fact calls `sage.categories.with_realizations.Realizations(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:

```
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.

TESTS:

```
sage: Semigroups().WithRealizations()
Join of Category of semigroups and Category of sets with realizations
sage: C = GradedHopfAlgebrasWithBasis(QQ).WithRealizations(); C
```

```
    Category of graded hopf algebras with basis over Rational Field with realizations
sage: C.super_categories()
[Join of Category of hopf algebras over Rational Field
     and Category of graded algebras over Rational Field]
sage: TestSuite(Semigroups().WithRealizations()).run()
```

class sage.categories.with_realizations.**WithRealizationsCategory**(*category*, **args*)

Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstruction

An abstract base class for all categories of parents with multiple realizations.

See also:

Sets().WithRealizations

The role of this base class is to implement some technical goodies, such as the name for that category.

CATEGORIES

13.1 Additive groups

class sage.categories.additive_groups.**AdditiveGroups**(*base_category*)

Bases: sage.categories.category_with_axiom.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
```

TESTS:
```
sage: C = AdditiveGroups()
sage: TestSuite(C).run()
```

AdditiveCommutative

alias of CommutativeAdditiveGroups

13.2 Additive Magmas

class sage.categories.additive_magmas.**AdditiveMagmas**(*s=None*)

Bases: [sage.categories.category_singleton.Category_singleton](#)

The category of additive magmas.

An additive magma is a set endowed with a binary operation $+$.

EXAMPLES:
```
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
```

The following axioms are defined by this category:
```
sage: AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
sage: AdditiveMagmas().AdditiveUnital()
Category of additive unital additive magmas
sage: AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative()
Category of commutative additive semigroups
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().AdditiveUnital()
Category of commutative additive monoids
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().AdditiveInve
Category of commutative additive groups
```

TESTS:
```
sage: C = AdditiveMagmas()
sage: TestSuite(C).run()
```

AdditiveAssociative
    alias of AdditiveSemigroups

class AdditiveCommutative(*base_category*)

Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_singleton](#)

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class Algebras(*category*, **args*)
    Bases: [sage.categories.algebra_functor.AlgebrasCategory](#)

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

extra_super_categories()

Implement the fact that the algebra of a commutative additive magmas is commutative.

EXAMPLES:

```
sage: AdditiveMagmas().AdditiveCommutative().Algebras(QQ).extra_super_categories()
[Category of commutative magmas]
```

```
sage: AdditiveMagmas().AdditiveCommutative().Algebras(QQ).super_categories()
[Category of additive magma algebras over Rational Field,
 Category of commutative magmas]
```

class AdditiveMagmas.AdditiveCommutative.**CartesianProducts**(*category*,
                                                                    *args)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

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 additive commutative additive magmas]
sage: C.axioms()
frozenset({'AdditiveCommutative'})
```

class AdditiveMagmas.**AdditiveUnital**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class **AdditiveInverse**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class **CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruc
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a p
sage: TestSuite(C).run()
```

class **ElementMethods**

AdditiveMagmas.AdditiveUnital.AdditiveInverse.CartesianProducts.**extra_super_c**
    Implement the fact that a cartesian product of additive magmas with inverses is an additive magma with inverse.

EXAMPLES:
```
sage: C = AdditiveMagmas().AdditiveUnital().AdditiveInverse().CartesianProducts()
sage: C.extra_super_categories();
[Category of additive inverse additive unital additive magmas]
sage: sorted(C.axioms())
['AdditiveInverse', 'AdditiveUnital']
```

class AdditiveMagmas.AdditiveUnital.**Algebras**(*category*, **args*)

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

class **ParentMethods**

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',
sage: A = S.algebra(ZZ)
sage: A.one_basis()
0
sage: A.one()
B[0]
sage: A(3)
3*B[0]
```

AdditiveMagmas.AdditiveUnital.Algebras.**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 mag
```

class AdditiveMagmas.AdditiveUnital.**CartesianProducts**(*category*, **args*)
Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
```

```
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

class **ParentMethods**

> **zero**()
> Returns the zero of this group
>
> EXAMPLE:
> ```
> sage: GF(8,'x').cartesian_product(GF(5)).zero()
> (0, 0)
> ```

AdditiveMagmas.AdditiveUnital.CartesianProducts.**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 AdditiveMagmas.AdditiveUnital.**ElementMethods**

class AdditiveMagmas.AdditiveUnital.**Homsets**(*category*, **args*)
> Bases: `sage.categories.homsets.HomsetsCategory`
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> Category of foo bars of modules with basis over Integer Ring
> sage: C.base_category()
> Category of modules with basis over Integer Ring
> sage: latex(C)
> \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
> sage: TestSuite(C).run()
> ```

class **ParentMethods**

> **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
```

TESTS:

sage: TestSuite(f).run()

`AdditiveMagmas.AdditiveUnital.Homsets.`**`extra_super_categories`**`()`

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

EXAMPLES:

```
sage: AdditiveMagmas().AdditiveUnital().Homsets().extra_super_categories()
[Category of additive unital additive magmas]
sage: AdditiveMagmas().AdditiveUnital().Homsets().super_categories()
[Category of additive unital additive magmas, Category of homsets]
```

class `AdditiveMagmas.AdditiveUnital.`**`ParentMethods`**

`is_empty``()`

Return whether this set is empty.

Since this set is an additive magma it has a zero element and hence is not empty. This method thus always returns `False`.

EXAMPLES:

```
sage: A = AdditiveAbelianGroup([3,3])
sage: A in AdditiveMagmas()
True
sage: A.is_empty()
False

sage: B = CommutativeAdditiveMonoids().example()
sage: B.is_empty()
False
```

TESTS:

We check that the method is_empty is inherited from this category in both examples above:

```
sage: A.is_empty.__module__
'sage.categories.additive_magmas'
sage: B.is_empty.__module__
'sage.categories.additive_magmas'
```

`zero``()`

Return the zero of this additive magma, that is the unique neutral element for $+$.

The default implementation is to coerce `0` into `self`.

It is recommended to override this method because the coercion from the integers:
  • is not always meaningful (except for 0), and
  • often uses `self.zero()` otherwise.

EXAMPLES:

```
sage: S = CommutativeAdditiveMonoids().example()
sage: S.zero()
0
```

`zero_element``()`

Backward compatibility alias for `self.zero()`.

TESTS:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 3)
sage: P.zero_element()
doctest:...: DeprecationWarning: .zero_element() is deprecated. Use .zero() instead
See http://trac.sagemath.org/17694 for details.
A 0-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex
```

class `AdditiveMagmas.AdditiveUnital.`**`SubcategoryMethods`**

> **`AdditiveInverse`**`()`
>> Return the full subcategory of the additive inverse objects of `self`.
>>
>> An inverse `additive magma` is a `unital additive magma` such that every element admits both an additive inverse on the left and on the right. Such an additive magma is also called an *additive loop*.
>>
>> **See also:**
>>
>> Wikipedia article Inverse_element, Wikipedia article Quasigroup
>>
>> EXAMPLES:
>> ```
>> sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
>> Category of additive inverse additive unital additive magmas
>> sage: from sage.categories.additive_monoids import AdditiveMonoids
>> sage: AdditiveMonoids().AdditiveInverse()
>> Category of additive groups
>> ```
>>
>> TESTS:
>> ```
>> sage: TestSuite(AdditiveMagmas().AdditiveUnital().AdditiveInverse()).run()
>> sage: CommutativeAdditiveMonoids().AdditiveInverse.__module__
>> 'sage.categories.additive_magmas'
>> ```

class `AdditiveMagmas.AdditiveUnital.`**`WithRealizations`**`(category, *args)`
> Bases: `sage.categories.with_realizations.WithRealizationsCategory`
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> Category of foo bars of modules with basis over Integer Ring
> sage: C.base_category()
> Category of modules with basis over Integer Ring
> sage: latex(C)
> \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
> sage: TestSuite(C).run()
> ```
>
> **class** **`ParentMethods`**
>
>> **`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:
```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: A.zero.__module__
'sage.categories.additive_magmas'
sage: A.zero()
0
```

TESTS:
```
sage: A.zero() is A.a_realization().zero()
True
sage: A._test_zero()
```

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

class AdditiveMagmas.**Algebras**(*category*, *\*args*)
Bases: `sage.categories.algebra_functor.AlgebrasCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:   _functor_category = "FooBars"
....:   _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

**algebra_generators**()
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 monoid: the free commutative monoid generated by ('a', 'b
sage: A = S.algebra(QQ)
sage: A.algebra_generators()
Finite family {0: B[a], 1: B[b], 2: B[c], 3: B[d]}
```

> **product_on_basis**(*g1*, *g2*)
>> Product, on basis elements, as per `MagmaticAlgebras.WithBasis.ParentMethods.product_on_ba`
>>
>> 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 monoid: the free commutative monoid generated by ('a', 'b
>> sage: A = S.algebra(QQ)
>> sage: a,b,c,d = A.algebra_generators()
>> sage: a * b + b * d * c
>> B[c + b + d] + B[a + b]
>> ```

> AdditiveMagmas.Algebras.**extra_super_categories**()
>> EXAMPLES:
>> ```
>> 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
>> ```

class AdditiveMagmas.**CartesianProducts**(*category*, **args*)
> Bases: `sage.categories.cartesian_product.CartesianProductsCategory`
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> Category of foo bars of modules with basis over Integer Ring
> sage: C.base_category()
> Category of modules with basis over Integer Ring
> sage: latex(C)
> \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
> sage: TestSuite(C).run()
> ```

> **class ElementMethods**

> AdditiveMagmas.CartesianProducts.**extra_super_categories**()
>> Implement the fact that a cartesian product of additive magmas is an additive magma.
>>
>> EXAMPLES:
>> ```
>> 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 AdditiveMagmas.**ElementMethods**

class AdditiveMagmas.**Homsets**(*category*, **args*)
> Bases: `sage.categories.homsets.HomsetsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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

    EXAMPLES:
    ```
    sage: AdditiveMagmas().Homsets().extra_super_categories()
    [Category of additive magmas]
    sage: AdditiveMagmas().Homsets().super_categories()
    [Category of additive magmas, Category of homsets]
    ```

class AdditiveMagmas.**ParentMethods**

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.

```
sage: R=IntegerModRing(5)
sage: R.addition_table()
+  a b c d e
 +----------
a| a b c d e
b| b c d e a
c| c d e a b
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.

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

```
sage: T=R.addition_table(names='digits')
sage: T
+  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)
+  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)
+  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()
sage: T.column_keys()
(0, 1, 2)
sage: sorted(T.translation().items())
[('a', 0), ('b', 1), ('c', 2)]
sage: T.change_names(['x', 'y', 'z'])
sage: sorted(T.translation().items())
[('x', 0), ('y', 1), ('z', 2)]
sage: T
+  x y z
 +------
x| x y z
y| y z x
z| z x y
```

summation (*x*, *y*)

> Return the sum of `x` and `y`.
>
> The binary addition operator of this additive magma.
>
> INPUT:
>
> > •x, y – elements of this additive magma
>
> EXAMPLES:
>
> ```
> sage: S = CommutativeAdditiveSemigroups().example()
> sage: (a,b,c,d) = S.additive_semigroup_generators()
> sage: S.summation(a, b)
> a + b
> ```
>
> 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: 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: 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: 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.

---

class `AdditiveMagmas.`**SubcategoryMethods**

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
```

TESTS:

```
sage: TestSuite(AdditiveMagmas().AdditiveAssociative()).run()
sage: Rings().AdditiveAssociative.__module__
'sage.categories.additive_magmas'
```

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
sage: AdditiveMagmas().AdditiveAssociative().AdditiveUnital().AdditiveCommutative()
Category of commutative additive monoids
sage: _ is CommutativeAdditiveMonoids()
True
```

TESTS:

```
sage: TestSuite(AdditiveMagmas().AdditiveCommutative()).run()
sage: Rings().AdditiveCommutative.__module__
'sage.categories.additive_magmas'
```

AdditiveUnital()

Return the subcategory of the unital objects of `self`.

An additive magma M is *unital* if it admits an element 0, called *neutral element*, such that for all $x \in M$,

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

This element is necessarily unique, and should be provided as `M.zero()`.

See also:

Wikipedia article Unital_magma#unital

EXAMPLES:

```
sage: 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
```

TESTS:

```
sage: TestSuite(AdditiveMagmas().AdditiveUnital()).run()
sage: CommutativeAdditiveSemigroups().AdditiveUnital.__module__
'sage.categories.additive_magmas'
```

AdditiveMagmas.**super_categories**()

EXAMPLES:

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

13.3 Additive monoids

class sage.categories.additive_monoids.**AdditiveMonoids**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of additive monoids.

An *additive monoid* is a unital class:$additivesemigroup < sage.categories.additive_semigroups.AdditiveSemigroups >$, that is a set endowed with a binary operation $+$ which is associative and admits a zero (see Wikipedia article Monoid).

EXAMPLES:
```
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
```

TESTS:
```
sage: C.Algebras(QQ).is_subcategory(AlgebrasWithBasis(QQ))
True
sage: TestSuite(C).run()
```

AdditiveCommutative

alias of `CommutativeAdditiveMonoids`

AdditiveInverse

alias of `AdditiveGroups`

class Homsets(*category*, **args*)

Bases: `sage.categories.homsets.HomsetsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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 `AdditiveMonoids.`**`ParentMethods`**

    `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 + c + 2*b
sage: S.sum(())
0
sage: S.sum(()).parent() == S
True
```

13.4 Additive semigroups

class `sage.categories.additive_semigroups.`**`AdditiveSemigroups`**(*base_category*)

    Bases: `sage.categories.category_with_axiom.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()
[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
```

TESTS:
```
sage: TestSuite(C).run()
```

AdditiveCommutative
    alias of `CommutativeAdditiveSemigroups`

AdditiveUnital
    alias of `AdditiveMonoids`

class **Algebras**(*category*, **args*)
    Bases: `sage.categories.algebra_functor.AlgebrasCategory`

    TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python n
sage: TestSuite(C).run()
```

    class **ParentMethods**

        algebra_generators()
            Return the generators of this algebra, as per `MagmaticAlgebras.ParentMethods.algebra_generator`

            They correspond to the generators of the additive semigroup.

            EXAMPLES:
```
sage: S = CommutativeAdditiveSemigroups().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b
sage: A = S.algebra(QQ)
sage: A.algebra_generators()
Finite family {0: B[a], 1: B[b], 2: B[c], 3: B[d]}
```

        product_on_basis(*g1*, *g2*)
            Product, on basis elements, as per `MagmaticAlgebras.WithBasis.ParentMethods.product_on_ba`

            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 monoid: the free commutative monoid generated by ('a', 'b
sage: A = S.algebra(QQ)
sage: a,b,c,d = A.algebra_generators()
sage: a * b + b * d * c
B[c + b + d] + B[a + b]
```

    AdditiveSemigroups.Algebras.**extra_super_categories**()
        EXAMPLES:

```
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 additive magma algebras over Rational Field]
```

class AdditiveSemigroups.**CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:      _functor_category = "FooBars"
....:      _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

extra_super_categories()

Implement the fact that a cartesian product of additive semigroups is an additive semigroup.

EXAMPLES:

```
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 AdditiveSemigroups.**Homsets**(*category*, **args*)

Bases: `sage.categories.homsets.HomsetsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:      _functor_category = "FooBars"
....:      _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

extra_super_categories()

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

EXAMPLES:

```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: AdditiveSemigroups().Homsets().extra_super_categories()
[Category of additive semigroups]
sage: AdditiveSemigroups().Homsets().super_categories()
[Category of homsets of additive magmas, Category of additive semigroups]
```

class AdditiveSemigroups.**ParentMethods**

13.5 Affine Weyl Groups

class sage.categories.affine_weyl_groups.**AffineWeylGroups**(*s=None*)

Bases: sage.categories.category_singleton.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: W = WeylGroup(["A",4,1]); W
Weyl Group of type ['A', 4, 1] (as a matrix group acting on the root space)
sage: W.category()
Category of affine weyl groups
```

TESTS:

```
sage: TestSuite(C).run()
```

class **ElementMethods**

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: W = WeylGroup(['A',2,1])
sage: w = W.from_reduced_word([0,2,1,0])
sage: la = w.affine_grassmannian_to_core(); la
[4, 2]
sage: type(la)
<class 'sage.combinat.core.Cores_length_with_category.element_class'>
sage: la.to_grassmannian() == w
True

sage: w = W.from_reduced_word([0,2,1])
sage: w.affine_grassmannian_to_core()
Traceback (most recent call last):
...
ValueError: Error! this only works on type 'A' affine Grassmannian elements
```

affine_grassmannian_to_partition()

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])
sage: w = W.from_reduced_word([0,2,1,0])
sage: la = w.affine_grassmannian_to_partition(); la
[2, 2]
sage: la.from_kbounded_to_grassmannian(k) == w
True
```

is_affine_grassmannian()

Tests 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 mimimal coset representative for W / cl W.

EXAMPLES:

```
sage: W=WeylGroup(['A',3,1])
sage: w=W.from_reduced_word([2,1,0])
sage: w.is_affine_grassmannian()
True
sage: w=W.from_reduced_word([2,0])
sage: w.is_affine_grassmannian()
False
sage: W.one().is_affine_grassmannian()
True
```

class `AffineWeylGroups.`**ParentMethods**

---

affine_grassmannian_elements_of_given_length(k)

Returns the affine Grassmannian elements of length k, as a list.

EXAMPLES:
```
sage: W=WeylGroup(['A',3,1])
sage: [x.reduced_word() 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()

Todo

should return an enumerated set, with iterator, ...

---

special_node()

Returns the distinguished special node of the underlying Dynkin diagram

EXAMPLES:
```
sage: W=WeylGroup(['A',3,1])
sage: W.special_node()
0
```

AffineWeylGroups.**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()
```

AffineWeylGroups.**super_categories**()

EXAMPLES:
```
sage: AffineWeylGroups().super_categories()
[Category of infinite weyl groups]
```

13.6 AlgebraIdeals

class sage.categories.algebra_ideals.**AlgebraIdeals**(A)

Bases: sage.categories.category_types.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]
sage: C = AlgebraIdeals(FreeAlgebra(QQ,2,'a,b'))
sage: C.super_categories()
[]
```

13.7 Algebra modules

class sage.categories.algebra_modules.**AlgebraModules**(*A*)
    Bases: [sage.categories.category_types.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]
```

13.8 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: `sage.categories.category_with_axiom.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:** `Algebras` will be eventually be replaced by `magmatic_algebras.MagmaticAlgebras`
> for consistency with e.g. Wikipedia article Algebras which assumes neither associativity nor the existence
> of a unit (see trac ticket #15043).

Todo

Should R be a commutative ring?

EXAMPLES:
```
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]
```

TESTS:
```
sage: TestSuite(Algebras(ZZ)).run()
```

class CartesianProducts(*category*, **args*)
    Bases: `sage.categories.cartesian_product.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#348f42ae77a66

> •http://en.wikipedia.org/wiki/Direct_product

extra_super_categories()
    A cartesian product of algebras is endowed with a natural algebra structure.

EXAMPLES:

```
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]
```

Algebras.**Commutative**

    alias of CommutativeAlgebras

class Algebras.**DualObjects**(*category*, **args*)

    Bases: sage.categories.dual.DualObjectsCategory

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    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 = 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 trac ticket #15647.

class Algebras.**ElementMethods**

Algebras.**Graded**

    alias of GradedAlgebras

class Algebras.**Quotients**(*category*, **args*)

    Bases: sage.categories.quotients.QuotientsCategory

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
```

```
....:        _functor_category = "FooBars"
....:        _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

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 {'y': B['y'], 'x': B['x'], 'b': 0, 'a': 0}
```

Todo

this could possibly remove the elements that retract to zero.

Algebras.**Semisimple**
    alias of SemisimpleAlgebras

class Algebras.**SubcategoryMethods**

Semisimple()
    Return the subcategory of semisimple objects of `self`.

Note: This mimics the syntax of axioms for a smooth transition if `Semisimple` becomes one.

    EXAMPLES:
```
sage: Algebras(QQ).Semisimple()
Category of semisimple algebras over Rational Field
sage: Algebras(QQ).WithBasis().FiniteDimensional().Semisimple()
Category of finite dimensional semisimple algebras with basis over Rational Field
```

class Algebras.**TensorProducts**(*category*, **args*)
    Bases: `sage.categories.tensor.TensorProductsCategory`

    TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:        _functor_category = "FooBars"
```

```
....:       _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ElementMethods

class Algebras.TensorProducts.**ParentMethods**

Algebras.TensorProducts.**extra_super_categories**()
   EXAMPLES:
```
sage: Algebras(QQ).TensorProducts().extra_super_categories()
[Category of algebras over Rational Field]
sage: Algebras(QQ).TensorProducts().super_categories()
[Category of algebras over Rational Field,
 Category of tensor products of vector spaces over Rational Field]
```

   Meaning: a tensor product of algebras is an algebra

Algebras.**WithBasis**
   alias of `AlgebrasWithBasis`

13.9 Algebras With Basis

class sage.categories.algebras_with_basis.**AlgebrasWithBasis**(*base_category*)
   Bases: `sage.categories.category_with_axiom.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 Rat

sage: A.category()
Category of algebras with basis over Rational Field

sage: A.one_basis()
word:
sage: A.one()
B[word: ]

sage: A.base_ring()
```

```
Rational Field
sage: A.basis().keys()
Finite Words over {'a', 'b', 'c'}

sage: (a,b,c) = A.algebra_generators()
sage: a^3, b^2
(B[word: aaa], B[word: bb])
sage: a*c*b
B[word: acb]

sage: A.product
<bound method FreeAlgebra_with_category._product_from_product_on_basis_multiply of
 An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over Ra
sage: A.product(a*b,b)
B[word: abb]

sage: TestSuite(A).run(verbose=True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_distributivity() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_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__
<class 'sage.categories.examples.algebras_with_basis.FreeAlgebra_with_category'>
sage: A.element_class
<class 'sage.combinat.free_module.FreeAlgebra_with_category.element_class'>
```

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

TESTS:
```
sage: TestSuite(AlgebrasWithBasis(QQ)).run()
```

class **CartesianProducts**(*category*, **args*)
    Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

    The category of algebras with basis, constructed as cartesian products of algebras with basis

---

Note: this construction give the direct products of algebras with basis. See comment in `Algebras.CartesianProducts`

class **ParentMethods**

> **one**()
> > TESTS:
> > ```
> > sage: A = AlgebrasWithBasis(QQ).example(); A
> > An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c
> > sage: B = cartesian_product((A, A, A))
> > sage: B.one()
> > B[(0, word: )] + B[(1, word: )] + B[(2, word: )]
> > ```

> **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
> > sage: A.one_basis()
> > word:
> >
> > sage: B = cartesian_product((A, A, A))
> > sage: B.one_from_cartesian_product_of_one_basis()
> > B[(0, word:)] + B[(1, word:)] + B[(2, word:)]
> > sage: B.one()
> > B[(0, word:)] + B[(1, word:)] + B[(2, word:)]
> >
> > sage: cartesian_product([SymmetricGroupAlgebra(QQ, 3), SymmetricGroupAlgebra(QQ, 4)])
> > B[(0, [1, 2, 3])] + B[(1, [1, 2, 3, 4])]
> > ```

> AlgebrasWithBasis.CartesianProducts.**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,
> >  Category of Cartesian products of vector spaces with basis over Rational Field]
> > ```

class AlgebrasWithBasis.**ElementMethods**

AlgebrasWithBasis.**FiniteDimensional**
> alias of `FiniteDimensionalAlgebrasWithBasis`

AlgebrasWithBasis.**Graded**
> alias of `GradedAlgebrasWithBasis`

class AlgebrasWithBasis.**ParentMethods**

one()
>   Return the multiplicative unit element.
>
>   EXAMPLES:
>   ```
>   sage: A = AlgebrasWithBasis(QQ).example()
>   sage: A.one_basis()
>   word:
>   sage: A.one()
>   B[word: ]
>   ```

class AlgebrasWithBasis.**TensorProducts**(*category*, **args*)
>   Bases: `sage.categories.tensor.TensorProductsCategory`
>
>   The category of algebras with basis constructed by tensor product of algebras with basis
>
>   **class ElementMethods**
>   >   Implements operations on elements of tensor products of algebras with basis
>
>   **class** AlgebrasWithBasis.TensorProducts.**ParentMethods**
>   >   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
>   >   >   sage: A.one_basis()
>   >   >   word:
>   >   >   sage: B = tensor((A, A, A))
>   >   >   sage: B.one_basis()
>   >   >   (word: , word: , word: )
>   >   >   sage: B.one()
>   >   >   B[word: ] # B[word: ] # B[word: ]
>   >   >   ```
>   >
>   >   **product_on_basis**(*t1*, *t2*)
>   >   >   The product of the algebra on the basis, as per `AlgebrasWithBasis.ParentMethods.product_on_bas`
>   >   >
>   >   >   EXAMPLES:
>   >   >   ```
>   >   >   sage: A = AlgebrasWithBasis(QQ).example(); A
>   >   >   An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c
>   >   >   sage: (a,b,c) = A.algebra_generators()
>   >   >
>   >   >   sage: x = tensor((a, b, c)); x
>   >   >   B[word: a] # B[word: b] # B[word: c]
>   >   >   sage: y = tensor((c, b, a)); y
>   >   >   B[word: c] # B[word: b] # B[word: a]
>   >   >   sage: x*y
>   >   >   B[word: ac] # B[word: bb] # B[word: ca]
>   >   >
>   >   >   sage: x = tensor(((a+2*b), c))   ; x
>   >   >   B[word: a] # B[word: c] + 2*B[word: b] # B[word: c]
>   >   >   sage: y = tensor((c,   a)) + 1; y
>   >   >   B[word:] # B[word:] + B[word: c] # B[word: a]
>   >   >   sage: x*y
>   >   >   B[word: a] # B[word: c] + B[word: ac] # B[word: ca] + 2*B[word: b] # B[word: c] + 2*B
>   >   >   ```

TODO: optimize this implementation!

AlgebrasWithBasis.TensorProducts.**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]
```

AlgebrasWithBasis.**example**(*alphabet=('a', 'b', 'c')*)
Return an example of algebra with basis.

EXAMPLES:
```
sage: AlgebrasWithBasis(QQ).example()
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over
```

An other set of generators can be specified as optional argument:
```
sage: AlgebrasWithBasis(QQ).example((1,2,3))
An example of an algebra with basis: the free algebra on the generators (1, 2, 3) over Ratio
```

13.10 Associative algebras

class sage.categories.associative_algebras.**AssociativeAlgebras**(*base_category*)
Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

The category of associative algebras over a given base ring.

An associative algebra over a ring R is a module over R which is also a not necessarily unital ring.

> **Warning:** Until [trac ticket #15043](#) is implemented, `Algebras` is the category of associative unital algebras; thus, unlike the name suggests, `AssociativeAlgebras` is not a subcategory of `Algebras` but of `MagmaticAlgebras`.

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

TESTS:
```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: C is MagmaticAlgebras(ZZ).Associative()
True
sage: TestSuite(C).run()
```

class ElementMethods
An abstract class for elements of an associative algebra.

> **Note:** `Magmas.Element.__mul__` is preferable to `Modules.Element.__mul__` since the later does not handle products of two elements of `self`.

TESTS:

```
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: x = A.an_element()
sage: x
B[word: ] + 2*B[word: a] + 3*B[word: b] + B[word: bab]
sage: x.__mul__(x)
B[word: ] + 4*B[word: a] + 4*B[word: aa] + 6*B[word: ab]
+ 2*B[word: abab] + 6*B[word: b] + 6*B[word: ba]
+ 2*B[word: bab] + 2*B[word: baba] + 3*B[word: babb]
+ B[word: babbab] + 9*B[word: bb] + 3*B[word: bbab]
```

AssociativeAlgebras.**Unital**
> alias of [Algebras](#)

13.11 Bialgebras

class sage.categories.bialgebras.**Bialgebras**(*base*, *name=None*)
> Bases: [sage.categories.category_types.Category_over_base_ring](#)

> The category of bialgebras

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

> TESTS:
> ```
> sage: TestSuite(Bialgebras(ZZ)).run()
> ```

> class **ElementMethods**

> class Bialgebras.**ParentMethods**

> Bialgebras.**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()
> > ```

> Bialgebras.**super_categories**()
> > EXAMPLES:
```

```
sage: Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]
```

13.12 Bialgebras with basis

sage.categories.bialgebras_with_basis.**BialgebrasWithBasis**(*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]
```

TESTS:
```
sage: TestSuite(BialgebrasWithBasis(ZZ)).run()
```

13.13 Bimodules

class sage.categories.bimodules.**Bimodules**(*left_base*, *right_base*, *name=None*)

Bases: sage.categories.category.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

class Bimodules.**ParentMethods**

Bimodules.**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 Bimodules.**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 precisi
    ```

Bimodules.**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
    ```

Bimodules.**right_base_ring**()
    Return the right base ring over which elements of this category are defined.

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

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

13.14 Classical Crystals

class sage.categories.classical_crystals.**ClassicalCrystals**(*s=None*)
    Bases: `sage.categories.category_singleton.Category_singleton`

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

    EXAMPLES:
    ```
    sage: C = ClassicalCrystals()
    sage: C
    Category of classical crystals
    sage: C.super_categories()
    [Category of regular crystals,
     Category of finite crystals,
     Category of highest weight crystals]
    sage: C.example()
    Highest weight crystal of type A_3 of highest weight omega_1
    ```

    TESTS:
    ```
    sage: TestSuite(C).run()
    sage: B = ClassicalCrystals().example()
    sage: TestSuite(B).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_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

class **ElementMethods**

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 λ 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 α_i.

EXAMPLES:
```
sage: B = crystals.Tableaux(['A',3],shape=[2,1])
sage: b = B(rows=[[1,2],[4]])
sage: b.lusztig_involution()
[[1, 4], [3]]
sage: b.to_tableau().schuetzenberger_involution(n=4)
[[1, 4], [3]]

sage: all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involuti
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]],
[[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]]], [[[-3]], [[-3]]],
[[[-2]], [[2]]], [[[-1]], [[1]]]]

sage: C = CartanType(['E',6])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
```

```
sage: t = T[3]; t
[(-4, 2, 5)]
sage: t.lusztig_involution()
[(-2, -3, 4)]
```

class `ClassicalCrystals.`**`ParentMethods`**

`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)
sage: chiT == chi^2
True
```

One may specify an alternate `WeylCharacterRing`:

```
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
```

It should have the same Cartan type and use the same realization of the weight lattice as `self`:

```
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
```

`demazure_character``(w, f=None)`

Returns the Demazure character associated to `w`.

INPUT:

- w – an element of the ambient weight lattice realization of the crystal, or a reduced word, or an element in the associated Weyl group

OPTIONAL:

- f – a function from the crystal to a module

This is currently only supported for crystals whose underlying weight space is the ambient space.

The Demazure character is obtained by applying the Demazure operator D_w (see
`sage.categories.regular_crystals.RegularCrystals.ParentMethods.demazure_operator`
to the highest weight element of the classical crystal. The simple Demazure operators D_i (see
`sage.categories.regular_crystals.RegularCrystals.ElementMethods.demazure_operator`
do not braid on the level of crystals, but on the level of characters they do. That is why it makes sense
to input `w` either as a weight, a reduced word, or as an element of the underlying Weyl group.

EXAMPLES:
```
sage: T = crystals.Tableaux(['A',2], shape = [2,1])
sage: e = T.weight_lattice_realization().basis()
sage: weight = e[0] + 2*e[2]
sage: weight.reduced_word()
[2, 1]
sage: T.demazure_character(weight)
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x1*x3^2

sage: T = crystals.Tableaux(['A',3],shape=[2,1])
sage: T.demazure_character([1,2,3])
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x2^2*x3
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([1,2,3])
sage: T.demazure_character(w)
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*x2*x3 + x2^2*x3

sage: T = crystals.Tableaux(['B',2], shape = [2])
sage: e = T.weight_lattice_realization().basis()
sage: weight = -2*e[1]
sage: T.demazure_character(weight)
x1^2 + x1*x2 + x2^2 + x1 + x2 + x1/x2 + 1/x2 + 1/x2^2 + 1

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:

opposition_automorphism()
    Deprecated in trac ticket #15560. Use the corresponding method in Cartan type.

EXAMPLES:
```
sage: T = crystals.Tableaux(['A',5],shape=[1])
sage: T.opposition_automorphism()
doctest:...: DeprecationWarning: opposition_automorphism is deprecated.
Use opposition_automorphism from the Cartan type instead.
See http://trac.sagemath.org/15560 for details.
Finite family {1: 5, 2: 4, 3: 3, 4: 2, 5: 1}
```

class ClassicalCrystals.**TensorProducts**(*category*, **args*)
    Bases: `sage.categories.tensor.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]
```

ClassicalCrystals.**additional_structure**()
    Return `None`.

Indeed, the category of classical crystals defines no additional structure: it only states that its objects are $U_q(\mathfrak{g})$-crystals, where $\mathfrak{g}$ is of finite type.

See also:

`Category.additional_structure()`

EXAMPLES:
```
sage: ClassicalCrystals().additional_structure()
```

ClassicalCrystals.**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
```

ClassicalCrystals.**super_categories**()

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

13.15 Coalgebras

class sage.categories.coalgebras.**Coalgebras**(*base*, *name=None*)

Bases: `sage.categories.category_types.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]
```

TESTS:
```
sage: TestSuite(Coalgebras(ZZ)).run()
```

class **DualObjects**(*category*, **args*)

Bases: `sage.categories.dual.DualObjectsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
```

```
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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 Ratio
```

> > **Warning:** This is only correct in certain cases (finite dimension, ...). See trac ticket #15647.

class Coalgebras.**ElementMethods**

coproduct()
> Returns 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
sage: [a,b] = A.algebra_generators()
sage: a, a.coproduct()
(B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
sage: b, b.coproduct()
(B[(1,3)], B[(1,3)] # B[(1,3)])
```

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

class Coalgebras.**ParentMethods**

coproduct(*x*)
> Returns 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
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*)

Returns the counit 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
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.

tensor_square()

Returns the tensor square of `self`

EXAMPLES:
```
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: A.tensor_square()
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
```

class Coalgebras.**Realizations**(*category*, **args*)

Bases: `sage.categories.realizations.RealizationsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

coproduct_by_coercion(*x*)

Returns the coproduct by coercion if coproduct_by_basis is not implemented.

EXAMPLES:

```
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()
m[] # m[2, 1] + m[1] # m[2] + m[2] # m[1] + m[2, 1] # m[]

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()
R[] # R[1] + R[1] # R[]
```

class Coalgebras.**TensorProducts**(*category*, **args*)

    Bases: `sage.categories.tensor.TensorProductsCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    class ElementMethods

    class Coalgebras.TensorProducts.**ParentMethods**

    Coalgebras.TensorProducts.**extra_super_categories**()

        EXAMPLES:

```
sage: Coalgebras(QQ).TensorProducts().extra_super_categories()
[Category of coalgebras over Rational Field]
sage: Coalgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of vector spaces over Rational Field,
 Category of coalgebras over Rational Field]
```

        Meaning: a tensor product of coalgebras is a coalgebra

Coalgebras.**WithBasis**

    alias of `CoalgebrasWithBasis`

class Coalgebras.**WithRealizations**(*category*, **args*)

Bases: `sage.categories.with_realizations.WithRealizationsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

> **coproduct** (*x*)
> Returns 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])
> S[] # S[2] + S[1] # S[1] + S[2] # S[]
> ```

> **counit** (*x*)
> Returns 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
> 2*S[] + 2*S[1] + 3*S[1, 1]
> sage: N.counit(x)
> 2
> ```

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

13.16 Coalgebras with basis

class sage.categories.coalgebras_with_basis.**CoalgebrasWithBasis**(*base_category*)

    Bases: sage.categories.category_with_axiom.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]
```

    TESTS:
```
sage: TestSuite(CoalgebrasWithBasis(ZZ)).run()
```

    class ElementMethods

    class CoalgebrasWithBasis.**ParentMethods**

        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
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
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)])
```

        coproduct_on_basis(*i*)
            The coproduct of the algebra on the basis (optional).

            INPUT:
                •i – the indices of an element of the basis of self
            Returns the coproduct of the corresponding basis elements If implemented, the coproduct of the alge-
            bra is defined from it by linearity.

            EXAMPLES:
```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
sage: (a, b) = A._group.gens()
sage: A.coproduct_on_basis(a)
B[(1,2,3)] # B[(1,2,3)]
```

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

            EXAMPLES:
```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
sage: [a,b] = A.algebra_generators()
```

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

counit_on_basis(*i*)

The counit of the algebra on the basis (optional).

INPUT:

- i – the indices of an element of the basis of self

Returns the counit of the corresponding basis elements If implemented, the counit of the algebra is defined from it by linearity.

EXAMPLES:

```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
sage: (a, b) = A._group.gens()
sage: A.counit_on_basis(a)
1
```

13.17 Commutative additive groups

class sage.categories.commutative_additive_groups.**CommutativeAdditiveGroups**(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton, sage.categories.category_types.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.

TESTS:

```
sage: TestSuite(CommutativeAdditiveGroups()).run()
sage: sorted(CommutativeAdditiveGroups().CartesianProducts().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital']
```

The empty covariant functorial construction category classes CartesianProducts and Algebras are left here for the sake of nicer output since this is a commonly used category:

```
sage: CommutativeAdditiveGroups().CartesianProducts()
Category of Cartesian products of commutative additive groups
```

```
sage: CommutativeAdditiveGroups().Algebras(QQ)
Category of commutative additive group algebras over Rational Field
```

Also, it's likely that some code will end up there at some point.

class **Algebras**(*category*, **args*)

 Bases: `sage.categories.algebra_functor.AlgebrasCategory`

 TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class CommutativeAdditiveGroups.**CartesianProducts**(*category*, **args*)

 Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

 TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

13.18 Commutative additive monoids

class sage.categories.commutative_additive_monoids.**CommutativeAdditiveMonoids**(*base_category*)

 Bases: `sage.categories.category_with_axiom.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.

TESTS:
```
sage: TestSuite(CommutativeAdditiveMonoids()).run()
```

13.19 Commutative additive semigroups

class sage.categories.commutative_additive_semigroups.**CommutativeAdditiveSemigroups**(*base_category*)

Bases: sage.categories.category_with_axiom.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 monoid: the free commutative monoid 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.

TESTS:
```
sage: TestSuite(C).run()
```

13.20 Commutative algebra ideals

class sage.categories.commutative_algebra_ideals.**CommutativeAlgebraIdeals**(*A*)

Bases: sage.categories.category_types.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]
```

13.21 Commutative algebras

class sage.categories.commutative_algebras.**CommutativeAlgebras**(*base_category*)
    Bases: [sage.categories.category_with_axiom.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
```

    TESTS:
```
sage: Algebras(QQ).Commutative() is CommutativeAlgebras(QQ)
True
sage: TestSuite(CommutativeAlgebras(ZZ)).run()
```

    Todo:

        • product (= cartesian product)

        • coproduct (= tensor product over base ring)

13.22 Commutative ring ideals

class sage.categories.commutative_ring_ideals.**CommutativeRingIdeals**(*R*)
    Bases: [sage.categories.category_types.Category_ideal](#)

    The category of ideals in a fixed commutative ring.

    EXAMPLES:
```
sage: C = CommutativeRingIdeals(IntegerRing())
sage: C
Category of commutative ring ideals in Integer Ring
```

    super_categories()
        EXAMPLES:

```
sage: CommutativeRingIdeals(ZZ).super_categories()
[Category of ring ideals in Integer Ring]
```

13.23 Commutative rings

class sage.categories.commutative_rings.**CommutativeRings**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

    The category of commutative rings

    commutative rings with unity, i.e. rings with commutative * and a multiplicative identity

    EXAMPLES:
```
sage: C = CommutativeRings(); C
Category of commutative rings
sage: C.super_categories()
[Category of rings, Category of commutative monoids]
```

    TESTS:
```
sage: TestSuite(C).run()

sage: QQ['x,y,z'] in CommutativeRings()
True
sage: GroupAlgebra(DihedralGroup(3), QQ) in CommutativeRings()
False
sage: MatrixSpace(QQ,2,2) in CommutativeRings()
False
```

    GroupAlgebra should be fixed:
```
sage: GroupAlgebra(CyclicPermutationGroup(3), QQ) in CommutativeRings() # todo: not implemented
True
```

    class ElementMethods

    class CommutativeRings.**Finite**(*base_category*)

        Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

        TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

        class ParentMethods

            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 $R = \mathbf{Z}/n\mathbf{Z}$ the smallest element of each coset is sometimes callled 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
- `cosets` – an optional lists of elements of `self`. If provided, the function only return the list of cosets that contain some element from `cosets`.

OUTPUT:

A list of lists.

EXAMPLES:
```
sage: Zmod(11).cyclotomic_cosets(2)
[[0], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
sage: Zmod(15).cyclotomic_cosets(2)
[[0], [1, 2, 4, 8], [3, 6, 9, 12], [5, 10], [7, 11, 13, 14]]
```

Since the group of invertible elements of a finite field is cyclic, the set of squares is a particular case of cyclotomic coset:
```
sage: K = GF(25,'z')
sage: a = K.multiplicative_generator()
sage: K.cyclotomic_cosets(a**2,cosets=[1])
[[1, 2, 3, 4, z + 1, z + 3,
  2*z + 1, 2*z + 2, 3*z + 3,
  3*z + 4, 4*z + 2, 4*z + 4]]
sage: sorted(b for b in K if not b.is_zero() and b.is_square())
[1, 2, 3, 4, z + 1, z + 3,
 2*z + 1, 2*z + 2, 3*z + 3,
 3*z + 4, 4*z + 2, 4*z + 4]
```

We compute some examples of minimal polynomials:
```
sage: K = GF(27,'z')
sage: a = K.multiplicative_generator()
sage: R.<X> = PolynomialRing(K, 'X')
sage: a.minimal_polynomial('X')
X^3 + 2*X + 1
sage: cyc3 = Zmod(26).cyclotomic_cosets(3,cosets=[1]); cyc3
[[1, 3, 9]]
sage: prod(X - a**i for i in cyc3[0])
X^3 + 2*X + 1

sage: (a**7).minimal_polynomial('X')
X^3 + X^2 + 2*X + 1
sage: cyc7 = Zmod(26).cyclotomic_cosets(3,cosets=[7]); cyc7
[[7, 11, 21]]
sage: prod(X - a**i for i in cyc7[0])
X^3 + X^2 + 2*X + 1
```

Cyclotomic cosets of fields are useful in combinatorial design theory to provide so called difference families (see Wikipedia article Difference_set). This is illustrated on the following examples:

```
sage: K = GF(5)
sage: a = K.multiplicative_generator()
sage: H = K.cyclotomic_cosets(a**2, cosets=[1,2]); H
[[1, 4], [2, 3]]
sage: sorted(x-y for D in H for x in D for y in D if x != y)
[1, 2, 3, 4]

sage: K = GF(37)
sage: a = K.multiplicative_generator()
sage: H = K.cyclotomic_cosets(a**4, cosets=[1]); H
[[1, 7, 9, 10, 12, 16, 26, 33, 34]]
sage: sorted(x-y for D in H for x in D for y in D if x != y)
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5, ..., 33, 34, 34, 35, 35, 36, 36]
```

13.24 Complete Discrete Valuation Rings (CDVR) and Fields (CDVF)

class sage.categories.complete_discrete_valuation.**CompleteDiscreteValuationFields**(*s=None*)

Bases: sage.categories.category_singleton.Category_singleton

The category of complete discrete valuation fields

EXAMPLES:

```
sage: Zp(7) in CompleteDiscreteValuationFields()
False
sage: QQ in CompleteDiscreteValuationFields()
False
sage: LaurentSeriesRing(QQ,'u') in CompleteDiscreteValuationFields()
True
sage: Qp(7) in CompleteDiscreteValuationFields()
True
sage: TestSuite(CompleteDiscreteValuationFields()).run()
```

class **ElementMethods**

precision_absolute()

Return the absolute precision of this element.

EXAMPLES:

```
sage: K = Qp(7)
sage: x = K(7); x
7 + O(7^21)
sage: x.precision_absolute()
21
```

precision_relative()

Return the relative precision of this element.

EXAMPLES:

```
sage: K = Qp(7)
sage: x = K(7); x
7 + O(7^21)
sage: x.precision_relative()
20
```

```
CompleteDiscreteValuationFields.super_categories()
    EXAMPLES:
    sage: CompleteDiscreteValuationFields().super_categories()
    [Category of discrete valuation fields]
```

class sage.categories.complete_discrete_valuation.**CompleteDiscreteValuationRings**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of complete discrete valuation rings

EXAMPLES:

```
sage: Zp(7) in CompleteDiscreteValuationRings()
True
sage: QQ in CompleteDiscreteValuationRings()
False
sage: QQ[['u']] in CompleteDiscreteValuationRings()
True
sage: Qp(7) in CompleteDiscreteValuationRings()
False
sage: TestSuite(CompleteDiscreteValuationRings()).run()
```

class ElementMethods

precision_absolute()

Return the absolute precision of this element.

EXAMPLES:

```
sage: R = Zp(7)
sage: x = R(7); x
7 + O(7^21)
sage: x.precision_absolute()
21
```

precision_relative()

Return the relative precision of this element.

EXAMPLES:

```
sage: R = Zp(7)
sage: x = R(7); x
7 + O(7^21)
sage: x.precision_relative()
20
```

CompleteDiscreteValuationRings.**super_categories**()

EXAMPLES:

```
sage: CompleteDiscreteValuationRings().super_categories()
[Category of discrete valuation rings]
```

13.25 Coxeter Group Algebras

class sage.categories.coxeter_group_algebras.**CoxeterGroupAlgebras**(*category,
args)

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCategor
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python modul
sage: TestSuite(C).run()
```

class **ParentMethods**

demazure_lusztig_eigenvectors(*q1*, *q2*)

Return the family of eigenvectors for the Cherednik operators.

INPUT:

- self – a finite Coxeter group W
- q1, q2 – two elements of the ground ring K

The affine Hecke algebra $H_{q_1,q_2}(\tilde{W})$ acts on the group algebra of W through the Demazure-Lusztig operators T_i. Its Cherednik operators Y^λ 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.CherednikOperatorsEige

EXAMPLES:

```
sage: W = WeylGroup(["B",2])
sage: W.element_class._repr_=lambda x: "".join(str(i) for i in 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))*B[2121] + ((-q2)/(-q1+q2))*B[121] - B[212] + B[12]
```

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
- bar – a boolean (default False)

See 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)*B[323123] + (q1+q2)*B[123]
sage: KW.demazure_lusztig_operator_on_basis(w, 1, q1, q2)
q1*B[1231]
sage: KW.demazure_lusztig_operator_on_basis(w, 2, q1, q2)
q1*B[1232]
sage: KW.demazure_lusztig_operator_on_basis(w, 3, q1, q2)
(q1+q2)*B[123] + (-q2)*B[12]
```

At $q_1 = 1$ and $q_2 = 0$ we recover the action of the isobaric divided differences π_i:
```
sage: KW.demazure_lusztig_operator_on_basis(w, 0, 1, 0)
B[123]
sage: KW.demazure_lusztig_operator_on_basis(w, 1, 1, 0)
B[1231]
sage: KW.demazure_lusztig_operator_on_basis(w, 2, 1, 0)
B[1232]
sage: KW.demazure_lusztig_operator_on_basis(w, 3, 1, 0)
B[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)
B[323123]
sage: KW.demazure_lusztig_operator_on_basis(w, 1, 1, -1)
B[1231]
sage: KW.demazure_lusztig_operator_on_basis(w, 2, 1, -1)
B[1232]
sage: KW.demazure_lusztig_operator_on_basis(w, 3, 1, -1)
B[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 $R \to R$ obtained by interpolating between the simple projection π_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:
```
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
B[123]
sage: T[0](x)
(-q2)*B[323123] + (q1+q2)*B[123]
sage: T[1](x)
q1*B[1231]
sage: T[2](x)
q1*B[1232]
sage: T[3](x)
(q1+q2)*B[123] + (-q2)*B[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.

13.26 Coxeter Groups

class sage.categories.coxeter_groups.**CoxeterGroups**(*s=None*)

 Bases: `sage.categories.category_singleton.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 $(s_i s_j)^{m_{i,j}} = 1$.

 I is the *index set* of W and $|I|$ is the *rank* of W.

 See Wikipedia article Coxeter_group for details.

 EXAMPLES:
```
sage: C = CoxeterGroups(); C
Category of coxeter groups
sage: C.super_categories()
[Category of finitely generated 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:
```
sage: FiniteCoxeterGroups().example()
The 5-th dihedral group of order 10
sage: FiniteWeylGroups().example()
The symmetric group on {0, ..., 3}
sage: WeylGroup(["B", 3])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
```

 Those 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
```

Todo

add a demo of usual computations on Coxeter groups.

See also:

WeylGroups, sage.combinat.root_system

> **Warning:** It is assumed that morphisms in this category preserve the distinguished choice of simple reflec-
> tions. 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
> ```

TESTS:

```
sage: W = CoxeterGroups().example(); TestSuite(W).run(verbose = "True")
running ._test_an_element() . . . pass
running ._test_associativity() . . . 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_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_has_descent() . . . pass
running ._test_inverse() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_reduced_word() . . . pass
running ._test_simple_projections() . . . pass
running ._test_some_elements() . . . pass
```

Algebras
    alias of `CoxeterGroupAlgebras`

class ElementMethods

absolute_le(*other*)

Return whether `self` is smaller than `other` in the absolute order.

A general reflection is an element of the form ws_iw^{-1}, where s_i is a simple reflection. The absolute order is defined analogously to the weak order but using general reflections rather than just simple reflections.

This partial order can be used to define noncrossing partitions associated with this Coxeter group.

See also:

absolute_length()

EXAMPLES:

```
sage: W = WeylGroup(["A", 3])
sage: s = W.simple_reflections()
sage: w0 = s[1]
sage: w1 = s[1]*s[2]*s[3]
sage: w0.absolute_le(w1)
True
sage: w1.absolute_le(w0)
False
sage: w1.absolute_le(w1)
True
```

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.

See also:

absolute_le().

EXAMPLES:

```
sage: W = WeylGroup(["A", 3])
sage: s = W.simple_reflections()
sage: (s[1]*s[2]*s[3]).absolute_length()
3
```

apply_conjugation_by_simple_reflection(*i*)

Conjugates `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_demazure_product(*element*, *side='right'*, *length_increasing=True*)

Returns the Demazure or 0-Hecke product of `self` with another Coxeter group element.

See `CoxeterGroups.ParentMethods.simple_projections()`.

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:

```
sage: W = WeylGroup(['C',4],prefix="s")
sage: v = W.from_reduced_word([1,2,3,4,3,1])
sage: v.apply_demazure_product([1,3,4,3,3])
s4*s1*s2*s3*s4*s3*s1
sage: v.apply_demazure_product([1,3,4,3],side='left')
s3*s4*s1*s2*s3*s4*s2*s3*s1
sage: v.apply_demazure_product((1,3,4,3),side='left')
s3*s4*s1*s2*s3*s4*s2*s3*s1
sage: v.apply_demazure_product(v)
s2*s3*s4*s1*s2*s3*s4*s2*s3*s2*s1
```

apply_simple_projection(*i*, *side='right'*, *length_increasing=True*)

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

Returns the result of the application of the simple projection π_i (resp. $\overline{\pi}_i$) on self.

See `CoxeterGroups.ParentMethods.simple_projections()` for the definition of the simple projections.

EXAMPLE:

```
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")
sage: v = W.from_reduced_word([1,2,3,4,3,1])
sage: v
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(2)
s1*s2*s3*s4*s3*s1*s2
sage: v.apply_simple_projection(2, side='left')
s1*s2*s3*s4*s3*s1
sage: v.apply_simple_projection(1, length_increasing = False)
s1*s2*s3*s4*s3
```

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

Returns self multiplied by the simple reflection s[i]

INPUT:

•i – an element of the index set

•side – "left" or "right" (default: "right")

This default implementation simply calls `apply_simple_reflection_left()` or `apply_simple_reflection_right()`.

EXAMPLES:

```
sage: W=CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection(0, side = "left")
(0, 2, 3, 1)
```

```
sage: w.apply_simple_reflection(1, side = "left")
(2, 1, 3, 0)
sage: w.apply_simple_reflection(2, side = "left")
(1, 3, 2, 0)

sage: w.apply_simple_reflection(0, side = "right")
(2, 1, 3, 0)
sage: w.apply_simple_reflection(1, side = "right")
(1, 3, 2, 0)
sage: w.apply_simple_reflection(2, side = "right")
(1, 2, 0, 3)
```

By default, `side` is "right":
```
sage: w.apply_simple_reflection(0)
(2, 1, 3, 0)
```

TESTS:
```
sage: w.apply_simple_reflection_right.__module__
'sage.categories.coxeter_groups'
```

apply_simple_reflection_left(*i*)

Returns `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:
```
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)
```

TESTS:
```
sage: w.apply_simple_reflection_left.__module__
'sage.categories.coxeter_groups'
```

apply_simple_reflection_right(*i*)

Returns `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)
```

TESTS:

```
sage: w.apply_simple_reflection_right.__module__
'sage.categories.coxeter_groups'
```

apply_simple_reflections(*word*, *side='right'*)
 INPUT:
  •word – A sequence of indices of Coxeter generators
  •side – Indicates multiplying from left or right
 Returns the result of the (left/right) multiplication of word to self. self is not changed.

 EXAMPLES:
```
sage: W=CoxeterGroups().example()
sage: w=W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1])
(2, 3, 1, 0)
sage: w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1],side='left')
(0, 1, 3, 2)
```

binary_factorizations(*predicate=The constant function (...) -> True*)
 Returns 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])
sage: s = W.simple_reflections()
sage: w0 = W.from_reduced_word([1,2,3,1,2,1])
sage: w0.binary_factorizations().cardinality()
24
```

 The same number of factorizations, by bounded length:
```
sage: [w0.binary_factorizations(lambda u: u.length() <= l).cardinality() for l in [-1,0,
[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() for i in [1,2,3]]
[12, 12, 12]
sage: w0.binary_factorizations(lambda u: False).cardinality()
0
```

 TESTS:
```
sage: w0.binary_factorizations().category()
Category of finite enumerated sets
```

bruhat_le(*other*)
 Bruhat comparison

 INPUT:
  •other - an element of the same Coxeter group
 OUTPUT: a boolean

 Returns whether self <= other in the Bruhat order.

