Sage Reference Manual: Category Framework

Release 8.7

The Sage Development Team

Mar 25, 2019
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.81</td>
<td>Graded Lie Algebras With Basis</td>
<td>441</td>
</tr>
<tr>
<td>3.82</td>
<td>Graded modules</td>
<td>441</td>
</tr>
<tr>
<td>3.83</td>
<td>Graded modules with basis</td>
<td>443</td>
</tr>
<tr>
<td>3.84</td>
<td>Graphs</td>
<td>444</td>
</tr>
<tr>
<td>3.85</td>
<td>Group Algebras</td>
<td>445</td>
</tr>
<tr>
<td>3.86</td>
<td>Groupoid</td>
<td>450</td>
</tr>
<tr>
<td>3.87</td>
<td>Groups</td>
<td>450</td>
</tr>
<tr>
<td>3.88</td>
<td>Hecke modules</td>
<td>458</td>
</tr>
<tr>
<td>3.89</td>
<td>Highest Weight Crystals</td>
<td>459</td>
</tr>
<tr>
<td>3.90</td>
<td>Hopf algebras</td>
<td>466</td>
</tr>
<tr>
<td>3.91</td>
<td>Hopf algebras with basis</td>
<td>468</td>
</tr>
<tr>
<td>3.92</td>
<td>H-trivial semigroups</td>
<td>471</td>
</tr>
<tr>
<td>3.93</td>
<td>Infinite Enumerated Sets</td>
<td>472</td>
</tr>
<tr>
<td>3.94</td>
<td>Integral domains</td>
<td>473</td>
</tr>
<tr>
<td>3.95</td>
<td>J-trivial semigroups</td>
<td>473</td>
</tr>
<tr>
<td>3.96</td>
<td>Kac-Moody Algebras</td>
<td>474</td>
</tr>
<tr>
<td>3.97</td>
<td>Lattice posets</td>
<td>474</td>
</tr>
<tr>
<td>3.98</td>
<td>Left modules</td>
<td>475</td>
</tr>
<tr>
<td>3.99</td>
<td>Lie Algebras</td>
<td>476</td>
</tr>
<tr>
<td>3.100</td>
<td>Lie Algebras With Basis</td>
<td>486</td>
</tr>
<tr>
<td>3.101</td>
<td>Lie Groups</td>
<td>488</td>
</tr>
<tr>
<td>3.102</td>
<td>Loop Crystals</td>
<td>489</td>
</tr>
<tr>
<td>3.103</td>
<td>L-trivial semigroups</td>
<td>501</td>
</tr>
<tr>
<td>3.104</td>
<td>Magmas</td>
<td>502</td>
</tr>
<tr>
<td>3.105</td>
<td>Magmas and Additive Magmas</td>
<td>515</td>
</tr>
<tr>
<td>3.106</td>
<td>Non-unital non-associative algebras</td>
<td>517</td>
</tr>
<tr>
<td>3.107</td>
<td>Manifolds</td>
<td>521</td>
</tr>
<tr>
<td>3.108</td>
<td>Matrix algebras</td>
<td>524</td>
</tr>
<tr>
<td>3.109</td>
<td>Metric Spaces</td>
<td>525</td>
</tr>
<tr>
<td>3.110</td>
<td>Modular abelian varieties</td>
<td>527</td>
</tr>
<tr>
<td>3.111</td>
<td>Modules</td>
<td>528</td>
</tr>
<tr>
<td>3.112</td>
<td>Modules With Basis</td>
<td>537</td>
</tr>
<tr>
<td>3.113</td>
<td>Monoid algebras</td>
<td>563</td>
</tr>
<tr>
<td>3.114</td>
<td>Monoids</td>
<td>563</td>
</tr>
<tr>
<td>3.115</td>
<td>Number fields</td>
<td>569</td>
</tr>
<tr>
<td>3.116</td>
<td>Objects</td>
<td>570</td>
</tr>
<tr>
<td>3.117</td>
<td>Partially ordered monoids</td>
<td>571</td>
</tr>
<tr>
<td>3.118</td>
<td>Permutation groups</td>
<td>572</td>
</tr>
<tr>
<td>3.119</td>
<td>Pointed sets</td>
<td>572</td>
</tr>
<tr>
<td>3.120</td>
<td>Polyhedral subsets of free ZZ, QQ or RR-modules.</td>
<td>573</td>
</tr>
<tr>
<td>3.121</td>
<td>Posets</td>
<td>573</td>
</tr>
<tr>
<td>3.122</td>
<td>Principal ideal domains</td>
<td>582</td>
</tr>
<tr>
<td>3.123</td>
<td>Quotient fields</td>
<td>583</td>
</tr>
<tr>
<td>3.124</td>
<td>Quantum Group Representations</td>
<td>589</td>
</tr>
<tr>
<td>3.125</td>
<td>Regular Crystals</td>
<td>595</td>
</tr>
<tr>
<td>3.126</td>
<td>Regular Supercrystals</td>
<td>602</td>
</tr>
<tr>
<td>3.127</td>
<td>Right modules</td>
<td>608</td>
</tr>
<tr>
<td>3.128</td>
<td>Ring ideals</td>
<td>608</td>
</tr>
<tr>
<td>3.129</td>
<td>Rings</td>
<td>609</td>
</tr>
<tr>
<td>3.130</td>
<td>Rings</td>
<td>617</td>
</tr>
<tr>
<td>3.131</td>
<td>R-trivial semigroups</td>
<td>617</td>
</tr>
<tr>
<td>3.132</td>
<td>Schemes</td>
<td>617</td>
</tr>
<tr>
<td>3.133</td>
<td>Semigroups</td>
<td>619</td>
</tr>
<tr>
<td>3.134</td>
<td>Semirings</td>
<td>630</td>
</tr>
</tbody>
</table>
3.135 Semisimple Algebras ................................................. 630
3.136 Sets .......................................................................... 631
3.137 Sets With a Grading .................................................... 657
3.138 SetsWithPartialMaps .................................................. 659
3.139 Shephard Groups ....................................................... 660
3.140 Simplicial Complexes .................................................. 660
3.141 Simplicial Sets .......................................................... 661
3.142 Super Algebras .......................................................... 668
3.143 Super algebras with basis ............................................. 668
3.144 Super Hopf algebras with basis ..................................... 669
3.145 Super modules .......................................................... 669
3.146 Super modules with basis ............................................. 672
3.147 Topological Spaces .................................................... 674
3.148 Kac-Moody Algebras With Triangular Decomposition Basis . 675
3.149 Unique factorization domains ...................................... 676
3.150 Unital algebras ......................................................... 678
3.151 Vector Spaces .......................................................... 680
3.152 Weyl Groups ........................................................... 682
3.153 Technical Categories .................................................. 691

4 Functorial constructions .................................................. 695
4.1 Covariant Functorial Constructions ................................ 695
4.2 Cartesian Product Functorial Construction ...................... 700
4.3 Tensor Product Functorial Construction ......................... 702
4.4 Dual functorial construction ......................................... 703
4.5 Group algebras and beyond: the Algebra functorial construction . 704
4.6 Subquotient Functorial Construction .............................. 710
4.7 Quotients Functorial Construction ................................ 710
4.8 Subobjects Functorial Construction ............................... 711
4.9 Isomorphic Objects Functorial Construction .................... 712
4.10 Homset categories ..................................................... 712
4.11 Realizations Covariant Functorial Construction ............... 716
4.12 With Realizations Covariant Functorial Construction ......... 718

5 Examples of parents using categories ................................. 723
5.1 Examples of algebras with basis .................................... 723
5.2 Examples of commutative additive monoids ..................... 724
5.3 Examples of commutative additive semigroups ................. 725
5.4 Examples of Coxeter groups ........................................ 727
5.5 Example of a crystal ................................................... 727
5.6 Examples of CW complexes ...................................... 729
5.7 Example of facade set ................................................ 730
5.8 Examples of finite Coxeter groups ................................ 731
5.9 Example of a finite dimensional algebra with basis .......... 733
5.10 Examples of finite enumerated sets ............................... 734
5.11 Examples of a finite dimensional Lie algebra with basis .... 736
5.12 Examples of finite monoids ........................................ 739
5.13 Examples of finite semigroups ..................................... 741
5.14 Examples of finite Weyl groups .................................. 743
5.15 Examples of graded connected Hopf algebras with basis .... 745
5.16 Examples of graded modules with basis ........................ 747
5.17 Examples of graphs .................................................. 749
5.18 Examples of algebras with basis .................................. 750
5.19 Examples of infinite enumerated sets ............................ 752
5.20  Examples of manifolds ......................................................... 753
5.21  Examples of a Lie algebra ...................................................... 754
5.22  Examples of a Lie algebra with basis .................................... 755
5.23  Examples of monoids ........................................................... 757
5.24  Examples of posets ............................................................. 758
5.25  Examples of semigroups in cython ....................................... 760
5.26  Examples of semigroups ....................................................... 762
5.27  Examples of sets ............................................................... 767
5.28  Example of a set with grading .............................................. 773
5.29  Examples of parents endowed with multiple realizations ........ 774

6  Internals  ................................................................................. 781
6.1  Specific category classes ..................................................... 781
6.2  Singleton categories ............................................................ 784
6.3  Fast functions for the category framework ......................... 787
6.4  Coercion methods for categories ....................................... 789
6.5  Poor Man’s map ................................................................. 789

7  Indices and Tables .................................................................. 791

Bibliography ............................................................................... 793

Python Module Index .................................................................. 795

Index .......................................................................................... 801
1.1 Elements, parents, and categories in Sage: a (draft of) primer

Contents

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

1.1.1 Abstract

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

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

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

- blending in the Magma inspired concept of Parent/Element
- being built on top of (and not into) the standard Python object oriented and class hierarchy mechanism. This did not require changing the language, and could in principle be implemented in any language supporting the creation of new classes dynamically.
The general philosophy is that *Building mathematical information into the system yields more expressive, more conceptual and, at the end, easier to maintain and faster code* (within a programming realm; this would not necessarily apply to specialized libraries like gmp!).

**One line pitch for mathematicians**

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

**One line pitch for programmers**

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

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

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

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

**Some challenges**

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

Similar objects should behave similarly:

```
sage: Permutations(5).cardinality()
sage: GL(2,2).cardinality()
sage: A=random_matrix(ZZ,6,3,x=7)
sage: L=LatticePolytope(A.rows())
sage: L.npoints()
# oops! # random
```

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

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

### 1.1.3 A bit of help from abstract algebra

#### The hierarchy of categories

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

Sage knows about monoids:

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

and sure enough, binary powering is defined there:

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

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

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

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

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

```sage
sage: Sets()
Category of sets
sage: Groups()
Category of groups
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 infinite 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 infinite groups, Category of groups, Category of monoids, ...
, Category of magmas,
Category of infinite sets, ...]
```

Those can be viewed graphically:

```
sage: g = Groups().category_graph()
sage: g.set_latex_options(format="dot2tex")
sage: view(g)  # 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)  # 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.

**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 $\mathbb{Q}^3$ to $\mathbb{Q}^3$.

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

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

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

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

Unlike parents, elements in Sage are not necessarily unique:

```
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 $\mathbb{Z}$-algebras, and the category of Cartesian products of $\mathbb{Z}$-algebras:
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:
Category of euclidean domains, Category of principal ideal domains,
Category of unique factorization domains, Category of gcd domains,
Category of integral domains, Category of domains,
Category of commutative rings, Category of rings, ...
Category of magmas and additive magmas, ...
Category of monoids, Category of semigroups,
Category of commutative magmas, Category of unital magmas, Category of magmas,
Category of commutative additive groups, ..., Category of additive magmas,
Category of infinite enumerated sets, Category of enumerated sets,
Category of infinite sets, Category of metric spaces,
Category of topological spaces, Category of sets,
Category of sets with partial maps,
Category of objects]

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

1.1.4 A bit of help from computer science

Hierarchy of classes

How are the bookshelves implemented in practice?