EXAMPLES:
```
sage: W = WeylGroup(["A",3])
sage: u = W.from_reduced_word([1,2,1])
sage: v = W.from_reduced_word([1,2,3,2,1])
sage: u.bruhat_le(u)
True
sage: u.bruhat_le(v)
True
sage: v.bruhat_le(u)
False
sage: v.bruhat_le(v)
True
sage: s = W.simple_reflections()
sage: s[1].bruhat_le(W.one())
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 Mobius 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 of v, and c is the cost of the low level methods `first_descent()`, `has_descent()`, `apply_simple_reflection()`, etc. Those are typically $O(n)$, where n is the rank of the Coxeter group.

TESTS:

We now run consistency tests with permutations and `bruhat_lower_covers()`:
```
sage: W = WeylGroup(["A",3])
sage: P4 = Permutations(4)
sage: def P4toW(w): return W.from_reduced_word(w.reduced_word())
sage: for u in P4:
...         for v in P4:
...             assert u.bruhat_lequal(v) == P4toW(u).bruhat_le(P4toW(v))

sage: W = WeylGroup(["B",3])
sage: P = W.bruhat_poset() # This is built from bruhat_lower_covers
sage: Q = Poset((W, attrcall("bruhat_le")))                          # long time (10s
sage: all( u.bruhat_le(v) == P.is_lequal(u,v) for u in W for v in W ) # long time  (7s)
True
sage: all( P.is_lequal(u,v) == Q.is_lequal(u,v) for u in W for v in W)    # long time
True
```

bruhat_lower_covers()
> Returns 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 $\{ws_i, u_1s_i, u_2s_i, ..., u_js_i\}$, where the u_k are elements that ws_i covers that also do not have a descent at i.
>
> EXAMPLES:
> ```
> sage: W = WeylGroup(["A",3])
> sage: w = W.from_reduced_word([3,2,3])
> sage: print([v.reduced_word() for v in w.bruhat_lower_covers()])
> [[3, 2], [2, 3]]
>
> sage: W = WeylGroup(["A",3])
> sage: print([v.reduced_word() 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])
sage: print([v.reduced_word() for v in w.bruhat_lower_covers()])
[[2], [0]]
```

We now show how to construct the Bruhat poset:

```
sage: W = WeylGroup(["A",3])
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()
   Returns 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:
```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_lower_covers_reflections()
[(s1*s2*s1, s1*s2*s3*s2*s1), (s3*s2*s1, s2), (s3*s1*s2, s1)]
```

bruhat_upper_covers()
   Returns 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 ws_i is a cover; if w has a decent at i, then u_js_i is a cover of w where u_j is a cover of ws_i.

   EXAMPLES:
```
sage: W = WeylGroup(['A',3,1], prefix="s")
sage: w = W.from_reduced_word([1,2,1])
sage: w.bruhat_upper_covers()
[s1*s2*s1*s0, s1*s2*s0*s1, s0*s1*s2*s1, s3*s1*s2*s1, s2*s3*s1*s2, s1*s2*s3*s1]
```

```
sage: W = WeylGroup(['A',3])
sage: w = W.long_element()
sage: w.bruhat_upper_covers()
[]
```

```
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([1,2,1])
sage: S = [v for v in W if w in v.bruhat_lower_covers()]
sage: C = w.bruhat_upper_covers()
sage: set(S) == set(C)
True
```

bruhat_upper_covers_reflections()
   Returns 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:
```
sage: W = WeylGroup(['A',4], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_upper_covers_reflections()
[(s1*s2*s3*s2*s1, s3), (s2*s3*s1*s2*s1, s2*s3*s2), (s3*s4*s1*s2*s1, s4), (s4*s3*s1*s2*s1
```

canonical_matrix()

Return the matrix of `self` in the canonical faithful representation.

This is an n-dimension real faithful essential representation, where n is the number of generators of the Coxeter group. Note that this is not always the most natural matrix representation, for instance in type A_n.

EXAMPLES:
```
sage: W = WeylGroup(["A", 3])
sage: s = W.simple_reflections()
sage: (s[1]*s[2]*s[3]).canonical_matrix()
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

coset_representative(*index_set*, *side='right'*)

INPUT:
- `index_set` - a subset (or iterable) of the nodes of the Dynkin diagram
- `side` - 'left' or 'right'

Returns 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.

EXAMPLES:
```
sage: W = CoxeterGroups().example(5)
sage: s = W.simple_reflections()
sage: w = s[2]*s[1]*s[3]
sage: w.coset_representative([]).reduced_word()
[2, 3, 1]
sage: w.coset_representative([1]).reduced_word()
[2, 3]
sage: w.coset_representative([1,2]).reduced_word()
[2, 3]
sage: w.coset_representative([1,3]                ).reduced_word()
[2]
sage: w.coset_representative([2,3]                ).reduced_word()
[2, 1]
sage: w.coset_representative([1,2,3]              ).reduced_word()
[]
sage: w.coset_representative([],     side = 'left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([1],    side = 'left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([1,2],  side = 'left').reduced_word()
[3]
sage: w.coset_representative([1,3],  side = 'left').reduced_word()
[2, 3, 1]
sage: w.coset_representative([2,3],  side = 'left').reduced_word()
[1]
sage: w.coset_representative([1,2,3], side = 'left').reduced_word()
```

```
[]
```

cover_reflections(*side='right'*)

Returns the set of reflections `t` such that `self t` covers `self`.

If `side` is 'left', `t self` covers `self`.

EXAMPLES:
```
sage: W = WeylGroup(['A',4], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.cover_reflections()
[s3, s2*s3*s2, s4, s1*s2*s3*s4*s3*s2*s1]
sage: w.cover_reflections(side = 'left')
[s4, s2, s1*s2*s1, s3*s4*s3]
```

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^∞.

INPUT:
   •`c`– a Coxeter element.
OUTPUT:

the `c`-sorting word of `self` as a list of integers.

EXAMPLES:
```
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
```

deodhar_factor_element(*w*, *index_set*)

Returns Deodhar's Bruhat order factoring element.

INPUT:
   •`w` is an element of the same Coxeter group `W` as `self`
   •`index_set` is a subset of Dynkin nodes defining a parabolic subgroup `W'` of `W`
It is assumed that `v = self` and `w` are minimum length coset representatives for `W/W'` such that `v` $\leq$ `w` in Bruhat order.

OUTPUT:

Deodhar's element `f(v,w)` is the unique element of `W'` such that, for all `v'` and `w'` in `W'`, `vv'` $\leq$ `ww'` in `W` if and only if `v'` $\leq$ `f(v,w) * w'` in `W'` where $*$ is the Demazure product.

EXAMPLES:
```
sage: W = WeylGroup(['A',5],prefix="s")
sage: v = W.from_reduced_word([5])
sage: w = W.from_reduced_word([4,5,2,3,1,2])
sage: v.deodhar_factor_element(w,[1,3,4])
s3*s1
sage: W=WeylGroup(['C',2])
sage: w=W.from_reduced_word([2,1])
sage: w.deodhar_factor_element(W.from_reduced_word([2]),[1])
Traceback (most recent call last):
...
ValueError: [2, 1] is not of minimum length in its coset for the parabolic subgroup with
```

REFERENCES:

deodhar_lift_down(*w*, *index_set*)

Letting v = $self$, given a Bruhat relation v W' $\geq w$ W' among cosets with respect to the subgroup W' given by the Dynkin node subset `index_set`, returns the Bruhat-maximum lift x of wW' such that $v \geq x$.

INPUT:

- w is an element of the same Coxeter group W as `self`.
- `index_set` is a subset of Dynkin nodes defining a parabolic subgroup W'.

OUTPUT:

The unique Bruhat-maximum element x in W such that x W' = w W' and v `\ge` `x`.

See also:

`sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_up()`

EXAMPLES:

```
sage: W = WeylGroup(['A',3],prefix="s")
sage: v = W.from_reduced_word([1,2,3,2])
sage: w = W.from_reduced_word([3,2])
sage: v.deodhar_lift_down(w, [3])
s2*s3*s2
```

deodhar_lift_up(*w*, *index_set*)

Letting v = $self$, given a Bruhat relation v W' $\leq w$ W' among cosets with respect to the subgroup W' given by the Dynkin node subset `index_set`, returns the Bruhat-minimum lift x of wW' such that $v \leq x$.

INPUT:

- w is an element of the same Coxeter group W as `self`.
- `index_set` is a subset of Dynkin nodes defining a parabolic subgroup W'.

OUTPUT:

The unique Bruhat-minimum element x in W such that x W' = w W' and $v \leq x$.

See also:

`sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar_lift_down()`

EXAMPLES:

```
sage: W = WeylGroup(['A',3],prefix="s")
sage: v = W.from_reduced_word([1,2,3])
sage: w = W.from_reduced_word([1,3,2])
sage: v.deodhar_lift_up(w, [3])
s1*s2*s3*s2
```

descents(*side='right'*, *index_set=None*, *positive=False*)

INPUT:

- `index_set` - a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
- `side` - 'left' or 'right' (default: 'right')
- `positive` - a boolean (default: `False`)

Returns the descents of self, as a list of elements of the index_set.

The `index_set` option can be used to restrict to the parabolic subgroup indexed by `index_set`.

If positive is `True`, then returns the non-descents instead

TODO: find a better name for `positive`: complement? non_descent?

Caveat: the return type may change to some other iterable (tuple, ...) in the future. Please use keyword arguments also, as the order of the arguments may change as well.

EXAMPLES:

```
sage: W=CoxeterGroups().example()
sage: s=W.simple_reflections()
sage: w=s[0]*s[1]
sage: w.descents()
[1]
sage: w=s[0]*s[2]
sage: w.descents()
[0, 2]

TODO: side, index_set, positive
```

first_descent(*side='right'*, *index_set=None*, *positive=False*)
　　Returns the first left (resp. right) descent of self, as ane 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]
sage: w.first_descent()
0
sage: w = s[0]*s[1]
sage: w.first_descent()
1
```

has_descent(*i*, *side='right'*, *positive=False*)
　　Returns 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:

```
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*)

Returns 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:
```
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
```

TESTS:
```
sage: w.has_left_descent.__module__
'sage.categories.coxeter_groups'
```

has_right_descent(*i*)

Returns whether i is a right descent of self.

EXAMPLES:
```
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
sage: WeylGroup(['A',2]).long_element().has_right_descent(1)
True
```

inverse()

Returns the inverse of self

EXAMPLES:
```
sage: W=WeylGroup(['B',7])
sage: w=W.an_element()
sage: u=w.inverse()
sage: u==~w
True
sage: u*w==w*u
True
sage: u*w
[1 0 0 0 0 0 0]
```

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

inversions_as_reflections()

Returns the set of reflections `r` such that `self r < self`.

EXAMPLES:

```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.inversions_as_reflections()
[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:

```
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
sage: w.is_coxeter_sortable(c)
False
sage: w = W.from_reduced_word([0,2,1,0,2])
sage: w.coxeter_sorting_word(c)
[2, 0, 1, 2, 0]
sage: w.is_coxeter_sortable(c)
True
sage: W = CoxeterGroup(['A',3])
sage: c = W.from_reduced_word([1,2,3])
sage: len([w for w in W if w.is_coxeter_sortable(c)]) # number of c-sortable elements in
14
```

is_grassmannian(*side='right'*)

  INPUT:

  - `side` - "left" or "right" (default: "right")

  Tests whether `self` is Grassmannian, i.e. it has at most one descent on the right (resp. on the left).

  v 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
```

left_inversions_as_reflections()

Returns the set of reflections r such that r self < self.

EXAMPLES:
```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.left_inversions_as_reflections()
[s1, s3, s1*s2*s3*s2*s1, s2*s3*s2]
```

length()

Returns the length of self, that 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: sum((x^w.length()) for w in W) - expand(prod(sum(x^i for i in range(j+1)) for j in
0
```

SEE ALSO: reduced_word()

TODO: Should use reduced_word_iterator (or reverse_iterator)

lower_cover_reflections(*side='right'*)

Returns the reflections t such that self covers self t.

If side is 'left', self covers t self.

EXAMPLES:
```
sage: W = WeylGroup(['A',3],prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.lower_cover_reflections()
[s1*s2*s3*s2*s1, s2, s1]
```

```
sage: w.lower_cover_reflections(side = 'left')
[s2*s3*s2, s3, s1]
```

lower_covers(*side='right'*, *index_set=None*)

Returns all elements that `self` covers 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])
sage: w = W.from_reduced_word([3,2,1])
sage: [x.reduced_word() for x in w.lower_covers()]
[[3, 2]]
```

To obtain covers for left weak order, set the option side to 'left':

```
sage: [x.reduced_word() for x in w.lower_covers(side='left')]
[[2, 1]]
sage: w = W.from_reduced_word([3,2,3,1])
sage: [x.reduced_word() for x in w.lower_covers()]
[[2, 3, 2], [3, 2, 1]]
```

Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:

```
sage: [x.reduced_word() for x in w.lower_covers(index_set = [1,2])]
[[2, 3, 2]]
sage: [x.reduced_word() for x in w.lower_covers(side='left')]
[[3, 2, 1], [2, 3, 1]]
```

min_demazure_product_greater(*element*)

Finds the unique Bruhat-minimum element u such that v ≤ w * u where v is `self`, w is `element` and * is the Demazure product.

INPUT:
  • `element` is either an element of the same Coxeter group as `self` or a list (such as a reduced word) of elements from the index set of the Coxeter group.

EXAMPLES:

```
sage: W = WeylGroup(['A',4],prefix="s")
sage: v = W.from_reduced_word([2,3,4,1,2])
sage: u = W.from_reduced_word([2,3,2,1])
sage: v.min_demazure_product_greater(u)
s4*s2
sage: v.min_demazure_product_greater([2,3,2,1])
s4*s2
sage: v.min_demazure_product_greater((2,3,2,1))
s4*s2
```

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} = $ self, where the s_i are the simple reflections.

EXAMPLES:

```
sage: W=CoxeterGroups().example()
sage: s=W.simple_reflections()
sage: w=s[0]*s[1]*s[2]
```

```
sage: w.reduced_word()
[0, 1, 2]
sage: w=s[0]*s[2]
sage: w.reduced_word()
[2, 0]
```

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 $m \geq 2$).

This graph is always connected (a theorem due to Tits) and has no multiple edges.

EXAMPLES:

```
sage: W = WeylGroup(['A',3], prefix='s')
sage: w0 = W.long_element()
sage: G = w0.reduced_word_graph()
sage: G.num_verts()
16
sage: len(w0.reduced_words())
16
sage: G.num_edges()
18
sage: len([e for e in G.edges() if e[2] == 2])
10
sage: len([e for e in G.edges() if e[2] == 3])
8
```

TESTS:

```
sage: w1 = W.one()
sage: G = w1.reduced_word_graph()
sage: G.num_verts()
1
sage: G.num_edges()
0
```

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:

```
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.

   EXAMPLES:
```
sage: W=CoxeterGroups().example()
sage: s=W.simple_reflections()
sage: w=s[0]*s[2]
sage: w.reduced_words()
[[2, 0], [0, 2]]
sage: W=WeylGroup(['E',6])
sage: w=W.from_reduced_word([2,3,4,2])
sage: w.reduced_words()
[[3, 2, 4, 2], [2, 3, 4, 2], [3, 4, 2, 4]]
```

   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()`

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*)
   Returns 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])
sage: w = W.from_reduced_word([2,3])
sage: [x.reduced_word() for x in w.upper_covers()]
[[2, 3, 1], [2, 3, 2]]
```

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]]
```

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])]
[[2, 3, 1]]
sage: [x.reduced_word() for x in w.upper_covers(side = 'left', index_set = [1])]
[[1, 2, 3]]
```

weak_covers (*side='right'*, *index_set=None*, *positive=False*)
    Returns 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:
```
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([3,2,1])
sage: [x.reduced_word() for x in w.weak_covers()]
[[3, 2]]
```

To obtain instead elements that cover self, set `positive = True`:
```
sage: [x.reduced_word() for x in w.weak_covers(positive = True)]
[[3, 1, 2, 1], [2, 3, 2, 1]]
```

To obtain covers for left weak order, set the option side to 'left':
```
sage: [x.reduced_word() for x in w.weak_covers(side='left')]
[[2, 1]]
sage: w = W.from_reduced_word([3,2,3,1])
sage: [x.reduced_word() for x in w.weak_covers()]
[[2, 3, 2], [3, 2, 1]]
sage: [x.reduced_word() for x in w.weak_covers(side='left')]
[[3, 2, 1], [2, 3, 1]]
```

Covers w.r.t. a parabolic subgroup are obtained with the option `index_set`:
```
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

    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])
sage: u = W.from_reduced_word([1,2])
sage: v = W.from_reduced_word([1,2,3,2])
sage: u.weak_le(u)
True
```

```
sage: u.weak_le(v)
True
sage: v.weak_le(u)
False
sage: v.weak_le(v)
True
```

Comparison for left weak order is achieved with the option `side`:

```
sage: u.weak_le(v, side = 'left')
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: $O(l * c)$, where l is the minimum of the lengths of u and of v, and c is the cost of the low level methods `first_descent()`, `has_descent()`, `apply_simple_reflection()`. 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])
sage: P4 = Permutations(4)
sage: def P4toW(w): return W.from_reduced_word(w.reduced_word())
sage: for u in P4:      # long time (5s on sage.math, 2011)
...         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)
```

CoxeterGroups.**Finite**
   alias of `FiniteCoxeterGroups`

class CoxeterGroups.**ParentMethods**

bruhat_interval(*x*, *y*)
   Returns the list of t such that x <= t <= y.

   EXAMPLES:

```
sage: W = WeylGroup("A3", prefix="s")
sage: [s1,s2,s3]=W.simple_reflections()
sage: W.bruhat_interval(s2,s1*s3*s2*s1*s3)
[s1*s2*s3*s2*s1, s2*s3*s2*s1, s3*s1*s2*s1, s1*s2*s3*s1, s1*s2*s3*s2, s3*s2*s1, s2*s3*s1,
sage: W = WeylGroup(['A',2,1], prefix="s")
sage: [s0,s1,s2]=W.simple_reflections()
sage: W.bruhat_interval(1,s0*s1*s2)
[s0*s1*s2, s1*s2, s0*s2, s0*s1, s2, s1, s0, 1]
```

canonical_representation()
   Return the canonical faithful representation of `self`.

   EXAMPLES:

```
sage: W = WeylGroup("A3")
sage: W.canonical_representation()
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]
```

demazure_product(*Q*)
   Returns 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 `CoxeterGroups.ParentMethods.simple_projections()`.

EXAMPLES:

```
sage: W = WeylGroup(['A',2])
sage: w = W.demazure_product([2,2,1])
sage: w.reduced_word()
[2, 1]

sage: w = W.demazure_product([2,1,2,1,2])
sage: w.reduced_word()
[1, 2, 1]

sage: W = WeylGroup(['B',2])
sage: w = W.demazure_product([2,1,2,1,2])
sage: w.reduced_word()
[2, 1, 2, 1]
```

elements_of_length(*n*)

Return all elements of length n.

EXAMPLES:

```
sage: A = AffinePermutationGroup(['A',2,1])
sage: [len(list(A.elements_of_length(i))) for i in [0..5]]
[1, 3, 6, 9, 12, 15]

sage: W = CoxeterGroup(['H',3])
sage: [len(list(W.elements_of_length(i))) for i in range(4)]
[1, 3, 5, 7]

sage: W = CoxeterGroup(['A',2])
sage: [len(list(W.elements_of_length(i))) for i in range(6)]
[1, 2, 2, 1, 0, 0]
```

from_reduced_word(*word*)

INPUT:

- `word` - a list (or iterable) of elements of `self.index_set()`

Returns the group element corresponding to the given word. Namely, if `word` is $[i_1, i_2, \ldots, i_k]$, then this returns the corresponding product of simple reflections $s_{i_1} s_{i_2} \cdots s_{i_k}$.

Note: the main use case is for constructing elements from reduced words, hence the name of this method. But actually the input word need *not* be reduced.

EXAMPLES:

```
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)
```

See also :meth:'._test_reduced_word':

```
sage: W._test_reduced_word()
```

TESTS:

```
sage: W=WeylGroup(['E',6])
sage: W.from_reduced_word([2,3,4,2])
[ 0  1  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0]
[-1  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0]
[ 0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  0  1]
```

grassmannian_elements(*side='right'*)

    INPUT:

        •side: "left" or "right" (default: "right")

    Returns the left or right grassmanian elements of self, as an enumerated set

    EXAMPLES:

```
sage: S = CoxeterGroups().example()
sage: G = S.grassmannian_elements()
sage: G.cardinality()
12
sage: G.list()
[(0, 1, 2, 3), (1, 0, 2, 3), (2, 0, 1, 3), (3, 0, 1, 2), (0, 2, 1, 3), (1, 2, 0, 3), (0,
sage: sorted(tuple(w.descents()) for w in G)
[(), (0,), (0,), (0,), (1,), (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,), (1,), (2,), (2,), (2,)]
```

group_generators()

    Implements Groups.ParentMethods.group_generators() by returning the simple reflections of self.

    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)}
```

    Those give 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]}
```

index_set()

> Returns the index set of (the simple reflections of) `self`, as a list (or iterable).
>
> EXAMPLES:
> ```
> sage: W = FiniteCoxeterGroups().example(); W
> The 5-th dihedral group of order 10
> sage: W.index_set()
> [1, 2]
> ```

random_element_of_length(*n*)

> Return a random element of length `n` in `self`.
>
> Starts at the identity, then chooses an upper cover at random.
>
> Not very uniform: actually constructs a uniformly random reduced word of length n. Thus we most likely get elements with lots of reduced words!
>
> EXAMPLES:
> ```
> sage: A = AffinePermutationGroup(['A', 7, 1])
> sage: p = A.random_element_of_length(10)
> sage: p in A
> True
> sage: p.length() == 10
> True
>
> sage: W = CoxeterGroup(['A', 4])
> sage: p = W.random_element_of_length(5)
> sage: p in W
> True
> sage: p.length() == 5
> True
> ```

rank()

> Return the rank of `self`.
>
> EXAMPLES:
> ```
> sage: W = CoxeterGroups().example()
> sage: W.rank()
> 3
> ```

semigroup_generators()

> Implements `Groups.ParentMethods.group_generators()` by returning the simple reflections of `self`.
>
> 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)}
> ```
>
> Those give 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_projection(*i*, *side='right'*, *length_increasing=True*)

    INPUT:

        • `i` - an element of the index set of `self`

    Returns the simple projection π_i (or $\overline{\pi}_i$ if $length_increasing$ is False).

    See `simple_projections()` for the options and for the definition of the simple projections.

    EXAMPLES:

```
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)
```

simple_projections(*side='right'*, *length_increasing=True*)

    Returns the family of simple projections, also known as 0-Hecke or Demazure operators.

    INPUT:

        • `self` - a Coxeter group W

        • `side` - 'left' or 'right' (default: 'right')

        • `length_increasing` - a boolean (default: True) specifying whether the operator increases or decreases length

    Returns the simple projections of W, as a family.

    To each simple reflection s_i of W, corresponds a *simple projection* π_i from W to W defined by:

        $\pi_i(w) = ws_i$ if i is not a descent of w $\pi_i(w) = w$ otherwise.

    The simple projections $(\pi_i)_{i\in I}$ move elements down the right permutohedron, toward the maximal element. They satisfy the same braid relations as the simple reflections, but are idempotents $\pi_i^2 = \pi$ not involutions $s_i^2 = 1$. As such, the simple projections generate the 0-Hecke monoid.

    By symmetry, one can also define the projections $(\overline{\pi}_i)_{i\in I}$ (when the option `length_increasing` is False):

        $\overline{\pi}_i(w) = ws_i$ if i is a descent of w $\overline{\pi}_i(w) = w$ otherwise.

    as well as the analogues acting on the left (when the option `side` is 'left').

    EXAMPLES:

```
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: pi=W.simple_projections()
sage: pi
Finite family {0: <function <lambda> at ...>, 1: <function <lambda> at ...>, 2: <function
sage: pi[1](sigma)
(1, 3, 2, 0)
sage: W.simple_projection(1)(sigma)
(1, 3, 2, 0)
```

simple_reflection(*i*)

   INPUT:

   •i - an element from the index set.

   Returns the simple reflection s_i

   EXAMPLES:

```
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)
```

simple_reflections()

   Returns the simple reflections $(s_i)_{i \in I}$, as a family.

   EXAMPLES:

```
sage: W = CoxeterGroups().example()
sage: W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: s
Finite family {0: (1, 0, 2, 3), 1: (0, 2, 1, 3), 2: (0, 1, 3, 2)}
sage: s[0]
(1, 0, 2, 3)
sage: s[1]
(0, 2, 1, 3)
sage: s[2]
(0, 1, 3, 2)
```

   This default implementation uses index_set() and simple_reflection().

some_elements()

   Implements Sets.ParentMethods.some_elements() by returning some typical element of
   self.

   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.order()
24
```

weak_order_ideal(*predicate*, *side='right'*, *category=None*)

Returns 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()
7
sage: list(I)
[(), (1,), (1, 2), (1, 2, 1), (2,), (2, 1), (2, 1, 2)]
```

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]  [ 3 -2]  [ 1  0]  [-1  2]
[0 1], [ 0  1], [ 2 -1], [ 2 -1], [-2  3]
]
```

Even when the result is finite, some features of `FiniteEnumeratedSets` are not available:

```
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]  [ 3 -2]  [ 1  0]  [-1  2]
[0 1], [ 0  1], [ 2 -1], [ 2 -1], [-2  3]
]
```

Background

The weak order is returned as a `SearchForest`. This is achieved by assigning to each element $u1$ of the ideal a single ancestor $u = u1s_i$, 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).

CoxeterGroups.**super_categories**()

EXAMPLES:

```
sage: CoxeterGroups().super_categories()
[Category of finitely generated groups]
```

13.27 Crystals

class sage.categories.crystals.**Crystals**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of crystals.

See `sage.combinat.crystals.crystals` for an introduction to crystals.

EXAMPLES:

```
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 $\varepsilon_i(x)$)

- `x.phi(i)` (returning $\varphi_i(x)$)

EXAMPLES:

```
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(Crystals().element_class)
{'optional': [], 'required': ['e', 'epsilon', 'f', 'phi', 'weight']}
```

TESTS:

```
sage: TestSuite(C).run()
sage: B = Crystals().example()
sage: TestSuite(B).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_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

class **ElementMethods**

    Epsilon()
        EXAMPLES:
```
sage: C = crystals.Letters(['A',5])
sage: C(0).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(1).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(2).Epsilon()
(1, 0, 0, 0, 0, 0)
```

    Phi()
        EXAMPLES:
```
sage: C = crystals.Letters(['A',5])
sage: C(0).Phi()
(0, 0, 0, 0, 0, 0)
sage: C(1).Phi()
(1, 0, 0, 0, 0, 0)
sage: C(2).Phi()
(1, 1, 0, 0, 0, 0)
```

    all_paths_to_highest_weight(*index_set=None*)
        Iterate over all paths to the highest weight from `self` with respect to $index_set$.

        INPUT:
            •index_set – (optional) a subset of the index set of `self`
        EXAMPLES:
```
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])
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*)
    Returns $e_i(x)$ if it exists or `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 $e_{i_r}...e_{i_1}$ to self for $list = [i_1, ..., i_r]$

    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
```

f(*i*)
    Returns $f_i(x)$ if it exists or `None` otherwise.

    This method should be implemented by the element class of the crystal.

    EXAMPLES:
```
sage: C = Crystals().example(5)
sage: x = C[1]; x
2
sage: x.f(1), x.f(2), x.f(3)
(None, 3, None)
```

f_string(*list*)
    Applies $f_{i_r}...f_{i_1}$ to self for $list = [i_1, ..., i_r]$

EXAMPLES:
```
sage: C = crystals.Letters(['A',3])
sage: b = C(1)
sage: b.f_string([1,2])
3
sage: b.f_string([2,1])
```

index_set()
 EXAMPLES:
```
sage: C = crystals.Letters(['A',5])
sage: C(1).index_set()
(1, 2, 3, 4, 5)
```

is_highest_weight(*index_set=None*)
 Returns `True` if `self` is a highest weight. Specifying the option `index_set` to be a subset I of the index set of the underlying crystal, finds all highest weight vectors for arrows in I.

 EXAMPLES:
```
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
```

is_lowest_weight(*index_set=None*)
 Returns `True` if `self` is a lowest weight. Specifying the option `index_set` to be a subset I of the index set of the underlying crystal, finds all lowest weight vectors for arrows in I.

 EXAMPLES:
```
sage: C = crystals.Letters(['A',5])
sage: C(1).is_lowest_weight()
False
sage: C(6).is_lowest_weight()
True
sage: C(4).is_lowest_weight(index_set = [1,3])
True
```

phi(*i*)
 EXAMPLES:
```
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:
```
sage: C = crystals.Letters(['A',5])
sage: C(1).phi_minus_epsilon(1)
1
```

s(*i*)

Returns the reflection of `self` along its i-string

EXAMPLES:
```
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'*)
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

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]]]
```

to_highest_weight (*index_set=None*)
Return the highest weight element u and a list $[i_1, ..., i_k]$ such that $self = f_{i_1}...f_{i_k}u$, 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 crystals!
```

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 crystals!
```

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)
```

Crystals.**Finite**

alias of FiniteCrystals

class Crystals.**ParentMethods**

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: (
```

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

    EXAMPLES:
```
sage: C = crystals.Letters(['A',2])
sage: C.cartan_type()
['A', 2]
```

crystal_morphism(*g*, *index_set=None*, *automorphism=<function <lambda> at 0x7fabdc255c08>*, *direction='down'*, *direction_image='down'*, *similarity_factor=None*, *similarity_factor_domain=None*, *cached=False*, *acyclic=True*)
    Constructs a morphism from the crystal `self` to another crystal. The input g can either be a function of a (sub)set of elements of self to element in another crystal or a dictionary between certain elements. Usually one would map highest weight elements or crystal generators to each other using g. Specifying index_set gives the opportunity to define the morphism as I-crystals where $I =$ index_set. If index_set is not specified, the index set of self is used. It is also possible to define twisted-morphisms by specifying an automorphism on the nodes in te Dynkin diagram (or the index_set). The option direction and direction_image indicate whether to use f_i or e_i in self or the image crystal to construct the morphism, depending on whether the direction is set to 'down' or 'up'. It is also possible to set a similarity_factor. This should be a dictionary between the elements in the index set and positive integers. The crystal operator f_i then gets mapped to $f_i^{m_i}$ where $m_i =$ similarity_factor[i]. Setting similarity_factor_domain to a dictionary between the index set and positive integers has the effect that $f_i^{m_i}$ gets mapped to f_i where $m_i =$ similarity_factor_domain[i]. Finally, it is possible to set the option $acyclic = False$. This calculates an isomorphism for cyclic crystals (for example finite affine crystals). In this case the input function g is supposed to be given as a dictionary.

    EXAMPLES:
```
sage: C2 = crystals.Letters(['A',2])
sage: C3 = crystals.Letters(['A',3])
sage: g = {C2.module_generators[0] : C3.module_generators[0]}
sage: g_full = C2.crystal_morphism(g)
sage: g_full(C2(1))
1
sage: g_full(C2(2))
2
sage: g = {C2(1) : C2(3)}
sage: g_full = C2.crystal_morphism(g, automorphism = lambda i : 3-i, direction_image = '
sage: [g_full(b) for b in C2]
[3, 2, 1]
sage: T = crystals.Tableaux(['A',2], shape = [2])
sage: g = {C2(1) : T(rows=[[1,1]])}
sage: g_full = C2.crystal_morphism(g, similarity_factor = {1:2, 2:2})
sage: [g_full(b) for b in C2]
[[[1, 1]], [[2, 2]], [[3, 3]]]
sage: g = {T(rows=[[1,1]]) : C2(1)}
sage: g_full = T.crystal_morphism(g, similarity_factor_domain = {1:2, 2:2})
sage: g_full(T(rows=[[2,2]]))
2

sage: B1 = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: B2 = crystals.KirillovReshetikhin(['A',2,1],1,2)
sage: T = crystals.TensorProduct(B1,B2)
sage: T1 = crystals.TensorProduct(B2,B1)
sage: La = T.weight_lattice_realization().fundamental_weights()
sage: t = [b for b in T if b.weight() == -3*La[0] + 3*La[1]][0]
sage: t1 = [b for b in T1 if b.weight() == -3*La[0] + 3*La[1]][0]
sage: g={t:t1}
```

```
sage: f=T.crystal_morphism(g,acyclic = False)
sage: [[b,f(b)] for b in T]
[[[[[1]], [[1, 1]]], [[[1, 1]], [[1]]]],
 [[[[1]], [[1, 2]]], [[[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]], [[2, 2]]], [[[2, 2]], [[2]]]],
 [[[[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]], [[2, 3]]], [[[3, 3]], [[2]]]],
 [[[[3]], [[3, 3]]], [[[3, 3]], [[3]]]]]
```

digraph(*subset=None*, *index_set=None*)

Returns the DiGraph associated to `self`.

INPUT:

- `subset` – (Optional) A subset of vertices for which the digraph should be constructed
- `index_set` – (Optional) The index set to draw arrows

EXAMPLES:

```
sage: C = Crystals().example(5)
sage: C.digraph()
Digraph on 6 vertices
```

The edges of the crystal graph are by default colored using blue for edge 1, red for edge 2, and green for edge 3:

```
sage: C = Crystals().example(3)
sage: G = C.digraph()
sage: view(G, pdflatex=True, tightpage=True)  # optional - dot2tex graphviz, not tested
```

One may also overwrite the colors:

```
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, pdflatex=True, tightpage=True)  # optional - dot2tex graphviz, not tested
```

Or one may add colors to yet unspecified edges:

```
sage: C = Crystals().example(4)
sage: G = C.digraph()
sage: C.cartan_type()._index_set_coloring[4]="purple"
sage: view(G, pdflatex=True, tightpage=True)  # optional - dot2tex graphviz, not tested
```

Here is an example of how to take the top part up to a given depth of an infinite dimensional crystal:

```
sage: C = CartanType(['C',2,1])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[0])
sage: S = T.subcrystal(max_depth=3)
sage: G = T.digraph(subset=S); G
Digraph on 5 vertices
sage: sorted(G.vertices(), key=str)
```

```
[(-Lambda[0] + 2*Lambda[1] - delta,),
 (1/2*Lambda[0] + Lambda[1] - Lambda[2] - 1/2*delta, -1/2*Lambda[0] + Lambda[1] - 1/2*de
 (1/2*Lambda[0] - Lambda[1] + Lambda[2] - 1/2*delta, -1/2*Lambda[0] + Lambda[1] - 1/2*de
 (Lambda[0] - 2*Lambda[1] + 2*Lambda[2] - delta,),
 (Lambda[0],)]
```

Here is a way to construct a picture of a Demazure crystal using the `subset` option:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = CombinatorialFreeModule(QQ,B)
sage: t = B.highest_weight_vector()
sage: b = C(t)
sage: D = B.demazure_operator(b,[2,1]); D
B[[[1, 1], [2]]] + B[[[1, 2], [2]]] + B[[[1, 3], [2]]] + B[[[1, 1], [3]]] + B[[[1, 3], [
sage: G = B.digraph(subset=D.support())
sage: G.vertices()
[[[1, 1], [2]], [[1, 2], [2]], [[1, 3], [2]], [[1, 1], [3]], [[1, 3], [3]]]
sage: view(G, pdflatex=True, tightpage=True)  # optional - dot2tex graphviz, not tested
```

We can also choose to display particular arrows using the `index_set` option:

```
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: G = C.digraph(index_set=[1,3])
sage: len(G.edges())
20
sage: view(G, pdflatex=True, tightpage=True)  # optional - dot2tex graphviz, not tested
```

TODO: add more tests

dot_tex()
Returns a dot_tex string representation of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',2])
sage: C.dot_tex()
'digraph G { \n  node [ shape=plaintext ];\n  N_0 [ label = " ", texlbl = "$1$" ];\n  N_
```

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)
```

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_()  # optional - dot2tex graphviz
'...tikzpicture...'
sage: view(T, pdflatex = True, tightpage = True) # optional - dot2tex graphviz, not test
```

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"})   #optional - dot2tex g
'...tikzpicture...'
```

latex_file(*filename*)

    Exports 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: C.latex_file('/tmp/test.tex')   # optional - dot2tex graphviz
```

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

    a2 a3 a4 a1

    where c*a2 = a3 = 2*a4 =0 and a1=0, with c=2 for B2, c=1 for C2. Applying e(2) a1 times, e(1) a2 times, e(2) a3 times, e(1) a4 times returns to the highest weight. (Observe that Littelmann writes the roots in opposite of the usual order, so our e(1) is his e(2) for these Cartan types.) For type A2, the pattern has the form

    a3 a2 a1

    where applying e(1) a1 times, e(2) a2 times then e(3) a1 times returns to the highest weight. These data determine the vertex and may be translated into a Gelfand-Tsetlin pattern or tableau.

    EXAMPLES:
```
sage: C = crystals.Letters(['A', 2])
sage: C.metapost(tmp_filename())

sage: C = crystals.Letters(['A', 5])
sage: C.metapost(tmp_filename())
Traceback (most recent call last):
...
NotImplementedError
```

plot(***options*)

    Returns the plot of self as a directed graph.

    EXAMPLES:
```
sage: C = crystals.Letters(['A', 5])
sage: print(C.plot())
Graphics object consisting of 17 graphics primitives
```

plot3d(***options*)

    Returns the 3-dimensional plot of self as a directed graph.

    EXAMPLES:
```
sage: C = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: print(C.plot3d())
Graphics3d Object
```

subcrystal(*index_set=None*, *generators=None*, *max_depth=inf*, *direction='both'*)
Construct the subcrystal from `generators` using e_i and/or f_i for all i in `index_set`.

INPUT:
- `index_set` – (Default: `None`) The index set; if `None` then use the index set of the crystal
- `generators` – (Default: `None`) The list of generators; if `None` then use the module generators of the crystal
- `max_depth` – (Default: infinity) The maximum depth to build
- `direction` – (Default: `'both'`) The direction to build the subcrystal. It can be one of the following:
  - `'both'` - Using both e_i and f_i
  - `'upper'` - Using e_i
  - `'lower'` - Using f_i

EXAMPLES:
```
sage: C = crystals.KirillovReshetikhin(['A',3,1], 1, 2)
sage: S = list(C.subcrystal(index_set=[1,2])); S
[[[1, 1]], [[1, 2]], [[1, 3]], [[2, 2]], [[2, 3]], [[3, 3]]]
sage: C.cardinality()
10
sage: len(S)
6
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)]))
[[[1, 4]], [[2, 4]], [[1, 3]], [[2, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], max_depth=1))
[[[1, 4]], [[2, 4]], [[1, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], direction='upper'))
[[[1, 4]], [[1, 3]]]
sage: list(C.subcrystal(index_set=[1,3], generators=[C(1,4)], direction='lower'))
[[[1, 4]], [[2, 4]]]
```

tensor(**crystals*, ***options*)
Return the tensor product of `self` with the crystals `B`.

EXAMPLES:
```
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()
Returns the weight lattice realization used to express weights.

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.

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

class `Crystals.`**`SubcategoryMethods`**

   Methods for all subcategories.

   `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 `Crystals.`**`TensorProducts`**(*category*, **args*)

   Bases: `sage.categories.tensor.TensorProductsCategory`

   The category of crystals constructed by tensor product of crystals.

   `extra_super_categories``()`

      EXAMPLES:

```
sage: Crystals().TensorProducts().extra_super_categories()
[Category of crystals]
```

`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.
```

`Crystals.`**`super_categories`**`()`

   EXAMPLES:

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

13.28 Discrete Valuation Rings (DVR) and Fields (DVF)

class sage.categories.discrete_valuation.**DiscreteValuationFields**(*s=None*)

 Bases: sage.categories.category_singleton.Category_singleton

 The category of discrete valuation fields

 EXAMPLES:
```
sage: Qp(7) in DiscreteValuationFields()
True
sage: TestSuite(DiscreteValuationFields()).run()
```

 class ElementMethods

 valuation()
  Return the valuation of this element.

  EXAMPLES:
```
sage: x = Qp(5)(50)
sage: x.valuation()
2
```

 class DiscreteValuationFields.**ParentMethods**

 residue_field()
  Return the residue field of the ring of integers of this discrete valuation field.

  EXAMPLES:
```
sage: Qp(5).residue_field()
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()
5 + O(5^21)
```

 DiscreteValuationFields.**super_categories**()
  EXAMPLES:
```
sage: DiscreteValuationFields().super_categories()
[Category of fields]
```

class sage.categories.discrete_valuation.**DiscreteValuationRings**(*s=None*)

 Bases: sage.categories.category_singleton.Category_singleton

 The category of discrete valuation rings

 EXAMPLES:
```
sage: GF(7)[['x']] in DiscreteValuationRings()
True
sage: TestSuite(DiscreteValuationRings()).run()
```

class **ElementMethods**

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)
sage: x.is_unit()
False

sage: x = Zp(7)(50)
sage: x.is_unit()
True
```

lcm(*other*)

Return the least common multiple of self and other, normalized so that it is a power of the distinguished uniformizer.

valuation()

Return the valuation of this element.

EXAMPLES:
```
sage: x = Zp(5)(50)
sage: x.valuation()
2
```

class DiscreteValuationRings.**ParentMethods**

residue_field()

Return the residue field of this ring.

EXAMPLES:
```
sage: Zp(5).residue_field()
Finite Field of size 5

sage: K.<u> = QQ[[]]
sage: K.residue_field()
Rational Field
```

uniformizer()

Return a uniformizer of this ring.

EXAMPLES:
```
sage: Zp(5).uniformizer()
5 + O(5^21)

sage: K.<u> = QQ[[]]
sage: K.uniformizer()
u
```

DiscreteValuationRings.**super_categories**()

EXAMPLES:
```
sage: DiscreteValuationRings().super_categories()
[Category of principal ideal domains]
```

13.29 Distributive Magmas and Additive Magmas

class sage.categories.distributive_magmas_and_additive_magmas.**DistributiveMagmasAndAdditiveMa**

Bases: `sage.categories.category_with_axiom.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 DistributiveMagmasAndA
sage: C = DistributiveMagmasAndAdditiveMagmas(); C
Category of distributive magmas and additive magmas
sage: C.super_categories()
[Category of magmas and additive magmas]
```

TESTS:

```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: C is MagmasAndAdditiveMagmas().Distributive()
True
sage: C is (Magmas() & AdditiveMagmas()).Distributive()
True
sage: TestSuite(C).run()
```

class AdditiveAssociative(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class AdditiveCommutative(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class AdditiveUnital(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
```

```
                    <class 'sage.categories.finite_sets.FiniteSets_with_category'>
                    sage: type(C).__base__.__base__
                    <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

                    sage: TestSuite(C).run()
```

        class **Associative**(*base_category*)

          Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

          TESTS:

```
                    sage: C = Sets.Finite(); C
                    Category of finite sets
                    sage: type(C)
                    <class 'sage.categories.finite_sets.FiniteSets_with_category'>
                    sage: type(C).__base__.__base__
                    <class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

                    sage: TestSuite(C).run()
```

          AdditiveInverse
            alias of `Rngs`

          Unital
            alias of `Semirings`

class `DistributiveMagmasAndAdditiveMagmas.`**CartesianProducts**(*category*, **args*)

    Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    extra_super_categories()
        Implement the fact that a cartesian product of magmas distributing over additive magmas is a magma
        distributing over an additive magma.

        EXAMPLES:

```
sage: C = (Magmas() & AdditiveMagmas()).Distributive().CartesianProducts()
sage: C.extra_super_categories();
[Category of distributive magmas and additive magmas]
sage: C.axioms()
frozenset({'Distributive'})
```

class `DistributiveMagmasAndAdditiveMagmas.`**ParentMethods**

13.30 Division rings

class sage.categories.division_rings.**DivisionRings**(*base_category*)

    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

    The category of division rings

    A division ring (or skew field) is a not necessarily commutative ring where all non-zero elements have multiplicative inverses

    EXAMPLES:

```
sage: DivisionRings()
Category of division rings
sage: DivisionRings().super_categories()
[Category of domains]
```

    TESTS:

```
sage: TestSuite(DivisionRings()).run()
```

    Commutative

        alias of Fields

    class **ElementMethods**

    DivisionRings.**Finite_extra_super_categories**()

        Return extraneous super categories for DivisionRings().Finite().

        EXAMPLES:

        Any field is a division ring:

```
sage: Fields().is_subcategory(DivisionRings())
True
```

        This methods specifies that, by Weddeburn theorem, the reciprocal holds in the finite case: a finite division ring is commutative and thus a field:

```
sage: DivisionRings().Finite_extra_super_categories()
(Category of commutative magmas,)
sage: DivisionRings().Finite()
Category of finite 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.

        TESTS:

```
sage: DivisionRings().Finite() is Fields().Finite()
True
```

        This works also for subcategories:

```
sage: class Foo(Category):
....:     def super_categories(self): return [DivisionRings()]
sage: Foo().Finite().is_subcategory(Fields())
True
```

    class DivisionRings.**ParentMethods**

DivisionRings.**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

13.31 Domains

class sage.categories.domains.**Domains**(*base_category*)
> Bases: sage.categories.category_with_axiom.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
> ```
>
> TESTS:
> ```
> sage: TestSuite(C).run()
> ```
>
> **Commutative**
>> alias of IntegralDomains
>
> **class ElementMethods**
>
> **class** Domains.**ParentMethods**
>
> Domains.**super_categories**()
>> EXAMPLES:
>> ```
>> sage: Domains().super_categories()
>> [Category of rings]
>> ```

13.32 Enumerated Sets

class sage.categories.enumerated_sets.**EnumeratedSets**(*s=None*)
> Bases: sage.categories.category_singleton.Category_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 element 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()`: the 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.unrank(n)`: the `n-th` element of the set when `n` is a sage `Integer`. This is the equivanlent 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: 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
```

TESTS:
```
sage: C = EnumeratedSets()
sage: TestSuite(C).run()
```

class **CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
```

```
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

first()
   Return the first element.

   EXAMPLES:
   ```
   sage: cartesian_product([ZZ]*10).first()
   (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
   ```

class EnumeratedSets.**ElementMethods**

rank()
   Returns the rank of `self` in its parent.

   See also `EnumeratedSets.ElementMethods.rank()`

   EXAMPLES:
   ```
   sage: F = FiniteSemigroups().example(('a','b','c'))
   sage: L = list(F); L
   ['a', 'c', 'ac', 'b', 'ba', 'bc', 'cb', 'ca', 'bca', 'ab', 'bac', 'cab', 'acb', 'cba', '.
   sage: L[7].rank()
   7
   ```

EnumeratedSets.**Finite**
   alias of `FiniteEnumeratedSets`

EnumeratedSets.**Infinite**
   alias of `InfiniteEnumeratedSets`

class EnumeratedSets.**ParentMethods**

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
```

TESTS:

```
sage: F.is_empty.__module__
'sage.categories.enumerated_sets'
```

list()

Returns an error since the cardinality of self is not known.

EXAMPLES:

```
sage: class broken(UniqueRepresentation, Parent):
...       def __init__(self):
...           Parent.__init__(self, category = EnumeratedSets())
...
sage: broken().list()
Traceback (most recent call last):
...
NotImplementedError: unknown cardinality
```

map(*f*, *name=None*)

Returns the image $\{f(x)|x \in \text{self}\}$ of this enumerated set by f, as an enumerated set.

f is supposed to be injective.

EXAMPLES:

```
sage: R = SymmetricGroup(3).map(attrcall('reduced_word')); R
Image of Symmetric group of order 3! as a permutation group by *.reduced_word()
sage: R.cardinality()
6
sage: R.list()
[[], [1], [2, 1], [1, 2], [2], [1, 2, 1]]
sage: [ r for r in R]
[[], [1], [2, 1], [1, 2], [2], [1, 2, 1]]
```

> **Warning:** If the function is not injective, then there may be repeated elements:
>
> ```
> sage: P = SymmetricGroup(3)
> sage: P.list()
> [(), (1,2), (1,2,3), (1,3,2), (2,3), (1,3)]
> sage: P.map(attrcall('length')).list()
> [0, 1, 2, 2, 1, 3]
> ```

> **Warning:** `MapCombinatorialClass` needs to be refactored to use categories:
>
> ```
> sage: R.category()                # todo: not implemented
> Category of enumerated sets
> sage: TestSuite(R).run(skip=['_test_an_element', '_test_category', '_test_some_elements
> ```

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`()
    Returns 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.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 `EnumeratedSets()` which can be used when the method `__iter__` is provided. Its complexity is $O(r)$, where r is the rank of `obj`. For infinite enumerated sets, this won't terminate when x is not in `self`

    EXAMPLES:
```
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`()
    Returns some elements in `self`.

    See `TestSuite` for a typical use case.

    This is a generic implementation from the category `EnumeratedSets()` which can be used when the method `__iter__` is provided. It returns an iterator for up to the first 100 elements of `self`

    EXAMPLES:
```
sage: C = FiniteEnumeratedSets().example()
sage: list(C.some_elements()) # indirect doctest
[1, 2, 3]
```

unrank(*r*)

The `r`-th element of `self`

`self.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 `EnumeratedSets()` which can be used when the method `__iter__` is provided. Its complexity is $O(r)$, where r is the rank of `obj`.

EXAMPLES:
```
sage: C = FiniteEnumeratedSets().example()
sage: C.unrank(2) # indirect doctest
3
sage: C._unrank_from_iterator(5)
Traceback (most recent call last):
...
ValueError: the value must be between 0 and 2 inclusive
```

`EnumeratedSets.`**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()
```

`EnumeratedSets.`**super_categories**()

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

13.33 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: `sage.categories.category_singleton.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]
```

TESTS:

```
sage: TestSuite(EuclideanDomains()).run()
```

class **ElementMethods**

euclidean_degree()
    Return the degree of this element as an element of a euclidean domain, i.e., for elements a, b the
    euclidean degree f satisfies the usual properties:

    1. if b is not zero, then there are elements q and r such that $a = bq + r$ with $r = 0$ or $f(r) < f(b)$
    2. if a, b are not zero, then $f(a) \leq f(ab)$

    Note: The name euclidean_degree was chosen because the euclidean function has different
    names in different contexts, e.g., absolute value for integers, degree for polynomials.

    OUTPUT:

    For non-zero elements, a natural number. For the zero element, this might raise an exception or
    produce some other output, depending on the implementation.

    EXAMPLES:
    ```
    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 [Coh1996].

    REFERENCES:

    EXAMPLES:
    ```
    sage: EuclideanDomains().ElementMethods().gcd(6,4)
    2
    ```

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

    EXAMPLES:
    ```
    sage: R.<x> = QQ[]
    sage: x.quo_rem(x)
    (1, 0)
    ```

class EuclideanDomains.**ParentMethods**

is_euclidean_domain()
    Return True, since this in an object of the category of Euclidean domains.

    EXAMPLES:

```
sage: Parent(QQ,category=EuclideanDomains()).is_euclidean_domain()
True
```

EuclideanDomains.**super_categories**()

    EXAMPLES:

```
sage: EuclideanDomains().super_categories()
[Category of principal ideal domains]
```

13.34 Fields

class sage.categories.fields.**Fields**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of (commutative) fields, i.e. commutative rings where all non-zero elements have multiplicative inverses

EXAMPLES:

```
sage: K = Fields()
sage: K
Category of fields
sage: Fields().super_categories()
[Category of euclidean domains, Category of division rings]

sage: K(IntegerRing())
Rational Field
sage: K(PolynomialRing(GF(3), 'x'))
Fraction Field of Univariate Polynomial Ring in x over
Finite Field of size 3
sage: K(RealField())
Real Field with 53 bits of precision
```

TESTS:

```
sage: TestSuite(Fields()).run()
```

class **ElementMethods**

    euclidean_degree()

        Return the degree of this element as an element of a 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
```

    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)
sage: K(2).gcd(K(1))
1
sage: K(0).gcd(K(0))
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:

```
sage: gcd(15.0,12.0)
3.00000000000000
```

But for others floating point numbers, the gcd is just 0.0 or 1.0:

```
sage: gcd(3.2, 2.18)
1.00000000000000
```

```
sage: gcd(0.0, 0.0)
0.000000000000000
```

AUTHOR:
- Simon King (2011-02) – trac ticket #10771
- Vincent Delecroix (2015) – trac ticket #17671

is_unit()

Returns True if self has a multiplicative inverse.

EXAMPLES:

```
sage: QQ(2).is_unit()
True
sage: QQ(0).is_unit()
False
```

lcm(*other*)

Least common multiple.

> **Note:** Since we are in a field and the least common multiple is only determined up to a unit, it is correct to either return zero or one. Note that fraction fields of unique factorization domains provide a more sophisticated lcm.

EXAMPLES:

```
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:

```
sage: lcm(15.0,12.0)
60.0000000000000
```

But for others floating point numbers, it is just 0.0 or 1.0:

```
sage: lcm(3.2, 2.18)
1.00000000000000
```

```
sage: lcm(0.0, 0.0)
0.000000000000000
```

AUTHOR:
- Simon King (2011-02) – trac ticket #10771
- Vincent Delecroix (2015) – trac ticket #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:
```
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 `self` such that `r = s * self + t * other`. Since the computations are done over a field, `r` is zero if `self` and `other` are zero, and one otherwise.

AUTHORS:
- Julian Rueth (2012-10-19): moved here from `sage.structure.element.FieldElement`

EXAMPLES:
```
sage: K = GF(5)
sage: K(2).xgcd(K(1))
(1, 3, 0)
sage: K(0).xgcd(K(4))
(1, 0, 4)
sage: K(1).xgcd(K(1))
(1, 1, 0)
sage: GF(5)(0).xgcd(GF(5)(0))
(0, 0, 0)
```

The 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 gcd on **Z**:
```
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)
```

`Fields.`**Finite**

alias of `FiniteFields`

class `Fields.`**ParentMethods**

fraction_field()

Returns the *fraction field* of `self`, which is `self`.

EXAMPLES:

```
sage: QQ.fraction_field() is QQ
True
```

is_field(*proof=True*)

Returns True as self is a field.

EXAMPLES:
```
sage: QQ.is_field()
True
sage: Parent(QQ,category=Fields()).is_field()
True
```

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()
True
sage: Z5 = GF(5); Z5
Finite Field of size 5
sage: Z5.is_integrally_closed()
True
```

is_perfect()

Return whether this field is perfect, i.e., its characteristic is $p = 0$ or every element has a p-th root.

EXAMPLES:
```
sage: QQ.is_perfect()
True
sage: GF(2).is_perfect()
True
sage: FunctionField(GF(2), 'x').is_perfect()
False
```

Fields.**extra_super_categories**()

EXAMPLES:
```
sage: Fields().extra_super_categories()
[Category of euclidean domains]
```

13.35 Finite Coxeter Groups

class sage.categories.finite_coxeter_groups.**FiniteCoxeterGroups**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

The category of finite Coxeter groups.

EXAMPLES:
```
sage: FiniteCoxeterGroups()
Category of finite coxeter groups
sage: FiniteCoxeterGroups().super_categories()
[Category of coxeter groups,
 Category of finite groups,
 Category of finite finitely generated semigroups]
```

```
sage: G = FiniteCoxeterGroups().example()
sage: G.cayley_graph(side = "right").plot()
Graphics object consisting of 40 graphics primitives
```

Here are some further examples:
```
sage: FiniteWeylGroups().example()
The symmetric group on {0, ..., 3}
```

```
sage: WeylGroup(["B", 3])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
```

Those other examples will eventually be also in this category:
```
sage: SymmetricGroup(4)
Symmetric group of order 4! as a permutation group
sage: DihedralGroup(5)
Dihedral group of order 10 as a permutation group
```

class **ElementMethods**

    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, 2, 3, 4, 5, 1], [4, 5, 4, 1, 2, 1], [4, 5, 3, 1, 2, 1], [4, 5, 2, 3, 1, 2],
[4, 5, 1, 2, 3, 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 $\{ws_i, u_1 s_i, u_2 s_i, ..., u_j s_i\}$, where the u_k are those covers of ws_i that have a descent at i.

    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()
[(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()
[((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()
[(1, 3), (3, 1)]
sage: D.edges()
[]
```

TESTS:
```
sage: W = WeylGroup(['B',4], prefix='s')
sage: w = W.from_reduced_word([1,2])
sage: w.coxeter_knuth_graph()
Traceback (most recent call last):
...
NotImplementedError: This has only been implemented in finite type A so far!
```

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)
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)}
```

TESTS:
```
sage: W = WeylGroup(['B',4], prefix='s')
sage: word = [1,2]
sage: w = W.from_reduced_word(word)
sage: w.coxeter_knuth_neighbor(word)
Traceback (most recent call last):
...
NotImplementedError: This has only been implemented in finite type A so far!
```

class FiniteCoxeterGroups.**ParentMethods**

Ambiguity resolution: the implementation of `some_elements` is preferable to that of `FiniteGroups`. The same holds for `__iter__`, although a breath first search would be more natural; at least this maintains backward compatibility after trac ticket #13589.

TESTS:
```
sage: W = FiniteCoxeterGroups().example(3)
```

```
sage: W.some_elements.__module__
'sage.categories.coxeter_groups'
sage: W.__iter__.__module__
'sage.categories.coxeter_groups'
```

```
sage: W.some_elements()
[(1,), (2,), (), (1, 2)]
sage: list(W)
[(), (1,), (1, 2), (1, 2, 1), (2,), (2, 1)]
```

bruhat_poset (*facade=False*)

Returns the Bruhat poset of `self`.

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)
```

TESTS:
```
sage: [len(WeylGroup(["A", n]).bruhat_poset().cover_relations()) for n in [1,2,3]]
[1, 8, 58]
```

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.

long_element(*index_set=None*)
> INPUT:
>> • `index_set` - a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
>
> Returns the longest element of `self`, or of the parabolic subgroup corresponding to the given `index_set`.
>
> 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)
> ```

some_elements()
> Implements `Sets.ParentMethods.some_elements()` by returning some typical element of *self*.
>
> 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.order()
> 24
> ```

w0()
> Return the longest element of `self`.
>
> This attribute is deprecated.
>
> EXAMPLES:
> ```
> sage: D8 = FiniteCoxeterGroups().example(8)
> sage: D8.w0
> (1, 2, 1, 2, 1, 2, 1, 2)
> sage: D3 = FiniteCoxeterGroups().example(3)
> sage: D3.w0
> (1, 2, 1)
> ```

weak_lattice(*side='right'*, *facade=False*)
> INPUT:
>> • `side` – "left", "right", or "twosided" (default: "right")
>> • `facade` – a boolean (default: `False`)
>
> Returns the left (resp. right) poset for weak order. In this poset, u is smaller than v if some reduced word of u is a right (resp. left) factor of some reduced word of v.

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())
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:

```
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)
```

TESTS:

```
sage: [len(WeylGroup(["A", n]).weak_poset(side = "right").cover_relations()) for n in [1
[1, 6, 36]
sage: [len(WeylGroup(["A", n]).weak_poset(side = "left" ).cover_relations()) for n in [1
[1, 6, 36]
```

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.

---

weak_poset (*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.

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())
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:

```
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)
```

TESTS:

```
sage: [len(WeylGroup(["A", n]).weak_poset(side = "right").cover_relations()) for n in [1
[1, 6, 36]
sage: [len(WeylGroup(["A", n]).weak_poset(side = "left" ).cover_relations()) for n in [1
[1, 6, 36]
```

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.

13.36 Finite Crystals

class sage.categories.finite_crystals.**FiniteCrystals**(*base_category*)

    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

    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
```

TESTS:
```
sage: TestSuite(C).run()
sage: B = FiniteCrystals().example()
sage: TestSuite(B).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_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

class **TensorProducts**(*category*, **args*)
    Bases: `sage.categories.tensor.TensorProductsCategory`

    The category of finite crystals constructed by tensor product of finite crystals.

    extra_super_categories()
        EXAMPLES:
```
sage: FiniteCrystals().TensorProducts().extra_super_categories()
[Category of finite crystals]
```

FiniteCrystals.**example**(*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
```

FiniteCrystals.**extra_super_categories**()
    EXAMPLES:

```
sage: FiniteCrystals().extra_super_categories()
[Category of finite enumerated sets]
```

13.37 Finite dimensional algebras with basis

REFERENCES:

class sage.categories.finite_dimensional_algebras_with_basis.**FiniteDimensionalAlgebrasWithBas**
Bases: sage.categories.category_with_axiom.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 modules 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
```

TESTS:
```
sage: TestSuite(C).run()
sage: C is Algebras(QQ).FiniteDimensional().WithBasis()
True
sage: C is Algebras(QQ).WithBasis().FiniteDimensional()
True
```

class ElementMethods

> **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)
> sage: a = QS3([2,1,3])
> sage: a.to_matrix(side='left')
> [0 0 1 0 0 0]
> [0 0 0 0 1 0]
> [1 0 0 0 0 0]
> [0 0 0 0 0 1]
> [0 1 0 0 0 0]
> [0 0 0 1 0 0]
> sage: a.to_matrix(side='right')
> [0 0 1 0 0 0]
> [0 0 0 1 0 0]
> [1 0 0 0 0 0]
> [0 1 0 0 0 0]
> ```

```
[0 0 0 0 0 1]
[0 0 0 0 1 0]
sage: a.to_matrix(base_ring=RDF, side="left")
[0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0]
[1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0 0.0]
```

AUTHORS: Mike Hansen, ...

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)
sage: a = QS3([2,1,3])
sage: a.to_matrix(side='left')
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 0 1 0 0]
sage: a.to_matrix(side='right')
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 0 0 0 1]
[0 0 0 0 1 0]
sage: a.to_matrix(base_ring=RDF, side="left")
[0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0]
[1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0 0.0]
```

AUTHORS: Mike Hansen, ...

class `FiniteDimensionalAlgebrasWithBasis.`**`ParentMethods`**

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 $(P_i^L)_{i \in I}$ and $(P_i^R)_{i \in I}$ 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 $S_i^* = \operatorname{top} P_i^L$ and $S_i = \operatorname{top} P_i^R$.

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;
- $C_{i,j} = \dim Hom_A(P_j^R, P_i^R)$;
- $C_{i,j}$ counts how many times S_j appears as composition factor of P_i^R;
- $C_{i,j} = \dim Hom_A(P_i^L, P_j^L)$;
- $C_{i,j}$ counts how many times S_i^* appears as composition factor of P_j^L.

In the commutative case, the Cartan invariant matrix is diagonal. In the context of solving systems of multivariate polynomial equations of dimension zero, A is the quotient of the polynomial ring by the ideal generated by the equations, the simple modules correspond to the roots, and the numbers $C_{i,i}$ give the multiplicities of those roots.

Note: For simplicity, the current implementation, assumes that the index set I is of the form $\{0, \ldots, n-1\}$. Better indexations will be possible in the future.

ALGORITHM:

The Cartan invariant matrix of A is computed from the dimension of the summands of its peirce decomposition.

See also:

- `peirce_decomposition()`
- `isotypic_projective_modules()`

EXAMPLES:

For a semisimple algebra, in particular for group algebras in chararacteristic zero, the Cartan invariants matrix is the identity:

```
sage: A3 = SymmetricGroup(3).algebra(QQ)
sage: A3.cartan_invariants_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

For the path algebra of a quiver, the Cartan invariants matrix counts the number of paths between two vertices:

```
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()
sage: A.cartan_invariants_matrix()
[1 2]
[0 1]
```

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)
sage: A.cartan_invariants_matrix()
[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
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
sage: A.cartan_invariants_matrix()
[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
sage: center = A.center(); center
Center of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: center in Algebras(QQ).WithBasis().FiniteDimensional().Commutative()
True
sage: center.dimension()
1
sage: center.basis()
Finite family {0: B[0]}
sage: center.ambient() is A
True
sage: [c.lift() for c in center.basis()]
[x + y]
```

The center of a semisimple algebra is semisimple:

```
sage: DihedralGroup(6).algebra(QQ).center() in Algebras(QQ).Semisimple()
True
```

Todo

- Pickling by construction, as `A.center()`?
- Lazy evaluation of `_repr_`

TESTS:

```
sage: TestSuite(center).run()
```

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 π be the projection $A \to \overline{A}$ on its semisimple quotient. Let $\overline{x}$ 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 $\overline{x}$ of the semisimple quotient of A. Alternatively one may give as input the idempotent $\overline{x}$, 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 [CR62] 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_identity_decomposition_into_orthogonal_idempotents(*l*)

Return whether `l` is a decomposition of the identity into orthogonal idempotents.

INPUT:

- `l` – 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])
True
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x+a, y-a])
True
```

Here the idempotents do not sum up to 1:
```
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x])
False
```

Here $1 + x$ and $-x$ are neither idempotent nor orthogonal:
```
sage: A.is_identity_decomposition_into_orthogonal_idempotents([1+x,-x])
False
```

With the algebra of the 0-Hecke monoid:
```
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
sage: idempotents = A.orthogonal_idempotents_central_mod_radical()
sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

Todo

Add examples of elements that are orthogonal and sum to the identity yet are not idempotent, and reciprocally.

---

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:
```
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`:
```
sage: sum([Qi.dimension() for Qi in Q]) == A.dimension()
True
```

See also:

- orthogonal_idempotents_central_mod_radical()
- peirce_decomposition()

---

orthogonal_idempotents_central_mod_radical()

> Return a family of orthogonal idempotents of `self` that project on the central orthogonal idempotents of the semisimple quotient.
>
> OUTPUT:
>
> > • a list of orthogonal idempotents obtained by lifting the central orthogonal idempotents of the semisimple quotient.
>
> ALGORITHM:
>
> The orthogonal idempotents of A are obtained by lifting the central orthogonal idempotents of the semisimple quotient $\overline{A}$.
>
> Namely, let $(\overline{f_i})$ be the central orthogonal idempotents of the semisimple quotient of A. We recursively construct orthogonal idempotents of A by the following procedure: assuming $(f_i)_{i<n}$ is a set of already constructed orthogonal idempotent, we construct f_k by idempotent lifting of $(1-f)g(1-f)$, where g is any lift of $\overline{e_k}$ and $f = \sum_{i<k} f_i$.
>
> See [CR62] for correctness and termination proofs.
>
> **See also:**
>
> > • Algebras.SemiSimple.FiniteDimensional.WithBasis.ParentMethods.central_orthogo
> > • idempotent_lift()
>
> 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.orthogonal_idempotents_central_mod_radical()
> [x, y]
> ```
>
> ```
> sage: Z12 = Monoids().Finite().example(); Z12
> An example of a finite multiplicative monoid: the integers modulo 12
> sage: A = Z12.algebra(QQ)
> sage: idempotents = A.orthogonal_idempotents_central_mod_radical()
> sage: sorted(idempotents, key=str)
> [-1/2*B[3] + 1/2*B[9],
>  -1/2*B[8] + 1/2*B[4],
>  -B[0] + 1/2*B[3] + 1/2*B[9],
>  -B[0] + 1/2*B[4] + 1/2*B[8],
>  1/4*B[1] + 1/2*B[3] + 1/4*B[5] - 1/4*B[7] - 1/2*B[9] - 1/4*B[11],
>  1/4*B[1] + 1/4*B[11] - 1/4*B[5] - 1/4*B[7],
>  1/4*B[1] - 1/2*B[4] - 1/4*B[5] + 1/4*B[7] + 1/2*B[8] - 1/4*B[11],
>  B[0],
>  B[0] + 1/4*B[1] - 1/2*B[3] - 1/2*B[4] + 1/4*B[5] + 1/4*B[7] - 1/2*B[8] - 1/2*B[9] + 1/4
> sage: sum(idempotents) == 1
> True
> sage: all(e*e == e for e in idempotents)
> True
> sage: all(e*f == 0 and f*e == 0 for e in idempotents for f in idempotents if e != f)
> True
> ```
>
> This is best tested with:
>
> ```
> sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
> True
> ```
>
> We construct orthogonal idempotents for the algebra of the 0-Hecke monoid:
>
> ```
> sage: from sage.monoids.hecke_monoid import HeckeMonoid
> sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
> ```

```
sage: idempotents = A.orthogonal_idempotents_central_mod_radical()
sage: A.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

peirce_decomposition(*idempotents=None*, *check=True*)

Return a Peirce decomposition of `self`.

Let $(e_i)_i$ 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 $(S_i)_i$ are the simple modules of A and $C_{i,j}$ is the Cartan invariants matrix.

INPUT:

- `idempotents` – a list of orthogonal idempotents $(e_i)_{i=0,\dots,n}$ of the algebra that sum to 1 (default: the idempotents returned by `orthogonal_idempotents_central_mod_radical()`)
- `check` – (default:True) whether to check that the idempotents are indeed orthogonal

OUTPUT:

A list of lists l such that `l[i][j]` is the subspace $e_i A e_j$.

See also:

- `orthogonal_idempotents_central_mod_radical()`
- `cartan_invariants_matrix()`

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.orthogonal_idempotents_central_mod_radical()
[x, y]
sage: decomposition = A.peirce_decomposition(); decomposition
[[Free module generated by {0} over Rational Field,
  Free module generated by {0, 1} over Rational Field],
 [Free module generated by {} over Rational Field,
  Free module generated by {0} over Rational Field]]
sage: [ [[x.lift() for x in decomposition[i][j].basis()]
....:     for j in range(2)]
....:    for i in range(2)]
[[[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)
sage: decomposition = A.peirce_decomposition()      # long time
sage: [[decomposition[i][j].dimension()             # long time (4s)
....:    for j in range(len(decomposition))]
....:  for i in range(len(decomposition))]
[[1, 0, 0, 0, 0],
 [0, 9, 0, 0, 0],
 [0, 0, 4, 0, 0],
 [0, 0, 0, 9, 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)]
[1, 3, 2, 3, 1]
```

peirce_summand(*ei*, *ej*)

Return the Peirce decomposition summand $e_i A e_j$.

INPUT:

- `self` – an algebra A
- `ei`, `ej` – 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()
sage: A.peirce_summand(idemp[0], idemp[1])
Free module generated by {0, 1} over Rational Field
sage: A.peirce_summand(idemp[1], idemp[0])
Free module generated by {} over Rational Field
```

We recover the 2×2 block of $\mathbf{Q}[S_4]$ corresponding to the unique simple module of dimension 2 of the symmetric group S_4:

```
sage: A4 = SymmetricGroup(4).algebra(QQ)
sage: e = A4.central_orthogonal_idempotents()[2]
sage: A4.peirce_summand(e, e)
Free module generated by {0, 1, 2, 3} over Rational Field
```

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:

```
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: x, y, a, b = A.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 left ideal generated by x is one-dimensional:

```
sage: Ax = A.principal_ideal(x, side='left'); Ax
Free module generated by {0} over Rational Field
sage: [B.lift() for B in Ax.basis()]
[x]
```

Multiplication on the left by x annihilates only x and fixes the other basis elements:

```
sage: x*x, x*y, x*a, x*b
(x, 0, a, b)
```

so the right ideal generated by x is 3-dimensional:

```
sage: xA = A.principal_ideal(x, side='right'); xA
Free module generated by {0, 1, 2} over Rational Field
sage: [B.lift() for B in xA.basis()]
[x, a, b]
```

See also:

- peirce_summand()

radical()

Return the Jacobson radical of `self`.

This uses radical_basis(), whose default implementation handles algebras over fields of characteristic zero or fields of characteristic p in which we can compute $x^{1/p}$.

See also:

radical_basis(), 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]}
sage: radical.ambient() is A
True
sage: [c.lift() for c in radical.basis()]
[a, b]
```

Todo
- Tell Sage that the radical is in fact an ideal;
- Pickling by construction, as `A.center()`;
- Lazy evaluation of `_repr_`.

TESTS:

```
sage: TestSuite(radical).run()
```

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 $x^{1/p}$ [FR85], [Eb89].

REFERENCES:

[Eb89] Eberly, Wayne. "Computations for algebras and group representations." Ph.D. Thesis, University of Toronto, 1989.

[FR85] Friedl, Katalin, and Lajos Rónyai. "Polynomial time solutions of some problems of computational algebra." Proceedings of the seventeenth annual ACM symposium on Theory of computing. ACM, 1985.

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: A = KleinFourGroup().algebra(QQ)
```

This algebra belongs to the category of finite dimensional algebras over the rationals:
```
sage: A in Algebras(QQ).FiniteDimensional().WithBasis()
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: A in Algebras(QQ).Semisimple()
True
sage: A.radical_basis()
[]
```

Let's work instead over a field of characteristic 2:
```
sage: A = KleinFourGroup().algebra(GF(2))
sage: A in Algebras(GF(2)).Semisimple()
False
sage: A.radical_basis()
[B[()] + B[(1,2)(3,4)], B[(3,4)] + B[(1,2)(3,4)], B[(1,2)] + B[(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: class AnAlgebra(CombinatorialFreeModule):
....:     def __init__(self, F):
....:         R.<x> = PolynomialRing(F)
....:         I = R.ideal(x**F.characteristic()-F.one())
....:         self._xbar = R.quotient(I).gen()
....:         basis_keys = [self._xbar**i for i in range(F.characteristic())]
....:         CombinatorialFreeModule.__init__(self, F, basis_keys,
....:             category=Algebras(F).FiniteDimensional().WithBasis())
....:     def one(self):
....:         return self.basis()[self.base_ring().one()]
....:     def product_on_basis(self, w1, w2):
....:         return self.from_vector(vector(w1*w2))
```

```
sage: AnAlgebra(GF(3)).radical_basis()
[B[1] + 2*B[xbar^2], B[xbar] + 2*B[xbar^2]]
sage: AnAlgebra(GF(16,'a')).radical_basis()
[B[1] + B[xbar]]
sage: AnAlgebra(GF(49,'a')).radical_basis()
[B[1] + 6*B[xbar^6], B[xbar] + 6*B[xbar^6], B[xbar^2] + 6*B[xbar^6],
 B[xbar^3] + 6*B[xbar^6], B[xbar^4] + 6*B[xbar^6], B[xbar^5] + 6*B[xbar^6]]
```

TESTS:
```
sage: A = KleinFourGroup().algebra(GF(2))
sage: A.radical_basis()
[B[()] + B[(1,2)(3,4)], B[(3,4)] + B[(1,2)(3,4)], B[(1,2)] + B[(1,2)(3,4)]]
```

```
sage: A = KleinFourGroup().algebra(QQ, category=Monoids())
sage: A.radical_basis.__module__
'sage.categories.finite_dimensional_algebras_with_basis'
sage: A.radical_basis()
[]
```

semisimple_quotient()
    Return the semisimple quotient of `self`.

    This is the quotient of `self` by its radical.

    See also:

    radical()

    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,b,x,y = sorted(A.basis())
sage: S = A.semisimple_quotient(); S
Semisimple quotient of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: S in Algebras(QQ).Semisimple()
True
sage: S.basis()
Finite family {'y': B['y'], 'x': B['x']}
sage: xs,ys = sorted(S.basis())
sage: (xs + ys) * xs
B['x']
```

    Sanity check: the semisimple quotient of the n-th descent algebra of the symmetric group is of dimension the number of partitions of n:
```
sage: [ DescentAlgebra(QQ,n).B().semisimple_quotient().dimension()
....:   for n in range(6) ]
[1, 1, 2, 3, 5, 7]
sage: [Partitions(n).cardinality() for n in range(10)]
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

Todo
    •Pickling by construction, as `A.semisimple_quotient()`?
    •Lazy evaluation of `_repr_`

TESTS:

```
sage: TestSuite(S).run()
```

13.38 Finite dimensional bialgebras with basis

sage.categories.finite_dimensional_bialgebras_with_basis.**FiniteDimensionalBialgebrasWithBas**
    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 over Rational Field,
 Category of coalgebras with basis over Rational Field,
 Category of finite dimensional algebras with basis over Rational Field]
sage: C is Bialgebras(QQ).WithBasis().FiniteDimensional()
True
```

    TESTS:
```
sage: TestSuite(C).run()
```

13.39 Finite dimensional coalgebras with basis

sage.categories.finite_dimensional_coalgebras_with_basis.**FiniteDimensionalCoalgebrasWithBas**
    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 modules with basis over Rational Field]
sage: C is Coalgebras(QQ).WithBasis().FiniteDimensional()
True
```

    TESTS:
```
sage: TestSuite(C).run()
```

13.40 Finite dimensional Hopf algebras with basis

class sage.categories.finite_dimensional_hopf_algebras_with_basis.**FiniteDimensionalHopfAlgeb**
    Bases: `sage.categories.category_with_axiom.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()
```

```
[Category of hopf algebras with basis over Rational Field,
 Category of finite dimensional algebras with basis over Rational Field]
```

TESTS:
```
sage: TestSuite(FiniteDimensionalHopfAlgebrasWithBasis(ZZ)).run()
```

class **ElementMethods**

class FiniteDimensionalHopfAlgebrasWithBasis.**ParentMethods**

13.41 Finite dimensional modules with basis

class sage.categories.finite_dimensional_modules_with_basis.**FiniteDimensionalModulesWithBasis**

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of finite dimensional modules with a distinguished basis

EXAMPLES:
```
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
```

TESTS:
```
sage: TestSuite(C).run()
```

class **ElementMethods**

class FiniteDimensionalModulesWithBasis.**MorphismMethods**

> **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()
>> sage: Y = CombinatorialFreeModule(ZZ, [3,4]); y = Y.basis()
>> sage: phi = X.module_morphism(on_basis = {1: y[3] + 3*y[4], 2: 2*y[3] + 5*y[4]}.__getitem
>> ...                           codomain = Y)
>> sage: phi.matrix()
>> [1 2]
>> ```

```
[3 5]
sage: phi.matrix(side="right")
[1 3]
[2 5]

sage: phi.matrix().parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: phi.matrix(QQ).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

The resulting matrix is immutable:
```
sage: phi.matrix().is_mutable()
False
```

The zero morphism has a zero matrix:
```
sage: Hom(X,Y).zero().matrix()
[0 0]
[0 0]
```

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()
sage: phi = X.module_morphism(on_basis = {1: y[3] + 3*y[4], 2: 2*y[3] + 5/2*y[4]}.__geti
...                            codomain = Y)
sage: phi.matrix().parent()            # todo: not implemented
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

This currently does not work because, in this case, the morphism is just in the category of commutative additive groups (i.e. the intersection of the categories of modules over **Z** and over **Q**):
```
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
```

---

class `FiniteDimensionalModulesWithBasis.`**`ParentMethods`**

`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.

EXAMPLES:
```
sage: F = FiniteDimensionalAlgebrasWithBasis(QQ).example(); F
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: x,y,a,b = F.basis()
sage: A = F.annihilator([a + 3*b + 2*y]); A
Free module generated by {0} over Rational Field
sage: [b.lift() for b in A.basis()]
[-1/2*a - 3/2*b + x]
```

The category can be used to specify other properties of this subspace, like that this is a subalgebra:
```
sage: center = F.annihilator(F.basis(), F.bracket,
....:                        category=Algebras(QQ).Subobjects())
sage: (e,) = center.basis()
sage: e.lift()
x + y
sage: e * e == e
True
```

Taking annihilator is order reversing for inclusion:
```
sage: A   = F.annihilator([]);    A  .rename("A")
sage: Ax  = F.annihilator([x]);   Ax .rename("Ax")
sage: Ay  = F.annihilator([y]);   Ay .rename("Ay")
sage: Axy = F.annihilator([x,y]); Axy.rename("Axy")
sage: P = Poset(([A, Ax, Ay, Axy], attrcall("is_submodule")))
sage: sorted(P.cover_relations(), key=str)
[[Ax, A], [Axy, Ax], [Axy, Ay], [Ay, A]]
```

annihilator_basis(*S*, *action=<built-in function mul>*, *side='right'*)
  Return a basis of the annihilator of a finite set of elements.

  INPUT:
   - `S` – a finite set of objects
   - `action` – a function (default: `operator.mul`)
   - `side` – 'left' or 'right' (default: 'right'): on which side of `self` the elements of S acts.

  See `annihilator()` for the assumptions and definition of the annihilator.

  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:
```
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()`.

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")
sage: x = X.basis()
sage: Y = X.quotient_module([x[0]-x[1], x[1]-x[2]], already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by {2} over Rational Field
sage: y = Y.basis()
sage: y[2]
y[2]
sage: y[2].lift()
x[2]
sage: Y.retract(x[0]+2*x[1])
3*y[2]
```

See also:

- Modules.WithBasis.ParentMethods.submodule()
- Rings.ParentMethods.quotient()
- sage.modules.with_basis.subquotient.QuotientModuleWithBasis

13.42 Finite dimensional semisimple algebras with basis

class sage.categories.finite_dimensional_semisimple_algebras_with_basis.**FiniteDimensionalSemi**
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of finite dimensional semisimple algebras with a distinguished basis

EXAMPLES:

```
sage: from sage.categories.finite_dimensional_semisimple_algebras_with_basis import FiniteDimens
sage: C = FiniteDimensionalSemisimpleAlgebrasWithBasis(QQ); C
Category of finite dimensional semisimple algebras with basis over Rational Field
```

This category is best constructed as:

```
sage: D = Algebras(QQ).Semisimple().FiniteDimensional().WithBasis(); D
Category of finite dimensional semisimple algebras with basis over Rational Field
sage: D is C
True
```

TESTS:

```
sage: TestSuite(C).run()
```

class Commutative(*base_category*)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
```

```
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class ParentMethods

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:
```
sage: A4 = SymmetricGroup(4).algebra(QQ)
sage: Z4 = A4.center()
sage: idempotents = Z4.central_orthogonal_idempotents()
sage: idempotents
[1/24*B[0] + 1/24*B[1] + 1/24*B[2] + 1/24*B[3] + 1/24*B[4],
 3/8*B[0] + 1/8*B[1] - 1/8*B[2] - 1/8*B[4],
 1/6*B[0] + 1/6*B[2] - 1/12*B[3],
 3/8*B[0] - 1/8*B[1] - 1/8*B[2] + 1/8*B[4],
 1/24*B[0] - 1/24*B[1] + 1/24*B[2] + 1/24*B[3] - 1/24*B[4]]
```

Lifting those idempotents from the center, we recognize among them the sum and alternating sum of all permutations:
```
sage: [e.lift() for e in idempotents]
[1/24*() + 1/24*(3,4) + 1/24*(2,3) + 1/24*(2,3,4) + 1/24*(2,4,3)
 + 1/24*(2,4) + 1/24*(1,2) + 1/24*(1,2)(3,4) + 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*(1,3,4,2) + 1/24*(1,3) + 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*(2,4) - 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*(1,3,4,2) - 1/24*(1,3) + 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:
```
sage: Z4.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

class FiniteDimensionalSemisimpleAlgebrasWithBasis.**ParentMethods**

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).

INPUT:

- `self` – a semisimple algebra.

EXAMPLES:

For the algebra of the symmetric group S_3, we recover the sum and alternating sum of all permutations, together with a third idempotent:

```
sage: A3 = SymmetricGroup(3).algebra(QQ)
sage: idempotents = A3.central_orthogonal_idempotents()
sage: idempotents
[1/6*() + 1/6*(2,3) + 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) + 1/6*(1,3),
 2/3*() - 1/3*(1,2,3) - 1/3*(1,3,2),
 1/6*() - 1/6*(2,3) - 1/6*(1,2) + 1/6*(1,2,3) + 1/6*(1,3,2) - 1/6*(1,3)]
sage: A3.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

For the semisimple quotient of a quiver algebra, we recover the vertices of the quiver:

```
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver (containing
the arrows a:x->y and b:x->y) over Rational Field
sage: Aquo = A.semisimple_quotient()
sage: Aquo.central_orthogonal_idempotents()
[B['x'], B['y']]
```

13.43 Finite Enumerated Sets

class sage.categories.finite_enumerated_sets.**FiniteEnumeratedSets**(*base_category*)

    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of finite enumerated sets

EXAMPLES:

```
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]
```

TESTS:

```
sage: C = FiniteEnumeratedSets()
sage: TestSuite(C).run()
sage: sorted(C.Algebras(QQ).super_categories(), key=str)
[Category of finite dimensional modules with basis over Rational Field,
 Category of set algebras over Rational Field]
```

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: `sage.categories.cartesian_product.CartesianProductsCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    class **ParentMethods**

        TESTS:

        Ideally, these tests should be just after the declaration of the associated attributes. But doing this way, Sage will not consider them as a doctest.

        We check that cartesian products of finite enumerated sets inherit various methods from *Sets.CartesianProducts* and not from `EnumeratedSets.Finite`:

```
sage: C = cartesian_product([Partitions(10), Permutations(20)])
sage: C in EnumeratedSets().Finite()
True

sage: C.random_element.__module__
'sage.categories.sets_cat'

sage: C.cardinality.__module__
'sage.categories.sets_cat'

sage: C.__iter__.__module__
'sage.categories.enumerated_sets'
```

        cardinality()

            Return the cardinality of self.

            EXAMPLES:

```
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E,SymmetricGroup(4)])
sage: C.cardinality()
72

sage: E = FiniteEnumeratedSet([])
sage: C = cartesian_product([E, ZZ, QQ])
sage: C.cardinality()
0

sage: C = cartesian_product([ZZ, QQ])
sage: C.cardinality()
+Infinity
```

```
sage: cartesian_product([GF(5), Permutations(10)]).cardinality()
18144000
sage: cartesian_product([GF(71)]*20).cardinality() == 71**20
True
```

last()
> Return the last element

> EXAMPLES:
```
sage: C = cartesian_product([Zmod(42), Partitions(10), IntegerRange(5)])
sage: C.last()
(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:
```
sage: C = cartesian_product([Permutations(10)]*5)
sage: C.random_element()              # random
([2, 9, 4, 7, 1, 8, 6, 10, 5, 3],
 [8, 6, 5, 7, 1, 4, 9, 3, 10, 2],
 [5, 10, 3, 8, 2, 9, 1, 4, 7, 6],
 [9, 6, 10, 3, 2, 1, 5, 8, 7, 4],
 [8, 5, 2, 9, 10, 3, 7, 1, 4, 6])

sage: C = cartesian_product([ZZ]*10)
sage: c1 = C.random_element()
sage: c1                         # random
(3, 1, 4, 1, 1, -3, 0, -4, -17, 2)
sage: c2 = C.random_element(4,7)
sage: c2                         # random
(6, 5, 6, 4, 5, 6, 6, 4, 5, 5)
sage: all(4 <= i < 7 for i in c2)
True
```

rank(*x*)
> Return the rank of an element of this cartesian product.

> The *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:
```
sage: C = cartesian_product([GF(2), GF(11), GF(7)])
sage: C.rank(C((1,2,5)))
96
sage: C.rank(C((0,0,0)))
0

sage: for c in C: print C.rank(c)
0
1
2
3
```

```
        4
        5
...
        150
        151
        152
        153

sage: F1 = FiniteEnumeratedSet('abcdefgh')
sage: F2 = IntegerRange(250)
sage: F3 = Partitions(20)
sage: C = cartesian_product([F1, F2, F3])
sage: c = C(('a', 86, [7,5,4,4]))
sage: C.rank(c)
54213
sage: C.unrank(54213)
('a', 86, [7, 5, 4, 4])
```

unrank(*i*)

Return the `i`-th element of this cartesian product.

INPUT:

- `i` – integer between `0` and `n-1` where `n` is the cardinality of this set.

See also:

- `EnumeratedSets.ParentMethods.unrank()`
- `rank()`

EXAMPLES:

```
sage: C = cartesian_product([GF(3), GF(11), GF(7), GF(5)])
sage: c = C.unrank(123); c
(0, 3, 3, 3)
sage: C.rank(c)
123

sage: c = C.unrank(857); c
(2, 2, 3, 2)
sage: C.rank(c)
857

sage: C.unrank(2500)
Traceback (most recent call last):
...
IndexError: index i (=2) is greater than the cardinality
```

`FiniteEnumeratedSets.CartesianProducts.`**extra_super_categories**()

A cartesian product of finite enumerated sets is a finite enumerated set.

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().CartesianProducts()
sage: C.extra_super_categories()
[Category of finite enumerated sets]
```

class `FiniteEnumeratedSets.`**IsomorphicObjects**(*category*, **args*)

Bases: `sage.categories.isomorphic_objects.IsomorphicObjectsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
```

```
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

cardinality()
> Returns the cardinality of `self` which is the same as that of the ambient set `self` is isomorphic
> to.
>
> EXAMPLES:
> ```
> sage: A = FiniteEnumeratedSets().IsomorphicObjects().example(); A
> The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
> sage: A.cardinality()
> 3
> ```

FiniteEnumeratedSets.IsomorphicObjects.**example**()
> Returns an example of isomorphic object of a finite enumerated set, as per `Category.example`.
>
> EXAMPLES:
> ```
> sage: FiniteEnumeratedSets().IsomorphicObjects().example()
> The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
> ```

class FiniteEnumeratedSets.**ParentMethods**

cardinality(**ignored_args*, ***ignored_kwds*)
> The cardinality of `self`.
>
> OUTPUT: an `Integer`
>
> 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
> ```
>
> This is the default implementation of `cardinality()` from the category
> `FiniteEnumeratedSet()`. To test this, we need a fresh example:
> ```
> sage: from sage.categories.examples.finite_enumerated_sets import Example
> sage: class FreshExample(Example): pass
> sage: C = FreshExample(); C.rename("FreshExample")
> sage: C.cardinality
> <bound method FreshExample_with_category._cardinality_from_iterator of FreshExample>
> ```
>
> TESTS:

This method shall return an `Integer`; we test this here, because `_test_enumerated_set_iter_cardinality()` does not do it for us:

```
sage: type(C._cardinality_from_iterator())
<type 'sage.rings.integer.Integer'>
```

We ignore additional inputs since during doctests classes which override `cardinality()` call up to the category rather than their own `cardinality()` method (see trac ticket #13688):

```
sage: C = FiniteEnumeratedSets().example()
sage: C._cardinality_from_iterator(algorithm='testing')
3
```

Here is a more complete example:

```
sage: class TestParent(Parent):
...       def __init__(self):
...           Parent.__init__(self, category=FiniteEnumeratedSets())
...       def __iter__(self):
...           yield 1
...           return
...       def cardinality(self, dummy_arg):
...           return 1 # we don't want to change the semantics of cardinality()
sage: P = TestParent()
sage: P.cardinality(-1)
1
sage: v = P.list(); v
[1]
sage: P.cardinality()
1
sage: P.cardinality('use alt algorithm') # Used to break here: see :trac:'13688'
1
sage: P.cardinality(dummy_arg='use alg algorithm') # Used to break here: see :trac:'1368
1
```

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:

```
sage: C = FiniteEnumeratedSets().example()
sage: C.last()
3
sage: C._last_from_iterator()
3
```

list()

The list of the elements of `self`.

This default implementation from the category `FiniteEnumeratedSet()` computes the list of the elements of `self` from the iterator of `self` and caches the result. It moreover overrides the following methods to use this cache:

- `self.cardinality()`
- `self.__iter__()` (but see below)
- `self.unrank()`

See also:

```
_list_from_iterator(),                                _cardinality_from_list(),
_iterator_from_list(), and _unrank_from_list()
```

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().example()
sage: C.list()
[1, 2, 3]
```

> **Warning:** The overriding of `self.__iter__` to use the cache is ignored upon calls such as `for x in C:` or `list(C)` (which essentially ruins its purpose). Indeed, Python looks up the `__iter__` method directly in the class of `C`, bypassing `C`'s dictionary (see the Python reference manual, Special method lookup for new-style classes)
>
> Let's take an example:
>
> ```
> sage: class Example(Parent):
> ...       def __init__(self):
> ...           Parent.__init__(self, category = FiniteEnumeratedSets())
> ...       def __iter__(self):
> ...           print "hello!"
> ...           for x in [1,2,3]: yield x
> sage: C = Example()
> sage: list(C)
> hello!
> hello!
> [1, 2, 3]
> sage: list(C)
> hello!
> [1, 2, 3]
> ```
>
> Note that `hello!` actually gets printed twice in the first call to `list(C)`. That's because of the current (dubious) implementation of `Parent.__len__()`. Let's call `list()`:
>
> ```
> sage: C.list()
> [1, 2, 3]
> ```
>
> Now we would want the original iterator of `C` not to be called anymore, but that's not the case:
>
> ```
> sage: list(C)
> hello!
> [1, 2, 3]
> ```

TESTS:

To test if the caching and overriding works, we need a fresh finite enumerated set example, because the caching mechanism has already been triggered:

```
sage: from sage.categories.examples.finite_enumerated_sets import Example
sage: class FreshExample(Example): pass
sage: C = FreshExample(); C.rename("FreshExample")
sage: C.list
<bound method FreshExample_with_category.list of FreshExample>
sage: C.unrank
<bound method FreshExample_with_category._unrank_from_iterator of FreshExample>
sage: C.cardinality
<bound method FreshExample_with_category._cardinality_from_iterator of FreshExample>

sage: l1 = C.list(); l1
```

```
[1, 2, 3]
sage: C.list
<bound method FreshExample_with_category.list of FreshExample>
sage: C.unrank
<bound method FreshExample_with_category._unrank_from_list of FreshExample>
sage: C.cardinality
<bound method FreshExample_with_category._cardinality_from_list of FreshExample>
sage: C.__iter__
<bound method FreshExample_with_category._iterator_from_list of FreshExample>
```

We finally check that nothing breaks before and after calling explicitly the method `.list()`:

```
sage: class FreshExample(Example): pass
sage: import __main__; __main__.FreshExample = FreshExample # Fake FreshExample being de
sage: C = FreshExample()
sage: TestSuite(C).run()
sage: C.list()
[1, 2, 3]
sage: TestSuite(C).run()
```

random_element()

A random element in `self`.

`self.random_element()` returns a random element in `self` with uniform probability.

This is the default implementation from the category `EnumeratedSet()` which uses the method `unrank`.

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().example()
sage: C.random_element()
1
sage: C._random_element_from_unrank()
2
```

TODO: implement _test_random which checks uniformness

13.44 Finite Fields

class sage.categories.finite_fields.**FiniteFields**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of finite fields.

EXAMPLES:

```
sage: K = FiniteFields(); K
Category of finite fields
```

A finite field is a finite monoid with the structure of a field; it is currently assumed to be enumerated:

```
sage: K.super_categories()
[Category of fields,
 Category of finite commutative rings,
 Category of finite enumerated sets]
```

Some examples of membership testing and coercion:

```
sage: FiniteField(17) in K
True
```

```
sage: RationalField() in K
False
sage: K(RationalField())
Traceback (most recent call last):
...
TypeError: unable to canonically associate a finite field to Rational Field
```

TESTS:
```
sage: K is Fields().Finite()
True
sage: TestSuite(K).run()
```

class ElementMethods

class FiniteFields.**ParentMethods**

FiniteFields.**extra_super_categories**()
> Any finite field is assumed to be endowed with an enumeration.

> TESTS:
> ```
> sage: Fields().Finite().extra_super_categories()
> [Category of finite enumerated sets]
> sage: FiniteFields().is_subcategory(FiniteEnumeratedSets())
> True
> ```

13.45 FiniteGroups

class sage.categories.finite_groups.**FiniteGroups**(*base_category*)
> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

> The category of finite (multiplicative) groups.

> EXAMPLES:
> ```
> sage: C = FiniteGroups(); C
> Category of finite groups
> sage: C.super_categories()
> [Category of finite monoids, Category of groups]
> sage: C.example()
> General Linear Group of degree 2 over Finite Field of size 3
> ```

> TESTS:
> ```
> sage: TestSuite(C).run()
> ```

> **class Algebras**(*category*, **args*)
> > Bases: `sage.categories.algebra_functor.AlgebrasCategory`

> > TESTS:
> > ```
> > sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> > sage: class FooBars(CovariantConstructionCategory):
> > ....:     _functor_category = "FooBars"
> > ....:     _base_category_class = (Category,)
> > sage: Category.FooBars = lambda self: FooBars.category_of(self)
> > sage: C = FooBars(ModulesWithBasis(ZZ))
> > sage: C
> > ```

```
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

extra_super_categories()

Implement Maschke's theorem.

In characteristic 0 all finite group algebras are semisimple.

EXAMPLES:

```
sage: FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
sage: FiniteGroups().Algebras(FiniteField(7)).is_subcategory(Algebras(QQ).Semisimple())
False
sage: FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False
```

class FiniteGroups.**ElementMethods**

class FiniteGroups.**ParentMethods**

cardinality()

Returns the cardinality of self, as per EnumeratedSets.ParentMethods.cardinality().

This default implementation calls order() if available, and otherwise resorts to _cardinality_from_iterator(). This is for backward compatibility only. Finite groups should override this method instead of order().

EXAMPLES:

We need to use a finite group which uses this default implementation of cardinality:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
sage: G.cardinality.__module__
'sage.categories.finite_groups'
sage: G.cardinality()
24
```

cayley_graph_disabled(*connecting_set=None*)

AUTHORS:

- Bobby Moretti (2007-08-10)
- Robert Miller (2008-05-01): editing

conjugacy_classes()

Return a list with all the conjugacy classes of the group.

This will eventually be a fall-back method for groups not defined over GAP. Right now just raises a NotImplementedError, until we include a non-GAP way of listing the conjugacy classes representatives.

EXAMPLES:

```
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 <type 'sage.groups.group.FiniteGroup'> is not implemented
```

conjugacy_classes_representatives()

Return a list of the conjugacy classes representatives of the group.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: G.conjugacy_classes_representatives()
[(), (1,2), (1,2,3)]
```

monoid_generators()

Return monoid generators for `self`.

For finite groups, the group generators are also monoid generators. Hence, this default implementation calls `group_generators()`.

EXAMPLES:

```
sage: A = AlternatingGroup(4)
sage: A.monoid_generators()
Family ((2,3,4), (1,2,3))
```

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:

```
sage: A = AlternatingGroup(4)
sage: A.semigroup_generators()
Family ((2,3,4), (1,2,3))
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: A = AlternatingGroup(4)
sage: A.some_elements()
Family ((2,3,4), (1,2,3))
```

FiniteGroups.**example**()

Return an example of finite group, as per `Category.example()`.

EXAMPLES:

```
sage: G = FiniteGroups().example(); G
General Linear Group of degree 2 over Finite Field of size 3
```

13.46 Finite lattice posets

class sage.categories.finite_lattice_posets.**FiniteLatticePosets**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

The category of finite lattices, i.e. finite partially ordered sets which are also lattices.

EXAMPLES:
```
sage: FiniteLatticePosets()
Category of finite lattice posets
sage: FiniteLatticePosets().super_categories()
[Category of lattice posets, Category of finite posets]
sage: FiniteLatticePosets().example()
NotImplemented
```

See also:

FinitePosets, LatticePosets, LatticePoset

TESTS:
```
sage: C = FiniteLatticePosets()
sage: C is FiniteLatticePosets().Finite()
True
sage: TestSuite(C).run()
```

class **ParentMethods**

    is_lattice_morphism(*f*, *codomain*)
        INPUT:
            •f – a function from `self` to `codomain`
            •codomain – a lattice
        Returns whether f is a morphism of posets form `self` to `codomain`, that is

$$x \leq y \Rightarrow f(x) \leq f(y)$$

        EXAMPLES:

        We build the boolean lattice of $\{2, 2, 3\}$ and the lattice of divisors of 60, and check that the map $b \mapsto 5 \prod_{x \in b} x$ is a morphism of lattices:
```
sage: D = LatticePoset((divisors(60), attrcall("divides")))
sage: B = LatticePoset((Subsets([2,2,3]), attrcall("issubset")))
sage: def f(b): return D(5*prod(b))
sage: B.is_lattice_morphism(f, D)
True
```

        We construct the boolean lattice B_2:
```
sage: B = Posets.BooleanLattice(2)
sage: B.cover_relations()
[[0, 1], [0, 2], [1, 3], [2, 3]]
```

        And the same lattice with new top and bottom elements numbered respectively -1 and 3:
```
sage: L = LatticePoset(DiGraph({-1:[0], 0:[1,2], 1:[3], 2:[3],3:[4]}))
sage: L.cover_relations()
[[-1, 0], [0, 1], [0, 2], [1, 3], [2, 3], [3, 4]]

sage: f = { B(0): L(0), B(1): L(1), B(2): L(2), B(3): L(3) }.__getitem__
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()
    Returns 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:

meet_irreducibles(), join_irreducibles_poset()

join_irreducibles_poset()
    Returns the poset of join-irreducible elements of this finite lattice.

    A *join-irreducible element* of self is an element x that is not minimal and can not be written as the join of two elements different from x.

    EXAMPLES:
```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.join_irreducibles_poset()
Finite poset containing 3 elements
```

See also:

join_irreducibles()

meet_irreducibles()
    Returns the meet-irreducible elements of this finite lattice.

    A *meet-irreducible element* of self is an element x that is not maximal and that can not be written as the meet of two elements different from x.

    EXAMPLES:
```
sage: L = LatticePoset({0:[1,2],1:[3],2:[3,4],3:[5],4:[5]})
sage: L.meet_irreducibles()
[1, 3, 4]
```

See also:

join_irreducibles(), meet_irreducibles_poset()

meet_irreducibles_poset()
    Returns 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 can not be written as the meet of two elements different from x.

    EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.join_irreducibles_poset()
Finite poset containing 3 elements
```

See also:

`meet_irreducibles()`

13.47 Finite Monoids

class sage.categories.finite_monoids.**FiniteMonoids**(*base_category*)

Bases: `sage.categories.category_with_axiom.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]
```

TESTS:

```
sage: TestSuite(FiniteMonoids()).run()
```

class **ElementMethods**

pseudo_order()

Returns the pair $[k, j]$ with k minimal and $0 \le j < k$ such that `self^k == self^j`.

Note that j is uniquely determined.

EXAMPLES:

```
sage: M = FiniteMonoids().example(); M
An example of a finite multiplicative monoid: the integers modulo 12

sage: x = M(2)
sage: [ x^i for i in range(7) ]
[1, 2, 4, 8, 4, 8, 4]
sage: x.pseudo_order()
[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.

13.48 Finite Permutation Groups

class sage.categories.finite_permutation_groups.**FinitePermutationGroups**(*base_category*)

Bases: `sage.categories.category_with_axiom.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 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
```

TESTS:

```
sage: C is FinitePermutationGroups()
True
sage: TestSuite(C).run()

sage: G = FinitePermutationGroups().example()
sage: TestSuite(G).run(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . 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_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_inverse() . . . 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 **ElementMethods**

class FinitePermutationGroups.**ParentMethods**

> **cycle_index**(*parent=None*)
>> 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)
>>
>> Returns the *cycle index* of G, which is a gadget counting the elements of G by cycle type, averaged over the group:
>>
>> $$P = \frac{1}{|G|} \sum_{g \in G} p_{\text{cycle type}(g)}$$
>>
>> EXAMPLES:
>>
>> Among the permutations of the symmetric group S_4, there is the identity, 6 cycles of length 2, 3 products of two cycles of length 2, 8 cycles of length 3, and 6 cycles of length 4:
>> ```
>> sage: S4 = SymmetricGroup(4)
>> sage: P = S4.cycle_index()
>> sage: 24 * P
>> p[1, 1, 1, 1] + 6*p[2, 1, 1] + 3*p[2, 2] + 8*p[3, 1] + 6*p[4]
>> ```
>>
>> 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]) ]
>> 8
>> ```
>>
>> The cycle index plays an important role in the enumeration of objects modulo the action of a group (Polya 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 Polya 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()
>> p[1, 1, 1, 1, 1, 1] + p[2, 2, 2] + 2*p[3, 3] + 2*p[6]
>>
>> sage: 60 * AlternatingGroup(5).cycle_index()
>> p[1, 1, 1, 1, 1] + 15*p[2, 2, 1] + 20*p[3, 1, 1] + 24*p[5]
>> ```

```
sage: for G in TransitiveGroups(5):                # optional - database_gap # 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]
p[1, 1, 1, 1, 1] + 15*p[2, 2, 1] + 20*p[3, 1, 1] + 24*p[5]
p[1, 1, 1, 1, 1] + 10*p[2, 1, 1, 1] + 15*p[2, 2, 1] + 20*p[3, 1, 1] + 20*p[3, 2] + 30*p[
```

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 have a `term` and `sum` method:
```
sage: CyclicPermutationGroup(6).cycle_index(parent = QQ)
Traceback (most recent call last):
  ...
AssertionError: 'parent' should be (or behave as) a free module with basis indexed by pa
```

REFERENCES:

AUTHORS:
  • Nicolas Borie and Nicolas M. Thiery

TESTS:
```
sage: P = PermutationGroup([]); P
Permutation Group with generators [()]
sage: P.cycle_index()
p[1]
sage: P = PermutationGroup([[(1)]]); P
Permutation Group with generators [()]
sage: P.cycle_index()
p[1]
```

FinitePermutationGroups.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
```

FinitePermutationGroups.extra_super_categories()
    Any permutation group is assumed to be endowed with a finite set of generators.

    TESTS:

    sage: PermutationGroups().Finite().extra_super_categories() [Category of finitely generated magmas]

13.49 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: sage.categories.category_with_axiom.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()

    TESTS:

```
sage: C = FinitePosets()
sage: C is Posets().Finite()
True
sage: TestSuite(C).run()
```

    class ParentMethods

        antichains()

            Return all antichains of self.

            EXAMPLES:

```
sage: A = Posets.PentagonPoset().antichains(); A
Set of antichains of Finite lattice containing 5 elements
sage: list(A)
[[], [0], [1], [1, 2], [1, 3], [2], [3], [4]]
```

        birational_free_labelling(*linear_extension=None*, *prefix='x'*, *base_field=None*, *reduced=False*, *addvars=None*)

            Return the birational free labelling of self.

            Let us hold back defining this, and introduce birational toggles and birational rowmotion first. These notions have been introduced in [EP13] as generalizations of the notions of toggles (order_ideal_toggle()) and rowmotion on order ideals of a finite poset. They have been studied further in [GR13].

            Let $\mathbf{K}$ be a field, and P be a finite poset. Let $\widehat{P}$ denote the poset obtained from P by adding a new element 1 which is greater than all existing elements of P, and a new element 0 which is smaller than all existing elements of P and 1. Now, a $\mathbf{K}$-*labelling of* P will mean any function from $\widehat{P}$ to $\mathbf{K}$. The image of an element v of $\widehat{P}$ under this labelling will be called the *label* of this labelling at v. The set of all $\mathbf{K}$-labellings of P is clearly $\mathbf{K}^{\widehat{P}}$.

            For any $v \in P$, we now define a rational map $T_v : \mathbf{K}^{\widehat{P}} \dashrightarrow \mathbf{K}^{\widehat{P}}$ as follows: For every $f \in \mathbf{K}^{\widehat{P}}$, the image $T_v f$ should send every element $u \in \widehat{P}$ distinct from v to $f(u)$ (so the labels at all $u \neq v$ don't change), while v is sent to

$$\frac{1}{f(v)} \cdot \frac{\sum_{u \lessdot v} f(u)}{\sum_{u \gtrdot v} \frac{1}{f(u)}}$$

(both sums are over all $u \in \widehat{P}$ satisfying the respectively given conditions). Here, $\lessdot$ and $\gtrdot$ mean (respectively) "covered by" and "covers", interpreted with respect to the poset $\widehat{P}$. This rational map T_v is an involution and is called the *(birational) v-toggle*; see `birational_toggle()` for its implementation.

Now, *birational rowmotion* is defined as the composition $T_{v_1} \circ T_{v_2} \circ \cdots \circ T_{v_n}$, where $(v_1, v_2, \ldots, v_n)$ is a linear extension of P (written as a linear ordering of the elements of P). This is a rational map $\mathbf{K}^{\widehat{P}} \dashrightarrow \mathbf{K}^{\widehat{P}}$ which does not depend on the choice of the linear extension; it is denoted by R. See `birational_rowmotion()` for its implementation.

The definitions of birational toggles and birational rowmotion extend to the case of $\mathbf{K}$ being any semifield rather than necessarily a field (although it becomes less clear what constitutes a rational map in this generality). The most useful case is that of the `tropical semiring`, in which case birational rowmotion relates to classical constructions such as promotion of rectangular semistandard Young tableaux (page 5 of [EP13b] and future work, via the related notion of birational *promotion*) and rowmotion on order ideals of the poset ([EP13]).

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 $\widehat{P}$ to a, and sending the element 1 of $\widehat{P}$ to b, where the ground field $\mathbf{K}$ is the field of rational functions in $n + 2$ indeterminates $a, x_1, x_2, \ldots, x_n, b$ over $\mathbb{Q}$.

In Sage, a labelling f of a poset P is encoded as a 4-tuple $(\mathbf{K}, d, u, v)$, where $\mathbf{K}$ is the ground field of the labelling (i. e., its target), d is the dictionary containing the values of f at the elements of P (the keys being the respective elements of P), u is the label of f at 0, and v is the label of f at 1.

> **Warning:** The dictionary d is labelled by the elements of P. If P is a poset with `facade` option set to `False`, these might not be what they seem to be! (For instance, if `P == Poset({1: [2, 3]}, facade=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 $\mathbf{Q}$ to define the rational function field over; this is not going to be the base field of the labelling, because the latter will have indeterminates adjoined!
- `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)

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.

REFERENCES:

EXAMPLES:

We construct the birational free labelling on a simple poset:

```
sage: P = Poset({1: [2, 3]})
sage: l = P.birational_free_labelling(); l
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b over Rational Field,
 {...},
 a,
 b)
sage: sorted(l[1].items())
[(1, x1), (2, x2), (3, x3)]

sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2]); l
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b over Rational Field,
 {...},
 a,
 b)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]

sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], reduced=True, addvars=
(Fraction Field of Multivariate Polynomial Ring in x1, x2, x3, spam, eggs over Rational
 {...},
 1,
 1)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]

sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], prefix="wut", reduced=
(Fraction Field of Multivariate Polynomial Ring in wut1, wut2, wut3, spam, eggs over Rat
 {...},
 1,
 1)
sage: sorted(l[1].items())
[(1, wut1), (2, wut3), (3, wut2)]

sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], reduced=False, addvars=
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b, spam, eggs over Rat
 {...},
 a,
 b)
sage: sorted(l[1].items())
[(1, x1), (2, x3), (3, x2)]
sage: l[1][2]
x3
```

Illustrating the warning about facade:

```
sage: P = Poset({1: [2, 3]}, facade=False)
sage: l = P.birational_free_labelling(linear_extension=[1, 3, 2], reduced=False, addvars=
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b, spam, eggs over Rat
 {...},
 a,
 b)
sage: l[1][2]
Traceback (most recent call last):
...
KeyError: 2
sage: l[1][P(2)]
x3
```

Another poset:

```
sage: P = Posets.SSTPoset([2,1])
sage: lext = sorted(P)
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,
 {...},
 a,
 b)
sage: sorted(l[1].items())
[([[1, 1], [2]], x1), ([[1, 1], [3]], x2), ([[1, 2], [2]], x3), ([[1, 2], [3]], x4),
 ([[1, 3], [2]], x5), ([[1, 3], [3]], x6), ([[2, 2], [3]], x7), ([[2, 3], [3]], x8)]
```

See   `birational_rowmotion()`,   `birational_toggle()`   and
`birational_toggles()` for more substantial examples of what one can do with the bira-
tional free labelling.

TESTS:

The `linear_extension` keyword does not have to be given an actual linear extension:

```
sage: P = Posets.ChainPoset(2).product(Posets.ChainPoset(3))
sage: P
Finite lattice containing 6 elements
sage: lex = [(1,0),(0,0),(1,1),(0,1),(1,2),(0,2)]
sage: l = P.birational_free_labelling(linear_extension=lex,
....:                                  prefix="u", reduced=True)
sage: l
(Fraction Field of Multivariate Polynomial Ring in u1, u2, u3, u4, u5, u6 over Rational I
 {...},
 1,
 1)
sage: sorted(l[1].items())
[((0, 0), u2),
 ((0, 1), u4),
 ((0, 2), u6),
 ((1, 0), u1),
 ((1, 1), u3),
 ((1, 2), u5)]
```

For comparison, the standard linear extension:

```
sage: l = P.birational_free_labelling(prefix="u", reduced=True); l
(Fraction Field of Multivariate Polynomial Ring in u1, u2, u3, u4, u5, u6 over Rational I
 {...},
 1,
 1)
sage: sorted(l[1].items())
[((0, 0), u1),
 ((0, 1), u2),
 ((0, 2), u3),
 ((1, 0), u4),
 ((1, 1), u5),
 ((1, 2), u6)]
```

If you want your linear extension to be tested for being a linear extension, just call the
`linear_extension` method on the poset:

```
sage: lex = [(0,0),(0,1),(1,0),(1,1),(0,2),(1,2)]
sage: l = P.birational_free_labelling(linear_extension=P.linear_extension(lex),
....:                                  prefix="u", reduced=True)
sage: l
```

```
(Fraction Field of Multivariate Polynomial Ring in u1, u2, u3, u4, u5, u6 over Rational
 {...},
 1,
 1)
sage: sorted(l[1].items())
[((0, 0), u1),
 ((0, 1), u2),
 ((0, 2), u5),
 ((1, 0), u3),
 ((1, 1), u4),
 ((1, 2), u6)]
```

Nonstandard base field:

```
sage: P = Poset({1: [3], 2: [3,4]})
sage: lex = [1, 2, 4, 3]
sage: l = P.birational_free_labelling(linear_extension=lex,
....:                                  prefix="aaa",
....:                                  base_field=Zmod(13))
sage: l
(Fraction Field of Multivariate Polynomial Ring in a, aaa1, aaa2, aaa3, aaa4, b over Ring
 {...},
 a,
 b)
sage: l[1][4]
aaa3
```

The empty poset:

```
sage: P = Poset({})
sage: P.birational_free_labelling(reduced=False, addvars="spam, eggs")
(Fraction Field of Multivariate Polynomial Ring in a, b, spam, eggs over Rational Field,
 {},
 a,
 b)
sage: P.birational_free_labelling(reduced=True, addvars="spam, eggs")
(Fraction Field of Multivariate Polynomial Ring in spam, eggs over Rational Field,
 {},
 1,
 1)
sage: P.birational_free_labelling(reduced=True)
(Multivariate Polynomial Ring in no variables over Rational Field,
 {},
 1,
 1)
sage: P.birational_free_labelling(prefix="zzz")
(Fraction Field of Multivariate Polynomial Ring in a, b over Rational Field,
 {},
 a,
 b)
```

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 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: 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 Fi
 {...},
 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 Fi
 {...},
 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 [GR13] 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
sage: test_rectangle_periodicity(2, 3, 5)  # long time
True
```