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

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

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

Hence, all the objects mentioned above should be instances of some classes. For example, an integer in Sage is an
instance of the class \texttt{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_ring.PolynomialRing_field_with_category._element_class'>
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__  # not tested (Trac #24275)
'sage.categories.semigroups'
```

We see that integers and polynomials have each their own multiplication method: the multiplication algorithms are indeed unrelated and deeply tied to their respective datastructures. On the other hand, as we have seen above, they share the same powering method because the set \( \mathbb{Z} \) of integers, and the set \( \mathbb{Q}[x] \) of polynomials are both semigroups. Namely, the class for integers and the class for polynomials both derive from an abstract class for semigroup elements, which factors out the generic methods like \( \_\text{pow}\_ \). This illustrates the use of hierarchy of classes to share common code between classes having common behaviour.

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

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

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

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

### 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("_",r"\_"))
sage: g.set_latex_options(format="dot2tex")
sage: view(g)  # 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”.

1.1. Elements, parents, and categories in Sage: a (draft of) primer
1.1.5 Sage categories

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

Dynamic hierarchy of classes

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

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

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

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

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

We could implement the above hierarchies as usual:

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

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

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

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

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

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

class ParentMethods:
    # generic methods that apply to all groups

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

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

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

```python
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()`:

```python
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.magma.Magmas.Unital.Inverse.element_class'>)
```

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

```python
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.magma.Magmas.Unital.Inverse.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.magma.Magmas.Unital.Inverse.parent_class'>
...
```

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

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

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

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

1.1. Elements, parents, and categories in Sage: a (draft of) primer
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 \( \text{Cs}() \) is a super category of a category \( \text{Ds}() \) if Sage considers any object of \( \text{Ds}() \) to be an object of \( \text{Cs}() \), up to an implicit application of a canonical functor from \( \text{Ds}() \) to \( \text{Cs}() \). This functor is normally an inclusion of categories or a forgetful functor. Reciprocally, \( \text{Ds}() \) is said to be a subcategory of \( \text{Cs}() \).

**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.

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()
frozenseq(['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
```

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.

**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_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
    pass
    running ._test_elements_eq_reflexive() . . . pass
    running ._test_elements_eq_symmetric() . . . pass
    running ._test_elements_eq_transitive() . . . pass
    running ._test_elements_neq() . . . pass
    running ._test_enumerated_set_contains() . . . pass
    running ._test_enumerated_set_iter_cardinality() . . . pass
    running ._test_enumerated_set_iter_list() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    running ._test_some_elements() . . . pass
```

Tests can be run individually:
Here is how to access the code of this test:

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

Here is how to run the test on all elements:

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

See `TestSuite` for more information.

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

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

And rerun the test:

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

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

```python
sage: import pdb
sage: pdb.pm()  # not tested
```

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.

1.1.6 Case study

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

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

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

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

```python
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:

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

_pow\_int\_ is a generic method for all finite semigroups:

```python
sage: x._pow_int.__module__
sage: sage.categories.semigroups'
```

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

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

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

```python
sage: x._mul_.__module__
'sage.categories.coercion_methods'
```

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

```python
sage: x._mul_?? # not tested
sage: x._mul_.__module__
'sage.categories.coercion_methods'
sage: from six import get_method_function as gmf
sage: gmf(x._mul_) is gmf(Magmas.ElementMethods._mul_parent)
True
```

product is a mathematical method implemented by the parent:
Cayley graph is a generic method on the parent, provided by the `FiniteSemigroups` category:

```python
cayley_graph
```

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

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

```python
multiplication_table
```

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

Consider now the implementation of the semigroup:

```python
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:

```python
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:

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

See also the `abstract_method()` decorator.

Here is the code for the finite semigroups category:

```python
FiniteSemigroups?? # not tested
```

### 1.1.7 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):

```python
P = PermutationGroup([[1, 2, 3]]); P
Permutation Group with generators [(1,2,3)]
P.category()
```

```
sage: P.category()
Category of finite enumerated permutation groups
```

16 Chapter 1. The Sage Category Framework
In this case, the group is commutative, so we can specify this:

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

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

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

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

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

```python
sage: P = PermutationGroup([(1,2,3)], category=Sets()) # 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.

### 1.1.8 Scaling further: functorial constructions, axioms, …

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

**Functorial constructions**

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

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

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

```python
sage: A = AlgebrasWithBasis(QQ).example(); A.rename("A")
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:

```python
sage: C in Monoids()
True
```

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

```python
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:

```python
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`:

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

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

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

```python
sage: C.categories()
[Category of finite dimensional Cartesian products of algebras with basis over...
  Rational Field, ...
```

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

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

**Axioms**

**First examples**

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

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

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

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

In general, we can construct new categories in Sage by specifying the axioms that are satisfied by the operations of the super categories. For example, starting from the category of magmas, we can construct all the following categories just by specifying the axioms satisfied by the multiplication:

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

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

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

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

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

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

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

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

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

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

Note: When an axiom specifies the properties of some operations in Sage, the notations for those operations are tied to this axiom. For example, as we have seen above, we need two distinct axioms for associativity: the axiom “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 (ad-
ditive 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 name space? 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().Unital()
```

(continues on next page)
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 $\mathcal{C}_s$ be some subcategory of $\text{Modules}(\mathbb{Q})$, say the category of modules itself:

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

Then, $\mathcal{C}_s.\text{FiniteDimensional}()$ (respectively $\mathcal{C}_s.\text{Graded}()$) is the subcategory of the objects $\mathcal{O}$ of $\mathcal{C}_s$ which are finite dimensional (respectively graded).

Let also $\mathcal{D}_s$ be a subcategory of $\mathcal{C}_s$, 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 $\mathcal{D}_s$ is a subcategory of $\mathcal{C}_s$, then $\mathcal{D}_s.\text{A}()$ is a subcategory of $\mathcal{C}_s.\text{A}()$.

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

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

On the other hand, a graded module $\mathcal{O}$ which is also an algebra is not necessarily a graded algebra! Indeed, the grading on $\mathcal{O}$ may not be compatible with the product on $\mathcal{O}$:
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 $C_s$ of $D_s$, $D_s.A()$ is the intersection of the categories $D_s$ and $C_s.A()$:

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

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

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

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

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

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

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

Todo: Improve the printing of functorial constructions and joins to raise this potentially dangerous ambiguity.

Further reading on axioms

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

1.1. Elements, parents, and categories in Sage: a (draft of) primer
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.

1.1.9 Writing a new category

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

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

Each category should come with a good example, in sage.categories.examples.

Inserting the new category into the category graph

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

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

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

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

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.

**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:

```python
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 raw_getattr(Monoids().ParentMethods, "prod")
True
```

We recommend to study the code of one example:

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

**On the order of super categories**

The generic method `C.all_super_categories()` determines recursively the list of all super categories of `C`.

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

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

Since 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()`.
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
```

1.2 Categories

AUTHORS:

- David Kohel, William Stein and Nicolas M. Thiery

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

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

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

EXAMPLES:

We create a couple of categories:

```
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 finite dimensional vector spaces with basis
  over (number fields and quotient fields and metric spaces)
```
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 compatibility 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
```

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

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

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

This is achieved as follows:

```python
sage: from sage.categories.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 $A$, $B$, $C$, $D$ with a diamond inheritance. Think for example:

- $A$: the category of sets
- $B$: the category of additive groups
- $C$: the category of multiplicative monoids
- $D$: the category of rings

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

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

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

sage: class D(Category):
    ....:     def super_categories(self):
    ....:         return [B(), C()]
    ....:
    ....:       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:

```python
sage: import gc
sage: idA = id(A())
sage: _ = gc.collect()

sage: n == id(A())
False

sage: a = A()

sage: id(A()) == id(a)
True
```
We construct a parent in the category $\text{Ds}()$ (that, is an instance of $\text{Ds().parent_class}$), 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'
```

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

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

```
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'>, <... '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'>, <... 'object'>]
```

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

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

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

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

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

We now construct a parent in the usual way:

```python
sage: class myparent(Parent):
....:     def __init__(self):
....:         Parent.__init__(self, category=Ds())
....:     def g(self):
....:         return "myparent"
....:
....:     class Element(object):
....:         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'>,
 <class '__main__.Ds.parent_class'>,
 <class '__main__.Cs.parent_class'>,
 <class '__main__.Bs.parent_class'>,
 <class '__main__.As.parent_class'>,
 <... 'object'>]

sage: D.fA()
'A'

sage: D.fB()
'B'

sage: D.fC()
'C'

sage: D.fD()
'D'

sage: D.f()
'C'

sage: D.g()
'myparent'
```

```python
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__.myparent_with_category'>,
 <class '__main__.myparent'>,
 <type 'sage.structure.parent.Parent'>,
 <type 'sage.structure.category_object.CategoryObject'>,
 <class '__main__.Ds.parent_class'>,
 <class '__main__.Cs.parent_class'>,
 <class '__main__.Bs.parent_class'>,
 <class '__main__.As.parent_class'>,
 <... 'object'>]
```
_super_categories()

The immediate super categories of this category.

This lazy attribute caches the result of the mandatory method `super_categories()` for speed. It also does some mangling (flattening join categories, sorting,...). Whenever speed matters, developers are advised to use this lazy attribute rather than calling `super_categories()`.

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

EXAMPLES:

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

_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 to ensure that Python will find a proper Method Resolution Order for those classes. For background, see `sage.misc.c3_controlled`.

See also:

:_cmp_key().

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

EXAMPLES:

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

_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 \texttt{\_super_categories\_for\_classes()} as a side effect

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

\_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:

\texttt{all\_super\_categories()}

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

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

\_set\_of\_super\_categories()

The frozen set of all proper super categories of this category.

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

See also:

\texttt{all\_super\_categories()}

EXAMPLES:

\begin{verbatim}
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,
\end{verbatim}

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

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

INPUT:

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

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

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

Note:

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

See also:
CategoryWithParameters._make_named_class()

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

Note that, by default, the result is not cached:
Indeed this method is only meant to construct lazy attributes like `parent_class` which already handle this caching:

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

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

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

### `_repr_()`
Return the print representation of this category.

**EXAMPLES:**

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

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

**EXAMPLES:**

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

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

**See also:**

`TestSuite`.

**EXAMPLES:**

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

Let us now write a couple broken categories:

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

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

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

**See also:**

- `cmp_key()`

**INPUT:**

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

**OUTPUT:**

A sorted tuple of categories, possibly with repeats.

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

**EXAMPLES:**

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

static _sort_uniq(categories)
Return the categories after sorting them and removing redundant categories.

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

INPUT:
• categories – a list (or iterable) of categories

OUTPUT: a sorted tuple of mutually incomparable categories

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

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

static __classcall__(*args, **options)
Input mangling for unique representation.

Let C = Cs(...) 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 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

(continues on next page)
sage: C = GradedHopfAlgebrasWithBasis(QQ).Realizations(); C
Join of Category of graded hopf algebras with basis over Rational Field and
Category of realizations of hopf algebras over Rational Field
sage: C.super_categories()
[Category of graded hopf algebras with basis over Rational Field, Category of,
realizations of hopf algebras over Rational Field]

sage: TestSuite(C).run()

See also:

• Sets().WithRealizations
• ClasscallMetaClass

Todo: Add an optional argument to allow for:

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

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

INPUT:

• self – a category

See also:

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

• Various use cases:

  – SymmetricFunctions
  – QuasiSymmetricFunctions
  – NonCommutativeSymmetricFunctions
  – SymmetricFunctionsNonCommutingVariables
  – DescentAlgebra
  – algebras.Moebius
  – IwahoriHeckeAlgebra
  – ExtendedAffineWeylGroup

• The Implementing Algebraic Structures thematic tutorial.

• sage.categories.realizations

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

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

### Parent with realizations

Let us now explain the concept of realizations. A *parent with realizations* is a facade parent (see `SetsFacade`) admitting multiple concrete realizations where its elements are represented. Consider for example an algebra $A$ which admits several natural bases:

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

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

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

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

```python
sage: 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:

```python
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}] + 4*Out[{1, 2, 3}]
```

In our example, there are three realizations:

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

Instead of manually defining the shorthands $F$, $In$, and $Out$, as above one can just do:

```python
1.2. Categories 41
```
A subset algebra is defined over a Common Ring, often a Field, and a common basis which is then used in all the realizations. The common ring and the common basis define a common entry point. This is illustrated in the following examples, where we define a subset algebra, and show how the polynomial ring is created.

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

**Rationale**

Besides some goodies described below, the role of $A$ is threefold:

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

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

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

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

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

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

```python
sage: p = P([1, In[{1}], Out[{2}]]); p
Out[{2}]*x^2 + In[{1}]*x + F[{}]
sage: p^2
Out[{2}]*x^4
+ (-8*In[{}]) + 2*F[{1}]*x + F[{}]
```

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

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

```python
sage: p.map_coefficients(In)
(-4*In[{}]) + 2*In[{1}]*x + In[{}]
```

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

The set of all realizations of $A$, together with the coercion morphisms is a category (whose class inherits from \texttt{Category realization of parent}):

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

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

\begin{verbatim}sage: A.F() in A.Realizations()
True
\end{verbatim}

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

\begin{verbatim}sage: A in Algebras(QQ).WithRealizations()
True
\end{verbatim}

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

Typically, \texttt{VectorSpaces(QQ).FiniteDimensional().WithRealizations()} will eventually be in charge, whenever a coercion $\phi: A \hookrightarrow B$ is registered, to register $\phi^{-1}$ as coercion $B \hookrightarrow A$ if there is none defined yet. To achieve this, \texttt{FiniteDimensionalVectorSpaces} would provide a nested class \texttt{WithRealizations} implementing the appropriate logic.

\texttt{WithRealizations} is a \textit{regressive covariant functorial construction}. On our example, this simply means that $A$ is automatically in the category of rings with realizations (covariance):

\begin{verbatim}sage: A in Rings().WithRealizations()
True
\end{verbatim}

and in the category of algebras (regressiveness):

\begin{verbatim}sage: A in Algebras(QQ)
True
\end{verbatim}

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

```
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 $\ast$ that shall be preserved by magma morphisms. On the other hand the category of rings does not define additional structure: a function between two rings that is both a unital magma morphism and a unital additive magma morphism is automatically a ring morphism.

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

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

EXAMPLES:

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

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

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

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

Most categories with axiom don’t define additional structure:

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

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

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

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

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

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

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

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

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

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

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

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

- an euclidean domain morphism preserve euclidean division?
  
  sage: EuclideanDomains().additional_structure()  
  Category of euclidean domains  

- an enumerated set morphism preserve the distinguished enumeration?
  
  sage: EnumeratedSets().additional_structure()  

- a module with basis morphism preserve the distinguished basis?
  
  sage: Modules().additional_structure()  
  Category of modules over Integer Ring
See also:

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

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

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
```
classmethod an_instance()

Return an instance of this class.

EXAMPLES:

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

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

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

axioms()

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

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

EXAMPLES:

```sage
sage: Monoids().axioms()
frozenset({'Associative', 'Unital'})
sage: (EnumeratedSets().Infinite() & Sets().Facade()).axioms()
frozenset({'Enumerated', 'Facade', 'Infinite'})
```

category()

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

EXAMPLES:

```sage
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
sage: C = Algebras(QQ)
sage: G = C.category_graph()
sage: G.is_directed_acyclic()
True
```
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
```

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

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

See also:

parent_class()

example(*args, **keywords)
Returns an object in this category. Most of the time, this is a parent.

This serves three purposes:

• Give a typical example to better explain what the category is all about. (and by the way prove that the category is non empty :-))

• Provide a minimal template for implementing other objects in this category

• Provide an object on which to test generic code implemented by the category

For all those applications, the implementation of the object shall be kept to a strict minimum. The object is therefore not meant to be used for other applications; most of the time a full featured version is available elsewhere in Sage, and should be used 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
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
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
sage: Monoids().full_super_categories()
[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
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 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()) |  |
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)
(continues on next page)
**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]
```

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

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

If the optional parameter `as_list` is `True`, this returns the super categories of the join as a list, without constructing the join category itself:

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

**static meet** *(categories)*

Returns the meet of a list of categories

**INPUT:**

- `categories` - a non empty list (or iterable) of categories

**See also:**

- `__or__()` for a shortcut

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

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

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

**morphism_class()**

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

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

See also:

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

**EXAMPLES:**

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

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

Return category or self if category is None.

**INPUT:**

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

**OUTPUT:**

- a category

**EXAMPLES:**

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

(continues on next page)
... ValueError: Subcategory of `Category of monoids` required; got `Category of...

Otherwise, the two categories are joined together:

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

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

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

See also:

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

**EXAMPLES:**

```
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() # py2
{'element': {'optional': ['_add_', '_mul_'], 'required': ['__nonzero__']},
 'parent': {'optional': ['algebra_generators'], 'required': ['__contains__']}}
sage: Algebras(QQ).required_methods() # py3
{'element': {'optional': ['_add_', '_mul_'], 'required': ['__bool__']},
 'parent': {'optional': ['algebra_generators'], 'required': ['__contains__']}}
```

**structure()**
Return the structure `self` is endowed with.

### 1.2. Categories

55
This method returns the structure that morphisms in this category shall be preserving. For example, it
tells that a ring is a set endowed with a structure of both a unital magma and an additive unital magma
which satisfies some further axioms. In other words, a ring morphism is a function that preserves the unital
magma and additive unital magma structure.

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

EXAMPLES:

```python
sage: Objects().structure()
frozenset()
```

```python
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:

```python
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 subcate-
gory of some other category, and for documentation purposes. It is computed recursively from the result of
`additional_structure()` on the super categories of self.

**subcategory_class()**

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

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

See also:

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

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

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

**super_categories()**

Return the immediate super categories of self.

**OUTPUT:**

- a duplicate-free list of categories.

Every category should implement this method.

**EXAMPLES:**

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

**Note:** Since 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.

```python
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:**

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

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

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

INPUT:

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

ASSUMPTION:

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

OUTPUT:

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

Note:

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

See also:

CategoryWithParameters._make_named_class()

EXAMPLES:

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

Note that, by default, the result is not cached:
Indeed this method is only meant to construct lazy attributes like `parent_class` which already handle this caching:

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

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

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

sage: loads(dumps(PC)) is Rings().parent_class
True
```

## 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:**

```python
sage: from sage.categories.category import JoinCategory
sage: J = JoinCategory((Groups(), CommutativeAdditiveMonoids())); J
Join of Category of groups and Category of commutative additive monoids
sage: J.super_categories()
[Category of groups, Category of commutative additive monoids]

sage: J.all_super_categories(proper=True)
[Category of groups, ..., Category of magmas,
 Category of commutative additive monoids, ..., Category of additive magmas,
 Category of sets, Category of sets with partial maps, Category of objects]
```

By 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:

```python
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
```

### Category._repr_object_names()

Return the name of the objects of this category.

**EXAMPLES:**

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

### Category._repr_()

Return the print representation of this category.
EXAMPLES:

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

Category._without_axioms(named=False)

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

INPUT:

- named – a boolean (default: False)

Todo: Improve this explanation.

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

EXAMPLES:

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

additional_structure()

Return None.

Indeed, a join category defines no additional structure.

See also:

Category.additional_structure()

EXAMPLES:

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

is_subcategory(C)

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

EXAMPLES:

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

super_categories()

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

EXAMPLES:

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

```python
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
```

Return a sample of categories.

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

EXAMPLES:

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

Returns True if x is a category.

EXAMPLES:

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

This documentation covers how to implement axioms and proceeds with an overview of the implementation of the axiom infrastructure. It assumes that the reader is familiar with the category primer, and in particular its section about axioms.

1.3.1 Implementing axioms

Simple case involving a single predefined axiom

Suppose that one wants to provide code (and documentation, tests, ...) for the objects of some existing category $C_s()$ that satisfy some predefined axiom $A$. The first step is to open the hood and check whether there already exists a class implementing the category $C_s().A()$. For example, taking $C_s=$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 $C_s$, 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")
```
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`:

### 1.3. Axioms
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`:

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

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

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

### Making the category with axiom directly callable

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

```python
sage: Semigroups()
Category of semigroups
sage: Semigroups() is Magmas().Associative()
True
sage: FiniteGroups()
Category of finite groups
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`).
In our example, the attribute \texttt{\_base_category_class_and_axiom} was set upon calling \texttt{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 \texttt{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')
....:     def foo(self):
....:         print("I am a method on finite C's")
```

Having to set explicitly this link back from \texttt{FiniteCs} to \texttt{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 \texttt{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 \texttt{\_base_category_class_and_axiom()} for the details). This typically covers the following examples:

```
sage: FiniteCoxeterGroups()
Category of finite coxeter groups
sage: FiniteCoxeterGroups() is CoxeterGroups().Finite()
True
sage: FiniteCoxeterGroups._base_category_class_and_axiom_origin
'deduced by \_base_category_class_and_axiom'
```

(continues on next page)
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:

```python
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).

**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:

```python
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:

```python
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 $C$ and $D$ 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 $\ast$ and $\ast$. 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 subclass:

```
sage: from sage.categories.magmas_and_additive_magmas import MagmasAndAdditiveMagmas
sage: Magmas() & AdditiveMagmas() is MagmasAndAdditiveMagmas()
False
sage: Magmas() & AdditiveMagmas()
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:
Todo: Other categories that would be better implemented via an axiom depending on a join category include:

- **Algebras**: defining an associative unital algebra as a ring and a module satisfying the suitable compatibility axiom between inner multiplication and multiplication by scalars (bilinearity). Of course this should be implemented at the level of `MagmaticAlgebras`, if not higher.

- **Bialgebras**: defining an bialgebra as an algebra and coalgebra where the coproduct is a morphism for the product.

- **Bimodules**: defining a bimodule as a left and right module where the two actions commute.

Todo:

- Design and implement an idiom for the definition of an axiom by a join category.

- Or support more advanced joins, through some hook or registration process to specify that a given category is the intersection of two (or more) categories.