While computations with the birational free labelling quickly run out of memory due to the complexity of the rational functions involved, it is computationally cheap to check properties of birational rowmotion on examples in the tropical semiring:
```
sage: def test_rectangle_periodicity_tropical(n, m, k):
....:     P = Posets.ChainPoset(n).product(Posets.ChainPoset(m))
....:     TT = TropicalSemiring(ZZ)
....:     t0 = (TT, {v: TT(floor(random()*100)) for v in P}, TT(0), TT(124))
....:     t = t0
....:     for i in range(k):
....:         t = P.birational_rowmotion(t)
....:     return t == t0
sage: test_rectangle_periodicity_tropical(7, 6, 13)
True
```

Tropicalization is also what relates birational rowmotion to classical rowmotion on order ideals. In fact, if T denotes the `tropical semiring` of **Z** and P is a finite poset, then we can define an embedding ϕ from the set $J(P)$ of all order ideals of P into the set $T^{\hat{P}}$ 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 ϕ has the property that $R \circ \phi = \phi \circ r$, where R denotes birational rowmotion, and r denotes classical rowmotion on $J(P)$. An example:

```
sage: P = Posets.IntegerPartitions(5)
sage: TT = TropicalSemiring(ZZ)
sage: def indicator_labelling(I):
....:     # send order ideal 'I' to a 'T'-labelling of 'P'.
....:     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
```

TESTS:

Facade set to false works:

```
sage: P = Poset({1: [2, 3], 2: [4], 3: [4]}, facade=False)
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 Fie
 {...},
 a,
 b)
sage: t = P.birational_rowmotion(t); t
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, b over Rational Fie
 {...},
 a,
 b)
sage: t[1][P(2)]
(x1*x2*b + x1*x3*b)/(x2*x4)
sage: t = P.birational_rowmotion(t)
sage: t[1][P(2)]
a*b/x3
```

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: 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: 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: 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: 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: all( V.birational_toggle(i, V.birational_toggle(i, s)) == s
....:      for i in V )
True
```

We can also start with a less generic labelling:

```
sage: t = (QQ, {1: 3, 2: 6, 3: 7}, 2, 10)
sage: t1 = V.birational_toggle(1, t); t1
(Rational Field, {...}, 2, 10)
sage: sorted(t1[1].items())
[(1, 28/13), (2, 6), (3, 7)]
sage: t13 = V.birational_toggle(3, t1); t13
(Rational Field, {...}, 2, 10)
sage: sorted(t13[1].items())
[(1, 28/13), (2, 6), (3, 40/13)]
```

However, labellings have to be sufficiently generic, lest denominators vanish:

```
sage: t = (QQ, {1: 3, 2: 5, 3: -5}, 1, 15)
sage: t1 = V.birational_toggle(1, t)
Traceback (most recent call last):
...
ZeroDivisionError: Rational division by zero
```

We don't get into zero-division issues in the tropical semiring (unless the zero of the tropical semiring appears in the labelling):

```
sage: TT = TropicalSemiring(QQ)
sage: t = (TT, {1: TT(2), 2: TT(4), 3: TT(1)}, TT(6), TT(0))
sage: t1 = V.birational_toggle(1, t); t1
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t1[1].items())
[(1, 8), (2, 4), (3, 1)]
sage: t12 = V.birational_toggle(2, t1); t12
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t12[1].items())
[(1, 8), (2, 4), (3, 1)]
sage: t123 = V.birational_toggle(3, t12); t123
(Tropical semiring over Rational Field, {...}, 6, 0)
sage: sorted(t123[1].items())
[(1, 8), (2, 4), (3, 7)]
```

We turn to more interesting posets. Here is the 6-element poset arising from the weak order on S_3:

```
sage: P = Posets.SymmetricGroupWeakOrderPoset(3)
sage: sorted(list(P))
['123', '132', '213', '231', '312', '321']
sage: t = (TT, {'123': TT(4), '132': TT(2), '213': TT(3), '231': TT(1), '321': TT(1), '3
sage: t1 = P.birational_toggle('123', t); t1
(Tropical semiring over Rational Field, {...}, 7, 1)
sage: sorted(t1[1].items())
[('123', 6), ('132', 2), ('213', 3), ('231', 1), ('312', 2), ('321', 1)]
sage: t13 = P.birational_toggle('213', t1); t13
(Tropical semiring over Rational Field, {...}, 7, 1)
sage: sorted(t13[1].items())
[('123', 6), ('132', 2), ('213', 4), ('231', 1), ('312', 2), ('321', 1)]
```

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:

```
sage: all( P.birational_toggle(v, P.birational_toggle(v, t)) == t
....:      for v in P )
True
```

Furthermore, two toggles T_v and T_w commute unless one of v or w covers the other:

```
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
```

TESTS:

Setting `facade` to `False` does not break `birational_toggle`:

```
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: all( P.birational_toggle(v, P.birational_toggle(v, t)) == t
....:      for v in P )
True
sage: t4 = P.birational_toggle(P('z'), t); t4
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, b over Rational Fie
 {...},
 a,
 b)
sage: t4[1][P('x')]
x1
```

```
sage: t4[1][P('y')]
x2
sage: t4[1][P('w')]
x3
sage: t4[1][P('z')]
(x2*b + x3*b)/x4
```

The one-element poset:

```
sage: P = Poset({8: []})
sage: t = P.birational_free_labelling()
sage: t8 = P.birational_toggle(8, t); t8
(Fraction Field of Multivariate Polynomial Ring in a, x1, b over Rational Field,
 {...},
 a,
 b)
sage: t8[1][8]
a*b/x1
```

birational_toggles(*vs*, *labelling*)

Return the result of applying a sequence of birational toggles (specified by `vs`) to the **K**-labelling `labelling` of the poset `self`.

See the documentation of `birational_free_labelling()` for a definition of birational toggles and **K**-labellings and for an explanation of how **K**-labellings are to be encoded to be understood by Sage. This implementation allows **K** to be a semifield, not just a field. The birational v-toggle is only a rational map, so an exception (most likely, `ZeroDivisionError`) will be thrown if the denominator is zero.

INPUT:

- `vs` – an iterable comprising elements of `self` (which must have `self` as parent if `self` is a `facade=False` poset)
- `labelling` – a **K**-labelling of `self` in the sense as defined in the documentation of `birational_free_labelling()`

OUTPUT:

The **K**-labelling $T_{v_n} T_{v_{n-1}} \cdots T_{v_1} f$ of `self`, where f is `labelling` and $(v_1, v_2, \ldots, v_n)$ is `vs` (written as list).

EXAMPLES:

```
sage: P = Posets.SymmetricGroupBruhatOrderPoset(3)
sage: sorted(list(P))
['123', '132', '213', '231', '312', '321']
sage: TT = TropicalSemiring(ZZ)
sage: t = (TT, {'123': TT(4), '132': TT(2), '213': TT(3), '231': TT(1), '321': TT(1), '3
sage: tA = P.birational_toggles(['123', '231', '312'], t); tA
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tA[1].items())
[('123', 6), ('132', 2), ('213', 3), ('231', 2), ('312', 1), ('321', 1)]
sage: tAB = P.birational_toggles(['132', '213', '321'], tA); tAB
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tAB[1].items())
[('123', 6), ('132', 6), ('213', 5), ('231', 2), ('312', 1), ('321', 1)]

sage: P = Poset({1: [2, 3], 2: [4], 3: [4]})
sage: Qx = PolynomialRing(QQ, 'x').fraction_field()
sage: x = Qx.gen()
sage: t = (Qx, {1: 1, 2: x, 3: (x+1)/x, 4: x^2}, 1, 1)
sage: t1 = P.birational_toggles((i for i in range(1, 5)), t); t1
```

```
     (Fraction Field of Univariate Polynomial Ring in x over Rational Field,
      {...},
      1,
      1)
sage: sorted(t1[1].items())
[(1, (x^2 + x)/(x^2 + x + 1)), (2, (x^3 + x^2)/(x^2 + x + 1)), (3, x^4/(x^2 + x + 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
```

Facade set to `False` works:

```
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*x2*x3/(x1*x2 + x1*x3)), (y, a*x3*x4/(x1*x2 + x1*x3)), (w, x3), (z, x4)]
```

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((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, 3, 6, 12], [4
```

TESTS:

```
sage: list(Poset().directed_subsets('up'))
[[]]
```

is_lattice()

Returns whether this poset is both a meet and a join semilattice.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.is_lattice()
True

sage: P = Poset([[1,2],[3],[3],[]])
sage: P.is_lattice()
True

sage: P = Poset({0:[2,3],1:[2,3]})
sage: P.is_lattice()
False
```

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.

See also:

`FiniteLatticePosets.ParentMethods.is_lattice_morphism()`

is_poset_morphism (*f*, *codomain*)
    Return whether f is a morphism of posets from `self` to `codomain`, that is

$$x \le y \implies f(x) \le f(y)$$

for all x and y in `self`.

INPUT:
    • `f` – a function from `self` to `codomain`
    • `codomain` – a poset
EXAMPLES:

We build the boolean lattice of the subsets of $\{2, 3, 5, 6\}$ and the lattice of divisors of 30, and check that the map $b \mapsto \gcd(\prod_{x \in b} x, 30)$ is a morphism of posets:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: B = Poset(([frozenset(s) for s in Subsets([2,3,5,6])], attrcall("issubset")))
sage: def f(b): return D(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, D)
True
```

Note: since `D` and `B` are not facade posets, `f` is responsible for the conversions between integers and subsets to elements of `D` and `B` and back.

---

f is also a morphism of posets to the chain of divisors of 30, ordered by usual comparison:

```
sage: P = Poset((divisors(30), operator.le))
sage: def f(b): return P(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, P)
True
```

FIXME: should this be `is_order_preserving_morphism`?

See also:

`is_poset_isomorphism()`

TESTS:

Base cases:
```
sage: P = Posets.ChainPoset(2)
sage: Q = Posets.AntichainPoset(2)
sage: f = lambda x: 1-x
sage: P.is_poset_morphism(f, P)
False
sage: P.is_poset_morphism(f, Q)
False
sage: Q.is_poset_morphism(f, Q)
True
sage: Q.is_poset_morphism(f, P)
True

sage: P = Poset(); P
Finite poset containing 0 elements
sage: P.is_poset_morphism(f, P)
True
```

is_selfdual()
> Returns whether this poset is *self-dual*, that is isomorphic to its dual poset.
>
> EXAMPLES:
> ```
> sage: P = Poset(([1,2,3],[[1,3],[2,3]]),cover_relations=True)
> sage: P.is_selfdual()
> False
>
> sage: P = Poset(([1,2,3,4],[[1,3],[1,4],[2,3],[2,4]]),cover_relations=True)
> sage: P.is_selfdual()
> True
>
> sage: P = Poset( {} )
> sage: P.is_selfdual()
> True
> ```

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])]); I
> [{2, 3}, {1}, {1, 2}, {1, 3}, {1, 2, 3}]
> sage: P.order_filter_generators(I)
> {{2, 3}, {1}}
> ```
>
> **See also:**

> order_ideal_generators()

order_ideal_complement_generators(*antichain*, *direction='up'*)

Return the Panyushev complement of the antichain `antichain`.

Given an antichain A of a poset P, the Panyushev complement of A is defined to be the antichain consisting of the minimal elements of the order filter B, where B is the (set-theoretic) complement of the order ideal of P generated by A.

Setting the optional keyword variable `direction` to `'down'` leads to the inverse Panyushev complement being computed instead of the Panyushev complement. The inverse Panyushev complement of an antichain A is the antichain whose Panyushev complement is A. It can be found as the antichain consisting of the maximal elements of the order ideal C, where C is the (set-theoretic) complement of the order filter of P generated by A.

`panyushev_complement()` is an alias for this method.

Panyushev complementation is related (actually, isomorphic) to rowmotion (`rowmotion()`).

INPUT:
- `antichain` – an antichain of `self`, as a list (or iterable), or, more generally, generators of an order ideal (resp. order filter)
- `direction` – 'up' or 'down' (default: 'up')

OUTPUT:
- the generating antichain of the complement order filter (resp. order ideal) of the order ideal (resp. order filter) generated by the antichain `antichain`

EXAMPLES:
```
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])]); I
[{}, {3}, {2}, {2, 3}, {1}, {1, 2}]
```

Then, we retrieve the generators of this ideal:

```
sage: P.order_ideal_generators(I)
{{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])]); I
[{2, 3}, {1}, {1, 2}, {1, 3}, {1, 2, 3}]
sage: P.order_ideal_generators(I, direction='up')
{{2, 3}, {1}}
```

Complexity: $O(n+m)$ where n is the cardinality of I, and m the number of upper covers of elements of I.

order_ideals_lattice (*as_ideals=True, facade=False*)
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

EXAMPLES:

```
sage: P = Posets.PentagonPoset(facade = True)
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: list(J)
[{}, {0}, {0, 2}, {0, 2, 3}, {0, 1}, {0, 1, 2}, {0, 1, 2, 3}, {0, 1, 2, 3, 4}]
```

As a lattice on antichains:

```
sage: J2 = P.order_ideals_lattice(False); J2
Finite lattice containing 8 elements
sage: list(J2)
[(0,), (1, 2), (1, 3), (1,), (2,), (3,), (4,), ()]
```

TESTS:

```
sage: J = Posets.DiamondPoset(4, facade = True).order_ideals_lattice(); J
Finite lattice containing 6 elements
sage: list(J)
[{}, {0}, {0, 2}, {0, 1}, {0, 1, 2}, {0, 1, 2, 3}]
sage: J.cover_relations()
[[{}, {0}], [{0}, {0, 2}], [{0}, {0, 1}], [{0, 2}, {0, 1, 2}], [{0, 1}, {0, 1, 2}], [{0,
```

> **Note:** we use facade posets in the examples above just to ensure a nicer ordering in the output.

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:

```
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

panyushev_orbit_iter (*antichain*, *element_constructor=<type 'set'>*, *stop=True*, *check=True*)

Iterate over the Panyushev orbit of an antichain `antichain` of `self`.

The Panyushev orbit of an antichain is its orbit under Panyushev complementation (see `panyushev_complement()`).

INPUT:
- `antichain` – an antichain of `self`, given as an iterable.

- •element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the antichains before they are yielded.
- •stop – a Boolean (default: True) determining whether the iterator should stop once it completes its cycle (this happens when it is set to True) or go on forever (this happens when it is set to False).
- •check – a Boolean (default: True) determining whether antichain should be checked for being an antichain.

OUTPUT:

- •an iterator over the orbit of the antichain antichain under Panyushev complementation. This iterator I has the property that I[0] == 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:

```
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.panyushev_orbit_iter(set([1, 2])))
[{1, 2}, {3}, set()]
sage: list(P.panyushev_orbit_iter([1, 2]))
[{1, 2}, {3}, set()]
sage: list(P.panyushev_orbit_iter([2, 1]))
[{1, 2}, {3}, set()]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=list))
[[1, 2], [3], []]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=frozenset))
[frozenset({1, 2}), frozenset({3}), frozenset()]
sage: list(P.panyushev_orbit_iter(set([1, 2]), element_constructor=tuple))
[(1, 2), (3,), ()]

sage: P = Poset( {} )
sage: list(P.panyushev_orbit_iter([]))
[set()]

sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.panyushev_orbit_iter([2], stop=False)
sage: next(Piter)
{2}
sage: next(Piter)
{3}
sage: next(Piter)
{2}
sage: next(Piter)
{3}
```

panyushev_orbits (*element_constructor=<type 'set'>*)

Return the Panyushev orbits of antichains in self.

The Panyushev orbit of an antichain is its orbit under Panyushev complementation (see panyushev_complement()).

INPUT:

- •element_constructor (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the antichains before they are returned.

OUTPUT:

- •the partition of the set of all antichains of self into orbits under Panyushev complementation. This is returned as a list of lists L such that for each L and i, cyclically: self.order_ideal_complement_generators(L[i]) == L[i+1]. The

entries `L[i]` are sets by default, but depending on the optional keyword variable `element_constructors` they can also be tuples, lists etc.

EXAMPLES:
```
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: P.panyushev_orbits()
[[{2}, {1}], [set(), {1, 2}, {3}]]
sage: P.panyushev_orbits(element_constructor=list)
[[[2], [1]], [[], [1, 2], [3]]]
sage: P.panyushev_orbits(element_constructor=frozenset)
[[frozenset({2}), frozenset({1})],
 [frozenset(), frozenset({1, 2}), frozenset({3})]]
sage: P.panyushev_orbits(element_constructor=tuple)
[[(2,), (1,)], [(), (1, 2), (3,)]]
sage: P = Poset( {} )
sage: P.panyushev_orbits()
[[set()]]
```

rowmotion(*order_ideal*)

The image of the order ideal `order_ideal` under rowmotion in `self`.

Rowmotion on a finite poset P is an automorphism of the set $J(P)$ of all order ideals of P. One way to define it is as follows: Given an order ideal $I \in J(P)$, we let F be the set-theoretic complement of I in P. Furthermore we let A be the antichain consisting of all minimal elements of F. Then, the rowmotion of I is defined to be the order ideal of P generated by the antichain A (that is, the order ideal consisting of each element of P which has some element of A above it).

Rowmotion is related (actually, isomorphic) to Panyushev complementation (`panyushev_complement()`).

INPUT:
- `order_ideal` – an order ideal of `self`, as a set

OUTPUT:
- the image of `order_ideal` under rowmotion, as a set again

EXAMPLES:
```
sage: P = Poset( {1: [2, 3], 2: [], 3: [], 4: [8], 5: [], 6: [5], 7: [1, 4], 8: []} )
sage: I = Set({2, 6, 1, 7})
sage: P.rowmotion(I)
{1, 3, 4, 5, 6, 7}

sage: P = Poset( {} )
sage: I = Set({})
sage: P.rowmotion(I)
Set of elements of {}
```

rowmotion_orbit_iter(*oideal*, *element_constructor=<type 'set'>*, *stop=True*, *check=True*)

Iterate over the rowmotion orbit of an order ideal `oideal` of `self`.

The rowmotion orbit of an order ideal is its orbit under rowmotion (see `rowmotion()`).

INPUT:
- `oideal` – an order ideal of `self`, given as an iterable.
- `element_constructor` (defaults to set) – a type constructor (set, tuple, list, frozenset, iter, etc.) which is to be applied to the order ideals before they are yielded.
- `stop` – a Boolean (default: `True`) determining whether the iterator should stop once it completes its cycle (this happens when it is set to `True`) or go on forever (this happens when it is set to `False`).
- `check` – a Boolean (default: `True`) determining whether `oideal` should be checked for being an order ideal.

OUTPUT:

- •an iterator over the orbit of the order ideal `oideal` under rowmotion. This iterator *I* has the property that `I[0] == oideal` and that every *i* satisfies `self.rowmotion(I[i]) == I[i+1]`, where `I[i+1]` has to be understood as `I[0]` if it is undefined. The entries `I[i]` are sets by default, but depending on the optional keyword variable `element_constructors` they can also be tuples, lists etc.

EXAMPLES:

```
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.rowmotion_orbit_iter(set([1, 2])))
[{1, 2}, {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter([1, 2]))
[{1, 2}, {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter([2, 1]))
[{1, 2}, {1, 2, 3}, set()]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=list))
[[1, 2], [1, 2, 3], []]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=frozenset))
[frozenset({1, 2}), frozenset({1, 2, 3}), frozenset()]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=tuple))
[(1, 2), (1, 2, 3), ()]

sage: P = Poset( {} )
sage: list(P.rowmotion_orbit_iter([]))
[set()]

sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.rowmotion_orbit_iter([1, 2, 3], stop=False)
sage: next(Piter)
{1, 2, 3}
sage: next(Piter)
{1, 2, 3, 4}
sage: next(Piter)
set()
sage: next(Piter)
{1}
sage: next(Piter)
{1, 2, 3}

sage: P = Poset({ 1: [4], 2: [4, 5], 3: [5] })
sage: list(P.rowmotion_orbit_iter([1, 2], element_constructor=list))
[[1, 2], [1, 2, 3, 4], [2, 3, 5], [1], [2, 3], [1, 2, 3, 5], [1, 2, 4], [3]]
```

rowmotion_orbits(*element_constructor=<type '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: [], 3: [], 4: [2]} )
sage: sorted(len(o) for o in P.rowmotion_orbits())
[3, 5]
sage: sorted(P.rowmotion_orbits(element_constructor=list))
[[[1, 3], [4], [1], [4, 1, 3], [4, 1, 2]], [[4, 1], [4, 1, 2, 3], []]]
sage: sorted(P.rowmotion_orbits(element_constructor=tuple))
[[(1, 3), (4,), (1,), (4, 1, 3), (4, 1, 2)], [(4, 1), (4, 1, 2, 3), ()]]
sage: P = Poset({})
sage: sorted(P.rowmotion_orbits(element_constructor=tuple))
[[()]]
```

toggling_orbit_iter(*vs*, *oideal*, *element_constructor=<type '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 $(I, T_2 T_1 I, T_2 T_1 T_2 T_1 I, \ldots)$ (where I denotes `oideal` and T_i means toggling at i) rather than $(I, T_1 I, T_2 T_1 I, T_1 T_2 T_1 I, \ldots)$.

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=<type '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 $(I, T_2 T_1 I, T_2 T_1 T_2 T_1 I, \ldots)$ (where I denotes an order ideal and T_i means toggling at i) rather than $(I, T_1 I, T_2 T_1 I, T_1 T_2 T_1 I, \ldots)$.

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`).

OUTPUT:
   - a partition of the order ideals of `self`, as a list of sets L such that for each L and i, cyclically:
     `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]
```

13.50 Finite semigroups

class sage.categories.finite_semigroups.**FiniteSemigroups**(*base_category*)

   Bases: `sage.categories.category_with_axiom.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 <code>enumerated set</code> 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 <code>semigroup_generators()</code> should now explicitly declare themselves in the category of <code>finitely generated semigroups</code>:
>
> ```
> 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')
```

TESTS:
```
sage: TestSuite(C).run()
```

class **ParentMethods**

    idempotents()
        Returns the idempotents of the semigroup

        EXAMPLES:
```
sage: S = FiniteSemigroups().example(alphabet=('x','y'))
sage: sorted(S.idempotents())
['x', 'xy', 'y', 'yx']
```

    j_classes()
        Returns the J-classes of the semigroup.

        Two elements u and v of a monoid are in the same J-class if u divides v and v divides u.

        OUTPUT:
            All the J-classes of self, as a list of lists.
        EXAMPLES:
```
sage: S = FiniteSemigroups().example(alphabet=('a','b', 'c'))
sage: sorted(map(sorted, S.j_classes()))
[['a'], ['ab', 'ba'], ['abc', 'acb', 'bac', 'bca', 'cab', 'cba'], ['ac', 'ca'], ['b'], [
```

    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'], [
```

j_transversal_of_idempotents()

> Returns a list of one idempotent per regular J-class

> EXAMPLES:
> ```
> sage: S = FiniteSemigroups().example(alphabet=('a','b', 'c'))
> sage: sorted(S.j_transversal_of_idempotents())
> ['a', 'ab', 'ac', 'acb', 'b', 'c', 'cb']
> ```

13.51 Finite sets

class sage.categories.finite_sets.**FiniteSets**(*base_category*)

> Bases: [sage.categories.category_with_axiom.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
> ```

> TESTS:
> ```
> sage: TestSuite(C).run()
> sage: C is Sets().Finite()
> True
> ```

> **class Algebras**(*category*, **args*)

> > Bases: [sage.categories.algebra_functor.AlgebrasCategory](#)

> > TESTS:
> > ```
> > sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> > sage: class FooBars(CovariantConstructionCategory):
> > ....:     _functor_category = "FooBars"
> > ....:     _base_category_class = (Category,)
> > sage: Category.FooBars = lambda self: FooBars.category_of(self)
> > sage: C = FooBars(ModulesWithBasis(ZZ))
> > sage: C
> > Category of foo bars of modules with basis over Integer Ring
> > sage: C.base_category()
> > Category of modules with basis over Integer Ring
> > sage: latex(C)
> > \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> > sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
> > sage: TestSuite(C).run()
> > ```

extra_super_categories()
> 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:
> ```
> sage: FiniteMonoids().Algebras(QQ).is_subcategory(AlgebrasWithBasis(QQ).FiniteDimensiona
> True
> ```

class FiniteSets.**ParentMethods**

> **is_finite**()
> > Return `True` since `self` is finite.
> >
> > EXAMPLES:
> > ```
> > sage: C = FiniteEnumeratedSets().example()
> > sage: C.is_finite()
> > True
> > ```

class FiniteSets.**Subquotients**(*category*, **args*)
> Bases: `sage.categories.subquotients.SubquotientsCategory`
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> Category of foo bars of modules with basis over Integer Ring
> sage: C.base_category()
> Category of modules with basis over Integer Ring
> sage: latex(C)
> \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
> sage: TestSuite(C).run()
> ```
>
> **extra_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
> > ```

13.52 Finite Weyl Groups

class sage.categories.finite_weyl_groups.**FiniteWeylGroups**(*base_category*)

> Bases: sage.categories.category_with_axiom.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}
> ```
>
> TESTS:
> ```
> sage: W = FiniteWeylGroups().example()
> sage: TestSuite(W).run(verbose = "True")
> running ._test_an_element() . . . pass
> running ._test_associativity() . . . 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_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_has_descent() . . . pass
> running ._test_inverse() . . . pass
> running ._test_not_implemented_methods() . . . pass
> running ._test_one() . . . pass
> running ._test_pickling() . . . pass
> running ._test_prod() . . . pass
> running ._test_reduced_word() . . . pass
> running ._test_simple_projections() . . . pass
> running ._test_some_elements() . . . pass
> ```
>
> **class ElementMethods**
>
> **class** FiniteWeylGroups.**ParentMethods**

13.53 Function fields

class sage.categories.function_fields.**FunctionFields**(*s=None*)

    Bases: `sage.categories.category.Category`

    The category of function fields.

    EXAMPLES:

    We create the category of function fields:

```
sage: C = FunctionFields()
sage: C
Category of function fields
```

    TESTS:

```
sage: TestSuite(FunctionFields()).run()
```

    class ElementMethods

    class FunctionFields.**ParentMethods**

    FunctionFields.**super_categories**()

        Returns the Category of which this is a direct sub-Category For a list off all super caategories see all_super_categories

        EXAMPLES:

```
sage: FunctionFields().super_categories()
[Category of fields]
```

13.54 G-Sets

class sage.categories.g_sets.**GSets**(*G*)

    Bases: `sage.categories.category.Category`

    The category of G-sets, for a group G.

    EXAMPLES:

```
sage: S = SymmetricGroup(3)
sage: GSets(S)
Category of G-sets for Symmetric group of order 3! as a permutation group
```

    TODO: should this derive from Category_over_base?

    classmethod an_instance()

        Returns an instance of this class.

        EXAMPLES:

```
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()
[Category of sets]
```

13.55 Gcd domains

class sage.categories.gcd_domains.**GcdDomains**(*s=None*)

 Bases: sage.categories.category_singleton.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]
```

 TESTS:
```
sage: TestSuite(GcdDomains()).run()
```

 class ElementMethods

 class GcdDomains.**ParentMethods**

 GcdDomains.**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()
```

 GcdDomains.**super_categories**()

 EXAMPLES:
```
sage: GcdDomains().super_categories()
[Category of integral domains]
```

13.56 Graded Algebras

class sage.categories.graded_algebras.**GradedAlgebras**(*base_category*)

 Bases: sage.categories.graded_modules.GradedModulesCategory

 The category of graded algebras

 EXAMPLES:
```
sage: GradedAlgebras(ZZ)
Category of graded algebras over Integer Ring
sage: GradedAlgebras(ZZ).super_categories()
[Category of algebras over Integer Ring,
 Category of graded modules over Integer Ring]
```

 TESTS:

```
sage: TestSuite(GradedAlgebras(ZZ)).run()
```

class ElementMethods

class GradedAlgebras.**ParentMethods**

13.57 Graded algebras with basis

class sage.categories.graded_algebras_with_basis.**GradedAlgebrasWithBasis**(*base_category*)

   Bases: sage.categories.graded_modules.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 algebras with basis over Integer Ring,
 Category of graded algebras over Integer Ring,
 Category of graded modules with basis over Integer Ring]
```

   TESTS:
```
sage: TestSuite(C).run()
```

   class ElementMethods

   degree()
       The degree of this element.

       Note: This raises an error if the element is not homogeneous. To obtain the maximum of the degrees of the homogeneous summands, use maximal_degree()

       EXAMPLES:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.homogeneous_degree()
2
sage: (x^3 + 4*y^2).homogeneous_degree()
6
sage: ((1 + x)^3).homogeneous_degree()
Traceback (most recent call last):
...
ValueError: Element is not homogeneous.
```

       TESTS:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.zero().degree()
Traceback (most recent call last):
...
ValueError: The zero element does not have a well-defined degree.
```

   homogeneous_degree()
       The degree of this element.

Note: This raises an error if the element is not homogeneous. To obtain the maximum of the degrees of the homogeneous summands, use `maximal_degree()`

EXAMPLES:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.homogeneous_degree()
2
sage: (x^3 + 4*y^2).homogeneous_degree()
6
sage: ((1 + x)^3).homogeneous_degree()
Traceback (most recent call last):
...
ValueError: Element is not homogeneous.
```

TESTS:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.zero().degree()
Traceback (most recent call last):
...
ValueError: The zero element does not have a well-defined degree.
```

is_homogeneous()
    Return whether this element is homogeneous.

    EXAMPLES:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: (3*x).is_homogeneous()
True
sage: (x^3 - y^2).is_homogeneous()
True
sage: ((x + y)^2).is_homogeneous()
False
```

maximal_degree()
    The maximum of the degrees of the homogeneous summands.

    EXAMPLES:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.maximal_degree()
2
sage: (x^3 + 4*y^2).maximal_degree()
6
sage: ((1 + x)^3).maximal_degree()
6
```

    TESTS:
```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S.zero().degree()
Traceback (most recent call last):
...
ValueError: The zero element does not have a well-defined degree.
```

class GradedAlgebrasWithBasis.**ParentMethods**

13.58 Graded bialgebras

sage.categories.graded_bialgebras.**GradedBialgebras**(*base_ring*)

> The category of graded bialgebras
>
> EXAMPLES:
> ```
> sage: C = GradedBialgebras(QQ); C
> Join of Category of graded algebras over Rational Field
>     and Category of bialgebras over Rational Field
> sage: C is Bialgebras(QQ).Graded()
> True
> ```
>
> TESTS:
> ```
> sage: TestSuite(C).run()
> ```

13.59 Graded bialgebras with basis

sage.categories.graded_bialgebras_with_basis.**GradedBialgebrasWithBasis**(*base_ring*)

> The category of graded bialgebras with a distinguished basis
>
> EXAMPLES:
> ```
> sage: C = GradedBialgebrasWithBasis(QQ); C
> Join of Category of ...
> sage: sorted(C.super_categories(), key=str)
> [Category of bialgebras over Rational Field,
>  Category of coalgebras with basis over Rational Field,
>  Category of graded algebras with basis over Rational Field]
> ```
>
> TESTS:
> ```
> sage: TestSuite(C).run()
> ```

13.60 Graded Coalgebras

sage.categories.graded_coalgebras.**GradedCoalgebras**(*base_ring*)

> The category of graded coalgebras
>
> EXAMPLES:
> ```
> sage: C = GradedCoalgebras(QQ); C
> Join of Category of graded modules over Rational Field
>     and Category of coalgebras over Rational Field
> sage: C is Coalgebras(QQ).Graded()
> True
> ```
>
> TESTS:
> ```
> sage: TestSuite(C).run()
> ```

13.61 Graded coalgebras with basis

sage.categories.graded_coalgebras_with_basis.**GradedCoalgebrasWithBasis**(*base_ring*)

The category of graded coalgebras with a distinguished basis

EXAMPLES:
```
sage: C = GradedCoalgebrasWithBasis(QQ); C
Join of Category of graded modules with basis over Rational Field
    and Category of coalgebras with basis over Rational Field
sage: C is Coalgebras(QQ).WithBasis().Graded()
True
```

TESTS:
```
sage: TestSuite(C).run()
```

13.62 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
sage: C is HopfAlgebras(QQ).Graded()
True
```

TESTS:
```
sage: TestSuite(C).run()
```

13.63 Graded Hopf algebras with basis

class sage.categories.graded_hopf_algebras_with_basis.**GradedHopfAlgebrasWithBasis**(*base_category*)

Bases: `sage.categories.graded_modules.GradedModulesCategory`

The category of graded Hopf algebras with a distinguished basis.

EXAMPLES:
```
sage: C = GradedHopfAlgebrasWithBasis(ZZ); C
Category of graded hopf algebras with basis over Integer Ring
sage: C.super_categories()
[Category of hopf algebras with basis over Integer Ring,
 Category of graded algebras with basis over Integer Ring]

sage: C is HopfAlgebras(ZZ).WithBasis().Graded()
True
sage: C is HopfAlgebras(ZZ).Graded().WithBasis()
False
```

TESTS:

```
sage: TestSuite(C).run()
```

class **ElementMethods**

class GradedHopfAlgebrasWithBasis.**ParentMethods**

class GradedHopfAlgebrasWithBasis.**WithRealizations**(*category*, **args*)

Bases: sage.categories.with_realizations.WithRealizationsCategory

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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]
```

TESTS:
```
sage: TestSuite(GradedHopfAlgebrasWithBasis(QQ).WithRealizations()).run()
```

13.64 Graded modules

class sage.categories.graded_modules.**GradedModules**(*base_category*)

Bases: sage.categories.graded_modules.GradedModulesCategory

The category of graded modules.

EXAMPLES:
```
sage: GradedModules(ZZ)
Category of graded modules over Integer Ring
sage: GradedModules(ZZ).super_categories()
[Category of modules over Integer Ring]
```

The category of graded modules defines the graded structure which shall be preserved by morphisms:
```
sage: Modules(ZZ).Graded().additional_structure()
Category of graded modules over Integer Ring
```

TESTS:
```
sage: TestSuite(GradedModules(ZZ)).run()
```

class **Connected**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

TESTS:
```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

class GradedModules.**ElementMethods**

class GradedModules.**ParentMethods**

class GradedModules.**SubcategoryMethods**

Connected()

Return the full subcategory of the connected objects of `self`.

EXAMPLES:
```
sage: Modules(ZZ).Graded().Connected()
Category of graded connected modules over Integer Ring
sage: Coalgebras(QQ).Graded().Connected()
Join of Category of graded connected modules over Rational Field
    and Category of coalgebras over Rational Field
sage: GradedAlgebrasWithBasis(QQ).Connected()
Category of graded connected algebras with basis over Rational Field
```

TESTS:
```
sage: TestSuite(Modules(ZZ).Graded().Connected()).run()
sage: Coalgebras(QQ).Graded().Connected.__module__
'sage.categories.graded_modules'
```

GradedModules.**extra_super_categories**()

Adds `VectorSpaces` to the super categories of `self` if the base ring is a field.

EXAMPLES:
```
sage: Modules(QQ).Graded().extra_super_categories()
[Category of vector spaces over Rational Field]
sage: Modules(ZZ).Graded().extra_super_categories()
[]
```

This makes sure that `Modules(QQ).Graded()` returns an instance of `GradedModules` and not a join category of an instance of this class and of `VectorSpaces(QQ)`:
```
sage: type(Modules(QQ).Graded())
<class 'sage.categories.graded_modules.GradedModules_with_category'>
```

Todo

Get rid of this workaround once there is a more systematic approach for the alias `Modules(QQ) ->` `VectorSpaces(QQ)`. Probably the later should be a category with axiom, and covariant constructions should play well with axioms.

---

class sage.categories.graded_modules.**GradedModulesCategory**(*base_category*)

Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstruction sage.categories.category_types.Category_over_base_ring

EXAMPLES:

```
sage: C = GradedAlgebras(QQ)
sage: C
Category of graded algebras over Rational Field
sage: C.base_category()
Category of algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of algebras over Rational Field,
 Category of graded modules over Rational Field]

sage: AlgebrasWithBasis(QQ).Graded().base_ring()
Rational Field
sage: GradedHopfAlgebrasWithBasis(QQ).base_ring()
Rational Field
```

TESTS:

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

13.65 Graded modules with basis

class sage.categories.graded_modules_with_basis.**GradedModulesWithBasis**(*base_category*)

Bases: sage.categories.graded_modules.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 graded modules over Integer Ring,
 Category of modules with basis over Integer Ring]
sage: C is ModulesWithBasis(ZZ).Graded()
True
```

TESTS:

```
sage: TestSuite(C).run()
```

class ElementMethods

degree()
The degree of this element in the graded module.

Note: This raises an error if the element is not homogeneous. Another implementation option would be to return the maximum of the degrees of the homogeneous summands.

EXAMPLES:

```
sage: A = GradedModulesWithBasis(ZZ).example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: x.degree()
6
sage: (x + 2*z).degree()
6
sage: (y - x).degree()
Traceback (most recent call last):
...
ValueError: Element is not homogeneous.
```

homogeneous_component(*n*)

Return the homogeneous component of degree n of this element.

EXAMPLES:

```
sage: A = GradedModulesWithBasis(ZZ).example()
sage: x = A.an_element(); x
2*P[] + 2*P[1] + 3*P[2]
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
```

TESTS:

Check that this really return A.zero() and not a plain 0:

```
sage: x.homogeneous_component(3).parent() is A
True
```

is_homogeneous()

Return whether this element is homogeneous.

EXAMPLES:

```
sage: A = GradedModulesWithBasis(ZZ).example()
sage: x=A(Partition((3,2,1)))
sage: y=A(Partition((4,4,1)))
sage: z=A(Partition((2,2,2)))
sage: (3*x).is_homogeneous()
True
sage: (x - y).is_homogeneous()
False
sage: (x+2*z).is_homogeneous()
True
```

truncate(*n*)

Return the sum of the homogeneous components of degree < n of this element

EXAMPLES:

```
sage: A = GradedModulesWithBasis(ZZ).example()
sage: x = A.an_element(); x
```

```
2*P[] + 2*P[1] + 3*P[2]
sage: x.truncate(0)
0
sage: x.truncate(1)
2*P[]
sage: x.truncate(2)
2*P[] + 2*P[1]
sage: x.truncate(3)
2*P[] + 2*P[1] + 3*P[2]
```

TESTS:

Check that this really return `A.zero()` and not a plain `0`:

```
sage: x.truncate(0).parent() is A
True
```

class GradedModulesWithBasis.**ParentMethods**

> **basis**(*d=None*)
> Returns the basis for (an homogeneous component of) this graded module
>
> INPUT:
>> •*d* – non negative integer or `None`, optional (default: `None`)
>> If *d* is None, returns a basis of the module. Otherwise, returns the basis of the homogeneous component of degree *d*.
>
> EXAMPLES:
> ```
> sage: A = GradedModulesWithBasis(ZZ).example()
> sage: A.basis(4)
> Lazy family (Term map from Partitions to An example of a graded module with basis: the f
> ```
>
> Without arguments, the full basis is returned:
> ```
> sage: A.basis()
> Lazy family (Term map from Partitions to An example of a graded module with basis: the f
> sage: A.basis()
> Lazy family (Term map from Partitions to An example of a graded module with basis: the f
> ```

13.66 Group Algebras

sage.categories.group_algebras.**GroupAlgebras**(*base_ring*)
> The category of group algebras over `base_ring`.
>
> EXAMPLES:
> ```
> sage: C = GroupAlgebras(QQ); C
> Category of group algebras over Rational Field
> sage: sorted(C.super_categories(), key=str)
> [Category of hopf algebras with basis over Rational Field,
>  Category of monoid algebras over Rational Field]
> ```
>
> This is just an alias for:
> ```
> sage: C is Groups().Algebras(QQ)
> True
> ```
>
> TESTS:

```
sage: TestSuite(GroupAlgebras(ZZ)).run()
```

13.67 Groupoid

class sage.categories.groupoid.**Groupoid**(*G=None*)

Bases: `sage.categories.category.CategoryWithParameters`

The category of groupoids, for a set (usually a group) G.

FIXME:

- Groupoid or Groupoids ?

- definition and link with http://en.wikipedia.org/wiki/Groupoid

- Should Groupoid inherit from Category_over_base?

EXAMPLES:

```
sage: Groupoid(DihedralGroup(3))
Groupoid with underlying set Dihedral group of order 6 as a permutation group
```

classmethod an_instance()

Returns an instance of this class.

EXAMPLES:

```
sage: Groupoid.an_instance() # indirect doctest
Groupoid with underlying set Symmetric group of order 8! as a permutation group
```

super_categories()

EXAMPLES:

```
sage: Groupoid(DihedralGroup(3)).super_categories()
[Category of sets]
```

13.68 Groups

class sage.categories.groups.**Groups**(*base_category*)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of (multiplicative) groups, i.e. monoids with inverses.

EXAMPLES:

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

TESTS:

```
sage: TestSuite(Groups()).run()
```

class Algebras(*category*, **args*)

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

The category of group algebras over a given base ring.

EXAMPLES:
```
sage: GroupAlgebras(IntegerRing())
Category of group algebras over Integer Ring
sage: GroupAlgebras(IntegerRing()).super_categories()
[Category of hopf algebras with basis over Integer Ring,
 Category of monoid algebras over Integer Ring]
```

Here is how to create the group algebra of a group G:
```
sage: G = DihedralGroup(5)
sage: QG = G.algebra(QQ); QG
Group algebra of Dihedral group of order 10 as a permutation group over Rational Field
```

and an example of computation:
```
sage: g = G.an_element(); g
(1,2,3,4,5)
sage: (QG.term(g) + 1)**3
B[()] + 3*B[(1,2,3,4,5)] + 3*B[(1,3,5,2,4)] + B[(1,4,2,5,3)]
```

Todo

  • Check which methods would be better located in `Monoid.Algebras` or `Groups.Finite.Algebras`.

TESTS:
```
sage: A = GroupAlgebras(QQ).example(GL(3, GF(11)))
sage: A.one_basis()
[1 0 0]
[0 1 0]
[0 0 1]
sage: A = SymmetricGroupAlgebra(QQ,4)
sage: x = Permutation([4,3,2,1])
sage: A.product_on_basis(x,x)
[1, 2, 3, 4]

sage: C = GroupAlgebras(ZZ)
sage: TestSuite(C).run()
```

class ElementMethods

  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:

```
sage: QS3 = SymmetricGroup(3).algebra(QQ)
sage: A = QS3([2,3,1]) + QS3([3,1,2])
sage: A.central_form()
B[(1,2,3)]
sage: QS4 = SymmetricGroup(4).algebra(QQ)
sage: B = sum(len(s.cycle_type())*QS4(s) for s in Permutations(4))
sage: B.central_form()
4*B[()] + 3*B[(1,2)] + 2*B[(1,2)(3,4)] + 2*B[(1,2,3)] + B[(1,2,3,4)]

sage: QG = GroupAlgebras(QQ).example(PermutationGroup([[(1,2,3),(4,5)],[(3,4)]]))
sage: sum(i for i in QG.basis()).central_form()
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
```

See also:

- `Groups.Algebras.ParentMethods.center_basis()`
- `Monoids.Algebras.ElementMethods.is_central()`

class `Groups.Algebras.ParentMethods`

algebra_generators()

Return generators of this group algebra (as an algebra).

EXAMPLES:

```
sage: GroupAlgebras(QQ).example(AlternatingGroup(10)).algebra_generators()
Finite family {(8,9,10): B[(8,9,10)], (1,2,3,4,5,6,7,8,9): B[(1,2,3,4,5,6,7,8,9)]}
```

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
Group algebra of Cyclic group of order 6 as a permutation group over Integer Ring
sage: g=CyclicPermutationGroup(6).an_element();g
(1,2,3,4,5,6)
sage: A.antipode_on_basis(g)
B[(1,6,5,4,3,2)]
sage: a=A.an_element();a
B[()] + 3*B[(1,2,3,4,5,6)] + 3*B[(1,3,5)(2,4,6)]
sage: a.antipode()
B[()] + 3*B[(1,5,3)(2,6,4)] + 3*B[(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_σ is the sum of the elements in the conjugacy class of σ.

OUTPUT:

•`list` of elements of `self`

> **Warning:**
>
> •This method requires the underlying group to have a method `conjugacy_classes` (every permutation group has one, thanks GAP!).

EXAMPLES:
```
sage: SymmetricGroup(3).algebra(QQ).center_basis()
[(), (2,3) + (1,2) + (1,3), (1,2,3) + (1,3,2)]
```

See also:

•`Groups.Algebras.ElementMethods.central_form()`
•`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
Group algebra of Cyclic group of order 6 as a permutation group over Integer Ring
sage: g=CyclicPermutationGroup(6).an_element();g
(1,2,3,4,5,6)
sage: A.coproduct_on_basis(g)
B[(1,2,3,4,5,6)] # B[(1,2,3,4,5,6)]
sage: a=A.an_element();a
B[()] + 3*B[(1,2,3,4,5,6)] + 3*B[(1,3,5)(2,4,6)]
sage: a.coproduct()
B[()] # B[()] + 3*B[(1,2,3,4,5,6)] # B[(1,2,3,4,5,6)] + 3*B[(1,3,5)(2,4,6)] # B[(1,3,
```

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
Group algebra of Cyclic group of order 6 as a permutation group over Integer Ring
sage: a=A.an_element();a
B[()] + 3*B[(1,2,3,4,5,6)] + 3*B[(1,3,5)(2,4,6)]
sage: a.counit()
7
```

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
Group algebra of Cyclic group of order 6 as a permutation group over Integer Ring
sage: g=CyclicPermutationGroup(6).an_element();g
(1,2,3,4,5,6)
```

```
sage: A.counit_on_basis(g)
1
```

group()
    Return the underlying group of the group algebra.

    EXAMPLES:
```
sage: GroupAlgebras(QQ).example(GL(3, GF(11))).group()
General Linear Group of degree 3 over Finite Field of size 11
sage: SymmetricGroup(10).algebra(QQ).group()
Symmetric group of order 10! as a permutation group
```

Groups.Algebras.**example**(*G=None*)
    Return an example of group algebra.

    EXAMPLES:
```
sage: GroupAlgebras(QQ['x']).example()
Group algebra of Dihedral group of order 8 as a permutation group over Univariate Polynor
```

    An other group can be specified as optional argument:
```
sage: GroupAlgebras(QQ).example(AlternatingGroup(4))
Group algebra of Alternating group of order 4!/2 as a permutation group over Rational Fie
```

Groups.Algebras.**extra_super_categories**()
    Implement the fact that the algebra of a group is a Hopf algebra.

    EXAMPLES:
```
sage: C = Groups().Algebras(QQ)
sage: C.extra_super_categories()
[Category of hopf algebras over Rational Field]
sage: sorted(C.super_categories(), key=str)
[Category of hopf algebras with basis over Rational Field,
 Category of monoid algebras over Rational Field]
```

class Groups.**CartesianProducts**(*category*, **args*)
    Bases: `sage.categories.cartesian_product.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

    group_generators()
        Return the group generators of `self`.

        EXAMPLES:
```
sage: C5 = CyclicPermutationGroup(5)
sage: C4 = CyclicPermutationGroup(4)
sage: S4 = SymmetricGroup(3)
sage: C = cartesian_product([C5, C4, S4])
sage: C.group_generators()
Family (((1,2,3,4,5), (), ()),
        ((), (1,2,3,4), ()),
        ((), (), (1,2)),
        ((), (), (2,3)))
```

        We check the other portion of trac ticket #16718 is fixed:

```
sage: len(C.j_classes())
1
```

An example with an infinitely generated group (a better output is needed):

```
sage: G = Groups.free([1,2])
sage: H = Groups.free(ZZ)
sage: C = cartesian_product([G, H])
sage: C.monoid_generators()
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))])
> sage: C.order()
> 87091200
> ```
>
> TESTS:
> ```
> sage: C.order.__module__
> 'sage.categories.groups'
> ```
>
> ---
>
> **Todo**
>
> this method is just here to prevent `FiniteGroups.ParentMethods` to call `_cardinality_from_iterator`.
>
> ---

Groups.CartesianProducts.**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 Groups.**Commutative** (*base_category*)

> Bases: `sage.categories.category_with_axiom.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)
> > Free abelian group indexed by Integer Ring
> > sage: Groups().Commutative().free(ZZ)
> > ```

```
Free abelian group indexed by Integer Ring
sage: Groups().Commutative().free(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: F.<x,y,z> = Groups().Commutative().free(); F
Multiplicative Abelian group isomorphic to Z x Z x Z
```

class Groups.**ElementMethods**

conjugacy_class()
> Return the conjugacy class of `self`.
>
> EXAMPLES:
>
> ```
> sage: D = DihedralGroup(5)
> sage: g = D((1,3,5,2,4))
> sage: g.conjugacy_class()
> Conjugacy class of (1,3,5,2,4) in Dihedral group of order 10 as a permutation group
>
> sage: H = MatrixGroup([matrix(GF(5),2,[1,2, -1, 1]), matrix(GF(5),2, [1,1, 0,1])])
> sage: h = H(matrix(GF(5),2,[1,2, -1, 1]))
> sage: h.conjugacy_class()
> 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
> ```

Groups.**Finite**
> alias of FiniteGroups

class Groups.**ParentMethods**

cayley_table(*names='letters'*, *elements=None*)
> Returns the "multiplication" table of this multiplicative group, which is also known as the "Cayley table".

> **Note:** The order of the elements in the row and column headings is equal to the order given by the table's `column_keys()` method. The association between the actual elements and the names/symbols used in the table can also be retrieved as a dictionary with the `translation()` method.

> For groups, this routine should behave identically to the `multiplication_table()` method for magmas, which applies in greater generality.
>
> INPUT:
> > •`names` - the type of names used, values are:

- **–**`'letters'` - lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by `list()`, padded to a common width with leading 'a's.
- **–**`'digits'` - base 10 representation of the elements' positions in the list given by `column_keys()`, padded to a common width with leading zeros.
- **–**`'elements'` - the string representations of the elements themselves.
- **–**a list - a list of strings, where the length of the list equals the number of elements.
- •`elements` - default = `None`. A list of elements of the group, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering is provided by the the group, which is reported by the `column_keys()` method. Or the `elements` can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.

OUTPUT: An object representing the multiplication table. This is an `OperationTable` object and even more documentation can be found there.

EXAMPLES:

Permutation groups, matrix groups and abelian groups can all compute their multiplication tables.

```
sage: G = DiCyclicGroup(3)
sage: T = G.cayley_table()
sage: T.column_keys()
((), (1,3,2,4)(5,7), ..., (1,2)(3,4)(5,7,6))
sage: T
*  a b c d e f g h i j k l
 +------------------------
a| a b c d e f g h i j k l
b| b e f j i h d k a l c g
c| c g d e h b k l j f i a
d| d k e h l g i a f b j c
e| e i h l a k j c b g f d
f| f d j i k e c g l h a b
g| g h b f j l e i c a d k
h| h j l a c i f d g k b e
i| i a k g b c l f e d h j
j| j c i k g d a b h e l f
k| k l g b f a h j d c e i
l| l f a c d j b e k i g h

sage: M=SL(2,2)
sage: M.cayley_table()
*  a b c d e f
 +------------
a| a b c d e f
b| b a d c f e
c| c f e b a d
d| d e f a b c
e| e d a f c b
f| f c b e d a

sage: A=AbelianGroup([2,3])
sage: A.cayley_table()
*  a b c d e f
 +------------
a| a b c d e f
b| b c a e f d
c| c a b f d e
d| d e f a b c
e| e f d b c a
f| f d e c a b
```

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)
sage: C.cayley_table(names='digits')
 *  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()
sage: names=['1', 'I', 'J', '-1', '-K', 'K', '-I', '-J']
sage: G.cayley_table(names=names)
 *   1  I  J -1 -K  K -I -J
  +-----------------------
 1|  1  I  J -1 -K  K -I -J
 I|  I -1  K -I  J -J  1 -K
 J|  J -K -1 -J -I  I  K  1
-1| -1 -I -J  1  K -K  I  J
-K| -K -J  I  K -1  1  J -I
 K|  K  J -I -K  1 -1 -J  I
-I| -I  1 -K  I -J  J -1  K
-J| -J  K  1  J  I -I -K -1

sage: A=AbelianGroup([2,2])
sage: A.cayley_table(names='elements')
    *      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)
sage: T=G.cayley_table()
sage: T.change_names('digits')
sage: T
* 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)
sage: G
Special Linear Group of degree 2 over Integer Ring
```

```
sage: identity = matrix(ZZ, [[1,0], [0,1]])
sage: G.cayley_table(elements=[identity, -identity])
*  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
sage: G=DiCyclicGroup(3)
sage: commutator = lambda x, y: x*y*x^-1*y^-1
sage: T=OperationTable(G, commutator)
sage: T
.  a b c d e f g h i j k l
 +------------------------
a| a a a a a a a a a a a a
b| a a h d a d h h a h d d
c| a d a a a d d a d d d a
d| a h a a a h h a h h h a
e| a a a a a a a a a a a a
f| a h h d a a d h h d a d
g| a d h d a h a h d a h d
h| a d a a a d d a d d d a
i| a a h d a d h h a h d d
j| a d h d a h a h d a h d
k| a h h d a a d h h d a d
l| a h a a a h h a h h h a
sage: trans = T.translation()
sage: comm = [trans['a'], trans['d'],trans['h']]
sage: comm
[(), (5,7,6), (5,6,7)]
sage: P=G.cayley_table(elements=comm)
sage: P
*  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 Trac #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])
sage: c = A.conjugacy_class(A.an_element())
```

```
sage: type(c)
<class 'sage.groups.conjugacy_classes.ConjugacyClass_with_category'>
```

group_generators()
    Returns group generators for self.

    This default implementation calls `gens()`, for backward compatibility.

    EXAMPLES:

```
sage: A = AlternatingGroup(4)
sage: A.group_generators()
Family ((2,3,4), (1,2,3))
```

holomorph()
    The holomorph of a group

    The holomorph of a group G is the semidirect product $G \rtimes_{id} Aut(G)$, where id is the identity function on $Aut(G)$, the automorphism group of G.

    See Wikipedia article Holomorph (mathematics)

    EXAMPLES:

```
sage: G = Groups().example()
sage: G.holomorph()
Traceback (most recent call last):
...
NotImplementedError: holomorph of General Linear Group of degree 4 over Rational Field n
```

monoid_generators()
    Return the generators of `self` as a monoid.

    Let G be a group with generating set X. In general, the generating set of G as a monoid is given by $X \cup X^{-1}$, where X^{-1} is the set of inverses of X. If G is a finite group, then the generating set as a monoid is X.

    EXAMPLES:

```
sage: A = AlternatingGroup(4)
sage: A.monoid_generators()
Family ((2,3,4), (1,2,3))
sage: F.<x,y> = FreeGroup()
sage: F.monoid_generators()
Family (x, y, x^-1, y^-1)
```

semidirect_product(*N*, *mapping*, *check=True*)
    The semi-direct product of two groups

    EXAMPLES:

```
sage: G = Groups().example()
sage: G.semidirect_product(G,Morphism(G,G))
Traceback (most recent call last):
...
NotImplementedError: semidirect product of General Linear Group of degree 4 over Rationa
```

Groups.**example**()
    EXAMPLES:

```
sage: Groups().example()
General Linear Group of degree 4 over Rational Field
```

static `Groups.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)
Free group indexed by Integer Ring
sage: Groups().free(ZZ)
Free group indexed by Integer Ring
sage: Groups().free(5)
Free Group on generators {x0, x1, x2, x3, x4}
sage: F.<x,y,z> = Groups().free(); F
Free Group on generators {x, y, z}
```

13.69 Hecke modules

class `sage.categories.hecke_modules.`**`HeckeModules`**(*R*)

Bases: `sage.categories.category_types.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 **Q**:
```
sage: C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
```

TODO: check that this is what we want:
```
sage: C.super_categories()
[Category of vector spaces with basis over Rational Field]
```

[Category of vector spaces over Rational Field]

Note that the base ring can be an arbitrary commutative ring:
```
sage: HeckeModules(IntegerRing())
Category of Hecke modules over Integer Ring
sage: HeckeModules(FiniteField(5))
Category of Hecke modules over Finite Field of size 5
```

The base ring doesn't have to be a principal ideal domain:

```
sage: HeckeModules(PolynomialRing(IntegerRing(), 'x'))
Category of Hecke modules over Univariate Polynomial Ring in x over Integer Ring
```

TESTS:
```
sage: TestSuite(HeckeModules(ZZ)).run()
```

class **Homsets**(*category*, **args*)

Bases: `sage.categories.homsets.HomsetsCategory`

TESTS:
```
sage: TestSuite(HeckeModules(ZZ).Homsets()).run()
```

class ParentMethods

HeckeModules.Homsets.**base_ring**()

EXAMPLES:
```
sage: HeckeModules(QQ).Homsets().base_ring()
Rational Field
```

HeckeModules.Homsets.**extra_super_categories**()

TESTS:

Check that Hom sets of Hecke modules are in the correct category (see trac ticket #17359):
```
sage: HeckeModules(ZZ).Homsets().super_categories()
[Category of modules over Integer Ring, Category of homsets]
sage: HeckeModules(QQ).Homsets().super_categories()
[Category of vector spaces over Rational Field, Category of homsets]
```

class HeckeModules.**ParentMethods**

HeckeModules.**super_categories**()

EXAMPLES:
```
sage: HeckeModules(QQ).super_categories()
[Category of vector spaces with basis over Rational Field]
```

13.70 Highest Weight Crystals

class sage.categories.highest_weight_crystals.**HighestWeightCrystals**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of highest weight crystals.