- Or at least improve the above workaround to avoid the last issue; this possibly could be achieved using a class `Magmas.Distributive` with a bit of `__classcall__` magic.

**Handling multiple axioms, arborescence structure of the code**

**Prelude**

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

```python
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:

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

However, the following may look suspicious at first:
Abstract model

As we have seen in the Primer, the objects of a category $\mathcal{C}$ can usually satisfy, or not, many different axioms. Out of all combinations of axioms, only a small number are relevant in practice, in the sense that we actually want to provide features for the objects satisfying these axioms.

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

- $\{\text{Associative}\} : \text{Semigroups}$
- $\{\text{Associative, Unital}\} : \text{Monoids}$
- $\{\text{Associative, Unital, Inverse}\} : \text{Groups}$
- $\{\text{Associative, Commutative}\} : \text{Commutative Semigroups}$
- $\{\text{Unital, Inverse}\} : \text{Loops}$

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

```python
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:

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

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 $(D_S)_{S \subseteq \mathcal{S}}$ of classes should be structured as an arborescence (or equivalently a rooted forest). The root is $\mathcal{C}$, 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 Algorithms). This is to be put in perspective with the manipulation of arbitrary collections of subsets (aka boolean functions) which can easily raise NP-hard problems.

Furthermore, thanks to its locality, the algorithms can be made suitably lazy: in particular, only the involved category classes need to be imported.

Flexibility

This design also brings in quite some flexibility, with the possibility to support features such as defining new axioms depending on other axioms and deduction rules. See below.

Asymmetry

As we have seen at the beginning of this section, this design introduces an asymmetry. It’s not so bad in practice, since in most practical cases, we want to work incrementally. It’s for example more natural to describe FiniteFields as Fields with the axiom Finite rather than Magmas and AdditiveMagmas with all (or at least sufficiently many) of the following axioms:

```sage
sage: sorted(Fields().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Associative', 'Commutative', 'Distributive', 'Division', 'NoZeroDivisors', 'Unital']
```

The main limitation is that the infrastructure currently imposes to be incremental by steps of a single axiom.

In practice, among the roughly 60 categories with axioms that are currently implemented in Sage, most admitted a (rather) natural choice of a base category and single axiom to add. For example, one usually thinks more naturally of a monoid as a semigroup which is unital rather than as a unital magma which is associative. Modeling this asymmetry in the code actually brings a bonus: it is used for printing out categories in a (heuristically) mathematician-friendly way:

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

Only in a few cases is a choice made that feels mathematically arbitrary. This is essentially in the chain of nested classes distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas.AdditiveAssociative.AdditiveCommutative.AdditiveUnital.Associative.

Placeholder classes

Given that we can only add a single axiom at a time when implementing a CategoryWithAxiom, we need to create a few category classes that are just placeholders. For the worst example, see the chain of nested classes distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas.AdditiveAssociative.AdditiveCommutative.AdditiveUnital.Associative.

This is suboptimal, but fits within the scope of the axiom infrastructure which is to reduce a potentially exponential number of placeholder category classes to just a couple.

1.3. Axioms
Note also that, in the above example, it’s likely that some of the intermediate classes will grow to non placeholder ones, as people will explore more weaker variants of rings.

Mismatch between the arborescence of nested classes and the hierarchy of categories

The fact that the hierarchy relation between categories is not reflected directly as a relation between the classes may sound suspicious at first! However, as mentioned in the primer, this is actually a big selling point of the axioms infrastructure: by calculating automatically the hierarchy relation between categories with axioms one avoids the nightmare of maintaining it by hand. Instead, only a rather minimal number of links needs to be maintained in the code (one per category with axiom).

Besides, with the flexibility introduced by runtime deduction rules (see below), the hierarchy of categories may depend on the parameters of the categories and not just their class. So it’s fine to make it clear from the onset that the two relations do not match.

Evolutivity

At this point, the arborescence structure has to be hardcoded by hand with the annoyances we have seen. This does not preclude, in a future iteration, to design and implement some idiom for categories with axioms that adds several axioms at once to a base category; maybe some variation around:

```python
class DistributiveMagmasAndAdditiveMagmas:
    ...

    @category_with_axiom(
        AdditiveAssociative,
        AdditiveCommutative,
        AdditiveUnital,
        AdditiveInverse,
        Associative)
    def _(): return LazyImport('sage.categories.rngs', 'Rngs', at_startup=True)
```

or:

```python
register_axiom_category(DistributiveMagmasAndAdditiveMagmas,
    {AdditiveAssociative,
     AdditiveCommutative,
     AdditiveUnital,
     AdditiveInverse,
     Associative},
    'sage.categories.rngs', 'Rngs', at_startup=True)
```

The infrastructure would then be in charge of building the appropriate arborescence under the hood. Or rely on some database (see discussion on 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:
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 `mags_and_additive_mags.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 \texttt{Rings().Division()}, the rule \texttt{Rings.Division-extra_super_categories()} simply adds \texttt{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, \texttt{DivisionRings().Finite()} coincides with \texttt{Fields().Finite()}:

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

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

A natural idiom would be to have \texttt{DivisionRings.Finite} be a link to \texttt{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 \texttt{DivisionRings().Finite()}, that \texttt{DivisionRings.Finite} is actually \texttt{Fields.Finite}, in order to construct appropriately \texttt{Fields().Finite()}; and reciprocally, upon computing the super categories of \texttt{Fields().Finite()}, to not try to add \texttt{DivisionRings().Finite()} as a super category.

Instead the current idiom is to have a method \texttt{DivisionRings.Finite_extra_super_categories} which mimicks the behavior of the would-be \texttt{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 \texttt{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 \texttt{DivisionRings}.

An alternative would be to implement the category of finite division rings (i.e. finite fields) in a class \texttt{DivisionRings.Finite} rather than \texttt{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 $C_1$s(), $C_2$s(), ... are mapped to the same category when applying some axiom $A$ (that is $C_1$s().$A() == C_2$s().$A() == ...), then one should be careful to implement this category in a single class $C_s.A$, and set up methods extra_super_categories or $A$ extra_super_categories methods as appropriate. Each such method should return something like $[C_2$s()$]$ and not $[C_2$s().$A()$] for the latter would likely lead to an infinite recursion.

**Design discussion**

Supporting similar deduction rules will be an important feature in the future, with quite a few occurrences already implemented in upcoming 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
```

(continues on next page)
.. code-block:: python

    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: As().B().C().super_categories()
[Category of e a1s, Category of f as, Category of b a1s, Category of c a1s]
```

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 $b \ c \ a_2s$, and choose to do so in $A2s.B.C$:

```python
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']

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

```python
sage: A2s().B().C() is A2s().E().F()
True
sage: C = A2s().E().F().D().B().C()

```

```python
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()  # idempotence
True
sage: C is A2s().D().E().F()
True
```

Finally, we “accidentally” implement the category of $b \ c \ a_1s$, both in $A3s.B.C$ and $A3s.E.F$:

```python
sage: class A3s(Category_singleton):
    ....:    def super_categories(self):
    ....:        return [As()]
    ....:
    ....:    class B(CategoryWithAxiom):
    ....:        class C(CategoryWithAxiom):
    ....:            def extra_super_categories(self):
    ....:                return [As().E(), As().F()]
    ....:
    ....:        class E(CategoryWithAxiom):
    ....:            class F(CategoryWithAxiom):
    ....:                def extra_super_categories(self):
    ....:                    return [As().B(), As().C()]

We can still construct, say:

1.3. Axioms
However,

```python
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).

### 1.3.2 Specifications

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

#### The lattice of constructible categories

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

```python
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.

#### Axioms

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

Consider the application $\phi_A$ which maps a category to its category of objects satisfying $A$. Equivalently, $\phi_A$ is computing the intersection with the defining category with axiom of $A$. It follows immediately from the latter that $\phi_A$ is a regressive endomorphism of the lattice of categories. It restricts to a regressive endomorphism $\text{Cs}() \rightarrow \text{Cs}().A()$ on the lattice of constructible categories.

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

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

Consider the intersection $S$ of such a fiber of $\phi_A$ with the upper set $I_A$ of categories that do not satisfy $A$. The fiber itself is a sublattice. However $I_A$ is not guaranteed to be stable under intersection (though exceptions should be rare). Therefore, there is a priori no guarantee that $S$ would be stable under intersection. Also it’s presumably finite, in fact small, but this is not guaranteed either.

Specifications

- Any constructible category $C$ should admit a finite number of larger constructible categories.
- The methods `super_categories`, `extra_super_categories`, and friends should always return strict supercategories.

For example, to specify that a finite division ring is a finite field, $\text{DivisionRings}$.\_\text{Finite}\_\text{extra_super_categories} should not return $\text{Fields}().\text{Finite}()$! It could possibly return $\text{Fields}()$; but it’s preferable to return the largest category that contains the relevant information, in this case $\text{Magmas}().\text{Commutative}()$, and to let the infrastructure apply the derivations.
- The base category of a $\text{CategoryWithAxiom}$ should be an implemented category (i.e. not a $\text{JoinCategory}$). This is checked by $\text{CategoryWithAxiom}.\_\text{test_category_with_axiom}()$.
- Arborescent structure: Let $\text{Cs}()$ be a category, and $S$ be some set of axioms defined in some super categories of $\text{Cs}()$ but not satisfied by $\text{Cs}()$. Suppose we want to provide a category with axiom for the elements of $\text{Cs}()$ satisfying the axioms in $S$. Then, there should be a single enumeration $A_1, A_2, \ldots, A_k$ without repetition of axioms in $S$ such that $\text{Cs}.A_1.A_2.\ldots.A_k$ is an implemented category. Furthermore, every intermediate step $\text{Cs}.A_1.A_2.\ldots.A_i$ with $i \leq k$ should be a category with axiom having $A_i$ as axiom and $\text{Cs}.A_1.A_2.\ldots.A_{i-1}$ as base category class; this base category class should not satisfy $A_i$. In particular, when some axioms of $S$ can be deduced from previous ones by deduction rules, they should not appear in the enumeration $A_1, A_2, \ldots, A_k$.
- In particular, if $\text{Cs}()$ is a category that satisfies some axiom $A$ (e.g. from one of its super categories), then it should not implement that axiom. For example, a category class $\text{Cs}$ can never have a nested class $\text{Cs}.A.A$. Similarly, applying the specification recursively, a category satisfying $A$ cannot have a nested class $\text{Cs}.A_1.A_2.A_3.A$ where $A_1, A_2, A_3$ are axioms.
- A category can only implement an axiom if this axiom is defined by some super category. The code has not been systematically checked to support having two super categories defining the same axiom (which should of course have the same semantic). You are welcome to try, at your own risk. :-)
- When a category defines an axiom or functorial construction $A$, this fixes the semantic of $A$ for all the subcategories. In particular, if two categories define $A$, then these categories should be independent, and either the semantic of $A$ should be the same, or there should be no natural intersection between the two hierarchies of subcategories.
- Any super category of a $\text{CategoryWithParameters}$ should either be a $\text{CategoryWithParameters}$ or a $\text{Category_singleton}$.
• 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()`.

1.3.3 Design goals

As pointed out in the primer, the main design goal of the axioms infrastructure is to subdue the potential combinatorial explosion of the category hierarchy by letting the developer focus on implementing a few bookshelves for which there is actual code or mathematical information, and let Sage compose dynamically and lazily these building blocks to construct the minimal hierarchy of classes needed for the computation at hand. This allows for the infrastructure to scale smoothly as bookshelves are added, extended, or reorganized.

Other design goals include:

• Flexibility in the code layout: the category of, say, finite sets can be implemented either within the Sets category (in a nested class `Sets.Finite`), or in a separate file (typically in a class `FiniteSets` in a lazily imported module `sage.categories.finite_sets`).

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

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

```
sage: Rings().Commutative().Finite().NoZeroDivisors()
Category of finite integral domains
sage: Rings().Finite().Division()
Category of finite enumerated fields
```

This will allow for progressively getting rid of all the entries like `GradedHopfAlgebrasWithBasis` which are polluting the global name space.

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

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

(continues on next page)
### 1.3.4 Upcoming features

**Todo:**

- Implement compatibility axiom / functorial constructions. For example, one would want to have:

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

- Once full subcategories are implemented (see trac ticket #10668), make the relevant categories with axioms be such. This can be done systematically for, e.g., the axioms `Associative` or `Commutative`, but not for the axiom `Unital`: a semigroup morphism between two monoids need not preserve the unit.

  Should all full subcategories be implemented in term of axioms?

### 1.3.5 Algorithms
Computing joins

The workhorse of the axiom infrastructure is the algorithm for computing the join $J$ of a set $C_1, \ldots, C_k$ of categories (see `Category.join()`). Formally, $J$ is defined as the largest constructible category such that $J \subset C_i$ for all $i$, and $J \subset C.A()$ for every constructible category $C \supset J$ and any axiom $A$ satisfied by $J$.

The join $J$ is naturally computed as a closure in the lattice of constructible categories: it starts with the $C_i$'s, gathers the set $S$ of all the axioms satisfied by them, and repeatedly adds each axiom $A$ to those categories that do not yet satisfy $A$ using `Category._with_axiom()`. Due to deduction rules or (extra) super categories, new categories or new axioms may appear in the process. The process stops when each remaining category has been combined with each axiom. In practice, only the smallest categories are kept along the way; this is correct because adding an axiom is covariant: $C.A()$ is a subcategory of $D.A()$ whenever $C$ is a subcategory of $D$.

As usual in such closure computations, the result does not depend on the order of execution. Furthermore, given that adding an axiom is an idempotent and regressive operation, the process is guaranteed to stop in a number of steps which is bounded by the number of super categories of $J$. In particular, it is a finite process.

Todo: Detail this a bit. What could typically go wrong is a situation where, for some category $C_1, C_1.A()$ specifies a category $C_2$ as super category such that $C_2.A()$ specifies $C_3$ as super category such that $\ldots$; this would clearly cause an infinite execution. Note that this situation violates the specifications since $C_1.A()$ is supposed to be a subcategory of $C_2.A()$, $\ldots$ so we would have an infinite increasing chain of constructible categories.

It’s reasonable to assume that there is a finite number of axioms defined in the code. There remains to use this assumption to argue that any infinite execution of the algorithm would give rise to such an infinite sequence.

Adding an axiom

Let $Cs$ be a category and $A$ an axiom defined for this category. To compute $Cs().A()$, there are two cases.

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

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

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

This is a highly recursive process. In fact, as such, it would run right away into an infinite loop! Indeed, the join of $Cs().B().A()$ 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 \( A_1 \) satisfied by \( D \), imposing to add \( A_1 \) to all of the above categories. Therefore the super categories are computed as the join of the above categories. Up to one twist: as is, the computation of this join would trigger recursively a recalculation of \( \text{Cs}(\cdot,A) \) ! To avoid this, \texttt{Category.join()} is given an optional argument to specify that the axiom \( A \) should not be applied to \( \text{Cs()} \).

**Sketch of proof of correctness and evaluation of complexity**

As we have seen, this is a highly recursive process! In particular, one needs to argue that, as long as the specifications are satisfied, the algorithm won’t run in an infinite recursion, in particular in case of deduction rule.

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

**Lemma**

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

**Proof**

Let \( D \) be a category involved in the join calculation for the super categories of \( C.A() \), and assume by induction that \( D \) is strictly smaller than \( C.A() \). A category \( E \) newly constructed from \( D \) can come from:

- \( D.(\text{extra}_\_\_\text{super_categories}()) \)
  - In this case, the specifications impose that \( E \) should be strictly smaller than \( D \) and therefore strictly smaller than \( C \).
- \( D.\text{with_axiom_as_tuple}('B') \) or \( D.B.\text{extra_super_categories}() \) for some axiom \( B \)
In this case, the axiom $B$ is satisfied by some subcategory of $\text{C.A()}$, and therefore must be satisfied by $\text{C.A()}$ itself. Since adding an axiom is a regressive construction, $E$ must be a subcategory of $\text{C.A()}$. If there is equality, then $E$ and $\text{C.A()}$ must have the same class, and therefore, $E$ must be directly constructed as $\text{C.A()}$. However the join construction explicitly prevents this call.

Note that a call to $D.\text{with_axiom_as_tuple('B')}$ does not trigger a direct join calculation; but of course, if $D$ implements $B$, the construction of the implemented category $E = D.\text{B()}$ will involve a strictly smaller join calculation.

### 1.3.6 Conclusion

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

### 1.3.7 Tests

**Note:** Quite a few categories with axioms are constructed early on during Sage’s startup. Therefore, when playing around with the implementation of the axiom infrastructure, it is easy to break Sage. The following sequence of tests is designed to test the infrastructure from the ground up even in a partially broken Sage. Please don’t remove the imports!

```python
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.seigroups 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
```

(continues on next page)
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 enumerated 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()

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.

class Commutative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class Connected(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class FiniteDimensional(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom
class Flying (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

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 blaqs

class SubcategoryMethods

Blue()
Commutative()
Connected()
FiniteDimensional()
Flying()
Unital()
class Unital (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom
class Blue (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

super_categories()
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'>
```

1.3. Axioms
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)

__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.

```python
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", \rightarrow "Blue"])
'flying blue semigroups'
sage: CategoryWithAxiom._repr_object\_names\_static(Algebras(QQ), ["Flying", \rightarrow "WithBasis", "Blue"])
'flying blue algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object\_names\_static(Algebras(QQ), ["WithBasis")
'algebras with basis over Rational Field'
sage: CategoryWithAxiom._repr_object\_names\_static(Sets().Finite().\rightarrow Subquotients(), ["Finite")
'subquotients of finite sets'
sage: CategoryWithAxiom._repr_object\_names\_static(Monoids(), ["Unital")
'monoids'
sage: CategoryWithAxiom._repr_object\_names\_static(Algebras(QQ['x']['y']), ["Flying", "WithBasis", "Blue")
'flying blue algebras with basis over Univariate Polynomial Ring in \_\rightarrow Univariate Polynomial Ring in \_ over Rational Field'
```

If the axioms is a set or frozen set, then they are first sorted using \_canonicalize\_axioms():

```
sage: CategoryWithAxiom._repr_object\_names\_static(Semigroups(), set(["Finite", \rightarrow "Commutative", "Facade"]))
'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()

```
\_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 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()
```

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: D = MagmasAndAdditiveMagmas().Distributive().Unital().axioms()
frozenset({'Distributive', 'Unital'})

1.3. Axioms

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

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:

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

Variants:

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

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

EXAMPLES:

```python
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

class sage.categories.category_with_axiom.CategoryWithAxiom_singleton(base_category):
    Bases:
        sage.categories.category_singleton.Category_singleton
        sage.categories.category_with_axiom.CategoryWithAxiom

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

class Facade(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class Finite(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class FiniteDimensional(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class FiniteDimensional(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class Finite(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class Unital(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class Commutative(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

class Unital(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom

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

A toy singleton category, for testing purposes.

See also:

Blahs

class Commutative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Facade(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Finite(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class FiniteDimensional(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class FiniteDimensional(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Finite(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Unital(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Commutative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class Unital(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

super_categories()

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")({})
    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")

(continues on next page)
sage: As().Finite()
(<>main__.As ... 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.magas.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
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'> to <class '__main__.FacadeSemigroups'> for axiom Facade?
sage: Semigroups.Facade = FacadeSemigroups
sage: base_category_class_and_axiom(FacadeSemigroups)
(<class 'sage.categories.semigroups.Semigroups'>, 'Facade')

Note: In the following example, we could possibly retrieve Sets from the class name. However this cannot be implemented robustly until 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_cat.Sets.Infinite'>.
...

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("JTrivialMonoids")
'j trivial monoids'
sage: sage.categories.category_with_axiom.uncamelcase("FiniteDimensionalAlgebras", "_")
'finite_dimensional_algebras'

1.4 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
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
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
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
sage: F = ForgetfulFunctor(FiniteFields(),Fields()) #indirect doctest
sage: F
The forgetful functor from Category of finite enumerated fields to Category of fields
sage: F(GF(3))
Finite Field of size 3
```

---

**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:

```python
sage: rings = Rings()
sage: abgrps = CommutativeAdditiveGroups()
sage: F = ForgetfulFunctor(rings, abgrps)
sage: F.domain()  # Category of rings
sage: F.codomain()  # Category of commutative additive groups
sage: from sage.categories.functor import 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:

```python
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.

```python
sage: R1.<x> = ZZ[]
sage: R2.<a,b> = QQ[]
sage: f = R1.hom([a+b],R2)
sage: f  # Ring morphism:
    From: Univariate Polynomial Ring in x over Integer Ring
    To:   Multivariate Polynomial Ring in a, b over Rational Field
    Defn: x |--> a + b
sage: F(f)  # Ring morphism:
    From: Fraction Field of Univariate Polynomial Ring in x over Integer Ring
    To:   Fraction Field of Multivariate Polynomial Ring in a, b over Rational Field
    Defn: x |--> a + b
sage: F(f)(1/x)
1/(a + b)
```
We can also apply a polynomial ring construction functor to our homomorphism. The result is a homomorphism that is defined on the base ring:

```
sage: F = QQ['t'].construction()[0]
sage: F
Poly[t]
sage: F(f)
Ring morphism:
  From: Univariate Polynomial Ring in t over Univariate Polynomial Ring in x over Integer Ring
  To:  Univariate Polynomial Ring in t over Multivariate Polynomial Ring in a, b over Rational Field
  Defn: Induced from base ring by
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:  Multivariate Polynomial Ring in a, b over Rational Field
      Defn: x |--> a + b
sage: p = R1['t']('(-x^2 + x)*t^2 + (x^2 - x)*t - 4*x^2 - x + 1')
sage: F(f)(p)
(-a^2 - 2*a*b - b^2 + a + b)*t^2 + (a^2 + 2*a*b + b^2 - a - b)*t - 4*a^2 - 8*a*b - 4*b^2 + a + b + 1
```

codomain() The codomain of self

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

domain() The domain of self

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

`sage.categories.functor.IdentityFunctor(C)` Construct the identity functor of the given category.

INPUT:
A category, C.