A crystal is highest weight if it is acyclic; in particular, every connected component has a unique highest weight element, and that element generate the component.

EXAMPLES:
```
sage: C = HighestWeightCrystals()
sage: C
Category of highest weight crystals
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

TESTS:
```
sage: TestSuite(C).run()
sage: B = HighestWeightCrystals().example()
sage: TestSuite(B).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_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

class **ElementMethods**

class HighestWeightCrystals.**ParentMethods**

> **highest_weight_vector**()
>> Returns the highest weight vector if there is a single one; otherwise, raises an error.
>>
>> Caveat: this assumes that highest_weight_vectors() returns a list or tuple.
>>
>> EXAMPLES:
>> ```
>> sage: C = crystals.Letters(['A',5])
>> sage: C.highest_weight_vector()
>> 1
>> ```
>
> **highest_weight_vectors**()
>> Returns the highest weight vectors of self
>>
>> This default implementation selects among the module generators those that are highest weight, and caches the result. A crystal element b is highest weight if $e_i(b) = 0$ for all i in the index set.
>>
>> EXAMPLES:
>> ```
>> sage: C = crystals.Letters(['A',5])
>> sage: C.highest_weight_vectors()
>> (1,)
>>
>> 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()

    Returns 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:

```
sage: C = crystals.Letters(['A',5])
sage: C.lowest_weight_vectors()
[6]

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.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 μ-weight space of $B(\lambda)$ (under the principal gradation) is equal to $\langle \rho^\vee, \lambda - \mu \rangle$ where $\langle \rho^\vee, \alpha_i \rangle = 1$ for all $i \in I$ (in particular, take $\rho^\vee = \sum_{i \in I} h_i$).

    The q-dimension of a highest weight crystal $B(\lambda)$ is defined as

$$\dim_q B(\lambda) := \sum_{j \geq 0} \dim(B_j) q^j,$$

    where B_j denotes the degree j portion of $B(\lambda)$. This can be expressed as the product

$$\dim_q B(\lambda) = \prod_{\alpha^\vee \in \Delta_+^\vee} \left(\frac{1 - q^{\langle \lambda + \rho, \alpha^\vee \rangle}}{1 - q^{\langle \rho, \alpha^\vee \rangle}} \right)^{\text{mult}\,\alpha},$$

    where $\Delta_+^\vee$ denotes the set of positive coroots. Taking the limit as $q \to 1$ gives the dimension of $B(\lambda)$. For more information, see [Kac] 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:

```
sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: qdim = C.q_dimension(); qdim
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
sage: qdim(1)
8
sage: len(C) == qdim(1)
True
sage: C.q_dimension(use_product=True) == qdim
True
sage: C.q_dimension(prec=20)
q^4 + 2*q^3 + 2*q^2 + 2*q + 1
sage: C.q_dimension(prec=2)
2*q + 1

sage: R.<t> = QQ[]
sage: C.q_dimension(q=t^2)
```

```
t^8 + 2*t^6 + 2*t^4 + 2*t^2 + 1

sage: C = crystals.Tableaux(['A',2], shape=[5,2])
sage: C.q_dimension()
q^10 + 2*q^9 + 4*q^8 + 5*q^7 + 6*q^6 + 6*q^5
 + 6*q^4 + 5*q^3 + 4*q^2 + 2*q + 1

sage: C = crystals.Tableaux(['B',2], shape=[2,1])
sage: qdim = C.q_dimension(); qdim
q^10 + 2*q^9 + 3*q^8 + 4*q^7 + 5*q^6 + 5*q^5
 + 5*q^4 + 4*q^3 + 3*q^2 + 2*q + 1
sage: qdim == C.q_dimension(use_product=True)
True

sage: C = crystals.Tableaux(['D',4], shape=[2,1])
sage: C.q_dimension()
q^16 + 2*q^15 + 4*q^14 + 7*q^13 + 10*q^12 + 13*q^11
 + 16*q^10 + 18*q^9 + 18*q^8 + 18*q^7 + 16*q^6 + 13*q^5
 + 10*q^4 + 7*q^3 + 4*q^2 + 2*q + 1
```

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 + O(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 + O(q^10)
sage: qdim = C.q_dimension(); qdim
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 + 22*q^10 + O(x^11)
sage: qdim.compute_coefficients(15)
sage: qdim
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 + 22*q^10 + 27*q^11
 + 36*q^12 + 44*q^13 + 57*q^14 + 70*q^15 + O(x^16)
```

REFERENCES:

class HighestWeightCrystals.**TensorProducts**(*category*, **args*)

    Bases: `sage.categories.tensor.TensorProductsCategory`

The category of highest weight crystals constructed by tensor product of highest weight crystals.

class **ParentMethods**

>   Implements operations on tensor products of crystals.

>   **highest_weight_vectors**()

>>   Return the highest weight vectors of `self`.

>>   This works by using a backtracing algorithm since if $b_2 \otimes b_1$ is highest weight then b_1 is highest weight.

>>   EXAMPLES:
```
sage: C = crystals.Tableaux(['D',4], shape=[2,2])
sage: D = crystals.Tableaux(['D',4], shape=[1])
sage: T = crystals.TensorProduct(D, C)
sage: T.highest_weight_vectors()
([[[1]], [[1, 1], [2, 2]]],
 [[[3]], [[1, 1], [2, 2]]],
 [[[-2]], [[1, 1], [2, 2]]])
sage: L = filter(lambda x: x.is_highest_weight(), T)
sage: tuple(L) == T.highest_weight_vectors()
True
```

>>   TESTS:

>>   We check this works with Kashiwara's convention for tensor products:
```
sage: C = crystals.Tableaux(['B',3], shape=[2,2])
sage: D = crystals.Tableaux(['B',3], shape=[1])
sage: T = crystals.TensorProduct(D, C)
sage: T.global_options(convention='Kashiwara')
sage: T.highest_weight_vectors()
([[[1, 1], [2, 2]], [[1]]],
 [[[1, 1], [2, 2]], [[3]]],
 [[[1, 1], [2, 2]], [[-2]]])
sage: T.global_options.reset()
sage: T.highest_weight_vectors()
([[[1]], [[1, 1], [2, 2]]],
 [[[3]], [[1, 1], [2, 2]]],
 [[[-2]], [[1, 1], [2, 2]]])
```

>   HighestWeightCrystals.TensorProducts.**extra_super_categories**()

>>   EXAMPLES:
```
sage: HighestWeightCrystals().TensorProducts().extra_super_categories()
[Category of highest weight crystals]
```

HighestWeightCrystals.**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:
```
sage: HighestWeightCrystals().additional_structure()
```

HighestWeightCrystals.**example**()
> Returns an example of highest weight crystals, as per `Category.example()`.
>
> EXAMPLES:
> ```
> sage: B = HighestWeightCrystals().example(); B
> Highest weight crystal of type A_3 of highest weight omega_1
> ```

HighestWeightCrystals.**super_categories**()
> EXAMPLES:
> ```
> sage: HighestWeightCrystals().super_categories()
> [Category of crystals]
> ```

13.71 Hopf algebras

class sage.categories.hopf_algebras.**HopfAlgebras**(*base*, *name=None*)
> Bases: `sage.categories.category_types.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]
> ```
>
> TESTS:
> ```
> sage: TestSuite(HopfAlgebras(ZZ)).run()
> ```
>
> **class DualCategory**(*base*, *name=None*)
> > Bases: `sage.categories.category_types.Category_over_base_ring`
> >
> > The category of Hopf algebras constructed as dual of a Hopf algebra
> >
> > **class ParentMethods**
>
> **class** HopfAlgebras.**ElementMethods**
>
> > **antipode**()
> > > Returns the antipode 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
> > > sage: [a,b] = A.algebra_generators()
> > > sage: a, a.antipode()
> > > (B[(1,2,3)], B[(1,3,2)])
> > > sage: b, b.antipode()
> > > (B[(1,3)], B[(1,3)])
> > > ```
> > >
> > > TESTS:
> > > ```
> > > sage: all(x.antipode() * x == A.one() for x in A.basis())
> > > True
> > > ```
>
> **class** HopfAlgebras.**Morphism**(*s=None*)
> > Bases: `sage.categories.category.Category`

The category of Hopf algebra morphisms

class HopfAlgebras.**ParentMethods**

class HopfAlgebras.**Realizations**(*category*, **args*)

Bases: `sage.categories.realizations.RealizationsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

> **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)
> > sage: R = N.ribbon()
> > sage: R.antipode_by_coercion.__module__
> > 'sage.categories.hopf_algebras'
> > sage: R.antipode_by_coercion(R[1,3,1])
> > -R[2, 1, 2]
> > ```

class HopfAlgebras.**TensorProducts**(*category*, **args*)

Bases: `sage.categories.tensor.TensorProductsCategory`

The category of Hopf algebras constructed by tensor product of Hopf algebras

class ElementMethods

class HopfAlgebras.TensorProducts.**ParentMethods**

HopfAlgebras.TensorProducts.**extra_super_categories**()

> EXAMPLES:
> ```
> sage: C = HopfAlgebras(QQ).TensorProducts()
> sage: C.extra_super_categories()
> [Category of hopf algebras over Rational Field]
> sage: sorted(C.super_categories(), key=str)
> [Category of hopf algebras over Rational Field,
>  Category of tensor products of algebras over Rational Field,
>  Category of tensor products of coalgebras over Rational Field]
> ```

HopfAlgebras.**WithBasis**

> alias of `HopfAlgebrasWithBasis`

HopfAlgebras.**dual**()
> Returns 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
> ```

HopfAlgebras.**super_categories**()
> EXAMPLES:
> ```
> sage: HopfAlgebras(QQ).super_categories()
> [Category of bialgebras over Rational Field]
> ```

13.72 Hopf algebras with basis

class sage.categories.hopf_algebras_with_basis.**HopfAlgebrasWithBasis**(*base_category*)
> Bases: sage.categories.category_with_axiom.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 algebras with basis over Rational Field,
>  Category of coalgebras 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
> An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a p
> sage: A.__custom_name = "A"
> sage: A.category()
> Category of finite dimensional hopf algebras with basis over Rational Field
>
> sage: A.one_basis()
> ()
> sage: A.one()
> B[()]
>
> sage: A.base_ring()
> Rational Field
> sage: A.basis().keys()
> Dihedral group of order 6 as a permutation group
>
> sage: [a,b] = A.algebra_generators()
> sage: a, b
> (B[(1,2,3)], B[(1,3)])
> sage: a^3, b^2
> (B[()], B[()])
> sage: a*b
> ```

```
B[(1,2)]

sage: A.product            # todo: not quite ...
<bound method MyGroupAlgebra_with_category._product_from_product_on_basis_multiply of A>
sage: A.product(b,b)
B[()]

sage: A.zero().coproduct()
0
sage: A.zero().coproduct().parent()
A # A
sage: a.coproduct()
B[(1,2,3)] # B[(1,2,3)]

sage: TestSuite(A).run(verbose=True)
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_distributivity() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_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__
<class 'sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra_with_category'>
sage: A.element_class
<class 'sage.combinat.free_module.MyGroupAlgebra_with_category.element_class'>
```

Let us look at the code for implementing A:

```
sage: A??                           # todo: not implemented
```

TESTS:

```
sage: TestSuite(A).run()
sage: TestSuite(C).run()
```

class **ElementMethods**

HopfAlgebrasWithBasis.**FiniteDimensional**

---

alias of `FiniteDimensionalHopfAlgebrasWithBasis`

`HopfAlgebrasWithBasis.`**`Graded`**
alias of `GradedHopfAlgebrasWithBasis`

class `HopfAlgebrasWithBasis.`**`ParentMethods`**

`antipode``()`
The antipode of this Hopf algebra.

If `antipode_basis()` is available, this constructs the antipode morphism from `self` to `self` by extending it by linearity. Otherwise, `self.antipode_by_coercion()` is used, if available.

EXAMPLES:
```
sage: A = HopfAlgebrasWithBasis(ZZ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: [a,b] = A.algebra_generators()
sage: a, A.antipode(a)
(B[(1,2,3)], B[(1,3,2)])
sage: b, A.antipode(b)
(B[(1,3)], B[(1,3)])
```

TESTS:
```
sage: all(A.antipode(x) * x == A.one() for x in A.basis())
True
```

`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:
```
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: W = A.basis().keys(); W
Dihedral group of order 6 as a permutation group
sage: w = W.an_element(); w
(1,2,3)
sage: A.antipode_on_basis(w)
B[(1,3,2)]
```

class `HopfAlgebrasWithBasis.`**`TensorProducts`**`(category, *args)`
Bases: `sage.categories.tensor.TensorProductsCategory`

The category of hopf algebras with basis constructed by tensor product of hopf algebras with basis

class `ElementMethods`

class `HopfAlgebrasWithBasis.TensorProducts.`**`ParentMethods`**

`HopfAlgebrasWithBasis.TensorProducts.`**`extra_super_categories`**`()`
EXAMPLES:
```
sage: C = HopfAlgebrasWithBasis(QQ).TensorProducts()
sage: C.extra_super_categories()
[Category of hopf algebras with basis over Rational Field]
sage: sorted(C.super_categories(), key=str)
[Category of hopf algebras with basis over Rational Field,
```

```
          Category of tensor products of algebras with basis over Rational Field,
          Category of tensor products of hopf algebras over Rational Field]
```

HopfAlgebrasWithBasis.**example**(*G=None*)

Returns an example of algebra with basis:

```
sage: HopfAlgebrasWithBasis(QQ['x']).example()
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as
```

An other group can be specified as optional argument:

```
sage: HopfAlgebrasWithBasis(QQ).example(SymmetricGroup(4))
An example of Hopf algebra with basis: the group algebra of the Symmetric group of order 4!
```

13.73 Infinite Enumerated Sets

AUTHORS:

- Florent Hivert (2009-11): initial revision.

class sage.categories.infinite_enumerated_sets.**InfiniteEnumeratedSets**(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of infinite enumerated sets

An infinite enumerated sets is a countable set together with a canonical enumeration of its elements.

EXAMPLES:

```
sage: InfiniteEnumeratedSets()
Category of infinite enumerated sets
sage: InfiniteEnumeratedSets().super_categories()
[Category of enumerated sets, Category of infinite sets]
sage: InfiniteEnumeratedSets().all_super_categories()
[Category of infinite enumerated sets,
 Category of enumerated sets,
 Category of infinite sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

TESTS:

```
sage: C = InfiniteEnumeratedSets()
sage: TestSuite(C).run()
```

class **ParentMethods**

list()

Returns an error since self is an infinite enumerated set.

EXAMPLES:

```
sage: NN = InfiniteEnumeratedSets().example()
sage: NN.list()
Traceback (most recent call last):
...
NotImplementedError: infinite list
```

random_element()
  Returns an error since 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?

13.74 Integral domains

class sage.categories.integral_domains.**IntegralDomains**(*base_category*)
  Bases: `sage.categories.category_with_axiom.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
```

  TESTS:
```
sage: TestSuite(C).run()
```

  class **ElementMethods**

  class IntegralDomains.**ParentMethods**

  is_integral_domain()
    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
```

13.75 Lattice posets

class sage.categories.lattice_posets.**LatticePosets**(*s=None*)
  Bases: `sage.categories.category.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:
```
sage: LatticePosets()
Category of lattice posets
sage: LatticePosets().super_categories()
[Category of posets]
sage: LatticePosets().example()
NotImplemented
```

See also:

`Posets`, `FiniteLatticePosets`, `LatticePoset()`

TESTS:
```
sage: C = LatticePosets()
sage: TestSuite(C).run()
```

Finite
    alias of `FiniteLatticePosets`

class **ParentMethods**

>   **join**(*x*, *y*)
>   Returns the join of x and y in this lattice
>
>   INPUT:
>   •x, y – elements of `self`
>   EXAMPLES:
>        ```
>        sage: D = LatticePoset((divisors(60), attrcall("divides")))
>        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:
>        ```
>        sage: D = LatticePoset((divisors(30), attrcall("divides")))
>        sage: D.meet( D(6), D(15) )
>        3
>        ```

LatticePosets.**super_categories**()
    Returns a list of the (immediate) super categories of `self`, as per `Category.super_categories()`.

    EXAMPLES:
    ```
    sage: LatticePosets().super_categories()
    [Category of posets]
    ```

13.76 Left modules

class sage.categories.left_modules.**LeftModules**(*base*, *name=None*)
    Bases: `sage.categories.category_types.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 multiplation 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]
```

TESTS:
```
sage: TestSuite(LeftModules(ZZ)).run()
```

class ElementMethods

class LeftModules.**ParentMethods**

LeftModules.**super_categories**()
> EXAMPLES:
> ```
> sage: LeftModules(QQ).super_categories()
> [Category of commutative additive groups]
> ```

13.77 Magmas

class sage.categories.magmas.**Magmas**(*s=None*)
> Bases: `sage.categories.category_singleton.Category_singleton`

> The category of (multiplicative) magmas.

> A magma is a set with a binary operation $*$.

> EXAMPLES:
> ```
> sage: Magmas()
> Category of magmas
> sage: Magmas().super_categories()
> [Category of sets]
> sage: Magmas().all_super_categories()
> [Category of magmas, Category of sets,
>  Category of sets with partial maps, Category of objects]
> ```

> The following axioms are defined by this category:
> ```
> sage: Magmas().Associative()
> Category of semigroups
> sage: Magmas().Unital()
> Category of unital magmas
> sage: Magmas().Commutative()
> Category of commutative magmas
> sage: Magmas().Unital().Inverse()
> Category of inverse unital magmas
> sage: Magmas().Associative()
> Category of semigroups
> sage: Magmas().Associative().Unital()
> Category of monoids
> sage: Magmas().Associative().Unital().Inverse()
> Category of groups
> ```

> TESTS:

```
sage: C = Magmas()
sage: TestSuite(C).run()
```

class **Algebras**(*category*, **args*)

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

extra_super_categories()

EXAMPLES:

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

This implements the fact that the algebra of a commutative magma is commutative:

```
sage: Magmas().Commutative().Algebras(QQ).super_categories()
[Category of magma algebras over Rational Field, Category of commutative magmas]
```

In particular, commutative monoid algebras are commutative algebras:

```
sage: Monoids().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

Magmas.**Associative**

alias of `Semigroups`

class Magmas.**CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

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);
sage: x = cartesian_product([A([1,3,2]), A([2,3,1])])
sage: y = cartesian_product([A([1,3,2]), A([2,3,1])])
sage: cartesian_product([A,A]).product(x,y)
B[(0, [1, 2, 3])] + B[(1, [3, 1, 2])]
sage: x*y
B[(0, [1, 2, 3])] + B[(1, [3, 1, 2])]
```

Magmas.CartesianProducts.**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 rings ...
sage: sorted(C.category().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Asso

sage: TestSuite(C).run()
```

Magmas.CartesianProducts.**extra_super_categories**()

    This implements the fact that a subquotient (and therefore a quotient or subobject) of a finite set is finite.

    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 Magmas.**Commutative**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

    TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class **Algebras**(*category*, **args*)

    Bases: `sage.categories.algebra_functor.AlgebrasCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:      _functor_category = "FooBars"
....:      _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

extra_super_categories()
    EXAMPLES:
      sage: Magmas().Commutative().Algebras(QQ).extra_super_categories() [Category of commutative magmas]
    This implements the fact that the algebra of a commutative magma is commutative:

```
sage: Magmas().Commutative().Algebras(QQ).super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

    In particular, commutative monoid algebras are commutative algebras:

```
sage: Monoids().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

class Magmas.Commutative.**ParentMethods**

    is_commutative()
        Return `True`, since commutative magmas are commutative.

        EXAMPLES:
```
sage: Parent(QQ,category=CommutativeRings()).is_commutative()
True
```

class Magmas.**ElementMethods**

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
```

Magmas.**FinitelyGeneratedAsMagma**
>   alias of `FinitelyGeneratedMagmas`

class Magmas.**ParentMethods**

>   **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.
>   ___

>   INPUTS:
>   •`names` - the type of names used
>   –`'letters'` - lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by `column_keys()`, padded to a common width with leading 'a's.
>   –`'digits'` - base 10 representation of the elements' positions in the list given by `column_keys()`, padded to a common width with leading zeros.
>   –`'elements'` - the string representations of the elements themselves.
>   –a list - a list of strings, where the length of the list equals the number of elements.
>   •`elements` - default = `None`. A list of elements of the magma, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering the `S.list` method returns. Or the `elements` can be a subset which is closed under the operation. In particular, this can be used when the base set is infinite.
>   OUTPUT: The multiplication table as an object of the class `OperationTable` which defines several methods for manipulating and displaying the table. See the documentation there for full details to supplement the documentation here.

>   EXAMPLES:

>   The default is to represent elements as lowercase ASCII letters.
>       ```
>       sage: G=CyclicPermutationGroup(5)
>       sage: G.multiplication_table()
>       *  a b c d e
>        +----------
>       a| a b c d e
>       b| b c d e a
>       c| c d e a b
>       d| d e a b c
>       e| e a b c d
>       ```

>   All that is required is that an algebraic structure has a multiplication defined. A `LeftRegularBand` is an example of a finite semigroup. The `names` argument allows displaying the elements in different ways.
>       ```
>       sage: from sage.categories.examples.finite_semigroups import LeftRegularBand
>       sage: L=LeftRegularBand(('a','b'))
>       sage: T=L.multiplication_table(names='digits')
>       sage: T.column_keys()
>       ('a', 'b', 'ab', 'ba')
>       sage: T
>       *  0 1 2 3
>        +--------
>       0| 0 2 2 2
>       1| 3 1 3 3
>       ```

```
2| 2 2 2 2
3| 3 3 3 3
```

Specifying the elements in an alternative order can provide more insight into how the operation behaves.

```
sage: L=LeftRegularBand(('a','b','c'))
sage: elts = sorted(L.list())
sage: L.multiplication_table(elements=elts)
*  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 c d d e e e
b| b b c 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 c
d| d e e d e e e e e e d d 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 j j i j
g| g g h 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
i| j j j j j i j j j i j i j j i j
j| j j j j j j j j j j j j j j j
k| l m m l m n o o n o k l m n o
l| l m m l m m m m m m l l m m m
m| m m m m m m m m m m m m m m m
n| o o o o o n o o n o n o o n o
o| o o o o o o o o o o o o o o o
```

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: L=LeftRegularBand(('a','b','c'))
sage: elts=['a', 'c', 'ac', 'ca']
sage: L.multiplication_table(names='elements', elements=elts)
   *  'a'  'c' 'ac' 'ca'
   +-------------------
 'a'|  'a' 'ac' 'ac' 'ac'
 'c'| 'ca'  'c' 'ca' 'ca'
'ac'| 'ac' 'ac' 'ac' 'ac'
'ca'| 'ca' 'ca' 'ca' 'ca'
```

The table returned can be manipulated in various ways. See the documentation for `OperationTable` for more comprehensive documentation.

```
sage: G=AlternatingGroup(3)
sage: T=G.multiplication_table()
sage: T.column_keys()
((), (1,2,3), (1,3,2))
sage: sorted(T.translation().items())
[('a', ()), ('b', (1,2,3)), ('c', (1,3,2))]
sage: T.change_names(['x', 'y', 'z'])
sage: sorted(T.translation().items())
[('x', ()), ('y', (1,2,3)), ('z', (1,3,2))]
sage: T
*  x y z
 +------
x| x y z
y| y z x
z| z x y
```

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 `Magmas()` must either implement `product()` in the parent class or `_mul_` in the element class. By default, the addition method on elements `x._mul_(y)` calls `S.product(x,y)`, and reciprocally.

    As a bonus, `S.product` models the binary function from `S` to `S`:

```
sage: bin = S.product
sage: bin(x,y)
'ab'
```

    Currently, `S.product` is just a bound method:

```
sage: bin
<bound method FreeSemigroup_with_category.product of An example of a semigroup: the free
```

    When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich `S.product` with extra mathematical structure. This will typically be implemented using lazy attributes.:

```
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 `Magmas()` must either implement `product()` in the parent class or `_mul_` in the element class. By default, the addition method on elements `x._mul_(y)` calls `S.product(x,y)`, and reciprocally.

    As a bonus, `S.product` models the binary function from `S` to `S`:

```
sage: bin = S.product
sage: bin(x,y)
'ab'
```

    Currently, `S.product` is just a bound method:

```
sage: bin
<bound method FreeSemigroup_with_category.product of An example of a semigroup: the free
```

When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich `S.product` with extra mathematical structure. This will typically be implemented using lazy attributes.:

```
sage: bin                       # todo: not implemented
Generic binary morphism:
From: (S x S)
To:   S
```

class Magmas.**Realizations**(*category*, **args*)

    Bases: `sage.categories.realizations.RealizationsCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    class **ParentMethods**

        product_by_coercion(*left*, *right*)

            Default implementation of product for realizations.

            This method coerces to the realization specified by `self.realization_of().a_realization()`, computes the product in that realization, and then coerces back.

            EXAMPLES:

```
sage: Out = Sets().WithRealizations().example().Out(); Out
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: Out.product
<bound method SubsetAlgebra.Out_with_category.product_by_coercion of The subset algeb
sage: Out.product.__module__
'sage.categories.magmas'
sage: x = Out.an_element()
sage: y = Out.an_element()
sage: Out.product(x, y)
Out[{}] + 4*Out[{1}] + 9*Out[{2}] + Out[{1, 2}]
```

class Magmas.**SubcategoryMethods**

    Associative()

        Return the full subcategory of the associative objects of `self`.

A (multiplicative) `magma` `Magmas` M is *associative* if, for all $x, y, z \in M$,

$$x * (y * z) = (x * y) * z$$

See also:

Wikipedia article Associative_property

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

TESTS:
```
sage: TestSuite(Magmas().Associative()).run()
sage: Rings().Associative.__module__
'sage.categories.magmas'
```

Commutative()
Return the full subcategory of the commutative objects of `self`.

A (multiplicative) `magma` `Magmas` M is *commutative* if, for all $x, y \in M$,

$$x * y = y * x$$

See also:

Wikipedia article Commutative_property

EXAMPLES:
```
sage: Magmas().Commutative()
Category of commutative magmas
sage: Monoids().Commutative()
Category of commutative monoids
```

TESTS:
```
sage: TestSuite(Magmas().Commutative()).run()
sage: Rings().Commutative.__module__
'sage.categories.magmas'
```

Distributive()
Return the full subcategory of the objects of `self` where $*$ is distributive on $+$.

INPUT:

• `self` – a subcategory of `Magmas` and `AdditiveMagmas`

Given that Sage does not yet know that the category MagmasAndAdditiveMagmas is the intersection of the categories `Magmas` and `AdditiveMagmas`, the method `MagmasAndAdditiveMagmas.SubcategoryMethods.Distributive()` is not available, as would be desirable, for this intersection.

This method is a workaround. It checks that `self` is a subcategory of both `Magmas` and `AdditiveMagmas` and upgrades it to a subcategory of MagmasAndAdditiveMagmas before applying the axiom. It complains overwise, since the `Distributive` axiom does not make sense for a plain magma.

EXAMPLES:
```
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 a
sage: Semigroups().Distributive()
Traceback (most recent call last):
...
ValueError: The distributive axiom only makes sense on a magma which is simultaneously a
```

TESTS:

```
sage: Semigroups().Distributive.__module__
'sage.categories.magmas'
sage: Rings().Distributive.__module__
'sage.categories.magmas_and_additive_magmas'
```

FinitelyGenerated()

Return the subcategory of the objects of self that are endowed with a distinguished finite set of (multiplicative) magma generators.

EXAMPLES:

This is a shorthand for `FinitelyGeneratedAsMagma()`, which see:

```
sage: Magmas().FinitelyGenerated()
Category of finitely generated magmas
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated groups
```

An error is raised if this is ambiguous:

```
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:

```
sage: Magmas().FinitelyGeneratedAsMagma()
Category of finitely generated magmas
```

Being finitely generated does depend on the structure: for a ring, being finitely generated as a magma, as an additive magma, or as a ring are different concepts. Hence the name of this axiom is explicit:

```
sage: Rings().FinitelyGeneratedAsMagma()
Category of finitely generated as magma rings
```

On the other hand, it does not depend on the multiplicative structure: for example a group is finitely generated if and only if it is finitely generated as a magma. A short hand is provided when there is no ambiguity, and the output tries to reflect that:

```
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated groups
```

```
sage: Semigroups().FinitelyGenerated().axioms()
frozenset({'Associative', '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.

TESTS:

```
sage: TestSuite(Magmas().FinitelyGeneratedAsMagma()).run()
sage: Semigroups().FinitelyGeneratedAsMagma.__module__
'sage.categories.magmas'
```

Unital()

Return the subcategory of the unital objects of `self`.

A (multiplicative) `magma` `Magmas` M is *unital* if it admits an element 1, called *unit*, such that for all $x \in M$,

$$1 * x = x * 1 = x$$

This element is necessarily unique, and should be provided as `M.one()`.

See also:

Wikipedia article Unital_magma#unital

EXAMPLES:

```
sage: Magmas().Unital()
Category of unital magmas
sage: Semigroups().Unital()
Category of monoids
sage: Monoids().Unital()
Category of monoids
sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: AssociativeAlgebras(QQ).Unital()
Category of algebras over Rational Field
```

TESTS:

```
sage: TestSuite(Magmas().Unital()).run()
sage: Semigroups().Unital.__module__
'sage.categories.magmas'
```

class Magmas.**Subquotients**(*category*, **args*)

Bases: `sage.categories.subquotients.SubquotientsCategory`

The category of subquotient magmas.

See `Sets.SubcategoryMethods.Subquotients()` for the general setup for subquotients. In the case of a subquotient magma S of a magma G, the condition that r be a morphism in `As` can be rewritten as follows:

- for any two $a, b \in S$ the identity $a \times_S b = r(l(a) \times_G l(b))$ holds.

This is used by this category to implement the product $\times_S$ of S from l and r and the product of G.

EXAMPLES:
```
sage: Semigroups().Subquotients().all_super_categories()
[Category of subquotients of semigroups, Category of semigroups,
 Category of subquotients of magmas, Category of magmas,
 Category of subquotients of sets, Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

class **ParentMethods**

    product (*x*, *y*)

        Return the product of two elements of `self`.

        EXAMPLES:
```
sage: S = Semigroups().Subquotients().example()
sage: S
An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup
sage: S.product(S(19), S(3))
19
```

        Here is a more elaborate example involving a sub algebra:
```
sage: Z = SymmetricGroup(5).algebra(QQ).center()
sage: B = Z.basis()
sage: B[3] * B[2]
4*B[2] + 6*B[3] + 5*B[6]
```

class Magmas.**Unital** (*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

    TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class **Algebras** (*category*, **args*)

    Bases: `sage.categories.algebra_functor.AlgebrasCategory`

    TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructio
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
```

```
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

extra_super_categories()
    EXAMPLES:
      sage: Magmas().Commutative().Algebras(QQ).extra_super_categories() [Category of commutative magmas]
    This implements the fact that the algebra of a commutative magma is commutative:

```
sage: Magmas().Commutative().Algebras(QQ).super_categories()
[Category of magma algebras over Rational Field,
 Category of commutative magmas]
```

    In particular, commutative monoid algebras are commutative algebras:

```
sage: Monoids().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

class Magmas.Unital.**CartesianProducts**(*category*, **args*)
    Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructio
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

    class ElementMethods

    class Magmas.Unital.CartesianProducts.**ParentMethods**

        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)
```

    Magmas.Unital.CartesianProducts.**extra_super_categories**()
      Implement the fact that a cartesian product of unital magmas is a unital magma

EXAMPLES:
```
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 Magmas.Unital.**Inverse**(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

class CartesianProducts(*category*, **args*)

Bases: sage.categories.cartesian_product.CartesianProductsCategory

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruc
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a p
sage: TestSuite(C).run()
```

extra_super_categories()

Implement the fact that a cartesian product of magmas with inverses is a magma with inverse.

EXAMPLES:
```
sage: C = Magmas().Unital().Inverse().CartesianProducts()
sage: C.extra_super_categories();
[Category of inverse unital magmas]
sage: sorted(C.axioms())
['Inverse', 'Unital']
```

class Magmas.Unital.**ParentMethods**

is_empty()

Return whether self is empty.

Since this set is a unital magma it is not empty and this method always return False.

EXAMPLES:
```
sage: S = SymmetricGroup(2)
sage: S.is_empty()
False

sage: M = Monoids().example()
sage: M.is_empty()
False
```

TESTS:
```
sage: S.is_empty.__module__
'sage.categories.magmas'
sage: M.is_empty.__module__
'sage.categories.magmas'
```

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 Magmas.Unital.**Realizations**(*category*, **args*)

Bases: `sage.categories.realizations.RealizationsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

class ParentMethods

one()

Return the unit element of `self`.
```
sage:    from sage.combinat.root_system.extended_affine_weyl_group  import  Ex-
tendedAffineWeylGroup sage:  PvW0 = ExtendedAffineWeylGroup(['A',2,1]).PvW0()
sage: PvW0 in Magmas().Unital().Realizations() True sage: PvW0.one() 1
```

class Magmas.Unital.**SubcategoryMethods**

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
> ```
>
> TESTS:
> ```
> sage: TestSuite(Magmas().Unital().Inverse()).run()
> sage: Algebras(QQ).Inverse.__module__
> 'sage.categories.magmas'
> ```

Magmas.Unital.**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
> ```

Magmas.**super_categories**()

> EXAMPLES:
> ```
> sage: Magmas().super_categories()
> [Category of sets]
> ```

13.78 Magmas and Additive Magmas

class sage.categories.magmas_and_additive_magmas.**MagmasAndAdditiveMagmas**(*s=None*)

> Bases: `sage.categories.category_singleton.Category_singleton`
>
> The category of sets $(S, +, *)$ with an additive operation '+' and a multiplicative operation $*$
>
> 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(
> Category of rngs
> ```

```
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital(
Category of semirings
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital(
Category of rings
```

This category is really meant to represent the intersection of the categories of `Magmas` and `AdditiveMagmas`; however Sage's infrastructure does not allow yet to model this:

```
sage: Magmas() & AdditiveMagmas()
Join of Category of magmas and Category of additive magmas
```

```
sage: Magmas() & AdditiveMagmas()          # todo: not implemented
Category of magmas and additive magmas
```

TESTS:
```
sage: TestSuite(MagmasAndAdditiveMagmas()).run()
```

Distributive
    alias of `DistributiveMagmasAndAdditiveMagmas`

class **SubcategoryMethods**

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 $x, y, z \in M$,

    $$x * (y + z) = x * y + x * z \text{ and } (x + y) * z = x * z + y * z$$

    EXAMPLES:
    ```
    sage: from sage.categories.magmas_and_additive_magmas 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:
    ```
    sage: Magmas() & AdditiveMagmas()
    Join of Category of magmas and Category of additive magmas
    ```

    Still, the natural syntax works:
    ```
    sage: (Magmas() & AdditiveMagmas()).Distributive()
    Category of distributive magmas and additive magmas
    ```

    thanks to a workaround implemented in `Magmas.SubcategoryMethods.Distributive()`:
    ```
    sage: (Magmas() & AdditiveMagmas()).Distributive.__module__
    'sage.categories.magmas'
    ```

    TESTS:
    ```
    sage: TestSuite(C).run()
    sage: Fields().Distributive.__module__
    'sage.categories.magmas_and_additive_magmas'
    ```

MagmasAndAdditiveMagmas.**additional_structure**()
    Return `None`.

Indeed, this category is meant to represent the join of `AdditiveMagmas` and `Magmas`. As such, it defines no additional structure.

See also:

`Category.additional_structure()`

EXAMPLES:
```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: MagmasAndAdditiveMagmas().additional_structure()
```

MagmasAndAdditiveMagmas.**super_categories**()
    EXAMPLES:
```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: MagmasAndAdditiveMagmas().super_categories()
[Category of magmas, Category of additive magmas]
```

13.79 Non-unital non-associative algebras

class sage.categories.magmatic_algebras.**MagmaticAlgebras**(*base*, *name=None*)
    Bases: `sage.categories.category_types.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 trac ticket #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 a
 Category of modules over Integer Ring]
```

    TESTS:
```
sage: TestSuite(C).run()
```

    Associative
        alias of `AssociativeAlgebras`

    class **ParentMethods**

        algebra_generators()
            Return a family of generators of this algebra.

            EXAMPLES:
```
sage: F = AlgebrasWithBasis(QQ).example(); F
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') o
sage: F.algebra_generators()
Family (B[word: a], B[word: b], B[word: c])
```

MagmaticAlgebras.**Unital**
> alias of UnitalAlgebras

class MagmaticAlgebras.**WithBasis**(*base_category*)
> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

> TESTS:
> ```
> sage: C = Modules(ZZ).FiniteDimensional(); C
> Category of finite dimensional modules over Integer Ring
> sage: type(C)
> <class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
> sage: type(C).__base__.__base__
> <class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>
>
> sage: TestSuite(C).run()
> ```

> **class ParentMethods**

>> **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()
>>> Finite family {{{-1, 1}}: P[{{-1, 1}}], {{-1}, {1}}: P[{{-1}, {1}}]}
>>> ```

>> **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()
>>> • _multiply() or _multiply_basis()
>>> • product_by_coercion()

>>> EXAMPLES:
>>> ```
>>> sage: A = AlgebrasWithBasis(QQ).example()
>>> sage: a, b, c = A.algebra_generators()
>>> sage: A.product(a + 2*b, 3*c)
>>> 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:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: Word = A.basis().keys()
sage: A.product_on_basis(Word("abc"),Word("cba"))
B[word: abccba]
```

MagmaticAlgebras.**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 compability between the magma and module structure.

EXAMPLES:

```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(ZZ).additional_structure()
```

MagmaticAlgebras.**super_categories**()

EXAMPLES:

```
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(ZZ).super_categories()
[Category of additive commutative additive associative additive unital distributive magmas a

sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: MagmaticAlgebras(ZZ).is_subcategory((AdditiveSemigroups() & Magmas()).Distributive())
True
```

13.80 Matrix algebras

class sage.categories.matrix_algebras.**MatrixAlgebras**(*base*, *name=None*)

Bases: `sage.categories.category_types.Category_over_base_ring`

The category of matrix algebras over a field.

EXAMPLES:

```
sage: MatrixAlgebras(RationalField())
Category of matrix algebras over Rational Field
```

TESTS:

```
sage: TestSuite(MatrixAlgebras(ZZ)).run()
```

super_categories()

EXAMPLES:

```
sage: MatrixAlgebras(QQ).super_categories()
[Category of algebras over Rational Field]
```

13.81 Modular abelian varieties

class sage.categories.modular_abelian_varieties.**ModularAbelianVarieties**(*Y*)

Bases: sage.categories.category_types.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: sage.categories.homsets.HomsetsCategory

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class Endset(*base_category*)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

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]
```

ModularAbelianVarieties.**base_field**()

EXAMPLES:

```
sage: ModularAbelianVarieties(QQ).base_field()
Rational Field
```

ModularAbelianVarieties.**super_categories**()
    EXAMPLES:
```
sage: ModularAbelianVarieties(QQ).super_categories()
[Category of sets]
```

13.82 Modules

class sage.categories.modules.**Modules**(*base*, *name=None*)
    Bases: `sage.categories.category_types.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:

    $$r * (x * s) = (r * x) * s \qquad \forall r, s \in R \text{ and } x \in M$$

    INPUT:

    • base_ring – a ring R or subcategory of `Rings()`

    • dispatch – a boolean (for internal use; default: `True`)

    When the base ring is a field, the category of vector spaces is returned instead (unless `dispatch == False`).

    > **Warning:** Outside of the context of symmetric modules over a commutative ring, the specifications of this category are fuzzy and not yet set in stone (see below). The code in this category and its subcategories is therefore prone to bugs or arbitrary limitations in this case.

    EXAMPLES:
```
sage: Modules(ZZ)
Category of modules over Integer Ring
sage: Modules(QQ)
Category of vector spaces over Rational Field

sage: Modules(Rings())
Category of modules over rings
sage: Modules(FiniteFields())
Category of vector spaces over finite 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

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
```

TESTS:
```
sage: TestSuite(Modules(ZZ)).run()
```

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*:

$$r * x = x * r \qquad \forall r \in R \text{ and } x \in M$$

- 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. trac ticket #15916).

---

class **ElementMethods**

class Modules.**FiniteDimensional**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

    TESTS:
```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

    extra_super_categories()

        Implement the fact that a finite dimensional module over a finite ring is finite.

        EXAMPLES:
```
sage: Modules(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
[Category of finite sets]
sage: Modules(ZZ).FiniteDimensional().extra_super_categories()
[]
sage: Modules(GF(5)).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: Modules(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
False
```

Modules.**Graded**

    alias of `GradedModules`

class Modules.**Homsets**(*category*, **args*)

    Bases: `sage.categories.homsets.HomsetsCategory`

The category of homomorphism sets $\hom(X, Y)$ for X, Y modules.

---

class **Endset** (*base_category*)

> Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`
>
> The category of endomorphism sets $End(X)$ for X a module (this is not used yet)
>
> **extra_super_categories**()
>
> > Implement the fact that the endomorphism set of a module is an algebra.
> >
> > **See also:**
> >
> > `CategoryWithAxiom.extra_super_categories()`
> >
> > EXAMPLES:
> > ```
> > sage: Modules(ZZ).Endsets().extra_super_categories()
> > [Category of magmatic algebras over Integer Ring]
> >
> > sage: End(ZZ^3) in Algebras(ZZ)
> > True
> > ```

class Modules.Homsets.**ParentMethods**

> **base_ring**()
>
> > Return the base ring of `self`.
> >
> > EXAMPLES:
> > ```
> > sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
> > sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
> > sage: H = Hom(E, F)
> > sage: H.base_ring()
> > Integer Ring
> > ```
> >
> > This `base_ring` method is actually overridden by `sage.structure.category_object.CategoryObj`
> > ```
> > sage: H.base_ring.__module__
> > ```
> >
> > Here we call it directly:
> > ```
> > sage: method = H.category().parent_class.base_ring
> > sage: method.__get__(H)()
> > Integer Ring
> > ```
>
> **zero**()
>
> > EXAMPLES:
> > ```
> > sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
> > sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
> > sage: H = Hom(E, F)
> > sage: f = H.zero()
> > sage: f
> > Generic morphism:
> >   From: Free module generated by {1, 2, 3} over Integer Ring
> >   To:   Free module generated by {2, 3, 4} over Integer Ring
> > sage: f(E.monomial(2))
> > 0
> > sage: f(E.monomial(3)) == F.zero()
> > True
> > ```
> >
> > TESTS:
> >
> > We check that `H.zero()` is picklable:
> > ```
> > sage: loads(dumps(f.parent().zero()))
> > Generic morphism:
> > ```

```
From: Free module generated by {1, 2, 3} over Integer Ring
To:   Free module generated by {2, 3, 4} over Integer Ring
```

Modules.Homsets.**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.

Modules.Homsets.**extra_super_categories**()

> EXAMPLES:
```
sage: Modules(ZZ).Homsets().extra_super_categories()
[Category of modules over Integer Ring]
```

class Modules.**ParentMethods**

class Modules.**SubcategoryMethods**

> **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 trac ticket #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:
```
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```
>
> The dual of an algebra is a coalgebra:

```
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:
```
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
```

TESTS:
```
sage: C = VectorSpaces(QQ).DualObjects()
sage: C.base_category()
Category of vector spaces over Rational Field
sage: C.super_categories()
[Category of vector spaces over Rational Field]
sage: latex(C)
\mathbf{DualObjects}(\mathbf{VectorSpaces}_{\Bold{Q}})
sage: TestSuite(C).run()
```

FiniteDimensional()

    Return the full subcategory of the finite dimensional objects of `self`.

    EXAMPLES:
```
sage: Modules(ZZ).FiniteDimensional()
Category of finite dimensional modules over Integer Ring
sage: Coalgebras(QQ).FiniteDimensional()
Category of finite dimensional coalgebras over Rational Field
sage: AlgebrasWithBasis(QQ).FiniteDimensional()
Category of finite dimensional algebras with basis over Rational Field
```

    TESTS:
```
sage: TestSuite(Modules(ZZ).FiniteDimensional()).run()
sage: Coalgebras(QQ).FiniteDimensional.__module__
'sage.categories.modules'
```

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()
Join of Category of graded modules over Rational Field and Category of coalgebras over R

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.

TESTS:
```
sage: Coalgebras(QQ).Graded.__module__
'sage.categories.modules'
```

TensorProducts()

> Return the full subcategory of objects of `self` constructed as tensor products.
>
> **See also:**
>
> > • `tensor.TensorProductsCategory`
> > • `RegressiveCovariantFunctorialConstruction`.
>
> EXAMPLES:
> ```
> sage: ModulesWithBasis(QQ).TensorProducts()
> Category of tensor products of vector spaces with basis over Rational Field
> ```

WithBasis()

> Return the full subcategory of the objects of `self` with a distinguished basis.
>
> EXAMPLES:
> ```
> 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
> ```
>
> TESTS:
> ```
> sage: TestSuite(Modules(ZZ).WithBasis()).run()
> sage: Coalgebras(QQ).WithBasis.__module__
> 'sage.categories.modules'
> ```

base_ring()

> Return the base ring (category) for `self`.
>
> This implements a `base_ring` method for all subcategories of `Modules(K)`.
>
> EXAMPLES:
> ```
> 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()
> sage: C.base_ring.__module__
> 'sage.categories.modules'
> sage: C.base_ring()
> Rational Field
> ```

```
sage: C = QuasiSymmetricFunctions(QQ).F().category()
sage: C.base_ring.__module__
'sage.categories.modules'
sage: C.base_ring()
Rational Field
```

dual()

Return the category of spaces constructed as duals of spaces of `self`.

The *dual* of a vector space V is the space consisting of all linear functionals on V (see Wikipedia article Dual_space). Additional structure on V can endow its dual with additional structure; for example, if V is a finite dimensional algebra, then its dual is a coalgebra.

This returns the category of spaces constructed as dual of spaces in `self`, endowed with the appropriate additional structure.

> **Warning:**
> - This semantic of `dual` and `DualObject` is imposed on all subcategories, in particular to make `dual` a covariant functorial construction.
>   A subcategory that defines a different notion of dual needs to use a different name.
> - Typically, the category of graded modules should define a separate `graded_dual` construction (see trac ticket #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:
```
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```

The dual of an algebra is a coalgebra:
```
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:
```
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
```

TESTS:
```
sage: C = VectorSpaces(QQ).DualObjects()
sage: C.base_category()
Category of vector spaces over Rational Field
sage: C.super_categories()
[Category of vector spaces over Rational Field]
```

```
sage: latex(C)
\mathbf{DualObjects}(\mathbf{VectorSpaces}_{\Bold{Q}})
sage: TestSuite(C).run()
```

Modules.**WithBasis**

  alias of ModulesWithBasis

Modules.**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()
  ```

Modules.**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]
  ```

13.83 Modules With Basis

AUTHORS:

- Nicolas M. Thiery (2008-2014): initial revision, axiomatization

- Jason Bandlow and Florent Hivert (2010): Triangular Morphisms

- Christian Stump (2010): trac ticket #9648 module_morphism's to a wider class of codomains

class sage.categories.modules_with_basis.**ModulesWithBasis**(*base_category*)

  Bases: sage.categories.category_with_axiom.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:
  ```
  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:

```
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 $Hom(X, Y)$:

```
sage: X = CombinatorialFreeModule(QQ, [1,2]); X.__custom_name = "X"
sage: Y = CombinatorialFreeModule(QQ, [3,4]); Y.__custom_name = "Y"
sage: H = Hom(X, Y); H
Set of Morphisms from X to Y in Category of finite dimensional vector spaces with basis over Rat
```

The simplest morphism is the zero map:

```
sage: H.zero()              # todo: move this test into module once we have an example
Generic morphism:
  From: X
  To:   Y
```

which we can apply to elements of X:

```
sage: x = X.monomial(1) + 3 * X.monomial(2)
sage: H.zero()(x)
0
```

TESTS:

```
sage: f = H.zero().on_basis()
sage: f(1)
0
sage: f(2)
0
```

EXAMPLES:

We now construct a more interesting morphism by extending a function by linearity:

```
sage: phi = H(on_basis = lambda i: Y.monomial(i+2)); phi
Generic morphism:
  From: X
  To:   Y
sage: phi(x)
B[3] + 3*B[4]
```

We can retrieve the function acting on indices of the basis:

```
sage: f = phi.on_basis()
sage: f(1), f(2)
(B[3], B[4])
```

$Hom(X, Y)$ 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 `FiniteDimensionalModulesWithBasis` and `GradedModulesWithBasis`:

```
sage: H in ModulesWithBasis(QQ), H in Modules(QQ)
(False, True)
```

Some more playing around with categories and higher order homsets:

```
sage: H.category()
Category of homsets of modules with basis over Rational Field
sage: Hom(H, H).category()
Category of endsets of homsets of modules with basis over Rational Field
```

Todo

End(X) is an algebra.

TESTS:

```
sage: TestSuite(ModulesWithBasis(ZZ)).run()
```

class **CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

The category of modules with basis constructed by cartesian products of modules with basis.

class **ParentMethods**

ModulesWithBasis.CartesianProducts.**extra_super_categories**()

EXAMPLES:

```
sage: ModulesWithBasis(QQ).CartesianProducts().extra_super_categories()
[Category of vector spaces with basis over Rational Field]
sage: ModulesWithBasis(QQ).CartesianProducts().super_categories()
[Category of Cartesian products of modules with basis over Rational Field,
 Category of vector spaces with basis over Rational Field,
 Category of Cartesian products of vector spaces over Rational Field]
```

class ModulesWithBasis.**DualObjects**(*category*, **args*)

Bases: `sage.categories.dual.DualObjectsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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 wi
```

class ModulesWithBasis.**ElementMethods**

leading_coefficient(*cmp=None*)

Returns 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 function, cmp(x,y), can be provided. This should return a negative value if x < y, 0 if x == y and a positive value if x > y.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_coefficient()
1
sage: def cmp(x,y): return y-x
sage: x.leading_coefficient(cmp=cmp)
3

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.leading_coefficient()
-5
```

leading_item(*cmp=None*)

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, cmp(x,y), can be provided. This should return a negative value if x < y, 0 if x == y and a positive value if x > y.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)
sage: x.leading_item()
(3, 4)
sage: def cmp(x,y): return y-x
sage: x.leading_item(cmp=cmp)
(1, 3)

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.leading_item()
([3], -5)
```

leading_monomial(*cmp=None*)

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 function, `cmp(x,y)`, can be provided. This should return a negative value if
`x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_monomial()
B[3]
sage: def cmp(x,y): return y-x
sage: x.leading_monomial(cmp=cmp)
B[1]

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.leading_monomial()
s[3]
```

leading_support(*cmp=None*)

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 function, `cmp(x,y)`,
can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if
`x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)
sage: x.leading_support()
3
sage: def cmp(x,y): return y-x
sage: x.leading_support(cmp=cmp)
1

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.leading_support()
[3]
```

leading_term(*cmp=None*)

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 function, `cmp(x,y)`, can be provided. This should return a negative value if `x < y`,
0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_term()
B[3]
sage: def cmp(x,y): return y-x
sage: x.leading_term(cmp=cmp)
3*B[1]

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.leading_term()
```

```
-5*s[3]
```

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 `self.parent()` obtained by applying the function `f` to all of the coefficients of `self`.
>
> EXAMPLES:
> ```
> sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
> sage: B = F.basis()
> sage: f = B['a'] - 3*B['c']
> sage: f.map_coefficients(lambda x: x+5)
> 6*B['a'] + 2*B['c']
> ```
>
> Killed coefficients are handled properly:
> ```
> sage: f.map_coefficients(lambda x: 0)
> 0
> sage: list(f.map_coefficients(lambda x: 0))
> []
>
> sage: s = SymmetricFunctions(QQ).schur()
> sage: a = s([2,1])+2*s([3,2])
> sage: a.map_coefficients(lambda x: x*2)
> 2*s[2, 1] + 4*s[3, 2]
> ```

map_item(*f*)

> Mapping a function on items.
>
> INPUT:
>
> > •f – a function mapping pairs (index, coeff) to other such pairs
>
> Return a new element of `self.parent()` obtained by applying the function *f* to all items (index, coeff) of `self`.
>
> EXAMPLES:
> ```
> sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])
> sage: x = B.an_element(); x
> 2*B[-1] + 2*B[0] + 3*B[1]
> sage: x.map_item(lambda i, c: (-i, 2*c))
> 6*B[-1] + 4*B[0] + 4*B[1]
> ```
>
> f needs not be injective:
> ```
> sage: x.map_item(lambda i, c: (1, 2*c))
> 14*B[1]
>
> sage: s = SymmetricFunctions(QQ).schur()
> sage: f = lambda m,c: (m.conjugate(), 2*c)
> sage: a = s([2,1]) + s([1,1,1])
> sage: a.map_item(f)
> 2*s[2, 1] + 2*s[3]
> ```

map_support(*f*)

> Mapping a function on the support.
>
> INPUT:
>
> > •f – an endofunction on the indices of the free module
>
> Return a new element of `self.parent()` obtained by applying the function `f` to all of the objects indexing the basis elements.

EXAMPLES:

```
sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])
sage: x = B.an_element(); x
2*B[-1] + 2*B[0] + 3*B[1]
sage: x.map_support(lambda i: -i)
3*B[-1] + 2*B[0] + 2*B[1]
```

f needs not be injective:

```
sage: x.map_support(lambda i: 1)
7*B[1]
```

```
sage: s = SymmetricFunctions(QQ).schur()
sage: a = s([2,1])+2*s([3,2])
sage: a.map_support(lambda x: x.conjugate())
s[2, 1] + 2*s[2, 2, 1]
```

TESTS:

```
sage: B.zero()          # This actually failed at some point!!! See #8890
0
```

```
sage: y = B.zero().map_support(lambda i: i/0); y
0
sage: y.parent() is B
True
```

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])
sage: x = B.an_element(); x
2*B[-1] + 2*B[0] + 3*B[1]
sage: x.map_support_skip_none(lambda i: -i if i else None)
3*B[-1] + 2*B[1]
```

f needs not be injective:

```
sage: x.map_support_skip_none(lambda i: 1 if i else None)
5*B[1]
```

TESTS:

```
sage: y = x.map_support_skip_none(lambda i: None); y
0
sage: y.parent() is B
True
```

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")
sage: X.monomial(2).support_of_term()
2
sage: X.term(3, 2).support_of_term()
3
```

An exception is raised if `self` has more than one term:

```
sage: (X.monomial(2) + X.monomial(3)).support_of_term()
Traceback (most recent call last):
...
ValueError: B[2] + B[3] is not a single term
```

tensor(**elements*)

Return the tensor product of its arguments, as an element of the tensor product of the parents of those elements.

EXAMPLES:

```
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()
sage: (a,b,c) = A.algebra_generators()
sage: a.tensor(b, c)
B[word: a] # B[word: b] # B[word: c]
```

FIXME: is this a policy that we want to enforce on all parents?

trailing_coefficient(*cmp=None*)

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 function `cmp(x,y)`, can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.trailing_coefficient()
3
sage: def cmp(x,y): return y-x
sage: x.trailing_coefficient(cmp=cmp)
1

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.trailing_coefficient()
2
```

trailing_item(*cmp=None*)

Returns the pair `(c, k)` where `c*self.parent().monomial(k)` is the trailing term of `self`.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when `self` is printed. If the default term ordering is not what is desired, a comparison function `cmp(x,y)`, can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.trailing_item()
(1, 3)
sage: def cmp(x,y): return y-x
sage: x.trailing_item(cmp=cmp)
(3, 1)

sage: s = SymmetricFunctions(QQ).schur()
```

```
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.trailing_item()
([1], 2)
```

trailing_monomial(*cmp=None*)

Return the trailing monomial of `self`.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when `self` is printed. If the default term ordering is not what is desired, a comparison function `cmp(x,y)`, can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.trailing_monomial()
B[1]
sage: def cmp(x,y): return y-x
sage: x.trailing_monomial(cmp=cmp)
B[3]

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.trailing_monomial()
s[1]
```

trailing_support(*cmp=None*)

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 function, `cmp(x,y)`, can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + 4*X.monomial(3)
sage: x.trailing_support()
1
sage: def cmp(x,y): return y-x
sage: x.trailing_support(cmp=cmp)
3

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.trailing_support()
[1]
```

trailing_term(*cmp=None*)

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 function `cmp(x,y)`, can be provided. This should return a negative value if `x < y`, 0 if `x == y` and a positive value if `x > y`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
```

```
sage: x.trailing_term()
3*B[1]
sage: def cmp(x,y): return y-x
sage: x.trailing_term(cmp=cmp)
B[3]

sage: s = SymmetricFunctions(QQ).schur()
sage: f = 2*s[1] + 3*s[2,1] - 5*s[3]
sage: f.trailing_term()
2*s[1]
```

ModulesWithBasis.**FiniteDimensional**
    alias of FiniteDimensionalModulesWithBasis

ModulesWithBasis.**Graded**
    alias of GradedModulesWithBasis

class ModulesWithBasis.**Homsets**(*category*, **args*)
    Bases: sage.categories.homsets.HomsetsCategory

    TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

    class **ParentMethods**

class ModulesWithBasis.**MorphismMethods**

    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()

sage: f = H(lambda x: Y.zero()).on_basis()
sage: f(2)
0

sage: f = lambda i: Y.monomial(i) + 2*Y.monomial(i+1)
sage: g = H(on_basis = f).on_basis()
sage: g(2)
```

```
      B[2] + 2*B[3]
sage: g == f
True
```

class ModulesWithBasis.**ParentMethods**

basis()

Return the basis of `self`.

EXAMPLES:
```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F.basis()
Finite family {'a': B['a'], 'c': B['c'], 'b': B['b']}

sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: list(QS3.basis())
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

echelon_form(*elements*)

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`.