OUTPUT:
The identity functor in C.

```
sage: rings = Rings()
sage: F = IdentityFunctor(rings)
sage: F(ZZ['x','y'])
is 22['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 \texttt{IdentityFunctor()}.

EXAMPLES:

```python
sage: F = IdentityFunctor(Fields())  # indirect doctest
sage: F
The identity functor on Category of fields
sage: F(RR)  # indirect doctest
is RR
True
sage: F(ZZ)
Traceback (most recent call last):
...:
TypeError: x (=Integer Ring) is not in Category of fields
```

\texttt{sage.categories.functor.is\_Functor(x)}

Test whether the argument is a functor

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

EXAMPLES:

```python
sage: from sage.categories.functor import is\_Functor
sage: F1 = QQ.construction()[0]
True
sage: is\_Functor(FractionField)
False
sage: F2 = ForgetfulFunctor(Fields(), Rings())
True
sage: is\_Functor(F2)
```

### 1.5 Implementing a new parent: a (draft of) tutorial

The easiest approach for implementing a new parent is to start from a close example in \texttt{sage.categories.examples}. Here, we will get through the process of implementing a new finite semigroup, taking as starting point the provided example:

```python
sage: S = FiniteSemigroups().example()
```

You may lookup the implementation of this example with:

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

Or by browsing the source code of \texttt{sage.categories.examples.finite\_semigroups.LeftRegularBand}.
Copy-paste this code into, say, a cell of the notebook, and replace every occurrence of `FiniteSemigroups().example(...)` in the documentation by `LeftRegularBand`. This will be equivalent to:

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

Now, try:

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

and play around with the examples in the documentation of `S` and of `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.
2.1 Base class for maps

AUTHORS:

• Robert Bradshaw: initial implementation
• Sebastien Besnier (2014-05-5): FormalCompositeMap contains a list of Map instead of only two Map. See trac ticket #16291.

```python
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.
```

EXAMPLES:

```python
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
```

(continues on next page)
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())
[Integral Ring, Rational Field]

first()
Return the first map in the formal composition.

If self represents $f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0$, then self.first() returns $f_0$. We have self == self.then() * self.first().

EXAMPLES:
sage: R.<x> = QQ[]
sage: S.<a> = QQ[]
sage: from sage.categories.morphism import SetMorphism
dsage: 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.

EXAMPLES:
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:

```python
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:

```python
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:

```python
sage: psi2 = V1.hom(Matrix([[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.

**EXAMPLES:**

```python
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:

```python
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:

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

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

```python
sage: FormalCompositeMap(Hom(V2, V1, phi32.category_for()), V2.
  ←hom(Matrix([[0], [0]]), V1), V1.hom(Matrix([[1]])).is_surjective()
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.

then()
Return the tail of the list of maps.

If self represents \( f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0 \), then self.first() returns \( f_n \circ f_{n-1} \circ \cdots \circ f_1 \). We have self == self.then() * self.first().

EXAMPLES:

```python
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:

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
sage: Y = ZZ
sage: phi = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])
sage: phi(x^2+2*x-1)
-1
```

category_for()
Returns the category self is a morphism for.

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

**EXAMPLES:**

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = 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: f = R.hom([x+y, x-y], R)
sage: f.category_for()
Join of Category of unique factorization domains
and Category of commutative algebras
over (number fields and quotient fields and metric spaces)
and Category of infinite sets
sage: f.category()
Category of endsets of unital magmas
and right modules over (number fields and quotient fields and metric spaces)
and left modules over (number fields and quotient fields and metric spaces)

FIXME: find a better name for this method

codomain
domain
domains()
Iterate over the domains of the factors of a (composite) map.
This default implementation simply yields the domain of this map.

See also:

FormalCompositeMap.domains()

EXAMPLES:

sage: list(QQ.coerce_map_from(ZZ).domains())
[Integer Ring]

extend_codomain(new_codomain)
INPUT:

• self – a member of Hom(X, Y)

• new_codomain – an object Z such that there is a canonical coercion \( \phi \) in \( \text{Hom}(Y, Z) \)

OUTPUT:

An element of \( \text{Hom}(X, Z) \) obtained by composing self with \( \phi \). If no canonical \( \phi \) exists, a TypeError is raised.

EXAMPLES:

sage: mor = QQ.coerce_map_from(ZZ)
sage: mor.extend_codomain(RDF)
Composite map:
  From: Integer Ring
  To:    Real Double Field
Defn: Natural morphism:
          From: Integer Ring
          To:    Rational Field
          then
Native morphism:
From: Rational Field
To: Real Double Field

```sage```
mor.extend_codomain(GF(7))
```
Traceback (most recent call last):
...
```TypeError```
No coercion from Rational Field to Finite Field of size 7

**extend_domain** *(new_domain)*

**INPUT:**

- self – a member of Hom(Y, Z)
- new_codomain – an object X such that there is a canonical coercion $\phi$ in Hom(X, Y)

**OUTPUT:**

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

**EXAMPLES:**

```sage```
mor = CDF.coerce_map_from(RDF)
mor.extend_domain(QQ)
Composite map:
  From: Rational Field
  To: Complex Double Field
  Defn: Native morphism:
    From: Rational Field
    To: Real Double Field
    then
    Native morphism:
    From: Real Double Field
    To: Complex Double Field
```

```sage```
mor.extend_domain(ZZ['x'])
```
Traceback (most recent call last):
...
```TypeError```
No coercion from Univariate Polynomial Ring in x over Integer Ring→to Real Double Field

**is_surjective** ()

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

**parent** ()

Return the homset containing this map.

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

**EXAMPLES:**

```sage```
Q = QuadraticField(-5)
```sage```
phi = CDF._internal_convert_map_from(Q)
We now demonstrate that the reference to the coercion map $\phi$ does not prevent $Q$ from being garbage collected:

```python
sage: import gc
sage: del Q
sage: _ = gc.collect()
sage: phi.parent()
Traceback (most recent call last):
  ... ValueErro: This map is in an invalid state, the domain has been garbage
collected
```

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

```python
sage: Q = QuadraticField(-5)
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 Double Field
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 Double Field
```

**post_compose** *(left)*

**INPUT:**

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

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

**Caveat:** see the current restrictions on $\text{Category.meet()}$

**EXAMPLES:**

```python
sage: from sage.categories.morphism import SetMorphism
sage: X.<x> = ZZ[]
sage: Y = ZZ
sage: Z = QQ
sage: phi_xy = SetMorphism(Hom(X, Y, Rings()), lambda p: p[0])
sage: phi_yz = SetMorphism(Hom(Y, Z, Monoids()), lambda y: QQ(y**2))
sage: phi_xz = phi_xy.post_compose(phi_yz); phi_xz
Composite map:
  From: Univariate Polynomial Ring in x over Integer Ring
  To: Rational Field
  Defn: Generic morphism:
    From: Univariate Polynomial Ring in x over Integer Ring
    To: Integer Ring
    then
```
Generic morphism:
From: Integer Ring
To: Rational Field
\texttt{sage}: \texttt{phi\_xz.category\_for()}
Category of monoids

\texttt{pre\_compose(right)}

\texttt{INPUT:}
\begin{itemize}
  \item \texttt{self} – a Map in some Hom(Y, Z, category\_left)
  \item \texttt{left} – a Map in some Hom(X, Y, category\_right)
\end{itemize}

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.

\texttt{EXAMPLES:}
\begin{verbatim}
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
\texttt{sage}: \texttt{phi\_xz.category\_for()}
Category of monoids
\end{verbatim}

\texttt{section()}

Return a section of self.

\texttt{NOTE:}

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

\texttt{class sage.categories.map.Section}

\texttt{Bases: sage.categories.map.Map}

A formal section of a map.

\texttt{NOTE:}

Call methods are not implemented for the base class Section.

\texttt{EXAMPLES:}
\begin{verbatim}
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])
\end{verbatim}
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'>

inverse()
   Return inverse of self.

sage.categories.map.is_Map(x)
   Auxiliary function: Is the argument a map?

   EXAMPLES:

sage: R.<x,y> = QQ[]
sage: f = R.hom([x+y, x-y], R)
sage: from sage.categories.map import is_Map
sage: is_Map(f)
True

sage.categories.map.unpickle_map(_class, parent, _dict, _slots)
   Auxiliary function for unpickling a map.

2.2 Homsets

The class \texttt{Hom} is the base class used to represent sets of morphisms between objects of a given category. \texttt{Hom} objects are usually "weakly" cached upon creation so that they don't have to be generated over and over but can be garbage collected together with the corresponding objects when these are not strongly ref'ed anymore.

   EXAMPLES:

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

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)
1
AUTHORS:

- David Kohel and William Stein
- David Joyner (2005-12-17): added examples
- Nicolas M. Thiery (2008-12-): Updated for the new category framework
- Simon King (2011-12): Use a weak cache for homsets
- Simon King (2013-02): added examples

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

**INPUT:**

- $X$ – anything
- 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 group of order 3!/2 as a permutation group in Category of finite enumerated permutation groups
```

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 finite groups
\texttt{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:

\texttt{sage: H = Hom(ZZ,ZZ,Rings())}
\texttt{sage: H.category()}
Category of endsets of unital magmas and additive unital additive magmas

\texttt{sage.categories.homset.Hom(X, Y, category=None, check=True)}
Create the space of homomorphisms from \(X\) to \(Y\) in the category \(\text{category}\).

\textbf{INPUT:}

- \(X\) – an object of a category
- \(Y\) – an object of a category
- \(\text{category}\) – 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.
- \(\text{check}\) – a boolean (default: True): whether to check the input, and in particular that \(X\) and \(Y\) belong to \(\text{category}\).

\textbf{OUTPUT:} a homset in \(\text{category}\)

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: V = VectorSpace(QQ,3)}
\texttt{sage: H = 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
\texttt{sage: G = AlternatingGroup(3)}
\texttt{sage: H = Hom(G, G)}
Set of Morphisms from Alternating group of order 3!/2 as a permutation group to Alternating group of order 3!/2 as a permutation group in Category of finite enumerated permutation groups
\texttt{sage: H = Hom(ZZ, QQ, Sets())}
Set of Morphisms from Integer Ring to Rational Field in Category of sets
\texttt{sage: H = Hom(FreeModule(ZZ,1), FreeModule(QQ,1))}
Set of Morphisms from Ambient free module of rank 1 over the principal ideal domain Integer Ring to Vector space of dimension 1 over Rational Field in Category of commutative additive groups
\texttt{sage: H = Hom(FreeModule(QQ,1), FreeModule(ZZ,1))}
Set of Morphisms from Vector space of dimension 1 over Rational Field to Ambient free module of rank 1 over the principal ideal domain Integer Ring in Category of commutative additive groups
\end{verbatim}

Here, we test against a memory leak that has been fixed at \texttt{trac ticket #11521} by using a weak cache:

\begin{verbatim}
\texttt{sage: for p in prime_range(10^3):}
...:     K = GF(p)
...:     a = K(0)
\texttt{sage: import gc}
\end{verbatim}
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 Category of enumerated monoids
```

Otherwise, if category is specified, then category is used, after checking that X and Y are indeed in category:

```
sage: Hom(X, Y, Magmas())
Set of Morphisms from Integer Ring to Symmetric group of order 3! as a permutation group in Category of magmas

sage: Hom(X, Y, Groups())
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
```

Todo:

- Design decision: how much of the homset comes from the category of X and Y, and how much from the specific X and Y. In particular, do we need several parent classes depending on X and Y, or does the difference only lie in the elements (i.e. the morphism), and of course how the parent calls their constructors.
• Specify the protocol for the _Hom_ hook in case of ambiguity (e.g. if both a parent and some category thereof provide one).

```python
class sage.categories.homset.Homset(X, Y, category=None, base=None, check=True):
    Bases: 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()) == 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'))) == AffineSpace(2, names='x,y')
True
sage: loads(dumps(H)) == H
True

Conversely, homsets of non-unique parents are non-unique:
```
sage: H = End(ProductProjectiveSpaces(QQ, [1, 1]))
sage: loads(dumps(ProductProjectiveSpaces(QQ, [1, 1]))) == ProductProjectiveSpaces(QQ, [1, 1])
False
sage: loads(dumps(H)) == 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() == f.parent().codomain()
True
```

domain()
Return the domain of this homset.

EXAMPLES:
```
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().domain()
Univariate Polynomial Ring in t over Integer Ring
sage: f.domain() == 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'>
```

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 enumerated permutation groups
```

identity()

The identity map of this homset.

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

EXAMPLES:

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

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()
```

(continues on next page)
Coercion morphism:
From: Univariate Polynomial Ring in t over Integer Ring
To: Univariate Polynomial Ring in t over Rational Field

```sage
H = Hom(QQ['t'],GF(3)['t'])
```

```
sage: H.natural_map()
Traceback (most recent call last):
  ...  
TypeError: natural coercion morphism from Univariate Polynomial Ring in t over Rational Field to Univariate Polynomial Ring in t over Finite Field of size 3 not defined
```

**one()**
The identity map of this homset.

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

**EXAMPLES:**

```sage
K = GaussianIntegers()
sage: End(K).one()
```

```
Identity endomorphism of Gaussian Integers in Number Field in I with defining polynomial x^2 + 1
```

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

```python
class sage.categories.homset.HomsetWithBase(X, Y, category=None, check=True, base=None)
Bases: sage.categories.homset.Homset
doce
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 \( \text{Hom}(X, Y)(f) \), where \( f \) is data that defines an element of \( \text{Hom}(X, Y) \).

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

2.3 Morphisms

AUTHORS:

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

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

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

    EXAMPLES:

    sage: E = End(Partitions(5))
    sage: E.identity().is_identity()
    True

    Check that trac ticket #15478 is fixed:

    sage: K.<z> = GF(4)
    sage: phi = End(K)([z^2])
    sage: R.<t> = K[]
    sage: psi = End(R)(phi)
    sage: psi.is_identity()
    False

    is_injective()
    Return whether this morphism is injective.

    EXAMPLES:

    sage: Hom(ZZ, ZZ).identity().is_injective()
    True

    is_surjective()
    Return whether this morphism is surjective.

    EXAMPLES:

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

    section()
    Return a section of this morphism.

    EXAMPLES:

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

class sage.categories.morphism.Morphism
    Bases: 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 enumerated sets and metric spaces)
and left modules over (euclidean domains and infinite enumerated sets and metric spaces)

\[
\begin{align*}
\text{sage: } & K = \text{CyclotomicField}(12) \\
\text{sage: } & L = \text{CyclotomicField}(132) \\
\text{sage: } & \phi = L._\text{internal_coerce_map_from}(K) \\
\text{sage: } & \phi.category() \\
\end{align*}
\]

\text{Category of homsets of number fields}

\text{is_endomorphism}()

Return True if this morphism is an endomorphism.

\text{EXAMPLES:}

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Z}[] \\
\text{sage: } & f = R.\text{hom}([t]) \\
\text{sage: } & f.\text{is_endomorphism()} \\
\text{True} \\
\text{sage: } & K = \text{CyclotomicField}(12) \\
\text{sage: } & L = \text{CyclotomicField}(132) \\
\text{sage: } & \phi = L._\text{internal_coerce_map_from}(K) \\
\text{sage: } & \phi.\text{is_endomorphism()} \\
\text{False} \\
\end{align*}
\]

\text{is_identity}()

Return True if this morphism is the identity morphism.

\text{Note: } Implemented only when the domain has a method gens()

\text{EXAMPLES:}

\[
\begin{align*}
\text{sage: } & R.<t> = \mathbb{Z}[] \\
\text{sage: } & f = R.\text{hom}([t]) \\
\text{sage: } & f.\text{is_identity()} \\
\text{True} \\
\text{sage: } & g = R.\text{hom}([t+1]) \\
\text{sage: } & g.\text{is_identity()} \\
\text{False} \\
\end{align*}
\]

A morphism between two different spaces cannot be the identity:

\[
\begin{align*}
\text{sage: } & R2.<t2> = \mathbb{Q}[] \\
\text{sage: } & h = R.\text{hom}([t2]) \\
\text{sage: } & h.\text{is_identity()} \\
\text{False} \\
\end{align*}
\]

\text{pushforward}()

\text{register_as_coercion}()

Register this morphism as a coercion to Sage’s coercion model (see \text{sage.structure.coerce}).

\text{EXAMPLES:}

By default, adding polynomials over different variables triggers an error:
Let us declare a coercion from \( \mathbb{Z}[x] \) to \( \mathbb{Z}[z] \):

```python
sage: Z.<z> = ZZ[]
sage: phi = Hom(X, Z)(z)
sage: phi(x^2+1)
z^2 + 1
```

Now we can add elements from \( \mathbb{Z}[x] \) and \( \mathbb{Z}[z] \), because the elements of the former are allowed to be implicitly coerced into the later:

```python
sage: x^2 + z
z^2 + z
```

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

```python
sage: phi = Hom(X, Y)(y)
sage: phi.register_as_coercion()
```

```python
Traceback (most recent call last):
  ...  
AssertionError: coercion from Univariate Polynomial Ring in x over Integer Ring to Univariate Polynomial Ring in y over Integer Ring already registered or discovered
```

```
register_as_conversion()
Register this morphism as a conversion to Sage’s coercion model

(see sage.structure.coerce).

EXAMPLES:
Let us declare a conversion from the symmetric group to \( \mathbb{Z} \) through the sign map:

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

```python
sage: phi(x)
1
```

```python
sage: phi.register_as_conversion()
```

```python
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)
```

## 2.4 Coercion via Construction Functors

**class** `sage.categories.pushout.AlgebraicClosureFunctor`

Bases: `sage.categories.pushout.ConstructionFunctor`

Algebraic Closure.

**EXAMPLES:**

```
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 \( \mathbb{R} \). Therefore, we do not merge with algebraic extension.

**class** `sage.categories.pushout.AlgebraicExtensionFunctor`

\( (\text{polys}, \text{names}, \text{embeddings=}, \text{structures=}, \text{cyclotomic=}, \text{precs=}, \text{implementations=}, **\text{kwds}) \)

Bases: `sage.categories.pushout.ConstructionFunctor`

Algebraic extension (univariate polynomial ring modulo principal ideal).

**EXAMPLES:**

```
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
    \( \overrightarrow{\text{over Integer Ring with modulus a}^3 + a^2 + 1} \)
```

Note that, even if a field is algebraically closed, the algebraic extension will be constructed as the quotient of a univariate polynomial ring:
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(...):

\[
\text{sage: } F = \text{F(ZZ)}
\]
Order in Number Field in a with defining polynomial \( x^3 + x^2 + 1 \)

This also holds for non-absolute number fields:

\[
\text{sage: } K.<a,b> = \text{NumberField([x^3+x^2+1,x^2+x+1])}
\]
\[
\text{sage: } F = K.\text{construction()[0]}
\]
\[
\text{sage: } O = F(ZZ); O
\]
Relative Order in Number Field in a with defining polynomial \( x^3 + x^2 + 1 \) over its base field
\[
\text{sage: } O.\text{ambient()} \text{ is } K
\]
True

**expand()**

Decompose the functor \( F \) into sub-functors, whose product returns \( F \).

**EXAMPLES:**

\[
\text{sage: } P.<x> = \text{QQ[]}
\]
\[
\text{sage: } K.<a> = \text{NumberField(x^3-5,embedding=0)}
\]
\[
\text{sage: } L.<b> = K.\text{extension(x^2+a)}
\]
\[
\text{sage: } F, R = L.\text{construction()}
\]
\[
\text{sage: } \text{prod(F.expand())(R)} == L
\]
True
\[
\text{sage: } K = \text{NumberField([x^2-2, x^2-3],'a')}
\]
\[
\text{sage: } F, R = K.\text{construction()}
\]
\[
\text{sage: } F
\]
AlgebraicExtensionFunctor
\[
\text{sage: } L = F.\text{expand()}; L
\]
[AlgebraicExtensionFunctor, AlgebraicExtensionFunctor]
\[
\text{sage: } L[-1](QQ)
\]
Number Field in a1 with defining polynomial \( x^2 - 3 \)

**merge(other)**

Merging with another \( \text{AlgebraicExtensionFunctor} \).

**INPUT:**

other – Construction Functor.

**OUTPUT:**

- If \( \text{self==other} \), \( \text{self} \) is returned.
- If \( \text{self and other} \) 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, prefix='z'); a = k.gen()
sage: l = GF(3^3, 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, 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 c1+c2 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 + 1
```

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^3 - 2, Number Field in b with defining polynomial x^6 - 2)
(continues on next page)
class sage.categories.pushout.BlackBoxConstructionFunctor(box)
Bases: sage.categories.pushout.ConstructionFunctor
Construction functor obtained from any callable object.

EXAMPLES:

```python
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'])
polynomial ring, over a field, global ordering
    // coefficients: QQ
    // number of vars : 1
    //   block 1 : ordering lp
    //       : names t
    //   block 2 : ordering C
sage: FG == FS
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:

```python
sage: R = Zp(5)
sage: R
5-adic Ring with capped relative precision 20
sage: F1 = R.construction()[0]
sage: F1
Completion[5, prec=20]
sage: F1(ZZ) is R
True
sage: F1(QQ)
5-adic Field with capped relative precision 20
sage: F2 = RR.construction()[0]
sage: F2
Completion[+Infinity, prec=53]
sage: F2(QQ) is RR
True
sage: P.<x> = ZZ[]
sage: Px = P.completion(x) # currently the only implemented completion of P
sage: Px
Power Series Ring in x over Integer Ring
sage: F3 = Px.construction()[0]
sage: F3(GF(3)['x'])
Power Series Ring in x over Finite Field of size 3
```

2.4. Coercion via Construction Functors
**commutes** *(other)*  
Completion commutes with fraction fields.

**EXAMPLES:**
```
sage: F1 = Zp(5).construction()[0]  
sage: F2 = QQ.construction()[0]  
sage: F1.commutes(F2)  
True
```

**merge** *(other)*
Two Completion functors are merged, if they are equal. If the precisions of both functors coincide, then a Completion functor is returned that results from updating the extras dictionary of self by other. extras. Otherwise, if the completion is at infinity then merging does not increase the set precision, and if the completion is at a finite prime, merging does not decrease the capped precision.

**EXAMPLES:**
```
sage: R1.<a> = Zp(5,prec=20)[]  
sage: R2 = Qp(5,prec=40)  
sage: R2(1)+a  # indirect doctest  
(1 + 0(5^20))*a + 1 + 0(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
```

**class** sage.categories.pushout.CompositeConstructionFunctor(*args)
```
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],ZZ['y'].construction()[0])
```

(continues on next page)
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 Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)

```
expand()
```

Return expansion of a CompositeConstructionFunctor.

**NOTE:**
The product over the list of components, as returned by the `expand()` method, is equal to `self`.

**EXAMPLES:**

```
sage: from sage.categories.pushout import CompositeConstructionFunctor
sage: F = CompositeConstructionFunctor(QQ.construction()[0],ZZ['x'].construction()[0],QQ.construction()[0],ZZ['y'].construction()[0])
sage: F
Poly[y](FractionField(Poly[x](FractionField(...))))
sage: prod(F.expand()) == F
True
```

2.4. Coercion via Construction Functors
When composing two construction functors, they are sometimes merged into one, as is the case in the Quotient construction:

```python
sage: Q15, R = (ZZ.quo(15*ZZ)).construction()
```

```python
sage: Q15
```

QuotientFunctor

```python
sage: Q35, R = (ZZ.quo(35*ZZ)).construction()
```

```python
sage: Q35
```

QuotientFunctor

```python
sage: Q15.merge(Q35)
```

```python
QuotientFunctor
```

```python
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:

```python
sage: P.<x,y> = ZZ[]
```

```python
sage: F = P.construction()[0]; F
```

```python
MPoly[x,y]
```

```python
sage: A.<a,b> = GF(5)[]
```

```python
sage: f = A.hom([a+b,a-b],A)
```

```python
sage: F(A)
```

Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b, over Finite Field of size 5

```python
sage: F(f)
```

Ring endomorphism of Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b, over Finite Field of size 5

```python
Defn: Induced from base ring by
```

```python
Ring endomorphism of Multivariate Polynomial Ring in a, b, over Finite Field of size 5
```

```python
Defn: a |--> a + b
```

```python
b |--> a - b
```

```python
sage: F(f)(F(A)(x)*a)
```

(a + b)*x

```python
common_base(other_functor, self_bases, other_bases)
```

This function is called by `pushout()` when no common parent is found in the construction tower.