OUTPUT:

A list of elements of `self` whose expressions as vectors form a matrix in echelon form. If `base_ring` is specified, then the calculation is achieved in this base ring.

EXAMPLES:
```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: V = X.echelon_form([x[0]-x[1], x[0]-x[2],x[1]-x[2]]); V
[x[0] - x[2], x[1] - x[2]]
sage: matrix(map(vector, V))
[ 1  0 -1]
[ 0  1 -1]

sage: F = CombinatorialFreeModule(ZZ, [1,2,3,4])
sage: B = F.basis()
sage: elements = [B[1]-17*B[2]+6*B[3], B[1]-17*B[2]+B[4]]
sage: F.echelon_form(elements)
[B[1] - 17*B[2] + B[4], 6*B[3] - B[4]]

sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: a,b,c = F.basis()
sage: F.echelon_form([8*a+b+10*c, -3*a+b-c, a-b-c])
[B['a'] + B['c'], B['b'] + 2*B['c']]
```

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 $\dim X \times \dim Y$ or $\dim Y \times \dim X$

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 in 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:

$$ g\left(a, \sum_i \lambda_i x_i, \ c\right) = \sum_i \lambda_i f(a, i, c). $$

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]);    X.rename("X")
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4]); Y.rename("Y")
sage: phi = X.module_morphism(lambda i: Y.monomial(i) + 2*Y.monomial(i+1), codomain = Y)
sage: x = X.basis(); y = Y.basis()
sage: phi(x[1] + x[3])
B[1] + 2*B[2] + B[3] + 2*B[4]
```

```
sage: phi
Generic morphism:
From: X
To:   Y
```

By default, the category is the first of Modules(R).WithBasis().FiniteDimensional(), Modules(R).WithBasis(), Modules(R), and CommutativeAdditiveMonoids() that contains both the domain and the codomain:

```
sage: phi.category_for()
Category of finite dimensional vector spaces with basis over Rational Field
```

With the zero argument, one can define affine morphisms:

```
sage: phi = X.module_morphism(lambda i: Y.monomial(i) + 2*Y.monomial(i+1),
....:                         codomain = Y, zero = 10*y[1])
sage: phi(x[1] + x[3])
11*B[1] + 2*B[2] + B[3] + 2*B[4]
```

In this special case, the default category is Sets():

```
sage: phi.category_for()
Category of sets
```

One can construct morphisms with the base ring as codomain:

```
sage: X = CombinatorialFreeModule(ZZ,[1,-1])
sage: phi = X.module_morphism( on_basis=lambda i: i, codomain=ZZ )
sage: phi( 2 * X.monomial(1) + 3 * X.monomial(-1) )
-1
sage: phi.category_for()
Category of commutative additive semigroups
sage: phi.category_for() # todo: not implemented (ZZ is currently not in Modules(ZZ))
Category of modules over Integer Ring
```

Or more generaly any ring admitting a coercion map from the base ring:

```
sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=RR )
sage: phi( 2 * X.monomial(1) + 3 * X.monomial(-1) )
-1.00000000000000
sage: phi.category_for()
Category of commutative additive semigroups
sage: phi.category_for() # todo: not implemented (RR is currently not in Modules(ZZ))
Category of modules over Integer Ring

sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=Zmod(4) )
sage: phi( 2 * X.monomial(1) + 3 * X.monomial(-1) )
3

sage: phi = Y.module_morphism(on_basis=lambda i: i, codomain=Zmod(4) )
Traceback (most recent call last):
...
ValueError: codomain(=Ring of integers modulo 4) should be a module over the base ring o
```

On can also define module morphisms between free modules over different base rings; here we implement the natural map from $X = \mathbf{R}^2$ to $Y = \mathbf{C}$:

```
sage: X = CombinatorialFreeModule(RR,['x','y'])
sage: Y = CombinatorialFreeModule(CC,['z'])
sage: x = X.monomial('x')
sage: y = X.monomial('y')
sage: z = Y.monomial('z')
sage: def on_basis( a ):
....:     if a == 'x':
....:         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']
sage: phi.category_for()
Category of commutative additive semigroups
sage: phi.category_for() # todo: not implemented (CC is currently not in Modules(RR)!)
Category of vector spaces over Real Field with 53 bits of precision

sage: Y = CombinatorialFreeModule(CC['q'],['z'])
sage: z = Y.monomial('z')
sage: phi = X.module_morphism( on_basis=on_basis, codomain=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:

```
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 mod

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 o
```

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")
sage: phi = X.module_morphism(diagonal=factorial, codomain=X)
sage: x = X.basis()
sage: phi(x[1]), phi(x[2]), phi(x[3])
(B[1], 2*B[2], 6*B[3])
```

See also: `sage.modules.with_basis.morphism.DiagonalModuleMorphism`.

With the `matrix=m` argument, this constructs the module morphism whose matrix in the distinguished basis of X and Y is m:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X"); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); Y.rename("Y"); y = Y.basis()
sage: m = matrix([[0,1,2],[3,5,0]])
sage: phi = X.module_morphism(matrix=m, codomain=Y)
sage: phi(x[1])
3*B[4]
sage: phi(x[2])
B[3] + 5*B[4]
```

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 = range(1,200)
sage: X = CombinatorialFreeModule(QQ, I); X.rename("X"); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, I); Y.rename("Y"); y = Y.basis()
sage: f = Y.sum_of_monomials * divisors
sage: phi = X.module_morphism(f, triangular="upper", codomain = Y)
sage: phi(x[2])
B[1] + B[2]
sage: phi(x[6])
B[1] + B[2] + B[3] + B[6]
sage: phi(x[30])
B[1] + B[2] + B[3] + B[5] + B[6] + B[10] + B[15] + B[30]
sage: phi.preimage(y[2])
-B[1] + B[2]
sage: phi.preimage(y[6])
B[1] - B[2] - B[3] + B[6]
sage: phi.preimage(y[30])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
```

```
sage: (phi^-1)(y[30])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
```

Since trac ticket #8678, one can also define a triangular morphism from a function:

```
sage: X = CombinatorialFreeModule(QQ, [0,1,2,3,4]); x = X.basis()
sage: from sage.modules.with_basis.morphism import TriangularModuleMorphismFromFunction
sage: def f(x): return x + X.term(0, sum(x.coefficients()))
sage: phi = X.module_morphism(function=f, codomain=X, triangular="upper")
sage: phi(x[2] + 3*x[4])
4*B[0] + B[2] + 3*B[4]
sage: phi.preimage(_)
B[2] + 3*B[4]
```

For details and further optional arguments, see `sage.modules.with_basis.morphism.TriangularModule`

> **Warning:** As a temporary measure, until multivariate morphisms are implemented, the constructed morphism is in `Hom(codomain, domain, 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

TESTS:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X")
sage: phi = X.module_morphism(codomain=X)
Traceback (most recent call last):
...
ValueError: module_morphism() takes exactly one option
out of 'matrix', 'on_basis', 'function', 'diagonal'

sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X")
sage: phi = X.module_morphism(diagonal=factorial, matrix=matrix(), codomain=X)
Traceback (most recent call last):
...
ValueError: module_morphism() takes exactly one option
out of 'matrix', 'on_basis', 'function', 'diagonal'

sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X")
sage: phi = X.module_morphism(matrix=factorial, codomain=X)
Traceback (most recent call last):
...
ValueError: matrix (=factorial) should be a matrix

sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); X.rename("X")
sage: phi = X.module_morphism(diagonal=3, codomain=X)
Traceback (most recent call last):
...
ValueError: diagonal (=3) should be a function
```

submodule (*gens*, *check=True*, *already_echelonized=False*, *category=None*)
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.

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.

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 **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:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: gens = [x[0] - x[1], x[1] - x[2]]; gens
[x[0] - x[1], x[1] - x[2]]
sage: Y = X.submodule(gens, already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by {0, 1} over Rational Field
sage: y = Y.basis()
sage: y[1]
y[1]
sage: y[1].lift()
x[1] - x[2]
sage: Y.retract(x[0]-x[2])
y[0] + y[1]
sage: Y.retract(x[0])
Traceback (most recent call last):
...
ValueError: x[0] is not in the image
```

By using a family to specify a basis of the submodule, we obtain a submodule whose index set coincides with the index set of the family:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
sage: x = X.basis()
sage: gens = Family({1 : x[0] - x[1], 3: x[1] - x[2]}); gens
Finite family {1: x[0] - x[1], 3: x[1] - x[2]}
sage: Y = X.submodule(gens, already_echelonized=True)
sage: Y.print_options(prefix='y'); Y
Free module generated by {1, 3} over Rational Field
sage: y = Y.basis()
sage: y[1]
y[1]
sage: y[1].lift()
x[0] - x[1]
sage: y[3].lift()
x[1] - x[2]
sage: Y.retract(x[0]-x[2])
y[1] + y[3]
sage: Y.retract(x[0])
Traceback (most recent call last):
...
ValueError: x[0] is not in the image
```

It is not necessary that the generators of the submodule form a basis (an explicit basis will be computed):

```
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
[x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
sage: Y = X.submodule(gens, already_echelonized=False)
sage: Y.print_options(prefix='y')
sage: Y
Free module generated by {0, 1} over Rational Field
sage: [b.lift() for b in Y.basis()]
[x[0] - x[2], x[1] - x[2]]
```

We now implement by hand the center of the algebra of the symmetric group S_3:

```
sage: S3 = SymmetricGroup(3)
sage: S3A = S3.algebra(QQ)
sage: basis = S3A.annihilator_basis(S3A.algebra_generators(), S3A.bracket)
sage: basis
[(), (2,3) + (1,2) + (1,3), (1,2,3) + (1,3,2)]
sage: center = S3A.submodule(basis,
....:                        category=AlgebrasWithBasis(QQ).Subobjects(),
....:                        already_echelonized=True)
sage: center
Free module generated by {0, 1, 2} over Rational Field
sage: center in Algebras
True
sage: center.print_options(prefix='c')
sage: c = center.basis()
sage: c[1].lift()
(2,3) + (1,2) + (1,3)
sage: c[0]^2
c[0]
sage: e = 1/6*(c[0]+c[1]+c[2])
sage: e.is_idempotent()
True
```

Of course, this center is best constructed using:

```
sage: center = S3A.center()
```

TESTS:

```
sage: TestSuite(Y).run()
sage: TestSuite(center).run()
```

tensor(*parents*)

Return the tensor product of the parents.

EXAMPLES:

```
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A")
sage: A.tensor(A,A)
A # A # A
sage: A.rename(None)
```

class ModulesWithBasis.**TensorProducts**(*category*, **args*)

Bases: `sage.categories.tensor.TensorProductsCategory`

The category of modules with basis constructed by tensor product of modules with basis.

---

class `ElementMethods`

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"); A.rename("A")
sage: B = CombinatorialFreeModule(ZZ, [3,4], prefix="B"); B.rename("B")
```

and f the bilinear morphism $(a, b) \mapsto b \otimes a$ from $A \times B$ to $B \otimes A$:

```
sage: def f(a,b):
....:     return tensor([b,a])
```

Now, calling applying f on $a \otimes b$ returns the same as $f(a, b)$:

```
sage: a = A.monomial(1) + 2 * A.monomial(2); a
A[1] + 2*A[2]
sage: b = B.monomial(3) - 2 * B.monomial(4); b
B[3] - 2*B[4]
sage: f(a,b)
B[3] # A[1] + 2*B[3] # A[2] - 2*B[4] # A[1] - 4*B[4] # A[2]
sage: tensor([a,b]).apply_multilinear_morphism(f)
B[3] # A[1] + 2*B[3] # A[2] - 2*B[4] # A[1] - 4*B[4] # A[2]
```

f may be a bilinear morphism to any module over the base ring of A and B. Here the codomain is $\mathbf{Z}$:

```
sage: def f(a,b):
....:     return sum(a.coefficients(), 0) * sum(b.coefficients(), 0)
sage: f(a,b)
-3
sage: tensor([a,b]).apply_multilinear_morphism(f)
-3
```

Mind the 0 in the sums above; otherwise f would not return 0 in $\mathbf{Z}$:

```
sage: def f(a,b):
....:     return sum(a.coefficients()) * sum(b.coefficients())
sage: type(f(A.zero(), B.zero()))
<type 'int'>
```

Which would be wrong and break this method:

```
sage: tensor([a,b]).apply_multilinear_morphism(f)
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:

```
sage: C = CombinatorialFreeModule(QQ, [(1,3),(2,4)], prefix="C"); C.rename("C")
sage: def f(a,b):
....:     return C.sum_of_terms( [((1,3), QQ(a[1]*b[3])), ((2,4), QQ(a[2]*b[4]))]
sage: f(a,b)
C[(1, 3)] - 4*C[(2, 4)]
sage: tensor([a,b]).apply_multilinear_morphism(f)
C[(1, 3)] - 4*C[(2, 4)]
```

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::

```
sage: Sym = SymmetricFunctions(QQ)
sage: s = Sym.schur()
sage: def f(a,b): return a*b.antipode()
sage: x = 4*s.an_element(); x
8*s[] + 8*s[1] + 12*s[2]
sage: x.coproduct().apply_multilinear_morphism(f)
8*s[]
sage: x.coproduct().apply_multilinear_morphism(f) == x.counit()
True
```

We recover the constant term of x, as desired.

Todo

Extract a method to linearize a multilinear morphism, and delegate the work there.

---

class `ModulesWithBasis.TensorProducts.`**`ParentMethods`**
   Implements operations on tensor products of modules with basis.

`ModulesWithBasis.TensorProducts.`**`extra_super_categories`**()
   EXAMPLES:
   ```
   sage: ModulesWithBasis(QQ).TensorProducts().extra_super_categories()
   [Category of vector spaces with basis over Rational Field]
   sage: ModulesWithBasis(QQ).TensorProducts().super_categories()
   [Category of tensor products of modules with basis over Rational Field,
    Category of vector spaces with basis over Rational Field,
    Category of tensor products of vector spaces over Rational Field]
   ```

`ModulesWithBasis.`**`is_abelian`**()
   Returns 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
   ```

13.84 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 algebras 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
```

TESTS:
```
sage: TestSuite(MonoidAlgebras(ZZ)).run()
```

13.85 Monoids

class sage.categories.monoids.**Monoids**(*base_category*)

    Bases: `sage.categories.category_with_axiom.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,
 Category of sets with partial maps,
 Category of objects]

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')
```

    TESTS:
```
sage: C = Monoids()
sage: TestSuite(C).run()
```

    class **Algebras**(*category*, **args*)

        Bases: `sage.categories.algebra_functor.AlgebrasCategory`

        TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ElementMethods**

is_central()
> Return whether the element `self` is central.

> EXAMPLES:
```
sage: SG4=SymmetricGroupAlgebra(ZZ,4)
sage: SG4(1).is_central()
True
sage: SG4(Permutation([1,3,2,4])).is_central()
False
sage: A=GroupAlgebras(QQ).example(); A
Group algebra of Dihedral group of order 8 as a permutation group over Rational Field
sage: sum(i for i in A.basis()).is_central()
True
```

class Monoids.Algebras.**ParentMethods**

algebra_generators()
> Return generators for this algebra.

> For a monoid algebra, the algebra generators are built from the monoid generators if available and from the semigroup generators otherwise.

> **See also:**

> > • `Semigroups.Algebras.ParentMethods.algebra_generators()`
> > • `MagmaticAlgebras.ParentMethods.algebra_generators()`.

> EXAMPLES:
```
sage: M = Monoids().example(); M
An example of a monoid:
the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.monoid_generators()
Finite family {'a': 'a', 'c': 'c', 'b': 'b', 'd': 'd'}
sage: M.algebra(ZZ).algebra_generators()
Finite family {'a': B['a'], 'c': B['c'], 'b': B['b'], '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()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3],  4: B[4],   5: B[5],
                6: B[6], 7: B[7], 8: B[8], 9: B[9], 10: B[10], 11: B[11]}
```

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)
sage: A.one_basis()
''
sage: A.one()
B['']
sage: A(3)
3*B['']
```

Monoids.Algebras.**extra_super_categories**()

EXAMPLES:

```
sage: Monoids().Algebras(QQ).extra_super_categories()
[Category of monoids]
sage: Monoids().Algebras(QQ).super_categories()
[Category of algebras with basis over Rational Field,
 Category of semigroup algebras over Rational Field,
 Category of unital magma algebras over Rational Field]
```

class Monoids.**CartesianProducts**(*category*, **args*)

Bases: `sage.categories.cartesian_product.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 ParentMethods

monoid_generators()

Return the generators of `self`.

EXAMPLES:

```
sage: M = Monoids.free([1,2,3])
sage: N = Monoids.free(['a','b'])
sage: C = cartesian_product([M, N])
sage: C.monoid_generators()
Family ((F[1], 1), (F[2], 1), (F[3], 1),
        (1, F['a']), (1, F['b']))
```

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()
Lazy family (gen(i))_{i in The cartesian product of (...)}
```

Monoids.CartesianProducts.**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 Monoids.**Commutative**(*base_category*)
    Bases: `sage.categories.category_with_axiom.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 specifing 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)
Free abelian monoid indexed by Integer Ring
sage: F.<x,y,z> = Monoids().Commutative().free(); F
Free abelian monoid indexed by {'x', 'y', 'z'}
```

class Monoids.**ElementMethods**

    is_one()
        Return whether `self` is the one of the monoid.

        The default implementation is to compare with `self.one()`.

        TESTS:

```
sage: S = Monoids().example()
sage: S.one().is_one()
True
sage: S("aa").is_one()
False
```

    powers(*n*)
        Return the list $[x^0, x^1, \ldots, x^{n-1}]$.

        EXAMPLES:

```
sage: A = Matrix([[1, 1], [-1, 0]])
sage: A.powers(6)
[
```

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

Monoids.**Finite**
> alias of [FiniteMonoids](#)

Monoids.**Inverse**
> alias of [Groups](#)

class Monoids.**ParentMethods**

> **one_element**()
> > Backward compatibility alias for `one()`.
> >
> > TESTS:
> > ```
> > sage: S = Monoids().example()
> > sage: S.one_element()
> > doctest:...: DeprecationWarning: .one_element() is deprecated. Please use .one() instead
> > See http://trac.sagemath.org/17694 for details.
> > ''
> > ```
>
> **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:
> > ```
> > 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:
> > ```
> > sage: M = Monoids().free([1,2,3])
> > sage: M.semigroup_generators()
> > 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:
> > ```
> > sage: R = IntegerModRing(15)
> > sage: M = R.submonoid([R(3),R(5)]); M
> > A submonoid of (Ring of integers modulo 15) with 2 generators
> > ```

```
sage: M.list()
[1, 3, 5, 9, 0, 10, 12, 6]
```

Not the presence of the unit, unlike in:

```
sage: S = R.subsemigroup([R(3),R(5)]); S
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: S.list()
[3, 5, 9, 0, 10, 12, 6]
```

This method is really a shorthand for subsemigroup:

```
sage: M2 = R.subsemigroup([R(3),R(5)], one=R.one())
sage: M2 is M
True
```

class Monoids.**Subquotients**(*category*, **args*)

Bases: `sage.categories.subquotients.SubquotientsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

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 Monoids.**WithRealizations**(*category*, **args*)

Bases: `sage.categories.with_realizations.WithRealizationsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
```

```
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

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
The subset algebra of {1, 2, 3} over Rational Field
sage: A.one.__module__
'sage.categories.monoids'
sage: A.one()
F[{}]
```

TESTS:

```
sage: A.one() is A.a_realization().one()
True
sage: A._test_one()
```

static `Monoids.`**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 specifing 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)
Free monoid indexed by Integer Ring
sage: Monoids().free(ZZ)
Free monoid indexed by Integer Ring
sage: F.<x,y,z> = Monoids().free(); F
Free monoid indexed by {'x', 'y', 'z'}
```

13.86 Number fields

class `sage.categories.number_fields.`**NumberFields**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of number fields.

EXAMPLES:

We create the category of number fields:

```
sage: C = NumberFields()
sage: C
Category of number fields
```

Notice that the rational numbers **Q** *are* considered as an object in this category:

```
sage: RationalField() in C
True
```

However, we can define a degree 1 extension of **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
```

TESTS:

```
sage: TestSuite(NumberFields()).run()
```

class **ElementMethods**

class NumberFields.**ParentMethods**

NumberFields.**super_categories**()
    EXAMPLES:
    ```
    sage: NumberFields().super_categories()
    [Category of fields]
    ```

13.87 Objects

class sage.categories.objects.**Objects**(*s=None*)
    Bases: sage.categories.category_singleton.Category_singleton

    The category of all objects the basic category

    EXAMPLES:
    ```
    sage: Objects()
    Category of objects
    sage: Objects().super_categories()
    []
    ```

    TESTS:
    ```
    sage: TestSuite(Objects()).run()
    ```

    class **ParentMethods**
        Methods for all category objects

class Objects.**SubcategoryMethods**

> **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
>> ```
>>
>> This used to be called hom_category:
>> ```
>> sage: Sets().hom_category()
>> doctest:...: DeprecationWarning: hom_category is deprecated. Please use Homsets instead.
>> See http://trac.sagemath.org/10668 for details.
>> Category of homsets of sets
>> ```
>>
>> ---
>>
>> **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.MorphimMethods; 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.

> **hom_category**(*args*, **kwds*)
>> Deprecated: Use Homsets() instead. See trac ticket #10668 for details.

Objects.**additional_structure**()
> Return None

> Indeed, by convention, the category of objects defines no additional structure.

> **See also:**

> Category.additional_structure()

> EXAMPLES:
> ```
> sage: Objects().additional_structure()
> ```

Objects.**super_categories**()
> EXAMPLES:
> ```
> sage: Objects().super_categories()
> []
> ```

13.88 Partially ordered monoids

class sage.categories.partially_ordered_monoids.**PartiallyOrderedMonoids**(*s=None*)
> Bases: sage.categories.category_singleton.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 $x * z < y * z$ and $z * x < z * y$.

> http://en.wikipedia.org/wiki/Ordered_monoid

> EXAMPLES:
> ```
> sage: PartiallyOrderedMonoids()
> Category of partially ordered monoids
> sage: PartiallyOrderedMonoids().super_categories()
> [Category of posets, Category of monoids]
> ```

> TESTS:
> ```
> sage: TestSuite(PartiallyOrderedMonoids()).run()
> ```

> **class ElementMethods**

> **class** PartiallyOrderedMonoids.**ParentMethods**

> PartiallyOrderedMonoids.**super_categories**()
>> EXAMPLES:
>> ```
>> sage: PartiallyOrderedMonoids().super_categories()
>> [Category of posets, Category of monoids]
>> ```

13.89 Permutation groups

class sage.categories.permutation_groups.**PermutationGroups**(*s=None*)

Bases: `sage.categories.category.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:
```
sage: PermutationGroups().additional_structure()
Category of permutation groups
```

TESTS:
```
sage: C = PermutationGroups()
sage: TestSuite(C).run()
```

Finite

alias of `FinitePermutationGroups`

super_categories()

Return a list of the immediate super categories of `self`.

EXAMPLES:
```
sage: PermutationGroups().super_categories()
[Category of groups]
```

13.90 Pointed sets

class sage.categories.pointed_sets.**PointedSets**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of pointed sets.

EXAMPLES:
```
sage: PointedSets()
Category of pointed sets
```

TESTS:

```
sage: TestSuite(PointedSets()).run()
```

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

13.91 Polyhedral subsets of free ZZ, QQ or RR-modules.

class sage.categories.polyhedra.**PolyhedralSets**(*R*)
    Bases: sage.categories.category_types.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
    ```

    TESTS:
    ```
    sage: TestSuite(PolyhedralSets(RDF)).run()
    ```

    ```
    sage: P = Polyhedron()
    sage: P.parent().category().element_class
    <class 'sage.categories.polyhedra.PolyhedralSets.element_class'>
    sage: P.parent().category().element_class.mro()
    [<class 'sage.categories.polyhedra.PolyhedralSets.element_class'>,
     <class 'sage.categories.magmas.Magmas.Commutative.element_class'>,
     <class 'sage.categories.magmas.Magmas.element_class'>,
     <class 'sage.categories.additive_monoids.AdditiveMonoids.element_class'>,
     <class 'sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital.element_class'>,
     <class 'sage.categories.additive_semigroups.AdditiveSemigroups.element_class'>,
     <class 'sage.categories.additive_magmas.AdditiveMagmas.element_class'>,
     <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'>,
     <type 'object'>]
    sage: isinstance(P, P.parent().category().element_class)
    True
    ```

    super_categories()
        EXAMPLES:
        ```
        sage: PolyhedralSets(QQ).super_categories()
        [Category of commutative magmas, Category of additive monoids]
        ```

13.92 Posets

class sage.categories.posets.**Posets**(*s=None*)
    Bases: sage.categories.category.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 trac ticket #10130 will be resolved, this will be automatically provided by this category:
```
sage: x < y       # todo: not implemented
True
sage: x < x       # todo: not implemented
False
sage: x <= x      # todo: not implemented
True
sage: y >= x      # todo: not implemented
True
```

See also:

`Poset()`, `FinitePosets`, `LatticePosets`

TESTS:
```
sage: C = Posets()
sage: TestSuite(C).run()
```

class **ElementMethods**

Posets.**Finite**
    alias of FinitePosets

class Posets.**ParentMethods**

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 )
True
sage: D.ge( 3, 5 )
False
```

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
sage: P.is_antichain_of_poset(i for i in divisors(12) if (2 < i and i < 6))
True
sage: P.is_antichain_of_poset(i for i in divisors(12) if (2 <= i and i < 6))
False

sage: Q = Poset({2: [3, 1], 3: [4], 1: [4]})
sage: Q.is_antichain_of_poset((1, 2))
False
sage: Q.is_antichain_of_poset((2, 4))
False
sage: Q.is_antichain_of_poset((4, 2))
False
sage: Q.is_antichain_of_poset((2, 2))
True
sage: Q.is_antichain_of_poset((3, 4))
False
sage: Q.is_antichain_of_poset((3, 1))
True
sage: Q.is_antichain_of_poset((1, ))
True
sage: Q.is_antichain_of_poset(())
True
```

> An infinite poset:

```
sage: from sage.categories.examples.posets import FiniteSetsOrderedByInclusion
sage: R = FiniteSetsOrderedByInclusion()
sage: R.is_antichain_of_poset([R(set([3, 1, 2])), R(set([1, 4])), R(set([4, 5]))])
```

```
     True
sage: R.is_antichain_of_poset([R(set([3, 1, 2, 4])), R(set([1, 4])), R(set([4, 5]))])
     False
```

is_chain_of_poset(*o*, *ordered=False*)

Return whether an iterable `o` is a chain of `self`, including a check for `o` being ordered from smallest to largest element if the keyword `ordered` is set to `True`.

INPUT:

- •`o` – an iterable (e. g., list, set, or tuple) containing some elements of `self`
- •`ordered` – a Boolean (default: `False`) which decides whether the notion of a chain includes being ordered

OUTPUT:

If `ordered` is set to `False`, the truth value of the following assertion is returned: The subset of `self` formed by the elements of `o` is a chain in `self`.

If `ordered` is set to `True`, the truth value of the following assertion is returned: Every element of the list `o` is (strictly!) smaller than its successor in `self`. (This makes no sense if `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 PositiveIntegersOrderedByDivisibilityF
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(3),), 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({3, 4, 12})
False
sage: P.is_order_filter({3, 6, 12})
True
```

is_order_ideal(*o*)

Return whether o is an order ideal of self, assuming self has no infinite descending path.

INPUT:
  •o – a list (or set, or tuple) containing some elements of `self`

EXAMPLES:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_extension=True)
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_order_ideal([1, 3])
True
sage: P.is_order_ideal([])
True
sage: P.is_order_ideal({1, 3})
True
sage: P.is_order_ideal([1, 3, 4])
False
```

le (*x*, *y*)

   Return whether $x \leq y$ in the poset `self`.

   INPUT:
     •x, y – elements of `self`.

   EXAMPLES:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.le( 3, 6 )
True
sage: D.le( 3, 3 )
True
sage: D.le( 3, 5 )
False
```

lower_covers (*x*)

   Return the lower covers of x, that is, the elements y such that $y < x$ and there exists no z such that $y < z < x$.

   EXAMPLES:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lower_covers(15)
[3, 5]
```

lower_set (*elements*)

   Return the order ideal in `self` generated by the elements of an iterable `elements`.

   A subset I of a poset is said to be an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

   This is also called the lower set generated by these elements.

   EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.order_ideal([7,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

lt (*x*, *y*)

   Return whether $x < y$ in the poset `self`.

   INPUT:
     •x, y – elements of `self`.

   This default implementation delegates the work to `le()`.

   EXAMPLES:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lt( 3, 6 )
True
sage: D.lt( 3, 3 )
False
sage: D.lt( 3, 5 )
False
```

order_filter(*elements*)

Return the order filter generated by a list of elements.

A subset I of a poset is said to be an order filter if, for any x in I and y such that $y \geq x$, then y is in I.

This is also called the upper set generated by these elements.

EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.order_filter([3,8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

order_ideal(*elements*)

Return the order ideal in `self` generated by the elements of an iterable `elements`.

A subset I of a poset is said to be an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

This is also called the lower set generated by these elements.

EXAMPLES:

```
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:

```
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)
{1, 2, 4}
sage: P4 = Posets(4)
sage: all(all(all(P.order_ideal_toggle(P.order_ideal_toggle(I, i), i) == I
....:             for i in range(4))
....:         for I in P.order_ideals_lattice(facade=True))
```

```
....:      for P in P4)
True
```

order_ideal_toggles(*I*, *vs*)

Return the result of toggling the elements of the list (or iterable) `vs` (one by one, from left to right) in the order ideal `I`.

See `order_ideal_toggle()` for a definition of toggling.

EXAMPLES:
```
sage: P = Poset({1: [2,3], 2: [4], 3: []})
sage: I = Set({1, 2})
sage: P.order_ideal_toggles(I, [1,2,3,4])
{1, 3}
sage: P.order_ideal_toggles(I, (1,2,3,4))
{1, 3}
```

principal_lower_set(*x*)

Return the order ideal generated by an element `x`.

This is also called the lower set generated by this element.

EXAMPLES:
```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_order_ideal(6)
[0, 2, 4, 6]
```

principal_order_filter(*x*)

Return the order filter generated by an element `x`.

This is also called the upper set generated by this element.

EXAMPLES:
```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_order_filter(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

principal_order_ideal(*x*)

Return the order ideal generated by an element `x`.

This is also called the lower set generated by this element.

EXAMPLES:
```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_order_ideal(6)
[0, 2, 4, 6]
```

principal_upper_set(*x*)

Return the order filter generated by an element `x`.

This is also called the upper set generated by this element.

EXAMPLES:
```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_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]
```

upper_set(*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:
```
sage: B = Posets.BooleanLattice(4)
sage: B.order_filter([3,8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

Posets.**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
```

Posets.**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]
```

13.93 Principal ideal domains

class sage.categories.principal_ideal_domains.**PrincipalIdealDomains**(*s=None*)

Bases: `sage.categories.category_singleton.Category_singleton`

The category of (constructive) principal ideal domains

By constructive, we mean that a single generator can be constructively found for any ideal given by a finite set of generators. Note that this constructive definition only implies that finitely generated ideals are principal. It is not clear what we would mean by an infinitely generated ideal.

EXAMPLES:
```
sage: PrincipalIdealDomains()
Category of principal ideal domains
sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]
```

See also: http://en.wikipedia.org/wiki/Principal_ideal_domain

TESTS:
```
sage: TestSuite(PrincipalIdealDomains()).run()
```

class **ElementMethods**

class PrincipalIdealDomains.**ParentMethods**

PrincipalIdealDomains.**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()
>   ```

PrincipalIdealDomains.**super_categories**()
>   EXAMPLES:
>   ```
>   sage: PrincipalIdealDomains().super_categories()
>   [Category of unique factorization domains]
>   ```

13.94 Quotient fields

class sage.categories.quotient_fields.**QuotientFields**(*s=None*)
>   Bases: `sage.categories.category_singleton.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]
>   ```
>
>   TESTS:
>   ```
>   sage: TestSuite(QuotientFields()).run()
>   ```

>   class **ElementMethods**

>   >   **denominator**()
>   >   >   Constructor for abstract methods
>   >   >
>   >   >   EXAMPLES:
>   >   >   ```
>   >   >   sage: def f(x):
>   >   >   ...        "doc of f"
>   >   >   ...        return 1
>   >   >   ...
>   >   >   sage: x = abstract_method(f); x
>   >   >
>   >   >   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()
(x - 3)^-1 * x * (x^2 + 1)
```

    Here is an example to show that ticket #7868 has been resolved:
```
sage: R.<x,y> = GF(2)[]
sage: f = x*y/(x+y)
sage: f.factor()
(x + y)^-1 * y * x
```

gcd(*other*)
    Greatest common divisor

    Note: In a field, the greatest common divisor is not very informative, as it is only determined up to a
    unit. But in the fraction field of an integral domain that provides both gcd and lcm, it is possible to be
    a bit more specific and define the gcd uniquely up to a unit of the base ring (rather than in the fraction
    field).

    AUTHOR:
        •Simon King (2011-02): See [trac ticket #10771](#)
    EXAMPLES:
```
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 * (x^2 + 1/3)
sage: gcd(p,q)
(x + 1)/(x^7 + x^5 - x^2 - 1)
```

```
sage: factor(gcd(p,q))
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^4 + x^3 + x^2 + x + 1)^-1
sage: factor(gcd(p,1+x))
(x - 1)^-1 * (x + 1) * (x^4 + x^3 + x^2 + x + 1)^-1
sage: factor(gcd(1+x,q))
(x - 1)^-1 * (x + 1) * (x^2 + 1)^-1
```

TESTS:

The following tests that the fraction field returns a correct gcd even if the base ring does not provide lcm and gcd:

```
sage: R = ZZ.extension(x^2+1, names='i')
sage: i = R.1
sage: gcd(5, 3 + 4*i)
-i - 2
sage: P.<t> = R[]
sage: gcd(t, i)
Traceback (most recent call last):
...
NotImplementedError: Maximal Order in Number Field in i with defining polynomial x^2 + 1
sage: q = t/(t+1); q.parent()
Fraction Field of Univariate Polynomial Ring in t over Maximal Order in Number Field in
sage: gcd(q, q)
1
sage: q.gcd(0)
1
sage: (q*0).gcd(0)
0
```

lcm(*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 trac ticket #10771
EXAMPLES:

```
sage: lcm(2/3, 1/5)
2
```

Indeed $2/3 = 2^1 3^{-1} 5^0$ and $1/5 = 2^0 3^0 5^{-1}$, so $lcm(2/3, 1/5) = 2^1 3^0 5^0 = 2$.
   sage: lcm(1/3, 1/5) 1 sage: lcm(1/3, 1/6) 1/3
Some more involved examples:

```
sage: R.<x> = QQ[]
sage: p = (1+x)^3*(1+2*x^2)/(1-x^5)
sage: q = (1+x)^2*(1+3*x^2)/(1-x^4)
sage: factor(p)
```

```
(-2) * (x - 1)^-1 * (x + 1)^3 * (x^2 + 1/2) * (x^4 + x^3 + x^2 + x + 1)^-1
sage: factor(q)
(-3) * (x - 1)^-1 * (x + 1) * (x^2 + 1)^-1 * (x^2 + 1/3)
sage: factor(lcm(p,q))
(x - 1)^-1 * (x + 1)^3 * (x^2 + 1/3) * (x^2 + 1/2)
sage: factor(lcm(p,1+x))
(x + 1)^3 * (x^2 + 1/2)
sage: factor(lcm(1+x,q))
(x + 1) * (x^2 + 1/3)
```

TESTS:

The following tests that the fraction field returns a correct lcm even if the base ring does not provide lcm and gcd:

```
sage: R = ZZ.extension(x^2+1, names='i')
sage: i = R.1
sage: P.<t> = R[]
sage: lcm(t, i)
Traceback (most recent call last):
...
NotImplementedError: Maximal Order in Number Field in i with defining polynomial x^2 + 1
sage: q = t/(t+1); q.parent()
Fraction Field of Univariate Polynomial Ring in t over Maximal Order in Number Field in
sage: lcm(q, q)
1
sage: q.lcm(0)
0
sage: (q*0).lcm(0)
0
```

Check that it is possible to take lcm of a rational and an integer (trac ticket #17852):

```
sage: (1/2).lcm(2)
2
sage: type((1/2).lcm(2))
<type 'sage.rings.rational.Rational'>
```

numerator()
   Constructor for abstract methods

   EXAMPLES:

```
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 (default: True)

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: 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
sage: sum(parts) == q
True
sage: q = 2*t / (t + 3)^2
sage: q.partial_fraction_decomposition()
(0, [2/(t + 3), -6/(t^2 + 6*t + 9)])
sage: for p in q.partial_fraction_decomposition()[1]: print p.factor()
(2) * (t + 3)^-1
(-6) * (t + 3)^-2
sage: q.partial_fraction_decomposition(decompose_powers=False)
(0, [2*t/(t^2 + 6*t + 9)])
```

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)
sage: r.partial_fraction_decomposition()
(0, [(-1/2*I)/(x^2 - I), 1/2*I/(x^2 + I)])
```

We can also ask Sage to find the least extension where the denominator factors in linear terms:

```
sage: R.<x> = QQ[]
sage: r = 1/(x^4+2)
sage: N = r.denominator().splitting_field('a')
sage: 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 - 2
  (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
  (-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/
  (5/1344*a^7 - 43/1344*a^5 + 85/672*a^3 + 31/672*a)/(x1 + 5/168*a^7 - 43/168*a^5 + 85/8
```

Or we may work directly over an algebraically closed field:

```
sage: R.<x> = QQbar[]
sage: r =  1/(x^4+1)
sage: r.partial_fraction_decomposition()
(0,
 [(-0.1767766952966369? - 0.1767766952966369?*I)/(x - 0.7071067811865475? - 0.70710678118
  (-0.1767766952966369? + 0.1767766952966369?*I)/(x - 0.7071067811865475? + 0.70710678118
  (0.1767766952966369? - 0.1767766952966369?*I)/(x + 0.7071067811865475? - 0.70710678118
  (0.1767766952966369? + 0.1767766952966369?*I)/(x + 0.7071067811865475? + 0.70710678118
```

We do the best we can over inexact fields:

```
sage: R.<x> = RealField(20)[]
sage: q = 1/(x^2 + x + 2)^2 + 1/(x-1); q
(x^4 + 2.0000*x^3 + 5.0000*x^2 + 5.0000*x + 3.0000)/(x^5 + x^4 + 3.0000*x^3 - x^2 - 4.00
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 - 4.00
```

TESTS:

We test partial fraction for irreducible denominators:

```
sage: R.<x> = ZZ[]
sage: q = x^2/(x-1)
sage: q.partial_fraction_decomposition()
(x + 1, [1/(x - 1)])
sage: q = x^10/(x-1)^5
sage: whole, parts = q.partial_fraction_decomposition()
sage: whole + sum(parts) == q
True
```

And also over finite fields (see trac #6052, #9945):

```
sage: R.<x> = GF(2)[]
sage: q = (x+1)/(x^3+x+1)
sage: q.partial_fraction_decomposition()
(0, [(x + 1)/(x^3 + x + 1)])

sage: R.<x> = GF(11)[]
sage: q = x + 1 + 1/(x+1) + x^2/(x^3 + 2*x + 9)
sage: q.partial_fraction_decomposition()
(x + 1, [1/(x + 1), x^2/(x^3 + 2*x + 9)])
```

And even the rationals:

```
sage: (26/15).partial_fraction_decomposition()
(1, [1/3, 2/5])
sage: (26/75).partial_fraction_decomposition()
(-1, [2/3, 3/5, 2/25])
```

A larger example:

```
sage: S.<t> = QQ[]
sage: r = t / (t^3+1)^5
sage: r.partial_fraction_decomposition()
(0,
 [-35/729/(t + 1),
  -35/729/(t^2 + 2*t + 1),
  -25/729/(t^3 + 3*t^2 + 3*t + 1),
  -4/243/(t^4 + 4*t^3 + 6*t^2 + 4*t + 1),
  -1/243/(t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1),
  (35/729*t - 35/729)/(t^2 - t + 1),
```

```
      (25/729*t - 8/729)/(t^4 - 2*t^3 + 3*t^2 - 2*t + 1),
      (-1/81*t + 5/81)/(t^6 - 3*t^5 + 6*t^4 - 7*t^3 + 6*t^2 - 3*t + 1),
      (-2/27*t + 1/9)/(t^8 - 4*t^7 + 10*t^6 - 16*t^5 + 19*t^4 - 16*t^3 + 10*t^2 - 4*t + 1),
      (-2/27*t + 1/27)/(t^10 - 5*t^9 + 15*t^8 - 30*t^7 + 45*t^6 - 51*t^5 + 45*t^4 - 30*t^3 +
sage: sum(r.partial_fraction_decomposition()[1]) == r
True
```

Some special cases:

```
sage: R = Frac(QQ['x']); x = R.gen()
sage: x.partial_fraction_decomposition()
(x, [])
sage: R(0).partial_fraction_decomposition()
(0, [])
sage: R(1).partial_fraction_decomposition()
(1, [])
sage: (1/x).partial_fraction_decomposition()
(0, [1/x])
sage: (1/x+1/x^3).partial_fraction_decomposition()
(0, [1/x, 1/x^3])
```

This was fixed in trac ticket #16240:

```
sage: R.<x> = QQ['x']
sage: p=1/(-x + 1)
sage: whole,parts = p.partial_fraction_decomposition()
sage: p == sum(parts)
True
sage: p=3/(-x^4 + 1)
sage: whole,parts = p.partial_fraction_decomposition()
sage: p == sum(parts)
True
sage: p=(6*x^2 - 9*x + 5)/(-x^3 + 3*x^2 - 3*x + 1)
sage: whole,parts = p.partial_fraction_decomposition()
sage: p == sum(parts)
True
```

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*self + t*other.

Note: In a field, the greatest common divisor is not very informative, as it is only determined up to a unit. But in the fraction field of an integral domain that provides both xgcd and lcm, it is possible to be a bit more specific and define the gcd uniquely up to a unit of the base ring (rather than in the fraction field).

EXAMPLES:

```
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 * (x^2 + 1/3)
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)
sage: O = K.maximal_order()
sage: R = PolynomialRing(O, 'x')
sage: F = R.fraction_field()
sage: x = F.gen(0)
sage: x.gcd(x+1)
1
sage: x.xgcd(x+1)
(1, 1/x, 0)
sage: zero = F.zero()
sage: zero.gcd(x)
1
sage: zero.xgcd(x)
(1, 0, 1/x)
sage: zero.xgcd(zero)
(0, 0, 0)
```

class QuotientFields.**ParentMethods**

QuotientFields.**super_categories**()
  EXAMPLES:
```
sage: QuotientFields().super_categories()
[Category of fields]
```

13.95 Regular Crystals

class sage.categories.regular_crystals.**RegularCrystals**(*s=None*)
  Bases: `sage.categories.category_singleton.Category_singleton`

  The category of regular crystals.

  A crystal is called *regular* if every vertex b satisfies

  $$\varepsilon_i(b) = \max\{k \mid e_i^k(b) \neq 0\} \quad \text{and} \quad \varphi_i(b) = \max\{k \mid f_i^k(b) \neq 0\}.$$

Note: Regular crystals are sometimes referred to as *normal*. When only one of the conditions (on either φ_i or ε_i) holds, these crystals are sometimes called *seminormal* or *semiregular*.

EXAMPLES:
```
sage: C = RegularCrystals()
sage: C
Category of regular crystals
```

```
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

TESTS:

```
sage: TestSuite(C).run()
sage: B = RegularCrystals().example()
sage: TestSuite(B).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_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

class **ElementMethods**

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_i^k for $0 \leq k \leq r$.
- If $r < 0$, this returns the opposite of the sum of the elements obtained by application of e_i^k for $0 < k < -r$.

REFERENCES:

EXAMPLES:

```
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()
Free module generated by The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]

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[[[2, 3]]] + B[[[1, 2]]]
```

TESTS:
```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: x = K.an_element(); x
[[1]]
sage: x.demazure_operator_simple(0)
0
sage: x.demazure_operator_simple(0, ring = QQ).parent()
Free module generated by Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r,s)=(1,
```

dual_equivalence_class(*index_set=None*)

Return the dual equivalence class indexed by `index_set` of `self`.

The dual equivalence class of an element $b \in B$ is the set of all elements of B reachable from b via sequences of i-elementary dual equivalence relations (i.e., i-elementary dual equivalence transformations and their inverses) for i in the index set of B.

For this to be well-defined, the element b has to be of weight 0 with respect to I; that is, we need to have $\varepsilon_j(b) = \varphi_j(b)$ for all $j \in I$.

See [Assaf08]. 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:
```
sage: T = crystals.Tableaux(['A',3], shape=[2,2])
sage: G = T(2,1,4,3).dual_equivalence_class()
sage: sorted(G.edges())
[([[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: sorted(G.edges())
[([[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 ε_i of `self`.

EXAMPLES:

```
sage: C = crystals.Letters(['A',5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```

phi(*i*)

Return φ_i of `self`.

EXAMPLES:

```
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:

```
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:

```
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:

```
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t=T(rows=[[1,2],[2]])
sage: t.stembridgeDelta_depth(1,2)
0
sage: s=T(rows=[[2,3],[3]])
sage: s.stembridgeDelta_depth(1,2)
-1
```

> **stembridgeDelta_rise**(*i, j*)
>
> Return the difference in the j-rise of 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:
> ```
> sage: T = crystals.Tableaux(['A',2], shape=[2,1])
> sage: t=T(rows=[[1,2],[2]])
> sage: t.stembridgeDelta_rise(1,2)
> -1
> sage: s=T(rows=[[2,3],[3]])
> sage: s.stembridgeDelta_rise(1,2)
> 0
> ```

> **stembridgeTriple**(*i, j*)
>
> Let A be the Cartan matrix of the crystal, x a crystal element, and let i and j be in the index set of the crystal. Further, set b=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:
> ```
> sage: T = crystals.Tableaux(['A',2], shape=[2,1])
> sage: t=T(rows=[[1,1],[2]])
> sage: t.stembridgeTriple(1,2)
> sage: s=T(rows=[[1,2],[2]])
> sage: s.stembridgeTriple(1,2)
> (-1, 0, -1)
>
> sage: T = crystals.Tableaux(['B',2], shape=[2,1])
> sage: t=T(rows=[[1,2],[2]])
> sage: t.stembridgeTriple(1,2)
> (-2, 0, -2)
> sage: s=T(rows=[[-1,-1],[0]])
> sage: s.stembridgeTriple(1,2)
> (-2, -2, 0)
> sage: u=T(rows=[[0,2],[1]])
> sage: u.stembridgeTriple(1,2)
> (-2, -1, -1)
> ```

> **weight**()
>
> Return the weight of this crystal element.
>
> EXAMPLES:
> ```
> sage: C = crystals.Letters(['A',5])
> sage: C(1).weight()
> (1, 0, 0, 0, 0, 0)
> ```

class RegularCrystals.**ParentMethods**

---

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:

```
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])
2*B[[[1, 1], [2]]] + 2*B[[[1, 2], [2]]] + 2*B[[[1, 3], [2]]] + 2*B[[[1, 1], [3]]]
+ 2*B[[[1, 2], [3]]] + 2*B[[[1, 3], [3]]] + 2*B[[[2, 2], [3]]] + 2*B[[[2, 3], [3]]]
```

    The Demazure operator is idempotent:

```
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
sage: e == T.demazure_operator(e,[1])
True

sage: all(T.demazure_operator(T.demazure_operator(C(t),[1]),[1]) == T.demazure_operator(
True
```

dual_equivalence_graph(*X=None*, *index_set=None*, *directed=True*)
    Return the dual equivalence graph indexed by `index_set` on the subset X of `self`.

    Let $b \in B$ be an element of weight 0, so $\varepsilon_j(b) = \varphi_j(b)$ for all $j \in I$, where I is the indexing set. We say b' is an i-elementary dual equivalence transformation of b (where $i \in I$) if
        •$\varepsilon_i(b) = 1$ and $\varepsilon_{i-1}(b) = 0$, and
        •$b' = f_{i-1}f_i e_{i-1}e_i b$.
    We can do the inverse procedure by interchanging i and $i - 1$ above.

    Note: If the index set is not an ordered interval, we let $i - 1$ mean the index appearing before i in I.

    This definition comes from [Assaf08] 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 $\mathcal{G}(\mathcal{X})$ in [Assaf08] Section 4 except we do not require $\varepsilon_i(b) = 0, 1$ for all i.

    This definition can be generalized by choosing a subset X of the set of all vertices of B of weight 0, and restricting the dual equivalence graph to the vertex set X.

    INPUT:
        •X – (optional) the vertex set X (default: the whole set of vertices of `self` of weight 0)

- index_set – (optional) the index set I (default: the whole index set of `self`); this has to be a subset of the index set of `self` (as a list or tuple)
- `directed` – (default: `True`) whether to have the dual equivalence graph be directed, where the head of an edge $b - b'$ is b and the tail is $b' = f_{i-1}f_i e_{i-1}e_i b$)

See also:

`sage.combinat.partition.Partition.dual_equivalence_graph()`

REFERENCES:

EXAMPLES:
```
sage: T = crystals.Tableaux(['A',3], shape=[2,2])
sage: G = T.dual_equivalence_graph()
sage: sorted(G.edges())
[([[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: sorted(G.edges())
[([[1, 3, 5], [2, 4]], [[1, 3, 4], [2, 5]], 4),
 ([[1, 3, 5], [2, 4]], [[1, 2, 5], [3, 4]], 2),
 ([[1, 3, 4], [2, 5]], [[1, 2, 4], [3, 5]], 2),
 ([[1, 2, 5], [3, 4]], [[1, 3, 5], [2, 4]], 3),
 ([[1, 2, 4], [3, 5]], [[1, 2, 3], [4, 5]], 3),
 ([[1, 2, 3], [4, 5]], [[1, 2, 4], [3, 5]], 4)]

sage: T = crystals.Tableaux(['A',4], shape=[3,1])
sage: G = T.dual_equivalence_graph(index_set=[1,2,3])
sage: G.vertices()
[[[1, 3, 4], [2]], [[1, 2, 4], [3]], [[1, 2, 3], [4]]]
sage: G.edges()
[([[1, 3, 4], [2]], [[1, 2, 4], [3]], 2),
 ([[1, 2, 4], [3]], [[1, 2, 3], [4]], 3)]
```

TESTS:
```
sage: T = crystals.Tableaux(['A',4], shape=[3,1])
sage: G = T.dual_equivalence_graph(index_set=[2,3])
sage: sorted(G.edges())
[([[1, 2, 4], [3]], [[1, 2, 3], [4]], 3),
 ([[2, 4, 5], [3]], [[2, 3, 5], [4]], 3)]
sage: sorted(G.vertices())
[[[1, 3, 4], [2]],
 [[1, 2, 4], [3]],
 [[2, 4, 5], [3]],
 [[1, 2, 3], [4]],
 [[2, 3, 5], [4]],
 [[1, 1, 1], [5]],
 [[1, 1, 5], [5]],
 [[1, 5, 5], [5]],
 [[2, 3, 4], [5]]]
```

class `RegularCrystals.`**`TensorProducts`**(*category*, *args*)

   Bases: `sage.categories.tensor.TensorProductsCategory`

   The category of regular crystals constructed by tensor product of regular crystals.

   `extra_super_categories`()
       EXAMPLES:

```
sage: RegularCrystals().TensorProducts().extra_super_categories()
[Category of regular crystals]
```

RegularCrystals.**additional_structure**()

Return `None`.

Indeed, the category of regular crystals defines no new structure: it only relates ε_a and φ_a to e_a and f_a respectively.

See also:

[Category.additional_structure()](#)

Todo

Should this category be a `CategoryWithAxiom`?

EXAMPLES:
```
sage: RegularCrystals().additional_structure()
```

RegularCrystals.**example**(*n=3*)

Returns an example of highest weight crystals, as per [Category.example()](#).

EXAMPLES:
```
sage: B = RegularCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1
```

RegularCrystals.**super_categories**()

EXAMPLES:
```
sage: RegularCrystals().super_categories()
[Category of crystals]
```

13.96 Right modules

class sage.categories.right_modules.**RightModules**(*base*, *name=None*)

Bases: [sage.categories.category_types.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 multiplation 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]
```

TESTS:
```
sage: TestSuite(RightModules(ZZ)).run()
```

class **ElementMethods**

class RightModules.**ParentMethods**

RightModules.**super_categories**()

EXAMPLES:

```
sage: RightModules(QQ).super_categories()
[Category of commutative additive groups]
```

13.97 Ring ideals

class sage.categories.ring_ideals.**RingIdeals**(*R*)

Bases: `sage.categories.category_types.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]
```

13.98 Rings

class sage.categories.rings.**Rings**(*base_category*)

Bases: `sage.categories.category_with_axiom.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
```

TESTS:

```
sage: TestSuite(Rings()).run()
```

Todo

(see: http://trac.sagemath.org/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**

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: MS = MatrixSpace(ZZ, 2)
> sage: MS.one().is_unit()
> True
> sage: MS.zero().is_unit()
> False
> sage: MS([1,2,3,4]).is_unit()
> Traceback (most recent call last):
> ...
> NotImplementedError
> ```

Rings.**NoZeroDivisors**

> alias of `Domains`

class Rings.**ParentMethods**

bracket(*x*, *y*)

> Returns the Lie bracket $[x, y] = xy - yx$ of x and y.

> INPUT:
> - x, y – elements of `self`

> EXAMPLES:

> ```
> sage: F = AlgebrasWithBasis(QQ).example()
> sage: F
> ```

```
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c')
sage: a,b,c = F.algebra_generators()
sage: F.bracket(a,b)
B[word: ab] - B[word: ba]
```

This measures the default of commutation between x and y. F endowed with the bracket operation is a Lie algebra; in particular, it satisfies Jacobi's identity:

```
sage: F.bracket( F.bracket(a,b), c) + F.bracket(F.bracket(b,c),a) + F.bracket(F.bracket(
0
```

characteristic()
    Return the characteristic of this ring.

    EXAMPLES:
```
sage: QQ.characteristic()
0
sage: GF(19).characteristic()
19
sage: Integers(8).characteristic()
8
sage: Zp(5).characteristic()
0
```

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 trac ticket #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.

    EXAMPLE:
```
sage: MS = MatrixSpace(QQ,2,2)
sage: isinstance(MS,Ring)
False
sage: MS in Rings()
True
sage: MS.ideal(2)
Twosided Ideal
(
  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS.ideal([MS.0,MS.1],side='right')
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 trac ticket #7797.

EXAMPLE:

```
sage: MS = MatrixSpace(QQ,2,2)
sage: isinstance(MS,Ring)
False
sage: MS in Rings()
True
sage: MS.ideal_monoid()
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices
over Rational Field
```

Note that the monoid is cached:

```
sage: MS.ideal_monoid() is MS.ideal_monoid()
True
```

`is_ring`()

Return True, since this in an object of the category of rings.

EXAMPLES:

```
sage: Parent(QQ,category=Rings()).is_ring()
True
```

`is_zero`()

Return `True` if this is the zero ring.

EXAMPLES:

```
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)[]
sage: R.quo(77).is_zero()
True
sage: R.quo(x^2+1).is_zero()
False
```

`quo`(*I*, *names=None*)

Quotient of a ring by a two-sided ideal.

NOTE:

This is a synonym for `quotient()`.

EXAMPLE:

```
sage: MS = MatrixSpace(QQ,2)
sage: I = MS*MS.gens()*MS
```

`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)
False
sage: isinstance(MS,Rings().parent_class)
True
sage: MS.quo(I,names = ['a','b','c','d'])
Quotient of Full MatrixSpace of 2 by 2 dense matrices over Rational Field by the ideal
(
  [1 0]
  [0 0],

  [0 1]
  [0 0],

  [0 0]
  [1 0],

  [0 0]
  [0 1]
)
```

quotient(*I*, *names=None*)

Quotient of a ring by a two-sided ideal.

INPUT:

- `I`: A twosided ideal of this ring.
- `names`: a list of strings to be used as names for the variables in the quotient ring.

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:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: from sage.rings.noncommutative_ideals import Ideal_nc
sage: class PowerIdeal(Ideal_nc):
...      def __init__(self, R, n):
...          self._power = n
...          Ideal_nc.__init__(self,R,[R.prod(m) for m in CartesianProduct(*[R.gens()]*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)
sage: Q = Rings().parent_class.quotient(F,I); Q
Quotient of Free Algebra on 3 generators (x, y, z) over Rational Field by the ideal (x^3
sage: Q.0
xbar
sage: Q.1
ybar
sage: Q.2
zbar
sage: Q.0*Q.1
xbar*ybar
sage: Q.0*Q.1*Q.0
0
```

quotient_ring(*I*, *names=None*)

Quotient of a ring by a two-sided ideal.

NOTE:

This is a synonyme for `quotient()`.

EXAMPLE:
```
sage: MS = MatrixSpace(QQ,2)
sage: I = MS*MS.gens()*MS
```

`MS` is not an instance of `Ring`, but it is an instance of the parent class of the category of rings. The quotient method is inherited from there:
```
sage: isinstance(MS,sage.rings.ring.Ring)
False
sage: isinstance(MS,Rings().parent_class)
True
sage: MS.quotient_ring(I,names = ['a','b','c','d'])
Quotient of Full MatrixSpace of 2 by 2 dense matrices over Rational Field by the ideal
(
  [1 0]
  [0 0],

  [0 1]
  [0 0],

  [0 0]
  [1 0],

  [0 0]
  [0 1]
)
```

class `Rings.`**`SubcategoryMethods`**

    `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.

        Note: This could be generalized to `MagmasAndAdditiveMagmas.Distributive.AdditiveUnital`.

        EXAMPLES:
```
sage: Rings().Division()
Category of division rings
sage: Rings().Commutative().Division()
Category of fields
```

        TESTS:
```
sage: TestSuite(Rings().Division()).run()
sage: Algebras(QQ).Division.__module__
'sage.categories.rings'
```

    `NoZeroDivisors``()`

        Return the full subcategory of the objects of `self` having no nonzero zero divisors.

        A *zero divisor* in a ring R is an element $x \in R$ such that there exists a nonzero element $y \in R$ such that $x \cdot y = 0$ or $y \cdot x = 0$ (see Wikipedia article Zero_divisor).

        EXAMPLES:

```
sage: Rings().NoZeroDivisors()
Category of domains
```

Note: This could be generalized to `MagmasAndAdditiveMagmas.Distributive.AdditiveUnital`.

TESTS:
```
sage: TestSuite(Rings().NoZeroDivisors()).run()
sage: Algebras(QQ).NoZeroDivisors.__module__
'sage.categories.rings'
```

13.99 Rngs

class sage.categories.rngs.**Rngs**(*base_category*)
    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

    The category of rngs.

    An *rng* $(S, +, *)$ is similar to a ring but not necessarilly unital. In other words, it is a combination of a commutative additive group $(S, +)$ and a multiplicative semigroup $(S, *)$, where $*$ distributes over $+$.

    EXAMPLES:
```
sage: C = Rngs(); C
Category of rngs
sage: sorted(C.super_categories(), key=str)
[Category of associative additive commutative additive associative additive unital distributive
 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
```

    TESTS:
```
sage: TestSuite(C).run()
```

    Unital
        alias of `Rings`

13.100 Schemes

class sage.categories.schemes.**Schemes**(*s=None*)
    Bases: `sage.categories.category.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.

TESTS:
```
sage: TestSuite(Schemes()).run()
```

Check that Hom sets of schemes are in the correct category:
```
sage: Schemes().Homsets().super_categories()
[Category of homsets]
```

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

class sage.categories.schemes.**Schemes_over_base**(*base*, *name=None*)
    Bases: `sage.categories.category_types.Category_over_base`

The category of schemes over a given base scheme.

EXAMPLES:
```
sage: Schemes(Spec(ZZ))
Category of schemes over Integer Ring
```

TESTS:
```
sage: C = Schemes(ZZ)
sage: TestSuite(C).run()
```

base_scheme()
    EXAMPLES:
```
sage: Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring
```

super_categories()
    EXAMPLES:
```
sage: Schemes(Spec(ZZ)).super_categories()
[Category of schemes]
```

13.101 Semigroups

class sage.categories.semigroups.**Semigroups**(*base_category*)

> Bases: sage.categories.category_with_axiom.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:
> ```
> sage: C = Semigroups(); C
> Category of semigroups
> sage: C.super_categories()
> [Category of magmas]
> sage: C.all_super_categories()
> [Category of semigroups, Category of magmas,
>  Category of sets, Category of sets with partial maps, Category of objects]
> sage: C.axioms()
> frozenset({'Associative'})
> sage: C.example()
> An example of a semigroup: the left zero semigroup
> ```

> TESTS:
> ```
> sage: TestSuite(C).run()
> ```

> **class Algebras**(*category*, **args*)

> > Bases: sage.categories.algebra_functor.AlgebrasCategory

> > TESTS:
> > ```
> > sage: TestSuite(Semigroups().Algebras(QQ)).run()
> > sage: TestSuite(Semigroups().Finite().Algebras(QQ)).run()
> > ```

> > **class ParentMethods**

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

> > > > They correspond to the generators of the semigroup.

> > > > EXAMPLES:
> > > > ```
> > > > 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()
> > > > Finite family {0: B['a'], 1: B['b'], 2: B['c'], 3: B['d']}
> > > > ```

> > > **product_on_basis**(*g1*, *g2*)
> > > > Product, on basis elements, as per MagmaticAlgebras.WithBasis.ParentMethods.product_on_ba

> > > > 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',
sage: A = S.algebra(QQ)
sage: a,b,c,d = A.algebra_generators()
sage: a * b + b * d * c * d
B['ab'] + B['bdc']
```

Semigroups.Algebras.**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]
```

class Semigroups.**CartesianProducts**(*category*, **args*)
Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

extra_super_categories()
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 Semigroups.**ElementMethods**

Semigroups.**Finite**
alias of `FiniteSemigroups`

Semigroups.**FinitelyGeneratedAsMagma**
alias of `FinitelyGeneratedSemigroups`

class Semigroups.**ParentMethods**

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
Dihedral group of order 8 as a permutation group
sage: G = D4.cayley_graph()
sage: show(G, color_by_label=True, edge_labels=True)
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.show3d(color_by_label=True, edge_size=0.01, edge_size2=0.02, vertex_size=0.03)
sage: G.show3d(vertex_size=0.03, edge_size=0.01, edge_size2=0.02, vertex_colors={(1,1,1)
sage: G.num_edges()
120
```

```
sage: w = WeylGroup(['A',3])
sage: d = w.cayley_graph(); d
Digraph on 24 vertices
sage: d.show3d(color_by_label=True, edge_size=0.01, vertex_size=0.03)
```

Alternative generators may be specified:

```
sage: G = A5.cayley_graph(generators=[A5.gens()[0]])
sage: G.num_edges()
60
sage: g=PermutationGroup([(i+1,j+1) for i in range(5) for j in range(5) if j!=i])
sage: g.cayley_graph(generators=[(1,2),(2,3)])
Digraph on 120 vertices
```

If `elements` is specified, then only the subgraph induced and those elements is returned. Here we use it to display the Cayley graph of the free monoid truncated on the elements of length at most 3:

```
sage: 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
sage: G = M.cayley_graph(elements = elements)
sage: G.num_verts(), G.num_edges()
(85, 84)
sage: G.show3d(color_by_label=True, edge_size=0.001, vertex_size=0.01)
```

We now illustrate the `side` and `simple` options on a semigroup:

```
sage: S = FiniteSemigroups().example(alphabet=('a','b'))
sage: g = S.cayley_graph(simple=True)
sage: g.vertices()
['a', 'ab', 'b', 'ba']
sage: g.edges()
[('a', 'ab', None), ('b', 'ba', None)]
```

```
sage: g = S.cayley_graph(side="left", simple=True)
sage: g.vertices()
```

```
['a', 'ab', 'b', 'ba']
sage: g.edges()
[('a', 'ba', None), ('ab', 'ba', None), ('b', 'ab', None),
('ba', 'ab', None)]

sage: g = S.cayley_graph(side="twosided", simple=True)
sage: g.vertices()
['a', 'ab', 'b', 'ba']
sage: g.edges()
[('a', 'ab', None), ('a', 'ba', None), ('ab', 'ba', None),
('b', 'ab', None), ('b', 'ba', None), ('ba', 'ab', None)]

sage: g = S.cayley_graph(side="twosided")
sage: g.vertices()
['a', 'ab', 'b', 'ba']
sage: g.edges()
[('a', 'a', (0, 'left')), ('a', 'a', (0, 'right')), ('a', 'ab', (1, 'right')), ('a', 'ba

sage: s1 = SymmetricGroup(1); s = s1.cayley_graph(); s.vertices()
[()]
```

TESTS:

```
sage: SymmetricGroup(2).cayley_graph(side="both")
Traceback (most recent call last):
...
ValueError: option 'side' must be 'left', 'right' or 'twosided'
```

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)
- Robert Miller (2008-05-01): editing
- 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')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

prod(*args*)

Return the product of the list of elements `args` inside `self`.

EXAMPLES:

```
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
```

semigroup_generators()

Return distinguished semigroup generators for `self`.

OUTPUT: a family

This method is optional.

EXAMPLES:

```
sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

subsemigroup(*generators*, *one=None*, *category=None*)

Return the multiplicative subsemigroup generated by `generators`.

INPUT:

- `generators` – a finite family of elements of `self`, or a list, iterable, ... that can be converted into one (see `Family`).
- `one` – a unit for the subsemigroup, or `None`.
- `category` – a category

This implementation lazily constructs all the elements of the semigroup, and the right Cayley graph relations between them, and uses the latter as an automaton.

See `sage.sets.monoids.AutomaticSemigroup` for details.

EXAMPLES:

```
sage: R = IntegerModRing(15)
sage: M = R.subsemigroup([R(3),R(5)]); M
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: M.list()
[3, 5, 9, 0, 10, 12, 6]
```

By default, M is just in the category of subsemigroups:

```
sage: M in Semigroups().Subobjects()
True
```

In the following example, we specify that M is a submonoid of the finite monoid R (it shares the same unit), and a group by itself:

```
sage: M = R.subsemigroup([R(-1)],
....:     category=Monoids().Finite().Subobjects() & Groups()); M
A submonoid of (Ring of integers modulo 15) with 1 generators
sage: M.list()
[1, 14]
sage: M.one()
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 explictly:

```
sage: M = R.subsemigroup([R(5)],
....:     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),
....:      category=Semigroups().Finite().Subobjects() & Groups()); M
A subsemigroup of (Ring of integers modulo 15) with 1 generators
sage: M in Groups()
True
sage: M.list()
[10, 5]
sage: M.one()
10
```

TESTS:

```
sage: TestSuite(M).run()
Failure in _test_inverse:
Traceback (most recent call last):
...
The following tests failed: _test_inverse
```

Todo

- Fix the above failure 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?

---

class Semigroups.**Quotients**(*category*, **args*)

Bases: sage.categories.quotients.QuotientsCategory

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:      _functor_category = "FooBars"
....:      _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

semigroup_generators()

Return semigroup generators for self by retracting the semigroup generators of the ambient semigroup.

EXAMPLES:

```
sage: S = FiniteSemigroups().Quotients().example().semigroup_generators() # todo: not
```

Semigroups.Quotients.**example**()

Return an example of quotient of a semigroup, as per Category.example().

EXAMPLES:

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

class Semigroups.**Subquotients**(*category*, **args*)

Bases: `sage.categories.subquotients.SubquotientsCategory`

The category of subquotient semi-groups.

EXAMPLES:
```
sage: Semigroups().Subquotients().all_super_categories()
[Category of subquotients of semigroups,
 Category of semigroups,
 Category of subquotients of magmas,
 Category of magmas,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]

[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]
```

example()

Returns an example of subquotient of a semigroup, as per `Category.example()`.

EXAMPLES:
```
sage: Semigroups().Subquotients().example()
An example of a (sub)quotient semigroup: a quotient of the left zero semigroup
```

Semigroups.**Unital**

alias of `Monoids`

Semigroups.**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')
```

13.102 Semirngs

class `sage.categories.semirings.`**`Semirings`**(*base_category*)

Bases: `sage.categories.category_with_axiom.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 $+$.

EXAMPLES:

```
sage: Semirings()
Category of semirings
sage: Semirings().super_categories()
[Category of associative additive commutative additive associative additive unital distributive
 Category of monoids]

sage: sorted(Semirings().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveUnital', 'Associative', 'Distributive',

sage: Semirings() is (CommutativeAdditiveMonoids() & Monoids()).Distributive()
True

sage: Semirings().AdditiveInverse()
Category of rings
```

TESTS:

```
sage: TestSuite(Semirings()).run()
```

13.103 Semisimple Algebras

class `sage.categories.semisimple_algebras.`**`SemisimpleAlgebras`**(*base*, *name=None*)

Bases: `sage.categories.category_types.Category_over_base_ring`

The category of semisimple algebras over a given base ring.

EXAMPLES:

```
sage: from sage.categories.semisimple_algebras import SemisimpleAlgebras
sage: C = SemisimpleAlgebras(QQ); C
Category of semisimple algebras over Rational Field
```

This category is best constructed as:

```
sage: D = Algebras(QQ).Semisimple(); D
Category of semisimple algebras over Rational Field
sage: D is C
True

sage: C.super_categories()
[Category of algebras over Rational Field]
```

Typically, finite group algebras are semisimple:

```
sage: DihedralGroup(5).algebra(QQ) in SemisimpleAlgebras
True
```

Unless the characteristic of the field divides the order of the group:

```
sage: DihedralGroup(5).algebra(IntegerModRing(5)) in SemisimpleAlgebras
False
```

```
sage: DihedralGroup(5).algebra(IntegerModRing(7)) in SemisimpleAlgebras
True
```

See also:

http://en.wikipedia.org/wiki/Semisimple_algebra

TESTS:

```
sage: TestSuite(C).run()
```

class **FiniteDimensional**(*base_category*)

    Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

    TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>
```

```
sage: TestSuite(C).run()
```

    WithBasis

        alias of `FiniteDimensionalSemisimpleAlgebrasWithBasis`

class SemisimpleAlgebras.**ParentMethods**

    radical_basis(***keywords*)

        Return a basis of the Jacobson radical of this algebra.

            •keywords – for compatibility; ignored.

        OUTPUT: the empty list since this algebra is semisimple.

        EXAMPLES:

```
sage: A = SymmetricGroup(4).algebra(QQ)
sage: A.radical_basis()
[]
```

        TESTS:

```
sage: A.radical_basis.__module__
'sage.categories.finite_dimensional_algebras_with_basis'
```

SemisimpleAlgebras.**super_categories**()

    EXAMPLES:

```
sage: Algebras(QQ).Semisimple().super_categories()
[Category of algebras over Rational Field]
```

13.104 Sets

exception `sage.categories.sets_cat.`**`EmptySetError`**

    Bases: `exceptions.ValueError`

    Exception raised when some operation can't be performed on the empty set.

    EXAMPLES:

```
sage: def first_element(st):
...      if not st: raise EmptySetError, "no elements"
...      else: return st[0]
sage: first_element(Set((1,2,3)))
1
sage: first_element(Set([]))
Traceback (most recent call last):
...
EmptySetError: no elements
```

class `sage.categories.sets_cat.`**`Sets`**(*s=None*)

    Bases: `sage.categories.category_singleton.Category_singleton`

    The category of sets.

    The base category for collections of elements with = (equality).

    This is also the category whose objects are all parents.

    EXAMPLES:

```
sage: Sets()
Category of sets
sage: Sets().super_categories()
[Category of sets with partial maps]
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'>
<type 'sage.misc.fast_methods.WithEqualityById'>
<type 'sage.structure.parent.Parent'>
<type 'sage.structure.category_object.CategoryObject'>
<type 'sage.structure.sage_object.SageObject'>
```

```
<class 'sage.categories.sets_cat.Sets.parent_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.parent_class'>
<class 'sage.categories.objects.Objects.parent_class'>
<type 'object'>
```

We run some generic checks on P:

```
sage: TestSuite(P).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_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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

Now, we manipulate some elements of P:

```
sage: P.an_element()
47
sage: x = P(3)
sage: x.parent()
Set of prime numbers
sage: x in P, 4 in P
(True, False)
sage: x.is_prime()
True
```

They get their methods from the following classes:

```
sage: for cl in x.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits.Element'>
<type 'sage.rings.integer.IntegerWrapper'>
<type 'sage.rings.integer.Integer'>
<type 'sage.structure.element.EuclideanDomainElement'>
<type 'sage.structure.element.PrincipalIdealDomainElement'>
<type 'sage.structure.element.DedekindDomainElement'>
<type 'sage.structure.element.IntegralDomainElement'>
<type 'sage.structure.element.CommutativeRingElement'>
<type 'sage.structure.element.RingElement'>
<type 'sage.structure.element.ModuleElement'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element'>
<type 'sage.structure.element.Element'>
<type 'sage.structure.sage_object.SageObject'>
<class 'sage.categories.sets_cat.Sets.element_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.element_class'>
<class 'sage.categories.objects.Objects.element_class'>
```

```
<type 'object'>
```

FIXME: Objects.element_class is not very meaningful ...

TESTS:
```
sage: TestSuite(Sets()).run()
```

class **Algebras** (*category*, **args*)

    Bases: `sage.categories.algebra_functor.AlgebrasCategory`

    TESTS:
```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python n
sage: TestSuite(C).run()
```

    extra_super_categories ()
        EXAMPLES:
```
sage: Sets().Algebras(ZZ).super_categories()
[Category of modules with basis over Integer Ring]

sage: Sets().Algebras(QQ).extra_super_categories()
[Category of vector spaces with basis over Rational Field]

sage: Sets().example().algebra(ZZ).categories()
[Category of set algebras over Integer Ring,
 Category of modules with basis over Integer Ring,
 ...
 Category of objects]
```

class Sets.**CartesianProducts** (*category*, **args*)

    Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

    EXAMPLES:
```
sage: C = Sets().CartesianProducts().example()
sage: C
The cartesian product of (Set of prime numbers (basic implementation),
 An example of an infinite enumerated set: the non negative integers,
 An example of a finite enumerated set: {1,2,3})
sage: C.category()
Category of Cartesian products of sets
sage: C.categories()
[Category of Cartesian products of sets, Category of sets,
 Category of sets with partial maps,
 Category of objects]
sage: TestSuite(C).run()
```

class **ElementMethods**

cartesian_factors()
> Return the cartesian factors of `self`.

> EXAMPLES:
```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: H = CombinatorialFreeModule(ZZ, [4,7]); H.__custom_name = "H"
sage: S = cartesian_product([F, G, H])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6)) + 4 * S.m
sage: x.cartesian_factors()
(B[4] + 2*B[5], 3*B[6], 4*B[4] + 5*B[7])
sage: [s.parent() for s in x.cartesian_factors()]
[F, G, H]
sage: S.zero().cartesian_factors()
(0, 0, 0)
sage: [s.parent() for s in S.zero().cartesian_factors()]
[F, G, H]
```

cartesian_projection(*i*)
> Return the projection of `self` onto the i-th factor of the cartesian product.

> INPUTS:
> •`i` – the index of a factor of the cartesian product
> EXAMPLES:
```
sage: F = CombinatorialFreeModule(ZZ, [4,5]); F.__custom_name = "F"
sage: G = CombinatorialFreeModule(ZZ, [4,6]); G.__custom_name = "G"
sage: S = cartesian_product([F, G])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.monomial((1,6))
sage: x.cartesian_projection(0)
B[4] + 2*B[5]
sage: x.cartesian_projection(1)
3*B[6]
```

summand_projection(**args*, ***kwds*)
> Deprecated: Use `cartesian_projection()` instead. See trac ticket #10963 for details.

summand_split(**args*, ***kwds*)
> Deprecated: Use `cartesian_factors()` instead. See trac ticket #10963 for details.

class Sets.CartesianProducts.**ParentMethods**

an_element()
> EXAMPLES:
```
sage: C = Sets().CartesianProducts().example(); C
The cartesian product of (Set of prime numbers (basic implementation),
 An example of an infinite enumerated set: the non negative integers,
 An example of a finite enumerated set: {1,2,3})
sage: C.an_element()
(47, 42, 1)
```

cardinality()
> Return the cardinality of self.

> EXAMPLES:

```
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E,SymmetricGroup(4)])
sage: C.cardinality()
72

sage: E = FiniteEnumeratedSet([])
sage: C = cartesian_product([E, ZZ, QQ])
sage: C.cardinality()
0

sage: C = cartesian_product([ZZ, QQ])
sage: C.cardinality()
+Infinity

sage: cartesian_product([GF(5), Permutations(10)]).cardinality()
18144000
sage: cartesian_product([GF(71)]*20).cardinality() == 71**20
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})
sage: x = C.an_element(); x
(47, 42, 1)
sage: pi = C.cartesian_projection(1)
sage: pi(x)
42
```

is_empty()

Return whether this set is empty.

EXAMPLES:
```
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:
```
sage: E = FiniteEnumeratedSet([1,2,3])
sage: C = cartesian_product([E, SymmetricGroup(4)])
```

```
sage: C.is_finite()
True

sage: cartesian_product([ZZ,ZZ]).is_finite()
False
sage: cartesian_product([ZZ, Set(), ZZ]).is_finite()
True
```

random_element(**args*)
  Return a random element of this cartesian product.

  The extra arguments are passed down to each of the factors of the cartesian product.

  EXAMPLES:
```
sage: C = cartesian_product([Permutations(10)]*5)
sage: C.random_element()            # random
([2, 9, 4, 7, 1, 8, 6, 10, 5, 3],
 [8, 6, 5, 7, 1, 4, 9, 3, 10, 2],
 [5, 10, 3, 8, 2, 9, 1, 4, 7, 6],
 [9, 6, 10, 3, 2, 1, 5, 8, 7, 4],
 [8, 5, 2, 9, 10, 3, 7, 1, 4, 6])

sage: C = cartesian_product([ZZ]*10)
sage: c1 = C.random_element()
sage: c1                       # random
(3, 1, 4, 1, 1, -3, 0, -4, -17, 2)
sage: c2 = C.random_element(4,7)
sage: c2                       # random
(6, 5, 6, 4, 5, 6, 6, 4, 5, 5)
sage: all(4 <= i < 7 for i in c2)
True
```

Sets.CartesianProducts.**example**()
  EXAMPLES:
```
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})
```

Sets.CartesianProducts.**extra_super_categories**()
  A cartesian product of sets is a set.

  EXAMPLES:
```
sage: Sets().CartesianProducts().extra_super_categories()
[Category of sets]
sage: Sets().CartesianProducts().super_categories()
[Category of sets]
```

class Sets.**ElementMethods**

cartesian_product(**elements*)
  Return the cartesian product of its arguments, as an element of the cartesian product of the parents of those elements.

  EXAMPLES:
```
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()
sage: (a,b,c) = A.algebra_generators()
```

```
sage: a.cartesian_product(b, c)
B[(0, word: a)] + B[(1, word: b)] + B[(2, word: c)]
```

FIXME: is this a policy that we want to enforce on all parents?

Sets.**Facade**
 alias of FacadeSets

Sets.**Finite**
 alias of FiniteSets

class Sets.**Infinite**(*base_category*)
 Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

 TESTS:
```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

 class **ParentMethods**

  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
```

  TESTS:
```
sage: C.is_finite.im_func is sage.categories.sets_cat.Sets.Infinite.ParentMethods.is_
True
```

class Sets.**IsomorphicObjects**(*category*, **args*)
 Bases: sage.categories.isomorphic_objects.IsomorphicObjectsCategory

A category for isomorphic objects of sets.

EXAMPLES:

```
sage: Sets().IsomorphicObjects()
Category of isomorphic objects of sets
sage: Sets().IsomorphicObjects().all_super_categories()
[Category of isomorphic objects of sets,
 Category of subobjects of sets, Category of quotients of sets,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

class **ParentMethods**

class Sets.**MorphismMethods**

class Sets.**ParentMethods**

CartesianProduct
> alias of [CartesianProduct]

algebra(*base_ring*, *category=None*)
> Return the algebra of `self` over `base_ring`.

> INPUT:
> - `self` – a parent S
> - `base_ring` – a ring K
> - `category` – a super category of the category of S, or `None`

> This returns the K-free module with basis indexed by S, endowed with whatever structure can be induced from that of S. Note that the `category` keyword needs to be fed with the structure on S to be used, not the structure that one wants to obtain on the result; see the examples below.

> EXAMPLES:

> If S is a monoid, the result is the monoid algebra KS:

```
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
Free module generated by An example of a monoid: the free monoid generated by ('a', 'b',
sage: A.category()
Category of monoid algebras over Rational Field
```

> If S is a group, the result is the group algebra KS:

```
sage: S = Groups().example(); S
General Linear Group of degree 4 over Rational Field
sage: A = S.algebra(QQ); A
Group algebra of General Linear Group of degree 4 over Rational Field over Rational Fiel
sage: A.category()
Category of group algebras over Rational Field
```

> which is actually a Hopf algebra:

```
sage: A in HopfAlgebras(QQ)
True
```

> By Maschke's theorem, for a finite group whose cardinality does not divide the characteristic of the base field, the algebra is semisimple:

```
sage: SymmetricGroup(5).algebra(QQ) in Algebras(QQ).Semisimple()
True
sage: CyclicPermutationGroup(10).algebra(FiniteField(5)) in Algebras.Semisimple
False
sage: CyclicPermutationGroup(10).algebra(FiniteField(7)) in Algebras.Semisimple
True
```

One may specify for which category one takes the algebra:

```
sage: A = S.algebra(QQ, category=Sets()); A
Free module generated by General Linear Group of degree 4 over Rational Field over Ratio
sage: A.category()
Category of set algebras over Rational Field
```

One may construct as well algebras of additive magmas, semigroups, monoids, or groups:

```
sage: S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b',
sage: U = S.algebra(QQ); U
Free module generated by An example of a commutative monoid: the free commutative monoid
```

Despite saying "free module", this is really an algebra and its elements can be multiplied:

```
sage: U in Algebras(QQ)
True
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: U(a) * U(b)
B[a + b]
```

Constructing the algebra of a set endowed with both an additive and a multiplicative structure is ambiguous:

```
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 sem
Constructing its algebra is ambiguous.
Please use, e.g., S.algebra(QQ, category=Semigroups())
```

The ambiguity can be resolved using the `category` argument:

```
sage: A = Z3.algebra(QQ, category=Monoids()); A
Free module generated by 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
Free module generated by Ring of integers modulo 3 over Rational Field
sage: A.category()
Category of finite dimensional commutative additive group algebras over Rational Field
```

Similarly, on , we obtain for additive magmas, monoids, groups.

> **Warning:** As we have seen, in most practical use cases, the result is actually an algebra, hence the name of this method. In the other cases this name is misleading:
>
> ```
> sage: A = Sets().example().algebra(QQ); A
> Free module generated by Set of prime numbers (basic implementation) over Rational Fiel
> sage: A.category()
> Category of set algebras over Rational Field
> sage: A in Algebras(QQ)
> False
> ```
>
> Suggestions for a uniform, meaningful, and non misleading name are welcome!

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*)

   Return the cartesian product of the parents.

   EXAMPLES:
   ```
   sage: C = AlgebrasWithBasis(QQ)
   sage: A = C.example(); A.rename("A")
   sage: A.cartesian_product(A,A)
   A (+) A (+) A
   sage: ZZ.cartesian_product(GF(2), FiniteEnumeratedSet([1,2,3]))
   The cartesian product of (Integer Ring, Finite Field of size 2, {1, 2, 3})

   sage: C = ZZ.cartesian_product(A); C
   The cartesian product of (Integer Ring, A)
   ```

   TESTS:
   ```
   sage: type(C)
   <class 'sage.sets.cartesian_product.CartesianProduct_with_category'>
   sage: C.category()
   Join of Category of rings and ...
       and Category of Cartesian products of commutative additive groups
   ```

is_parent_of(*element*)

   Return whether `self` is the parent of `element`.

   INPUT:
      • element – any object
   EXAMPLES:
   ```
   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:

```
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: S.some_elements()
[]
```

This method should return an iterable, *not* an iterator.

class Sets.**Quotients**(*category*, **args*)
Bases: `sage.categories.quotients.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**

class Sets.**Realizations**(*category*, **args*)
Bases: `sage.categories.realizations.RealizationsCategory`

TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:       _functor_category = "FooBars"
....:       _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
```

```
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class **ParentMethods**

> **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()
>> The subset algebra of {1, 2, 3} over Rational Field
>> ```

class Sets.**SubcategoryMethods**

> **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:
>> ```
>> 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`
>>
>> TESTS:
>> ```
>> sage: TestSuite(Groups().Finite().Algebras(QQ)).run()
>> ```

> **CartesianProducts**()
>> Return the full subcategory of the objects of `self` constructed as cartesian products.
>>
>> See also:
>>> •`cartesian_product.CartesianProductFunctor`

> • RegressiveCovariantFunctorialConstruction

EXAMPLES:

```
sage: Sets().CartesianProducts()
Category of Cartesian products of sets
sage: Semigroups().CartesianProducts()
Category of Cartesian products of semigroups
sage: EuclideanDomains().CartesianProducts()
Join of Category of rings and Category of Cartesian products of ...
```

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 **Z** is modelled by the parent ZZ, 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])                        # random
{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:

```
sage: Posets().example("facade")
An example of a facade poset: the positive integers ordered by divisibility
```

Let us investigate in detail a close variant of this last example: let P be set of divisors of 12 partially ordered by divisibility. There are two options for representing its elements:

1.as plain integers:

```
sage: P = Poset((divisors(12), attrcall("divides")), facade=True)
```

2.as integers, modified to be aware that their parent is P:

```
sage: Q = Poset((divisors(12), attrcall("divides")), facade=False)
```

The advantage of option 1. is that one needs not do conversions back and forth between P and $\mathbf{Z}$. 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?:

```
sage: 2 < 3
True
```

To raise this ambiguity, one needs to explicitly specify the underlying poset as in $2 <_P 3$:

```
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:

```
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!:

```
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:

```
sage: Sets().example("facade")
Set of prime numbers (facade implementation)
sage: Sets().example("inherits")
Set of prime numbers
sage: Sets().example("wrapper")
Set of prime numbers (wrapper implementation)
```

Specifications

A parent which is a facade must either:
- call `Parent.__init__()` using the `facade` parameter to specify a parent, or tuple thereof.
- overload the method `facade_for()`.

Note: The concept of facade parents was originally introduced in the computer algebra system MuPAD.

TESTS:

Check that multiple categories initialisation works (trac ticket #13801):

```
sage: class A(Parent):
....:    def __init__(self):
....:        Parent.__init__(self, category=(FiniteEnumeratedSets(),Monoids()), facade=Tr
sage: a = A()
```

TESTS:

```
sage: Posets().Facade()
Category of facade posets
sage: Posets().Facade().Finite() is  Posets().Finite().Facade()
True
```

Facades(**args*, ***kwds*)

Deprecated: Use `Facade()` instead. See trac ticket #17073 for details.

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
```

TESTS:

```
sage: TestSuite(Sets().Finite()).run()
sage: Rings().Finite.__module__
'sage.categories.sets_cat'
```

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
```

TESTS:

```
sage: TestSuite(Sets().Infinite()).run()
sage: Rings().Infinite.__module__
'sage.categories.sets_cat'
```

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:

```
sage: A in FiniteEnumeratedSets()
True
sage: A.cardinality()
3
sage: A.list()
[1, 4, 9]
```

The isomorphism from B to A is available as:

```
sage: A.retract(3)
9
```

and its inverse as:

```
sage: A.lift(9)
3
```

It often is natural to declare those morphisms as coercions so that one can do `A(b)` and `B(a)` to go back and forth between A and B (TODO: refer to a category example where the maps are declared as a coercion). This is not done by default. Indeed, in many cases one only wants to transport part of the structure of B to A. Assume for example, that one wants to construct the set of integers $B = ZZ$, endowed with max as addition, and $+$ as multiplication instead of the usual $+$ and $*$. 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:

```
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]
```

Unless something specific about isomorphic objects is implemented for this category, one actually get an optimized super category:

```
sage: C = Semigroups().IsomorphicObjects(); C
Join of Category of quotients of semigroups
    and Category of isomorphic objects of sets
```

See also:

- `Subquotients()` for background
- `isomorphic_objects.IsomorphicObjectsCategory`
- `RegressiveCovariantFunctorialConstruction`

TESTS:

```
sage: TestSuite(Sets().IsomorphicObjects()).run()
```

`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: 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: EuclideanDomains().Quotients()
Join of Category of euclidean domains
    and Category of subquotients of monoids
    and Category of quotients of semigroups
```

TESTS:
```
sage: TestSuite(C).run()
```

`Subobjects()`

Return the full subcategory of the objects of `self` constructed as subobjects.

Given a concrete category `As()` (i.e. a subcategory of `Sets()`), `As().Subobjects()` returns the category of objects of `As()` endowed with a distinguished embedding into some other object of `As()`.

Implementing an object of `As().Subobjects()` is done in the same way as for `As().Subquotients()`; namely by providing an ambient space and a lift and a retract map. In the case of a trivial embedding, the two maps will typically be identity maps that just change the parent of their argument. See `Subquotients()` for detailed instructions.

See also:

- `Subquotients()` for background
- `subobjects.SubobjectsCategory`

• `RegressiveCovariantFunctorialConstruction`

EXAMPLES:
```
sage: C = Sets().Subobjects(); C
Category of subobjects of sets
```
```
sage: C.super_categories()
[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:
```
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.

TESTS:
```
sage: Semigroups().Subobjects().is_subcategory(Semigroups().Subquotients())
True
sage: TestSuite(C).run()
```

Subquotients()
   Return the full subcategory of the objects of `self` constructed as subquotients.

   Given a concrete category `self == As()` (i.e. a subcategory of `Sets()`), `As().Subquotients()` returns the category of objects of `As()` endowed with a distinguished description as subquotient of some other object of `As()`.

   EXAMPLES:
```
sage: Monoids().Subquotients()
Category of subquotients of monoids
```

   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:

   $$l : A \to B' \text{ and } r : B' \to A$$

   called respectively the *lifting map* and *retract map* such that $r \circ l$ is the identity of A and r is a morphism in `As()`.

   Todo

   Draw the typical commutative diagram.

   It follows that, for each operation op of the category, we have some property like:

   $$op_A(e) = r(op_B(l(e))), \text{ for all } e \in A$$

   This allows for implementing the operations on A from those on B.

   The two most common use cases are:

- *homomorphic images* (or *quotients*), when $B' = B$, r is an homomorphism from B to A (typically a canonical quotient map), and l a section of it (not necessarily a homomorphism); see `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) 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
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 $\mathbf{Z}/n\mathbf{Z}$.

See also:

- `Quotients()`, `Subobjects()`, `IsomorphicObjects()`
- `subquotients.SubquotientsCategory`

  •RegressiveCovariantFunctorialConstruction

TESTS:
```
sage: TestSuite(Sets().Subquotients()).run()
```

class Sets.**Subobjects**(*category*, **args*)
  Bases: `sage.categories.subobjects.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**

class Sets.**Subquotients**(*category*, **args*)
  Bases: `sage.categories.subquotients.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**

    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 Sets.Subquotients.**ParentMethods**

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:**
>
> `Sets.SubcategoryMethods.Subquotients` for the specifications, `ambient()`, `retract()`, and also `Sets.Subquotients.ElementMethods.lift()`.

retract(*x*)

> Retract x to `self`.
>
> INPUT:
> •x – an element of the ambient space for `self`
>
> **See also:**
>
> `Sets.SubcategoryMethods.Subquotients` for the specifications, `ambient()`, `retract()`, and also `Sets.Subquotients.ElementMethods.retract()`.
>
> EXAMPLES:
> ```
> sage: S = Semigroups().Subquotients().example()
> sage: s = S.ambient().an_element()
> sage: s, s.parent()
> (42, An example of a semigroup: the left zero semigroup)
> sage: S.retract(s), S.retract(s).parent()
> (42, An example of a (sub)quotient semigroup: a quotient of the left zero semigroup)
> ```

class Sets.**WithRealizations**(*category*, **args*)

> Bases: `sage.categories.with_realizations.WithRealizationsCategory`
>
> TESTS:
> ```
> sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> sage: class FooBars(CovariantConstructionCategory):
> ....:     _functor_category = "FooBars"
> ....:     _base_category_class = (Category,)
> sage: Category.FooBars = lambda self: FooBars.category_of(self)
> sage: C = FooBars(ModulesWithBasis(ZZ))
> sage: C
> ```

```
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

class ParentMethods

> **class Realizations**(*parent_with_realization*)
>
> > Bases: `sage.categories.realizations.Category_realization_of_parent`
> >
> > TESTS:
> > ```
> > sage: from sage.categories.realizations import Category_realization_of_parent
> > sage: A = Sets().WithRealizations().example(); A
> > The subset algebra of {1, 2, 3} over Rational Field
> > sage: C = A.Realizations(); C
> > Category of realizations of The subset algebra of {1, 2, 3} over Rational Field
> > sage: isinstance(C, Category_realization_of_parent)
> > True
> > sage: C.parent_with_realization
> > The subset algebra of {1, 2, 3} over Rational Field
> > sage: TestSuite(C).run(skip=["_test_category_over_bases"])
> > ```
> >
> > ---
> >
> > **Todo**
> >
> > Fix the failing test by making `C` a singleton category. This will require some fiddling with the assertion in `Category_singleton.__classcall__()`
> >
> > ---
> >
> > **super_categories**()
> > > EXAMPLES:
> > > ```
> > > sage: A = Sets().WithRealizations().example(); A
> > > The subset algebra of {1, 2, 3} over Rational Field
> > > sage: A.Realizations().super_categories()
> > > [Category of realizations of sets]
> > > ```
>
> `Sets.WithRealizations.ParentMethods.`**a_realization**()
> > Return a 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
> > ```
>
> `Sets.WithRealizations.ParentMethods.`**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 su
> >
> > 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)
```

Sets.WithRealizations.ParentMethods.**inject_shorthands**(*verbose=True*)
  Import standard shorthands into the global namespace.

  INPUT:
   • verbose – boolean (default True) if True, prints the defined shorthands
  EXAMPLES:
```
sage: Q = QuasiSymmetricFunctions(ZZ)
sage: Q.inject_shorthands()
Injecting M as shorthand for Quasisymmetric functions over
the Integer Ring in the Monomial basis
Injecting F as shorthand for Quasisymmetric functions over
the Integer Ring in the Fundamental basis
Injecting dI as shorthand for Quasisymmetric functions over
the Integer Ring in the dualImmaculate basis
Injecting QS as shorthand for Quasisymmetric functions over
the Integer Ring in the Quasisymmetric Schur basis
sage: F[1,2,1] + 5*M[1,3] + F[2]^2
5*F[1, 1, 1, 1] - 5*F[1, 1, 2] - 3*F[1, 2, 1] + 6*F[1, 3] +
2*F[2, 2] + F[3, 1] + F[4]
sage: F
Quasisymmetric functions over the Integer Ring in the
Fundamental basis
sage: M
Quasisymmetric functions over the Integer Ring in the
Monomial basis
```

Sets.WithRealizations.ParentMethods.**realizations**()
  Return all the realizations of self that self is aware of.

  EXAMPLES:
```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: A.realizations()
[The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis, The su
```

  Note: Constructing a parent P in the category A.Realizations() automatically adds P to
  this list by calling A._register_realization(A)

Sets.WithRealizations.**example**(*base_ring=None*, *set=None*)
  Return an example of set with multiple realizations, as per Category.example().

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

sage: Sets().WithRealizations().example(ZZ, Set([1,2]))
The subset algebra of {1, 2} over Integer Ring
```

Sets.WithRealizations.**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]
```

Sets.**example**(*choice=None*)
Returns 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)
```

Sets.**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:
```
sage: Sets().super_categories()
[Category of sets with partial maps]
```

sage.categories.sets_cat.**print_compare**(*x*, *y*)
Helper method used in Sets.ParentMethods._test_elements_eq_symmetric(), Sets.ParentMethods._test_elements_eq_transitive().

INPUT:

- x – an element

- y – an element

EXAMPLES:
```
sage: from sage.categories.sets_cat import print_compare
sage: print_compare(1,2)
1 != 2
sage: print_compare(1,1)
1 == 1
```

13.105 Sets With a Grading

class sage.categories.sets_with_grading.**SetsWithGrading**(*s=None*)
Bases: sage.categories.category.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 $\mathbf{N}$ 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 `graded_component()` or `subset()`. If only `subset()` is implemented, the first argument must be the grading for compatibility with `graded_component()`. Additionally either the parent must implement `grading()` or its elements must implement a method `grade()`. See the example `sage.categories.examples.sets_with_grading.NonNegativeIntegers`.

Finally, if the graded set is enumerated (see `EnumeratedSets`) then each graded component should be enumerated. The contrary needs not be true.

EXAMPLES:

A typical example of a set with a grading is the set of non-negative integers graded by themselves:
```
sage: N = SetsWithGrading().example(); N
Non negative integers
sage: N.category()
Category of facade sets with grading
sage: N.grading_set()
Non negative integers
```

The *grading function* is given by `N.grading`:
```
sage: N.grading(4)
4
```

The graded component S_i is the set of all integer partitions of i:
```
sage: N.graded_component(grade = 5)
{5}
sage: N.graded_component(grade = 42)
{42}
```

Here are some information about this category:
```
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]
```

Todo

•This should be moved to `Sets().WithGrading()`.

- Should the grading set be a parameter for this category?

- Does the enumeration need to be compatible with the grading? Be careful that the fact that graded components are allowed to be finite or infinite make the answer complicated.

TESTS:
```
sage: C = SetsWithGrading()
sage: TestSuite(C).run()
```

class **ParentMethods**

generating_series()
> Default implementation for generating series.

> OUTPUT:

> A series, indexed by the grading set.

> EXAMPLES:
> ```
> sage: N = SetsWithGrading().example(); N
> Non negative integers
> sage: N.generating_series()
> 1/(-z + 1)
> ```

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
Integer vectors weighted by [3, 2, 1]
sage: W.subset(4)
Integer vectors of 4 weighted by [3, 2, 1]
```

SetsWithGrading.**super_categories**()
> EXAMPLES:
```
sage: SetsWithGrading().super_categories()
[Category of sets]
```

13.106 SetsWithPartialMaps

class sage.categories.sets_with_partial_maps.**SetsWithPartialMaps**(*s=None*)
> Bases: sage.categories.category_singleton.Category_singleton

The category whose objects are sets and whose morphisms are maps that are allowed to raise a ValueError on some inputs.

This category is equivalent to the category of pointed sets, via the equivalence sending an object X to X union {error}, a morphism f to the morphism of pointed sets that sends x to f(x) if f does not raise an error on x, or to error if it does.

EXAMPLES:
```
sage: SetsWithPartialMaps()
Category of sets with partial maps

sage: SetsWithPartialMaps().super_categories()
[Category of objects]
```

TESTS:
```
sage: TestSuite(SetsWithPartialMaps()).run()
```

super_categories()
> EXAMPLES:
```
sage: SetsWithPartialMaps().super_categories()
[Category of objects]
```

13.107 Unique factorization domains

class sage.categories.unique_factorization_domains.**UniqueFactorizationDomains**(*s=None*)
> Bases: sage.categories.category_singleton.Category_singleton

The category of unique factorization domains constructive unique factorization domains, i.e. where one 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]
```

TESTS:
```
sage: TestSuite(UniqueFactorizationDomains()).run()
```

class **ElementMethods**

class UniqueFactorizationDomains.**ParentMethods**

> **is_unique_factorization_domain**(*proof=True*)
>> Return True, since this in an object of the category of unique factorization domains.
>>
>> EXAMPLES:
>> ```
>> sage: Parent(QQ,category=UniqueFactorizationDomains()).is_unique_factorization_domain()
>> True
>> ```

UniqueFactorizationDomains.**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:
> ```
> sage: UniqueFactorizationDomains().additional_structure()
> ```

UniqueFactorizationDomains.**super_categories**()
> EXAMPLES:
> ```
> sage: UniqueFactorizationDomains().super_categories()
> [Category of gcd domains]
> ```

13.108 Unital algebras

class sage.categories.unital_algebras.**UnitalAlgebras**(*base_category*)
> Bases: sage.categories.category_with_axiom.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 s also a unital magma.
>
> > **Warning:** Until trac ticket #15043 is implemented, Algebras is the category of associative unital algebras; thus, unlike the name suggests, UnitalAlgebras is not a subcategory of Algebras but of MagmaticAlgebras.
>
> EXAMPLES:
> ```
> sage: from sage.categories.unital_algebras import UnitalAlgebras
> sage: C = UnitalAlgebras(ZZ); C
> Category of unital algebras over Integer Ring
> ```
>
> TESTS:
> ```
> sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
> sage: C is MagmaticAlgebras(ZZ).Unital()
> ```

```
True
sage: TestSuite(C).run()
```

class **ElementMethods**
>   `Magmas.Element.__mul__` is preferable to `Modules.Element.__mul__` since the later does
>   not handle products of two elements of `self`.
>
>   TESTS:
>   ```
>   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: x = A.an_element()
>   sage: x
>   B[word: ] + 2*B[word: a] + 3*B[word: b] + B[word: bab]
>   sage: x.__mul__(x)
>   B[word: ] + 4*B[word: a] + 4*B[word: aa] + 6*B[word: ab]
>   + 2*B[word: abab] + 6*B[word: b] + 6*B[word: ba]
>   + 2*B[word: bab] + 2*B[word: baba] + 3*B[word: babb]
>   + B[word: babbab] + 9*B[word: bb] + 3*B[word: bbab]
>   ```

class `UnitalAlgebras`.**ParentMethods**

>   **from_base_ring**(*r*)
>   >   Return the canonical embedding of `r` into `self`.
>   >
>   >   INPUT:
>   >   >   •`r` – an element of `self.base_ring()`
>   >   EXAMPLES:
>   >   ```
>   >   sage: A = AlgebrasWithBasis(QQ).example(); A
>   >   An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c')
>   >   sage: A.from_base_ring(1)
>   >   B[word: ]
>   >   ```

class `UnitalAlgebras`.**WithBasis**(*base_category*)
>   Bases: [sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring](#)
>
>   TESTS:
>   ```
>   sage: C = Modules(ZZ).FiniteDimensional(); C
>   Category of finite dimensional modules over Integer Ring
>   sage: type(C)
>   <class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
>   sage: type(C).__base__.__base__
>   <class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>
>
>   sage: TestSuite(C).run()
>   ```

>   class **ParentMethods**
>
>   >   **from_base_ring**()
>   >   >   TESTS:
>   >   >   ```
>   >   >   sage: A = AlgebrasWithBasis(QQ).example()
>   >   >   sage: A.from_base_ring(3)
>   >   >   3*B[word: ]
>   >   >   ```
>   >
>   >   **from_base_ring_from_one_basis**(*r*)

Implement the canonical embeding from the ground ring.

INPUT:

• r – an element of the coefficient ring

EXAMPLES:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.from_base_ring_from_one_basis(3)
3*B[word: ]
sage: A.from_base_ring(3)
3*B[word: ]
sage: A(3)
3*B[word: ]
```

one()

Return the multiplicative unit element.

EXAMPLES:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one_basis()
word:
sage: A.one()
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:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one_basis()
word:
sage: A.one()
B[word: ]
sage: A.from_base_ring(4)
4*B[word: ]
```

one_from_one_basis()

Return the one of the algebra, as per Monoids.ParentMethods.one()

By default, this is implemented from one_basis(), if available.

EXAMPLES:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one_basis()
word:
sage: A.one_from_one_basis()
B[word: ]
sage: A.one()
B[word: ]
```

TESTS:

Try to check that trac ticket #5843 Heisenbug is fixed:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: B = AlgebrasWithBasis(QQ).example(('a', 'c'))
sage: A == B
False
sage: Aone = A.one_from_one_basis
sage: Bone = B.one_from_one_basis
```

```
sage: Aone is Bone
False
```

Even if called in the wrong order, they should returns their respective one:

```
sage: Bone().parent() is B
True
sage: Aone().parent() is A
True
```

13.109 Vector Spaces

class sage.categories.vector_spaces.**VectorSpaces**(*K*)

> Bases: `sage.categories.category_types.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: `sage.categories.cartesian_product.CartesianProductsCategory`
> >
> > TESTS:
> >
> > ```
> > sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
> > sage: class FooBars(CovariantConstructionCategory):
> > ....:     _functor_category = "FooBars"
> > ....:     _base_category_class = (Category,)
> > sage: Category.FooBars = lambda self: FooBars.category_of(self)
> > sage: C = FooBars(ModulesWithBasis(ZZ))
> > sage: C
> > Category of foo bars of modules with basis over Integer Ring
> > sage: C.base_category()
> > Category of modules with basis over Integer Ring
> > sage: latex(C)
> > \mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
> > sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
> > sage: TestSuite(C).run()
> > ```
> >
> > **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 VectorSpaces.DualObjects**(*category*, **args*)
>
> > Bases: `sage.categories.dual.DualObjectsCategory`
> >
> > TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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 VectorSpaces.**ElementMethods**

class VectorSpaces.**ParentMethods**

class VectorSpaces.**TensorProducts**(*category*, **args*)

    Bases: sage.categories.tensor.TensorProductsCategory

    TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstructionCat
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a python m
sage: TestSuite(C).run()
```

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 `VectorSpaces.`**`WithBasis`**(*base_category*)

   Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

   TESTS:

```
sage: C = Modules(ZZ).FiniteDimensional(); C
Category of finite dimensional modules over Integer Ring
sage: type(C)
<class 'sage.categories.modules.Modules.FiniteDimensional_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring'>

sage: TestSuite(C).run()
```

   class **`CartesianProducts`**(*category*, **args*)

      Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

      TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

      `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 `VectorSpaces.WithBasis.`**`TensorProducts`**(*category*, **args*)

      Bases: `sage.categories.tensor.TensorProductsCategory`

      TESTS:

```
sage: from sage.categories.covariant_functorial_construction import CovariantConstruction
sage: class FooBars(CovariantConstructionCategory):
....:     _functor_category = "FooBars"
....:     _base_category_class = (Category,)
sage: Category.FooBars = lambda self: FooBars.category_of(self)
sage: C = FooBars(ModulesWithBasis(ZZ))
sage: C
Category of foo bars of modules with basis over Integer Ring
sage: C.base_category()
Category of modules with basis over Integer Ring
sage: latex(C)
\mathbf{FooBars}(\mathbf{ModulesWithBasis}_{\Bold{Z}})
sage: import __main__; __main__.FooBars = FooBars # Fake FooBars being defined in a pyth
sage: TestSuite(C).run()
```

extra_super_categories()
> The category of vector spaces with basis is closed under tensor products:
> ```
> sage: C = VectorSpaces(QQ).WithBasis()
> sage: C.TensorProducts()
> Category of tensor products of vector spaces with basis over Rational Field
> sage: C in C.TensorProducts().super_categories()
> True
> ```

VectorSpaces.WithBasis.**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
> ```

VectorSpaces.**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()
> ```

VectorSpaces.**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
> ```

VectorSpaces.**super_categories**()
> EXAMPLES:
> ```
> sage: VectorSpaces(QQ).super_categories()
> [Category of modules over Rational Field]
> ```

13.110 Weyl Groups

class sage.categories.weyl_groups.**WeylGroups**(*s=None*)
> Bases: sage.categories.category_singleton.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
```

TESTS:

```
sage: C = WeylGroups()
sage: TestSuite(C).run()
```

class **ElementMethods**

    bruhat_lower_covers_coroots()

        Returns all 2-tuples (v, α) where v is covered by `self` and α is the positive coroot such that `self` = v s_α where s_α is the reflection orthogonal to α.

        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()
[(s1*s2*s1, alphacheck[1] + alphacheck[2] + alphacheck[3]), (s3*s2*s1, alphacheck[2]), (
```

    bruhat_upper_covers_coroots()

        Returns all 2-tuples (v, α) where v is covers `self` and α is the positive coroot such that `self` = v s_α where s_α is the reflection orthogonal to α.

        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*s
```

    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'*)

Returns 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.from_reduced_word([3,2,5]).is_pieri_factor()
True
sage: W.from_reduced_word([3,2,4,5]).is_pieri_factor()
False

sage: W = WeylGroup(['C',4,1])
sage: W.from_reduced_word([0,2,1]).is_pieri_factor()
True
sage: W.from_reduced_word([0,2,1,0]).is_pieri_factor()
False

sage: W = WeylGroup(['B',3])
sage: W.from_reduced_word([3,2,3]).is_pieri_factor()
False
sage: W.from_reduced_word([2,1,2]).is_pieri_factor()
True
```

left_pieri_factorizations(*max_length=+Infinity*)

Returns all factorizations of self as uv, where u is a Pieri factor and v is an element of the Weyl group.

See also:

- •WeylGroups.ParentMethods.pieri_factors()
- •sage.combinat.root_system.pieri_factors

EXAMPLES:

If we take $w = w_0$ the maximal element of a strict parabolic subgroup of type $A_{n_1} \times \cdots \times A_{n_k}$, then the Pieri factorizations are in correspondence with all Pieri factors, and there are $\prod 2^{n_i}$ of them:
```
sage: W = WeylGroup(['A', 4, 1])
sage: W.from_reduced_word([]).left_pieri_factorizations().cardinality()
1
sage: W.from_reduced_word([1]).left_pieri_factorizations().cardinality()
2
sage: W.from_reduced_word([1,2,1]).left_pieri_factorizations().cardinality()
4
sage: W.from_reduced_word([1,2,3,1,2,1]).left_pieri_factorizations().cardinality()
8

sage: W.from_reduced_word([1,3]).left_pieri_factorizations().cardinality()
4
sage: W.from_reduced_word([1,3,4,3]).left_pieri_factorizations().cardinality()
8

sage: W.from_reduced_word([2,1]).left_pieri_factorizations().cardinality()
3
sage: W.from_reduced_word([1,2]).left_pieri_factorizations().cardinality()
2
sage: [W.from_reduced_word([1,2]).left_pieri_factorizations(max_length=i).cardinality()
[0, 1, 2, 2]

sage: W = WeylGroup(['C',4,1])
sage: w = W.from_reduced_word([0,3,2,1,0])
sage: w.left_pieri_factorizations().cardinality()
```

```
7
sage: [(u.reduced_word(),v.reduced_word()) for (u,v) in w.left_pieri_factorizations()]
[([], [3, 2, 0, 1, 0]),
([0], [3, 2, 1, 0]),
([3], [2, 0, 1, 0]),
([3, 0], [2, 1, 0]),
([3, 2], [0, 1, 0]),
([3, 2, 0], [1, 0]),
([3, 2, 0, 1], [0])]

sage: W = WeylGroup(['B',4,1])
sage: W.from_reduced_word([0,2,1,0]).left_pieri_factorizations().cardinality()
6
```

quantum_bruhat_successors (*index_set=None*, *roots=False*, *quantum_only=False*)
    Returns the successors of `self` in the parabolic quantum Bruhat graph.

    INPUT:
        •`self` – a Weyl group element, which is assumed to be of minimum length in its coset with respect
         to the parabolic subgroup
        •**index_set – (default: None) indicates the set of simple reflections used to generate the parabolic subgroup**
         the default value indicates that the subgroup is the identity
        •**roots – (default: False) if True, returns the list of 2-tuples (w, α) where w is a** successor
         and α is the positive root associated with the successor relation.
        •`quantum_only` – (default: False) if True, returns only the quantum successors
    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`.

    EXAMPLES:

```
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2])
sage: w.quantum_bruhat_successors([1], roots = True)
[(s3, alpha[2]), (s1*s2*s3*s2, alpha[3]), (s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3])]
sage: w.quantum_bruhat_successors([1,3])
[1, s2*s3*s1*s2]
sage: w.quantum_bruhat_successors(roots = True)
[(s3*s1*s2*s1, alpha[1]), (s3*s1, alpha[2]), (s1*s2*s3*s2, alpha[3]), (s2*s3*s1*s2, alph
sage: w.quantum_bruhat_successors()
[s3*s1*s2*s1, s3*s1, s1*s2*s3*s2, s2*s3*s1*s2]
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 genera
```

reflection_to_coroot ()
    Returns the coroot associated with the reflection `self`.

    EXAMPLES:

```
sage: W=WeylGroup(['C',2],prefix="s")
sage: W.from_reduced_word([1,2,1]).reflection_to_coroot()
alphacheck[1] + alphacheck[2]
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()

Returns the root associated with the reflection `self`.

EXAMPLES:

```
sage: W=WeylGroup(['C',2],prefix="s")
sage: W.from_reduced_word([1,2,1]).reflection_to_root()
2*alpha[1] + alpha[2]
sage: W.from_reduced_word([1,2]).reflection_to_root()
Traceback (most recent call last):
...
ValueError: s1*s2 is not a reflection
sage: W.long_element().reflection_to_root()
Traceback (most recent call last):
...
ValueError: s2*s1*s2*s1 is not a reflection
```

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, 1, 1, 1] + m[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, 1, 1, 1] + m[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] + 5*m[2, 1, 1, 1, 1, 1, 1] + 4*m[2, 2, 1, 1, 1, 1]
+ 3*m[2, 2, 2, 1, 1] + 2*m[2, 2, 2, 2] + 4*m[3, 1, 1, 1, 1, 1] + 3*m[3, 2, 1, 1, 1]
+ 2*m[3, 2, 2, 1] + 2*m[3, 3, 1, 1] + m[3, 3, 2] + 3*m[4, 1, 1, 1, 1] + 2*m[4, 2, 1, 1]
+ m[4, 2, 2] + m[4, 3, 1]
```

One more example (trac ticket #14095):

```
sage: G = SymmetricGroup(4)
sage: w = G.from_reduced_word([3,2,3,1])
sage: w.stanley_symmetric_function()
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
```

REFERENCES:

stanley_symmetric_function_as_polynomial(*max_length=+Infinity*)
    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:

```
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
sage: W.from_reduced_word([1,2,3,1,2,1,0]).stanley_symmetric_function_as_polynomial() #
22*x1^7 + 11*x1^5*x2 + 5*x1^3*x2^2 + 3*x1^4*x3 + 2*x1*x2^3 + x1^2*x2*x3
sage: W.from_reduced_word([3,1,2,0,3,1,0]).stanley_symmetric_function_as_polynomial() #
8*x1^7 + 4*x1^5*x2 + 2*x1^3*x2^2 + x1*x2^3

sage: W = WeylGroup(['C',3,1])
sage: W.from_reduced_word([0,2,1,0]).stanley_symmetric_function_as_polynomial()
32*x1^4 + 16*x1^2*x2 + 8*x2^2 + 4*x1*x3

sage: W = WeylGroup(['B',3,1])
```

```
sage: W.from_reduced_word([3,2,1]).stanley_symmetric_function_as_polynomial()
2*x1^3 + x1*x2 + 1/2*x3
```

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.

WeylGroups.**Finite**
    alias of `FiniteWeylGroups`

class WeylGroups.**ParentMethods**

   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:
```
sage: W = WeylGroup(['A',5,1])
sage: PF = W.pieri_factors()
sage: PF.cardinality()
63
```

```
sage: W = WeylGroup(['B',3])
sage: PF = W.pieri_factors()
sage: [w.reduced_word() for w in PF]
[[1, 2, 3, 2, 1], [1, 2, 3, 2], [2, 3, 2], [2, 3], [3, 1, 2, 1], [1, 2, 1], [2], [1, 2],
```

```
sage: W = WeylGroup(['C',4,1])
sage: PF = W.pieri_factors()
sage: W.from_reduced_word([3,2,0]) in PF
True
```

   quantum_bruhat_graph(*index_set=()*)
        Returns the quantum Bruhat graph of the quotient of the Weyl group by a parabolic subgroup W_J.

        INPUT:
            •index_set – a tuple J of nodes of the Dynkin diagram (default: ())
        By default, the value for index_set indicates that the subgroup is trivial and the quotient is the full Weyl group.

        EXAMPLES:
```
sage: W = WeylGroup(['A',3], prefix="s")
sage: g = W.quantum_bruhat_graph((1,3))
sage: g
Parabolic Quantum Bruhat Graph of Weyl Group of type ['A', 3] (as a matrix group acting
sage: g.vertices()
[s2*s3*s1*s2, s3*s1*s2, s1*s2, s3*s2, s2, 1]
sage: g.edges()
[(s2*s3*s1*s2, s2, alpha[2]), (s3*s1*s2, s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3]),
(s3*s1*s2, 1, alpha[2]), (s1*s2, s3*s1*s2, alpha[2] + alpha[3]),
(s3*s2, s3*s1*s2, alpha[1] + alpha[2]), (s2, s1*s2, alpha[1] + alpha[2]),
```

```
      (s2, s3*s2, alpha[2] + alpha[3]), (1, s2, alpha[2])]
sage: W = WeylGroup(['A',3,1], prefix="s")
sage: g = W.quantum_bruhat_graph()
Traceback (most recent call last):
...
ValueError: The Cartan type ['A', 3, 1] is not finite
```

WeylGroups.**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.additional_structure()

    Todo

    Should this category be a CategoryWithAxiom?

    EXAMPLES:

```
sage: WeylGroups().additional_structure()
```

WeylGroups.**super_categories**()

    EXAMPLES:

```
sage: WeylGroups().super_categories()
[Category of coxeter groups]
```

TECHNICAL CATEGORIES

14.1 Facade Sets

For background, see *What is a facade set?*.

class sage.categories.facade_sets.**FacadeSets**(*base_category*)

    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

    TESTS:

```
sage: C = Sets.Finite(); C
Category of finite sets
sage: type(C)
<class 'sage.categories.finite_sets.FiniteSets_with_category'>
sage: type(C).__base__.__base__
<class 'sage.categories.category_with_axiom.CategoryWithAxiom_singleton'>

sage: TestSuite(C).run()
```

    class ParentMethods

        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 trac ticket #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:

```
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:

```
sage: -1 in S, S.is_parent_of(-1)
(False, True)
```

Furthermore, there is no coercion attempted:

```
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?

`FacadeSets.`**`example`**`(choice='subset')`

Returns an example of facade set, as per `Category.example()`.

INPUT:

- `choice` – 'union' or 'subset' (default: 'subset').

EXAMPLES:

```
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
```

EXAMPLES OF PARENTS USING CATEGORIES

15.1 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: sage.combinat.free_module.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
sage: A.algebra_generators()
Family (B[word: a], B[word: b], B[word: c])
```

    one_basis()
        Returns the empty word, which index the one of this algebra, as per
        AlgebrasWithBasis.ParentMethods.one_basis().

        EXAMPLES::r

            sage: A = AlgebrasWithBasis(QQ).example() sage: A.one_basis() word: sage: A.one() B[word:
            ]

    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[word: acbcba]
sage: (a,b,c) = A.algebra_generators()
sage: a * (1-b)^2 * c
B[word: abbc] - 2*B[word: abc] + B[word: ac]
```

15.2 Examples of commutative additive monoids

sage.categories.examples.commutative_additive_monoids.**Example**
    alias of FreeCommutativeAdditiveMonoid

class sage.categories.examples.commutative_additive_monoids.**FreeCommutativeAdditiveMonoid**(*alph*
'b',
'c',
'd')

Bases: sage.categories.examples.commutative_additive_semigroups.FreeCommutativeAdditiveSe

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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass
```

class Element (*parent*, *iterable*)
        Bases: sage.categories.examples.commutative_additive_semigroups.FreeCommutativeAdditi

        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 monoid: the free commutative monoid generated by ('a', 'b', 'c',
```

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}
```

FreeCommutativeAdditiveMonoid.**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',
sage: M.zero()
0
```

15.3 Examples of commutative additive semigroups

sage.categories.examples.commutative_additive_semigroups.**Example**
    alias of FreeCommutativeAdditiveSemigroup

class sage.categories.examples.commutative_additive_semigroups.**FreeCommutativeAdditiveSemigro**

    Bases: sage.structure.unique_representation.UniqueRepresentation,
    sage.structure.parent.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 monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd'
```

```
sage: S.category()
Category of commutative additive semigroups
```

    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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class **Element** (*parent*, *iterable*)

Bases: `sage.structure.element_wrapper.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 monoid: the free commutative monoid generated by ('a', 'b', 'c',
```

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}
```

FreeCommutativeAdditiveSemigroup.**additive_semigroup_generators**()

Returns the generators of the semigroup.

EXAMPLES:

```
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.additive_semigroup_generators()
Family (a, b, c, d)
```

FreeCommutativeAdditiveSemigroup.**an_element**()

Returns an element of the semigroup.

EXAMPLES:

```
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.an_element()
a + 3*c + 2*b + 4*d
```

FreeCommutativeAdditiveSemigroup.**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 + c + b
```

15.4 Examples of Coxeter groups

15.5 Example of a crystal

class sage.categories.examples.crystals.**HighestWeightCrystalOfTypeA**(*n=3*)
Bases: `sage.structure.unique_representation.UniqueRepresentation`,
`sage.structure.parent.Parent`

An example of a crystal: the highest weight crystal of type A_n of highest weight ω_1.

The purpose of this class is to provide a minimal template for implementing crystals. See
`CrystalOfLetters` for a full featured and optimized implementation.

EXAMPLES:
```
sage: C = Crystals().example()
sage: C
Highest weight crystal of type A_3 of highest weight omega_1
sage: C.category()
Category of classical crystals
```

The elements of this crystal are in the set $\{1, \ldots, n+1\}$:
```
sage: C.list()
[1, 2, 3,  4]
sage: C.module_generators[0]
1
```

The crystal operators themselves correspond to the elementary transpositions:
```
sage: b = C.module_generators[0]
sage: b.f(1)
2
sage: b.f(1).e(1) == b
True
```

TESTS:
```
sage: C = Crystals().example()
sage: TestSuite(C).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_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    running ._test_stembridge_local_axioms() . . . 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_fast_iter() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_stembridge_local_axioms() . . . pass
```

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:

```
sage: C.cardinality()
4
```

class **Element**

    Bases: `sage.structure.element_wrapper.ElementWrapper`

    e(i)

        Returns the action of e_i on `self`.

        EXAMPLES:

```
sage: C = Crystals().example(4)
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1], [3, 2, 2], [4, 3, 3], [5, 4, 4]]
```

    f(i)

        Returns the action of f_i on `self`.

        EXAMPLES:

```
sage: C = Crystals().example(4)
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2], [2, 2, 3], [3, 3, 4], [4, 4, 5]]
```

class sage.categories.examples.crystals.**NaiveCrystal**

    Bases: `sage.structure.unique_representation.UniqueRepresentation`,
    `sage.structure.parent.Parent`

    This is an example of a "crystal" which does not come from any kind of representation, designed primarily to
    test the Stembridge local rules with. The crystal has vertices labeled 0 through 5, with 0 the highest weight.

The code here could also possibly be generalized to create a class that automatically builds a crystal from an edge-colored digraph, if someone feels adventurous.

Currently, only the methods `highest_weight_vector()`, `e()`, and `f()` are guaranteed to work.

EXAMPLES:
```
sage: C = Crystals().example(choice='naive')
sage: C.highest_weight_vector()
0
```

class Element
    Bases: `sage.structure.element_wrapper.ElementWrapper`

    e(*i*)
        Returns the action of e_i on `self`.

        EXAMPLES:
```
sage: C = Crystals().example(choice='naive')
sage: [[c,i,c.e(i)] for i in C.index_set() for c in [C(j) for j in [0..5]] if c.e(i) is n
[[1, 1, 0], [2, 1, 1], [3, 1, 2], [5, 1, 3], [4, 2, 0], [5, 2, 4]]
```

    f(*i*)
        Returns the action of f_i on `self`.

        EXAMPLES:
```
sage: C = Crystals().example(choice='naive')
sage: [[c,i,c.f(i)] for i in C.index_set() for c in [C(j) for j in [0..5]] if c.f(i) is n
[[0, 1, 1], [1, 1, 2], [2, 1, 3], [3, 1, 5], [0, 2, 4], [4, 2, 5]]
```

15.6 Example of facade set

class `sage.categories.examples.facade_sets.`**IntegersCompletion**
    Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.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:
```
sage: S = Sets().Facade().example("union"); S
An example of a facade set: the integers completed by +-infinity
```

    TESTS:
```
sage: TestSuite(S).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_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
```

```
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class sage.categories.examples.facade_sets.**PositiveIntegerMonoid**

Bases: sage.structure.unique_representation.UniqueRepresentation,
sage.structure.parent.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
```

TESTS:

```
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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

15.7 Examples of finite Coxeter groups

class sage.categories.examples.finite_coxeter_groups.**DihedralGroup**(*n=5*)

Bases: sage.structure.unique_representation.UniqueRepresentation,
sage.structure.parent.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,),
 (1, 2),
 (1, 2, 1),
 (1, 2, 1, 2),
 (1, 2, 1, 2, 1),
 (2,),
 (2, 1),
 (2, 1, 2),
 (2, 1, 2, 1)]
```

This reduced word is unique, except for the longest element where the choosen reduced word is $(1, 2, 1, 2\ldots)$:
```
sage: G.long_element()
(1, 2, 1, 2, 1)
```

TESTS:
```
sage: TestSuite(G).run(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . 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_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_has_descent() . . . pass
running ._test_inverse() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
```

```
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_reduced_word() . . . pass
running ._test_simple_projections() . . . pass
running ._test_some_elements() . . . pass
```

```
sage: c = FiniteCoxeterGroups().example(3).cayley_graph()
sage: sorted(c.edges())
[((), (1,), 1),
 ((), (2,), 2),
 ((1,), (), 1),
 ((1,), (1, 2), 2),
 ((1, 2), (1,), 2),
 ((1, 2), (1, 2, 1), 1),
 ((1, 2, 1), (1, 2), 1),
 ((1, 2, 1), (2, 1), 2),
 ((2,), (), 2),
 ((2,), (2, 1), 1),
 ((2, 1), (1, 2, 1), 2),
 ((2, 1), (2,), 1)]
```

class Element
    Bases: `sage.structure.element_wrapper.ElementWrapper`

    apply_simple_reflection_right(*i*)
        Implements `CoxeterGroups.ElementMethods.apply_simple_reflection()`.

        EXEMPLES:
```
sage: D5 = FiniteCoxeterGroups().example(5)
sage: [i^2 for i in D5]
[(), (), (1, 2, 1, 2), (), (2, 1), (), (), (2, 1, 2, 1), (), (1, 2)]
sage: [i^5 for i in D5]
[(), (1,), (), (1, 2, 1), (), (1, 2, 1, 2, 1), (2,), (), (2, 1, 2), ()]
```

    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
```

        TESTS:
```
sage: D6._test_has_descent()
```

`DihedralGroup.`**index_set**`()`

Implements `CoxeterGroups.ParentMethods.index_set()`.

EXAMPLES:
```
sage: D4 = FiniteCoxeterGroups().example(4)
sage: D4.index_set()
[1, 2]
```

DihedralGroup.**one**()
Implements `Monoids.ParentMethods.one()`.

EXAMPLES:
```
sage: D6 = FiniteCoxeterGroups().example(6)
sage: D6.one()
()
```

sage.categories.examples.finite_coxeter_groups.**Example**
alias of `DihedralGroup`

15.8 Examples of finite enumerated sets

class sage.categories.examples.finite_enumerated_sets.**Example**
Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class sage.categories.examples.finite_enumerated_sets.**IsomorphicObjectOfFiniteEnumeratedSet**(*a*
e
a
p
o
a
fo
n
e
n
a
s
{
2
3

Bases: sage.structure.unique_representation.UniqueRepresentation,
sage.structure.parent.Parent

TESTS:
```
sage: C = FiniteEnumeratedSets().IsomorphicObjects().example()
sage: C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.category()
Category of facade isomorphic objects of finite enumerated sets
sage: TestSuite(C).run()
```

ambient()
    Returns the ambient space for self, as per `Sets.Subquotients.ParentMethods.ambient()`.

    EXAMPLES:
```
sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.ambient()
An example of a finite enumerated set: {1,2,3}
```

lift(*x*)

    INPUT:

        • x – an element of self

    Lifts x to the ambient space for self, as per `Sets.Subquotients.ParentMethods.lift()`.

    EXAMPLES:
```
sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.lift(9)
3
```

retract(*x*)

> INPUT:
>
> > • x – an element of the ambient space for self
>
> Retracts x from the ambient space to self, as per `Sets.Subquotients.ParentMethods.retract()`.
>
> EXAMPLES:
> ```
> sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
> The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
> sage: C.retract(3)
> 9
> ```

15.9 Examples of finite monoids

sage.categories.examples.finite_monoids.**Example**
> alias of `IntegerModMonoid`

class sage.categories.examples.finite_monoids.**IntegerModMonoid**(*n=12*)
> Bases: sage.structure.unique_representation.UniqueRepresentation,
> sage.structure.parent.Parent
>
> An example of a finite monoid: the integers mod n
>
> This class illustrates a minimal implementation of a finite monoid.
>
> EXAMPLES:
> ```
> sage: S = FiniteMonoids().example(); S
> An example of a finite multiplicative monoid: the integers modulo 12
>
> sage: S.category()
> Category of finitely generated finite monoids
> ```
>
> We conclude by running systematic tests on this monoid:
> ```
> sage: TestSuite(S).run(verbose = True)
> running ._test_an_element() . . . pass
> running ._test_associativity() . . . pass
> running ._test_cardinality() . . . pass
> running ._test_category() . . . pass
> running ._test_elements() . . .
>   Running the test suite of self.an_element()
>   running ._test_category() . . . pass
>   running ._test_eq() . . . 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_not_implemented_methods() . . . pass
> running ._test_one() . . . pass
> ```

```
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

class **Element**
    Bases: `sage.structure.element_wrapper.ElementWrapper`

    wrapped_class
        alias of `Integer`

IntegerModMonoid.**an_element**()
    Returns an element of the monoid, as per `Sets.ParentMethods.an_element()`.

    EXAMPLES:
```
sage: M = FiniteMonoids().example()
sage: M.an_element()
6
```

IntegerModMonoid.**one**()
    Return the one of the monoid, as per `Monoids.ParentMethods.one()`.

    EXAMPLES:
```
sage: M = FiniteMonoids().example()
sage: M.one()
1
```

IntegerModMonoid.**product**(*x*, *y*)
    Return the product of two elements x and y of the monoid, as per `Semigroups.ParentMethods.product()`.

    EXAMPLES:
```
sage: M = FiniteMonoids().example()
sage: M.product(M(3), M(5))
3
```

IntegerModMonoid.**semigroup_generators**()
    Returns a set of generators for `self`, as per `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)
```

15.10 Examples of finite semigroups

`sage.categories.examples.finite_semigroups.`**Example**
    alias of `LeftRegularBand`

class `sage.categories.examples.finite_semigroups.`**LeftRegularBand**(*alphabet=('a',* *'b', 'c', 'd')*)
    Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.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: S.list()
['a', 'c', 'b', 'bd', 'bda', 'd', 'bdc', 'bc', 'bcd', 'cb',
 'ca', 'ac', 'cba', 'ba', 'cbd', 'bdca', 'db', 'dc', 'cd',
 'bdac', 'ab', 'abd', 'da', 'ad', 'cbad', 'acb', 'abc',
 'abcd', 'acbd', 'cda', 'cdb', 'dac', 'dba', 'dbc', 'dbca',
 'dcb', 'abdc', 'cdab', 'bcda', 'dab', 'acd', 'dabc', 'cbda',
 'bca', 'dacb', 'bad', 'adb', 'bac', 'cab', 'adc', 'cdba',
 'dca', 'cad', 'adbc', 'adcb', 'dbac', 'dcba', 'acdb', 'bacd',
 'cabd', 'cadb', 'badc', 'bcad', 'dcab']
```

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()
Graphics object consisting of 60 graphics primitives
sage: S.j_transversal_of_idempotents() # random (arbitrary choice)
['acb', 'ac', 'ab', 'bc', 'a', 'c', 'b']
```

We conclude by running systematic tests on this semigroup:
```
sage: TestSuite(S).run(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
```

```
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class **Element**
    Bases: `sage.structure.element_wrapper.ElementWrapper`

LeftRegularBand.**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'
```

LeftRegularBand.**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'
```

LeftRegularBand.**semigroup_generators**()
    Returns the generators of the semigroup.

    EXAMPLES:
```
sage: S = FiniteSemigroups().example(alphabet=('x','y'))
sage: S.semigroup_generators()
Family ('x', 'y')
```

15.11 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: sage.structure.unique_representation.UniqueRepresentation,
     sage.structure.parent.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:
```
sage: S = FiniteWeylGroups().example()
sage: S
The symmetric group on {0, ..., 3}
sage: S.category()
Category of finite weyl groups
```

     The elements of this group are permutations of the set $\{0,\dots,3\}$:
```
sage: S.one()
(0, 1, 2, 3)
sage: S.an_element()
(1, 2, 3, 0)
```

     The group itself is generated by the elementary transpositions:
```
sage: S.simple_reflections()
Finite family {0: (1, 0, 2, 3), 1: (0, 2, 1, 3), 2: (0, 1, 3, 2)}
```

     TESTS:
```
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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_has_descent() . . . pass
running ._test_inverse() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
```

```
running ._test_prod() . . . pass
running ._test_reduced_word() . . . pass
running ._test_simple_projections() . . . pass
running ._test_some_elements() . . . pass
```

Only the following basic operations are implemented:

- `one()`
- `product()`
- `simple_reflection()`
- `Element.has_right_descent()`.

All the other usual Weyl group operations are inherited from the categories:

```
sage: S.cardinality()
24
sage: S.long_element()
(3, 2, 1, 0)
sage: S.cayley_graph(side = "left").plot()
Graphics object consisting of 120 graphics primitives
```

Alternatively, one could have implemented `sage.categories.coxeter_groups.CoxeterGroups.ElementMethod` instead of `simple_reflection()` and `product()`. See `CoxeterGroups().example()`.

class **Element**
    Bases: `sage.structure.element_wrapper.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()
```

SymmetricGroup.**index_set**()
    Implements `CoxeterGroups.ParentMethods.index_set()`.

    EXAMPLES:
```
sage: FiniteWeylGroups().example().index_set()
[0, 1, 2]
```

SymmetricGroup.**one**()
    Implements `Monoids.ParentMethods.one()`.

    EXAMPLES:
```
sage: FiniteWeylGroups().example().one()
(0, 1, 2, 3)
```

SymmetricGroup.**product**(*x*, *y*)
    Implements `Semigroups.ParentMethods.product()`.

    EXAMPLES:

```
sage: s = FiniteWeylGroups().example().simple_reflections()
sage: s[1] * s[2]
(0, 2, 3, 1)
```

SymmetricGroup.**simple_reflection**(*i*)

Implements `CoxeterGroups.ParentMethods.simple_reflection()` by returning the transposition $(i, i + 1)$.

EXAMPLES:

```
sage: FiniteWeylGroups().example().simple_reflection(2)
(0, 1, 3, 2)
```

15.12 Examples of graded modules with basis

sage.categories.examples.graded_modules_with_basis.**Example**

alias of `GradedPartitionModule`

class sage.categories.examples.graded_modules_with_basis.**GradedPartitionModule**(*base_ring*)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

This class illustrates an implementation of a graded module with basis: the free module over partitions.

INPUT:

- R - base ring

The implementation involves the following:

- A choice of how to represent elements. In this case, the basis elements are partitions. The algebra is constructed as a `CombinatorialFreeModule` on the set of partitions, so it inherits all of the methods for such objects, and has operations like addition already defined.

  ```
  sage: A = GradedModulesWithBasis(QQ).example()
  ```

- A basis function - this module is graded by the non-negative integers, so there is a function defined in this module, creatively called `basis()`, which takes an integer d as input and returns a family of partitions representing a basis for the algebra in degree d.

  ```
  sage: A.basis(2)
  Lazy family (Term map from Partitions to An example of a graded module with basis: the free
  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 `CombinatorialFreeModuleElement`. 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*)

Returns the basis for (an homogeneous component of) this graded module

INPUT:

•*d* – non negative integer or `None`, optional (default: `None`)

If *d* is None, returns a basis of the module. Otherwise, returns the basis of the homogeneous component of degree *d*.

EXAMPLES:
```
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.basis(4)
Lazy family (Term map from Partitions to An example of a graded module with basis: the free
```

Without arguments, the full basis is returned:
```
sage: A.basis()
Lazy family (Term map from Partitions to An example of a graded module with basis: the free
sage: A.basis()
Lazy family (Term map from Partitions to An example of a graded module with basis: the free
```

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))))
<type 'sage.rings.integer.Integer'>
```

15.13 Examples of algebras with basis

class sage.categories.examples.hopf_algebras_with_basis.**MyGroupAlgebra**(R, G)

 Bases: sage.combinat.free_module.CombinatorialFreeModule

 An 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
sage: A.algebra_generators()
Finite family {(1,3): B[(1,3)], (1,2,3): B[(1,2,3)]}
```

 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.coproduct_on_basis(a)
B[(1,2,3)] # B[(1,2,3)]
```

 counit_on_basis(g)

  Counit, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.counit_on_basis().

  The counit on the basis elements is 1.

  EXAMPLES:
```
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:
> ```
> 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:
> ```
> 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)]
> ```

15.14 Examples of infinite enumerated sets

sage.categories.examples.infinite_enumerated_sets.**Example**
> alias of NonNegativeIntegers

class sage.categories.examples.infinite_enumerated_sets.**NonNegativeIntegers**
> Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`
>
> An example of infinite enumerated set: the non negative integers
>
> This class provides a minimal implementation of an infinite enumerated set.
>
> EXAMPLES:
> ```
> sage: NN = InfiniteEnumeratedSets().example()
> sage: NN
> An example of an infinite enumerated set: the non negative integers
> sage: NN.cardinality()
> +Infinity
> sage: NN.list()
> Traceback (most recent call last):
> ...
> NotImplementedError: infinite list
> sage: NN.element_class
> <type 'sage.rings.integer.Integer'>
> sage: it = iter(NN)
> sage: [next(it), next(it), next(it), next(it), next(it)]
> [0, 1, 2, 3, 4]
> sage: x = next(it); type(x)
> <type 'sage.rings.integer.Integer'>
> sage: x.parent()
> Integer Ring
> sage: x+3
> ```

```
8
sage: NN(15)
15
sage: NN.first()
0
```

This checks that the different methods of NN 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_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_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
> ```

15.15 Examples of monoids

sage.categories.examples.monoids.**Example**
> alias of `FreeMonoid`

class sage.categories.examples.monoids.**FreeMonoid**(*alphabet=('a', 'b', 'c', 'd')*)
> Bases: `sage.categories.examples.semigroups.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(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . 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_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_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

class **Element**
    Bases: `sage.structure.element_wrapper.ElementWrapper`

`FreeMonoid.`**`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', 'c': 'c', 'b': 'b', 'd': 'd'}
      sage: a,b,c,d = M.monoid_generators()
      sage: a*d*c*b
      'adcb'
```

FreeMonoid.**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()
''
```

15.16 Examples of posets

class sage.categories.examples.posets.**FiniteSetsOrderedByInclusion**
    Bases: sage.structure.unique_representation.UniqueRepresentation,
    sage.structure.parent.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(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_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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

    class Element
        Bases: sage.structure.element_wrapper.ElementWrapper

        wrapped_class
            alias of Set_object_enumerated

FiniteSetsOrderedByInclusion.**an_element**()
> Returns an element of this poset

> EXAMPLES:
```
sage: B = Posets().example()
sage: B.an_element()
{1, 4, 6}
```

FiniteSetsOrderedByInclusion.**le**(x, y)
> Returns whether x is a subset of y

> EXAMPLES:
```
sage: P = Posets().example()
sage: P.le( P(Set([1,3])), P(Set([1,2,3])) )
True
sage: P.le( P(Set([1,3])), P(Set([1,3])) )
True
sage: P.le( P(Set([1,2])), P(Set([1,3])) )
False
```

class sage.categories.examples.posets.**PositiveIntegersOrderedByDivisibilityFacade**
> Bases: sage.structure.unique_representation.UniqueRepresentation,
> sage.structure.parent.Parent

> An example of a facade poset: the positive integers ordered by divisibility

> This class provides a minimal implementation of a facade poset

> EXAMPLES:
```
sage: P = Posets().example("facade"); P
An example of a facade poset: the positive integers ordered by divisibility
```

```
sage: 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
```

```
sage: 3 in P
True
sage: 0 in P
False
```

> **class element_class**(X)
> > Bases: sage.sets.set.Set_object_enumerated, sage.categories.sets_cat.Sets.parent_class

> > A finite enumerated set.

> PositiveIntegersOrderedByDivisibilityFacade.**le**(x, y)
> > Returns whether x is divisible by y

> > EXAMPLES:
```
sage: P = Posets().example("facade")
sage: P.le(3, 6)
True
sage: P.le(3, 3)
True
```

```
sage: P.le(3, 7)
False
```

15.17 Examples of semigroups in cython

class sage.categories.examples.semigroups_cython.**IdempotentSemigroups**(*s=None*)

Bases: `sage.categories.category.Category`

Initializes this category.

EXAMPLES:

```
sage: class SemiprimitiveRings(Category):
....:     def super_categories(self):
....:         return [Rings()]
....:
....:     class ParentMethods:
....:         def jacobson_radical(self):
....:             return self.ideal(0)
....:
sage: C = SemiprimitiveRings()
sage: C
Category of semiprimitive rings
sage: C.__class__
<class '__main__.SemiprimitiveRings_with_category'>
```

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.

ElementMethods

alias of `IdempotentSemigroupsElementMethods`

super_categories()

EXAMPLES:

```
sage: from sage.categories.examples.semigroups_cython import IdempotentSemigroups
sage: IdempotentSemigroups().super_categories()
[Category of semigroups]
```

class sage.categories.examples.semigroups_cython.**IdempotentSemigroupsElementMethods**

Bases: `object`

is_idempotent_cpdef()

EXAMPLES:

```
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: S(2).is_idempotent_cpdef()   # todo: not implemented (binding; see __getattr__)
True
```

class sage.categories.examples.semigroups_cython.**LeftZeroSemigroup**

Bases: `sage.categories.examples.semigroups.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. Also, the category itself includes some cython methods.

This is purely a proof of concept. The code obviously needs refactorisation!

Comments:

- nested classes seem not to be currently supported by Cython

- one cannot play ugly class surgery tricks (as with _mul_parent). available operations should really be declared to the coercion model!

EXAMPLES:

```
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup(); S
An example of a semigroup: the left zero semigroup
```

This is the semigroup which contains all sort of objects:

```
sage: S.some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

with product rule is 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         # todo: not implemented (see __getattr__)
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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_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.element_class.is_idempotent of 42>
sage: S(42).is_idempotent()
True

sage: S(42)._pow_                     # todo: not implemented (how to bind it?)
<method '_pow_' of 'sage.categories.examples.semigroups_cython.IdempotentSemigroupsElement' obje
sage: S(42)^10                        # todo: not implemented (see __getattr__)
```

```
42

sage: S(42).is_idempotent_cpdef    # todo: not implemented (how to bind it?)
<method 'is_idempotent_cpdef' of 'sage.categories.examples.semigroups_cython.IdempotentSemigroup
sage: S(42).is_idempotent_cpdef() # todo: not implemented (see __getattr__)
True
```

Element
    alias of `LeftZeroSemigroupElement`

class `sage.categories.examples.semigroups_cython.`**`LeftZeroSemigroupElement`**
    Bases: `sage.structure.element.Element`

    EXAMPLES:
```
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: x = S(3)
sage: TestSuite(x).run()
```

15.18 Examples of semigroups

class `sage.categories.examples.semigroups.`**`FreeSemigroup`**(*alphabet=('a', 'b', 'c', 'd')*)
    Bases: `sage.structure.unique_representation.UniqueRepresentation`,
    `sage.structure.parent.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 contatenation:
```
sage: S('dab') * S('acb')
'dabacb'
```

    TESTS:
```
sage: TestSuite(S).run()
```

    class Element
        Bases: `sage.structure.element_wrapper.ElementWrapper`

        The class for elements of the free semigroup.

    `FreeSemigroup.`**`an_element`**()
        Returns an element of the semigroup.

        EXAMPLES:

```
sage: F = Semigroups().example('free')
sage: F.an_element()
'abcd'
```

FreeSemigroup.**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'
```

FreeSemigroup.**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: sage.structure.unique_representation.UniqueRepresentation,
sage.structure.parent.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
```

 TESTS:
```
sage: S._test_not_implemented_methods()
Traceback (most recent call last):
  ...
AssertionError: Not implemented method: lift

sage: TestSuite(S).run(verbose = True)
running ._test_an_element() . . . pass
running ._test_associativity() . . . fail
Traceback (most recent call last):
  ...
NotImplementedError: <abstract method retract at ...>
------------------------------------------------------------
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_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_not_implemented_methods() . . . fail
Traceback (most recent call last):
  ...
AssertionError: Not implemented method: lift
----------------------------------------------------
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
The following tests failed: _test_associativity, _test_not_implemented_methods
```

  class **Element**
    Bases: `sage.structure.element_wrapper.ElementWrapper`

`IncompleteSubquotientSemigroup.`**`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: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.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
```

TESTS:
```
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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

> **class Element**
>
> > Bases: `sage.structure.element_wrapper.ElementWrapper`
> >
> > **is_idempotent()**
> >
> > > Trivial implementation of `Semigroups.Element.is_idempotent` since all elements of this semigroup are idempotent!
> > >
> > > EXAMPLES:
> > > ```
> > > sage: S = Semigroups().example()
> > > sage: S.an_element().is_idempotent()
> > > True
> > > sage: S(17).is_idempotent()
> > > True
> > > ```
>
> LeftZeroSemigroup.**an_element**()
>
> > Returns an element of the semigroup.
> >
> > EXAMPLES:
> > ```
> > sage: Semigroups().example().an_element()
> > 42
> > ```
>
> LeftZeroSemigroup.**product**(*x*, *y*)
>
> > Returns the product of x and y in the semigroup, as per `Semigroups.ParentMethods.product()`.
> >
> > EXAMPLES:
> > ```
> > sage: S = Semigroups().example()
> > sage: S('hello') * S('world')
> > 'hello'
> > sage: S(3)*S(1)*S(2)
> > 3
> > ```
>
> LeftZeroSemigroup.**some_elements**()
>
> > Returns a list of some elements of the semigroup.
> >
> > EXAMPLES:
> > ```
> > sage: Semigroups().example().some_elements()
> > [3, 42, 'a', 3.4, 'raton laveur']
> > ```

class `sage.categories.examples.semigroups.`**QuotientOfLeftZeroSemigroup**(*category=None*)

> Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`
>
> Example of a quotient semigroup
>
> EXAMPLES:

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

This is the quotient of:
```
sage: S.ambient()
An example of a semigroup: the left zero semigroup
```

obtained by setting $x = 42$ for any $x \geq 42$:
```
sage: S(100)
42
sage: S(100) == S(42)
True
```

The product is inherited from the ambient semigroup:
```
sage: S(1)*S(2) == S(1)
True
```

TESTS:
```
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_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class **Element**
  Bases: sage.structure.element_wrapper.ElementWrapper

QuotientOfLeftZeroSemigroup.**ambient**()
  Returns the ambient semigroup.

  EXAMPLES:
```
sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
An example of a semigroup: the left zero semigroup
```

QuotientOfLeftZeroSemigroup.**an_element**()
  Returns an element of the semigroup.

  EXAMPLES:

```
sage: S = Semigroups().Subquotients().example()
sage: S.an_element()
42
```

QuotientOfLeftZeroSemigroup.**lift**(*x*)

Lift the element x into the ambient semigroup.

INPUT:

- •x – an element of `self`.

OUTPUT:

- •an element of `self.ambient()`.

EXAMPLES:

```
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
```

QuotientOfLeftZeroSemigroup.**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
```

TESTS:

```
sage: S.retract(L(171)) in S
True
```

QuotientOfLeftZeroSemigroup.**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]
```

QuotientOfLeftZeroSemigroup.**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
> ```

15.19 Examples of sets

class sage.categories.examples.sets_cat.**PrimeNumbers**
> Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent
>
> An example of parent in the category of sets: the set of prime numbers.
>
> The elements are represented as plain integers in **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)
> <type '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)
> <type 'sage.rings.integer.Integer'>
>
> sage: TestSuite(P).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_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

an_element()
>   Implements `Sets.ParentMethods.an_element()`.
>
>   TESTS:
>   ```
>   sage: P = Sets().example()
>   sage: x = P.an_element(); x
>   47
>   sage: x.parent()
>   Integer Ring
>   ```

element_class
>   alias of `Integer`

class sage.categories.examples.sets_cat.**PrimeNumbers_Abstract**
>   Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.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: `sage.structure.element.Element`
>   >
>   >   **is_prime**()
>   >   >   Returns if a prime number is prime = True !
>   >   >
>   >   >   EXAMPLES:
>   >   >   ```
>   >   >   sage: P = Sets().example("inherits")
>   >   >   sage: x = P.an_element()
>   >   >   sage: P.an_element().is_prime()
>   >   >   True
>   >   >   ```
>   >
>   >   **next**()
>   >   >   Returns the next prime number
>   >   >
>   >   >   EXAMPLES:
>   >   >   ```
>   >   >   sage: P = Sets().example("inherits")
>   >   >   sage: next(P.an_element())
>   >   >   53
>   >   >   ```

PrimeNumbers_Abstract.**an_element**()
> Implements `Sets.ParentMethods.an_element()`.
>
> TESTS:
> ```
> sage: P = Sets().example("inherits")
> sage: x = P.an_element(); x
> 47
> sage: x.parent()
> Set of prime numbers
> ```

PrimeNumbers_Abstract.**next**(*i*)
> Returns 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
> ```

PrimeNumbers_Abstract.**some_elements**()
> Returns 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: `sage.categories.examples.sets_cat.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 **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)
> <type '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)
> <type 'sage.rings.integer.Integer'>
> ```

```
sage: z = P.next(x); z
17
sage: type(z)
<type 'sage.rings.integer.Integer'>
sage: z.parent()
Integer Ring
```

The disadvantage of this implementation is that the element doesn't know that they are primes so that prime testing is slow:

```
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. Then the elements themselve "know" that they are prime:

```
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
```

And moreover, the next methods for the element does not exists:

```
sage: pf.next()
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'next'
```

whereas:

```
sage: next(pw)
53
sage: next(pi)
53
```

TESTS:

```
sage: TestSuite(P).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_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_not_implemented_methods() . . . pass
```

```
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

element_class
     alias of Integer

class sage.categories.examples.sets_cat.**PrimeNumbers_Inherits**
     Bases: sage.categories.examples.sets_cat.PrimeNumbers_Abstract

An example of parent in the category of sets: the set of prime numbers. In this implementation, the element are stored as object of a new class which inherits from the class Integer (technically IntegerWrapper).

EXAMPLES:

```
sage: P = Sets().example("inherits")
sage: P
Set of prime numbers
sage: P(12)
Traceback (most recent call last):
...
ValueError: 12 is not a prime number
sage: a = P.an_element()
sage: a.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)
<type 'sage.rings.integer.Integer'>
sage: y.parent()
Integer Ring
sage: type(P(13)+P(17))
<type 'sage.rings.integer.Integer'>
sage: type(P(2)+P(3))
<type 'sage.rings.integer.Integer'>

sage: z = P.next(x); z
17
sage: type(z)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'>
sage: z.parent()
Set of prime numbers

sage: TestSuite(P).run(verbose=True)
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
```

```
  running ._test_eq() . . . 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_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

See also:
```
sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: P = Sets().example("wrapper")
```

class **Element** (*parent*, *p*)

> Bases: sage.rings.integer.IntegerWrapper, sage.categories.examples.sets_cat.PrimeNumbe

> TESTS:
> ```
> sage: P = Sets().example("inherits")
> sage: P(12)
> Traceback (most recent call last):
> ...
> ValueError: 12 is not a prime number
> sage: x = P(13); type(x)
> <class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category.element_class'
> sage: x.parent() is P
> True
> ```

class sage.categories.examples.sets_cat.**PrimeNumbers_Wrapper**

> Bases: sage.categories.examples.sets_cat.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:
> ```
> 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)
<type 'sage.rings.integer.Integer'>

sage: z = P.next(x); z
17
sage: type(z)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Wrapper_with_category.element_class'>
sage: z.parent()
Set of prime numbers (wrapper implementation)
```

TESTS:
```
sage: TestSuite(P).run()
```

class **Element**
    Bases: sage.structure.element_wrapper.ElementWrapper,
    sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element

PrimeNumbers_Wrapper.**ElementWrapper**
    alias of ElementWrapper

15.20 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: sage.structure.unique_representation.UniqueRepresentation,
    sage.structure.parent.Parent

    Non negative integers graded by themselves.

    EXAMPLES:
```
sage: E = SetsWithGrading().example()
sage: E
Non negative integers
sage: E.graded_component(0)
{0}
sage: E.graded_component(100)
{100}
```

    an_element()
        Returns 0.

        EXAMPLES:
```
sage: SetsWithGrading().example().an_element()
0
```

    generating_series(*var='z'*)
        Returns $1/(1 - z)$.

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
```

graded_component(*grade*)
> Returns the component with grade `grade`.

> EXAMPLES:
> ```
> sage: N = SetsWithGrading().example()
> sage: N.graded_component(65)
> {65}
> ```

grading(*elt*)
> Returns the grade of `elt`.

> EXAMPLES:
> ```
> sage: N = SetsWithGrading().example()
> sage: N.grading(10)
> 10
> ```

15.21 Examples of parents endowed with multiple realizations

class sage.categories.examples.with_realizations.**SubsetAlgebra**(*R, S*)
> Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.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:

$$In_s = \sum_{t \subset s} F_t \qquad \text{and} \qquad F_s = \sum_{t \supset s} Out_t$$

> 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

> 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`:

```
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:

```
sage: A.inject_shorthands()
Injecting F as shorthand for The subset algebra of {1, 2, 3} over Rational Field in the Fundamen
...
Injecting In as shorthand for The subset algebra of {1, 2, 3} over Rational Field in the In basi
...
```

Accessing the basis elements is done with `basis()` method:

```
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:

```
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:

```
sage: F[2,3]
F[{2, 3}]
sage: In[1,3]
In[{1, 3}]
```

Some conversions:

```
sage: F(In[2,3])
F[{}] + F[{2}] + F[{3}] + F[{2, 3}]
sage: In(F[2,3])
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
```

We can now mix expressions:

```
sage: (1 + Out[1]) * In[2,3]
Out[{}] + 2*Out[{1}] + 2*Out[{2}] + 2*Out[{3}] + 2*Out[{1, 2}] + 2*Out[{1, 3}] + 4*Out[{2, 3}] +
```

---

class Bases (*parent_with_realization*)

　　Bases: `sage.categories.realizations.Category_realization_of_parent`

　　The category of the realizations of the subset algebra

　　class ParentMethods

　　　　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[{}]
sage: Out.one()
Out[{}] + Out[{1}] + Out[{2}] + Out[{3}] + Out[{1, 2}] + Out[{1, 3}] + Out[{2, 3}] +
```

　　`SubsetAlgebra.Bases.`**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()
[Join of Category of algebras over Rational Field and Category of realizations of unital
 Category of realizations of The subset algebra of {1, 2, 3} over Rational Field,
 Category of algebras with basis over Rational Field]
```

class SubsetAlgebra.**Fundamental** (*A*)

　　Bases: `sage.combinat.free_module.CombinatorialFreeModule`, `sage.misc.bindable_class.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()
sage: A.one_basis()
word:
sage: A.one()
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[{}]
```

product_on_basis(*left*, *right*)

　　Product of basis elements, as per `AlgebrasWithBasis.ParentMethods.product_on_basis()`.

　　INPUT:

　　　　•`left`, `right` – sets indexing basis elements

　　EXAMPLES:

```
sage: F = Sets().WithRealizations().example().F(); F
The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis
sage: S = F.basis().keys(); S
Subsets of {1, 2, 3}
sage: F.product_on_basis(S([]), S([]))
F[{}]
sage: F.product_on_basis(S({1}), S({3}))
F[{1, 3}]
sage: F.product_on_basis(S({1,2}), S({2,3}))
F[{1, 2, 3}]
```

class SubsetAlgebra.**In**(*A*)

　　Bases: 　　　　　　　　　`sage.combinat.free_module.CombinatorialFreeModule`,
　　`sage.misc.bindable_class.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
```

　　TESTS:

The product in this basis is computed by converting to the fundamental basis, computing the product there, and then converting back:

```
sage: In = Sets().WithRealizations().example().In(); In
The subset algebra of {1, 2, 3} over Rational Field in the In basis
sage: x = In.an_element()
sage: y = In.an_element()
sage: In.product
<bound method ....product_by_coercion ...>
sage: In.product.__module__
'sage.categories.magmas'
sage: In.product(x, y)
-21*In[{}] - 2*In[{1}] + 19*In[{2}] + 53*In[{1, 2}]
```

class SubsetAlgebra.**Out**(*A*)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.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
```

TESTS:

The product in this basis is computed by converting to the fundamental basis, computing the product there, and then converting back:

```
sage: Out = Sets().WithRealizations().example().Out(); Out
The subset algebra of {1, 2, 3} over Rational Field in the Out basis
sage: x = Out.an_element()
sage: y = Out.an_element()
sage: Out.product
<bound method ....product_by_coercion ...>
sage: Out.product.__module__
'sage.categories.magmas'
sage: Out.product(x, y)
Out[{}] + 4*Out[{1}] + 9*Out[{2}] + Out[{1, 2}]
```

SubsetAlgebra.**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
```

SubsetAlgebra.**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}
```

SubsetAlgebra.**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}
```

SubsetAlgebra.**indices_cmp**(*x*, *y*)
    A comparison function on sets which gives a linear extension of the inclusion order.

    INPUT:

        •x, y – sets

    EXAMPLES:
```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: sorted(A.indices(), A.indices_cmp)
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

SubsetAlgebra.**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}]
```

MISCELLANEOUS

16.1 Group, ring, etc. actions on objects.

The terminology and notation used is suggestive of groups acting on sets, but this framework can be used for modules, algebras, etc.

A group action $G \times S \to S$ is a functor from G to Sets.

> **Warning:** An `Action` object only keeps a weak reference to the underlying set which is acted upon. This decision was made in trac ticket #715 in order to allow garbage collection within the coercion framework (this is where actions are mainly used) and avoid memory leaks.
>
> ```
> sage: from sage.categories.action import Action
> sage: class P: pass
> sage: A = Action(P(),P())
> sage: import gc
> sage: _ = gc.collect()
> sage: A
> <repr(<sage.categories.action.Action at 0x...>) failed: RuntimeError: This action acted on a set th
> ```
>
> To avoid garbage collection of the underlying set, it is sufficient to create a strong reference to it before the action is created.
>
> ```
> sage: _ = gc.collect()
> sage: from sage.categories.action import Action
> sage: class P: pass
> sage: q = P()
> sage: A = Action(P(),q)
> sage: gc.collect()
> 0
> sage: A
> Left action by <__main__.P instance at ...> on <__main__.P instance at ...>
> ```

AUTHOR:

- Robert Bradshaw: initial version

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

> Bases: `sage.categories.functor.Functor`
>
> **`act`**(g, a)
>
> > This is a consistent interface for acting on a by g, regardless of whether it's a left or right action.
>
> **`actor`**()

codomain()

domain()

is_left()

left_domain()

operation()

right_domain()

class sage.categories.action.**ActionEndomorphism**

    Bases: `sage.categories.morphism.Morphism`

    The endomorphism defined by the action of one element.

    EXAMPLES:
```
sage: A = ZZ['x'].get_action(QQ, self_on_left=False, op=operator.mul)
sage: A
Left scalar multiplication by Rational Field on Univariate Polynomial
Ring in x over Integer Ring
sage: A(1/2)
Action of 1/2 on Univariate Polynomial Ring in x over Integer Ring
under Left scalar multiplication by Rational Field on Univariate
Polynomial Ring in x over Integer Ring.
```

class sage.categories.action.**InverseAction**

    Bases: `sage.categories.action.Action`

    An action that acts as the inverse of the given action.

    TESTS:

    This illustrates a shortcoming in the current coercion model. See the comments in _call_ below:
```
sage: x = polygen(QQ,'x')
sage: a = 2*x^2+2; a
2*x^2 + 2
sage: a / 2
x^2 + 1
sage: a /= 2
sage: a
x^2 + 1
```

    codomain()

class sage.categories.action.**PrecomposedAction**

    Bases: `sage.categories.action.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 trac ticket #14711 did not become a problem:
```
sage: E = ModularSymbols(11).2
sage: s = E.modular_symbol_rep()
sage: del E,s
sage: import gc
sage: _ = gc.collect()
sage: E = ModularSymbols(11).2
sage: v = E.manin_symbol_rep()
sage: c,x = v[0]
```

```
sage: y = x.modular_symbol_rep()
sage: A = y.parent().get_action(QQ, self_on_left=False, op=operator.mul)
sage: A
Left scalar multiplication by Rational Field on Abelian Group of all
Formal Finite Sums over Rational Field
with precomposition on right by Conversion 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()

16.2 Fast functions for the category framework

AUTHOR:

- Simon King (initial version)

class sage.categories.category_cy_helper.**AxiomContainer**

Bases: dict

A fast container for axioms.

This is derived from 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:
```
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:
```
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.**canonicalize_axioms**(*all_axioms*, *axioms*)

Canonicalize a set of axioms.

INPUT:

- all_axioms – all available axioms

- axioms – a set (or iterable) of axioms

Note: AxiomContainer provides a fast container for axioms, and the collection of axioms is stored in sage.categories.category_with_axiom. In order to avoid circular imports, we expect that the collection of all axioms is provided as an argument to this auxiliary function.

OUTPUT:

A set of axioms as a tuple sorted according to the order of the tuple `all_axioms` in `sage.categories.category_with_axiom`.

EXAMPLES:
```
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', '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
sage: get_axiom_index(all_axioms, 'AdditiveCommutative') == all_axioms['AdditiveCommutative']
True
```

sage.categories.category_cy_helper.**join_as_tuple**(*categories*, *axioms*, *ignore_axioms*)

Helper for `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:
```
sage: from sage.categories.category_cy_helper import join_as_tuple
sage: T = (Coalgebras(QQ), Sets().Finite(), Algebras(ZZ), SimplicialComplexes())
sage: join_as_tuple(T,(),())
(Category of algebras over Integer Ring,
 Category of finite monoids,
 Category of coalgebras over Rational Field,
 Category of 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 simplicial complexes)
sage: join_as_tuple(T,(),((Monoids(),'Finite'),))
(Category of algebras over Integer Ring,
 Category of coalgebras over Rational Field,
 Category of finite sets,
 Category of simplicial complexes)
```

16.3 Poor Man's map

class sage.categories.poor_man_map.**PoorManComposeMap**(*f*, *g*)

    Bases: [sage.categories.poor_man_map.PoorManMap](#)

    EXAMPLES:

```
sage: from sage.categories.poor_man_map import PoorManMap
sage: f = PoorManMap(factorial, domain = [1,2,3], codomain = [1,2,6])
sage: g = PoorManMap(sqrt,   domain = [1,4,9], codomain = [1,2,3])
sage: h = f*g
sage: h.codomain()
[1, 2, 6]
sage: h.domain()
[1, 4, 9]
sage: TestSuite(h).run()
```

class sage.categories.poor_man_map.**PoorManMap**(*function*, *domain=None*, *codomain=None*, *name=None*)

    Bases: sage.structure.sage_object.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

    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:

```
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

[D1974] M. Demazure, Desingularisation des varietes de Schubert, Ann. E. N. S., Vol. 6, (1974), p. 163-172

[M2009] Sarah Mason, An Explicit Construction of Type A Demazure Atoms, Journal of Algebraic Combinatorics, Vol. 29, (2009), No. 3, p.295-313. Arxiv 0707.4267

[Deodhar]   22. Deodhar, A splitting criterion for the Bruhat orderings on Coxeter groups. Comm. Algebra, 15:1889-1894, 1987.

[Coh1996] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer, 1996.

[CR62] Curtis, Charles W.; Reiner, Irving "Representation theory of finite groups and associative algebras." Pure and Applied Mathematics, Vol. XI Interscience Publishers, a division of John Wiley & Sons, New York-London 1962 pp 545–547

[Ker1991] A. Kerber. Algebraic combinatorics via finite group actions, 2.2 p. 70. BI-Wissenschaftsverlag, Mannheim, 1991.

[EP13] David Einstein, James Propp. *Combinatorial, piecewise-linear, and birational homomesy for products of two chains*. Arxiv 1310.5294v1.

[EP13b] David Einstein, James Propp. *Piecewise-linear and birational toggling*. Extended abstract for FPSAC 2014. http://faculty.uml.edu/jpropp/fpsac14.pdf

[GR13] Darij   Grinberg,   Tom   Roby.   *Iterative   properties   of   birational   rowmotion   I*. http://web.mit.edu/~darij/www/algebra/skeletal.pdf

[Kac] Victor G. Kac. *Infinite-dimensional Lie Algebras*. Third edition. Cambridge University Press, Cambridge, 1990.

[L1995] Peter Littelmann, Crystal graphs and Young tableaux, J. Algebra 175 (1995), no. 1, 65–87.

[K1993] Masaki Kashiwara, The crystal base and Littelmann's refined Demazure character formula, Duke Math. J. 71 (1993), no. 3, 839–858.

[Assaf08] Sami Assaf. *A combinatorial realization of Schur-Weyl duality via crystal graphs and dual equivalence graphs*. FPSAC 2008, 141-152, Discrete Math. Theor. Comput. Sci. Proc., AJ, Assoc. Discrete Math. Theor. Comput. Sci., (2008). Arxiv 0804.1587v1

[BH1994] S. Billey, M. Haiman. *Schubert polynomials for the classical groups*. J. Amer. Math. Soc., 1994.

[Lam2008] T. Lam. *Schubert polynomials for the affine Grassmannian*. J. Amer. Math. Soc., 2008.

[LSS2009] T. Lam, A. Schilling, M. Shimozono. *Schubert polynomials for the affine Grassmannian of the symplectic group*. Mathematische Zeitschrift 264(4) (2010) 765-811 (Arxiv 0710.2720)

[Pon2010] S. Pon. *Types B and D affine Stanley symmetric functions*, unpublished PhD Thesis, UC Davis, 2010.

C

Symbols

A

B

D

F

H

I

M

P

Q

S

T

U

V