**Note:** The main use is for multivariate construction functors, which use this function to implement recursion for `pushout()`.

**INPUT:**

- `other_functor` – a construction functor.
- `self_bases` – the arguments passed to this functor.
- `other_bases` – the arguments passed to the functor `other_functor`.  

(continued from previous page)
OUTPUT:
Nothing, since a `CoercionException` is raised.

\textbf{Note:} Overload this function in derived class, see e.g. `MultivariateConstructionFunctor`.

\texttt{commutes}(\texttt{other})
Determine whether \texttt{self} commutes with another construction functor.

\textbf{NOTE:}
By default, \texttt{False} is returned in all cases (even if the two functors are the same, since in this case \texttt{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.

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\texttt{expand}()
Decompose \texttt{self} into a list of construction functors.

\textbf{NOTE:}
The default is to return the list only containing \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
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]]
\end{verbatim}

\texttt{merge}(\texttt{other})
Merge \texttt{self} with another construction functor, or return None.

\textbf{Note:} The default is to merge only if the two functors coincide. But this may be overloaded for subclasses, such as the quotient functor.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.merge(F)
\end{verbatim}

(continues on next page)
FractionField

\[ \text{sage}: \text{F.merge(P)} \]
\[ \text{sage}: \text{P.merge(F)} \]
\[ \text{sage}: \text{P.merge(P)} \]
\[ \text{Poly}[t] \]

\textit{pushout} (other)
Composition of two construction functors, ordered by their ranks.

\textbf{Note:}
- This method seems not to be used in the coercion model.
- By default, the functor with smaller rank is applied first.

\textbf{class} \ sage.categories.pushout.FractionField

Bases: \ sage.categories.pushout.ConstructionFunctor

Construction functor for fraction fields.

\textbf{EXAMPLES:}

\[ \text{sage}: \text{F} = \text{QQ.construction()[0]} \]
\[ \text{sage}: \text{F} \]
\[ \text{FractionField} \]
\[ \text{sage}: \text{F.domain()} \]
\[ \text{Category of integral domains} \]
\[ \text{sage}: \text{F.codomain()} \]
\[ \text{Category of fields} \]
\[ \text{sage}: \text{F(GF(5)) is GF(5)} \]
\[ \text{True} \]
\[ \text{sage}: \text{F(ZZ['t'])} \]
\[ \text{Fraction Field of Univariate Polynomial Ring in t over Integer Ring} \]
\[ \text{sage}: \text{P.<x,y> = QQ[]} \]
\[ \text{sage}: \text{f = P.hom([x+2*y,3*x-y],P)} \]
\[ \text{sage}: \text{F(f)} \]
\[ \text{Ring endomorphism of Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field} \]
\[ \text{Defn: x |--> x + 2*y} \]
\[ \text{y |--> 3*x - y} \]
\[ \text{sage}: \text{F(f)(1/x)} \]
\[ \frac{1}{x + 2*y} \]
\[ \text{sage}: \text{F == loads(dumps(F))} \]
\[ \text{True} \]

\textbf{class} \ sage.categories.pushout.IdentityConstructionFunctor

Bases: \ sage.categories.pushout.ConstructionFunctor

A construction functor that is the identity functor.

\textbf{class} \ sage.categories.pushout.InfinitePolynomialFunctor\(\text{(gens, order, implementation)}\)

Bases: \ sage.categories.pushout.ConstructionFunctor

A Construction Functor for Infinite Polynomial Rings (see \textit{infinite_polynomial_ring}).

\textbf{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-monoïd of the monomials of the ambient Infinite Polynomial Ring, then they are removed from the base ring and merged with the generators of the ambient ring. However, if the orders don’t match, an error is raised, since there was a name conflict without merging.

**EXAMPLES:**

```python
sage: A.<a,b> = InfinitePolynomialRing(ZZ['t'])
sage: A.construction()
[InfPoly{[a,b], "lex", "dense"},
 Univariate Polynomial Ring in t over Integer Ring]
sage: type(_[0])
<class 'sage.categories.pushout.InfinitePolynomialFunctor'>
sage: B.<x,y,a_3,a_1> = PolynomialRing(QQ, order='lex')
sage: B.construction()
(MPoly[x,y,a_3,a_1], Rational Field)
sage: A.construction()[0]*B.construction()[0]
InfPoly{[a,b], "lex", "dense"}(MPoly[x,y](...))
```

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:

```python
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:

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

Another requirement is that after merging the order of the remaining variables must be unique. This is not the case in the following example, since it is not clear whether the variables $x, y$ should be greater or smaller than the variables $b_*$:

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

Since the construction functors are actually used to construct infinite polynomial rings, the following result is no surprise:
There is also an overlap in the next example:

```python
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:

```python
sage: P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
w_2 + z_3
sage: _.parent() is P
True
```

**expand()**

Decompose the functor \(F\) into sub-functors, whose product returns \(F\).

**EXAMPLES:**

```python
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:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ,order='degrevlex')
sage: X.construction()
[InfPoly{[x,y], "lex", "sparse"}, Rational Field]
sage: Y.construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
sage: Y.construction()[0].merge(Y.construction()[0])
InfPoly{[x,y], "degrevlex", "dense"}
sage: y[3] + X(x[2])
```

(continues on next page)
class sage.categories.pushout.LaurentPolynomialFunctor(var, multi_variate=False)

Bases: sage.categories.pushout.ConstructionFunctor

Construction functor for Laurent polynomial rings.

EXAMPLES:

```python
sage: L.<t> = LaurentPolynomialRing(ZZ)
sage: F = L.construction()[0]
sage: F
LaurentPolynomialFunctor
sage: F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
sage: K.<x> = LaurentPolynomialRing(ZZ)
sage: F(K)
Univariate Laurent Polynomial Ring in t over Univariate Laurent Polynomial Ring
  → in x over Integer Ring
sage: P.<x,y> = ZZ[]
sage: f = P.hom([x+2*y,3*x-y],P)
sage: F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t over Multivariate
  → Polynomial Ring in x, y over Integer Ring
  Defn: Induced from base ring by
  Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer
    → 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 multi-variate if one of the arguments is multivariate.

EXAMPLES:

```python
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(F2)(LaurentPolynomialRing(GF(2),'a'))
Univariate Laurent Polynomial Ring in t over Univariate Laurent Polynomial
  → Ring in a over Finite Field of size 2
```

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

Bases: sage.categories.pushout.ConstructionFunctor

A construction functor for matrices over rings.

EXAMPLES:
```python
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 Rational Field
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 in x, y over Rational Field
  Defn: Induced from base ring by
    Ring endomorphism of Multivariate Polynomial Ring in x, y over Rational Field
      Defn: x |--> x + y
              y |--> x - y
sage: M = R([x,y,x+y,x+y])
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.

**EXAMPLES:**

```python
sage: F1 = MatrixSpace(ZZ,2,2).construction()[0]
sage: F2 = MatrixSpace(ZZ,2,3).construction()[0]
sage: F3 = MatrixSpace(ZZ,2,2,sparse=True).construction()[0]
sage: F1.merge(F2)
MatrixFunctor
sage: F1.merge(F3)
MatrixFunctor
sage: F13 = F1.merge(F3)
sage: F13.is_sparse False
sage: F1.is_sparse False
sage: F3.is_sparse True
sage: F3.merge(F3).is_sparse True
```

```python
class sage.categories.pushout.MultiPolynomialFunctor(vars, term_order)
Bases: sage.categories.pushout.ConstructionFunctor

A constructor for multivariate polynomial rings.

**EXAMPLES:**

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

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 over Integer Ring
sage: T.<y,s> = QQ[]
sage: x + s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Multivariate Polynomial Ring in x, y, z over Integer Ring' and 'Multivariate Polynomial Ring in y, s over Rational Field'
sage: R = PolynomialRing(ZZ, 'x', 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.MultivariateConstructionFunctor
Bases: sage.categories.pushout.ConstructionFunctor

An abstract base class for functors that take multiple inputs (e.g. Cartesian products).

2.4. Coercion via Construction Functors
common_base(other_functor, self_bases, other_bases)

This function is called by pushout() when no common parent is found in the construction tower.

INPUT:
- other_functor – a construction functor.
- self_bases – the arguments passed to this functor.
- other_bases – the arguments passed to the functor other_functor.

OUTPUT:
A parent.

If no common base is found a sage.structure.coerce_exceptions.CoercionException is raised.

Note: Overload this function in derived class, see e.g. MultivariateConstructionFunctor.

class sage.categories.pushout.PermutationGroupFunctor(gens, domain)
Bases: 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.

EXAMPLES:

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)[]

```python
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:

```python
sage: R.<x> = PolynomialRing(GF(5), sparse=True)
```

```python
sage: F,B = R.construction()
```

```python
sage: F(B)
```

```
F is R
True
```

```python
sage: S.<x> = PolynomialRing(ZZ)
```

```python
sage: R.has_coerce_map_from(S)
```

```
False
```

```python
sage: S.has_coerce_map_from(R)
```

```
False
```

```python
sage: S.0 + R.0
```

```
2*x
```

```python
sage: (S.0 + R.0).parent()
```

```
Univariate Polynomial Ring in x over Finite Field of size 5
```

```python
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.

EXAMPLES:

```python
sage: P = ZZ['x'].construction()[0]
```

```python
sage: Q = ZZ['y','x'].construction()[0]
```

```python
sage: P.merge(Q)
```

```
F is P
```

```python
sage: P.merge(P)
```

```
F is P
```

```python
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.

EXAMPLES:

```python
sage: P.<x,y> = ZZ[]
```

```python
sage: Q = P.quo([x^2+y^2]*P)
```

```python
sage: F = Q.construction()[0]
```

```python
sage: F(QQ['x','y'])
```

```
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
```

(continues on next page)
merge (other)

Two quotient functors with coinciding names are merged by taking the gcd of their moduli.

EXAMPLES:

```python
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:

```python
sage: pushout(GF(5), Integers(5))
Finite Field of size 5
```

class sage.categories.pushout.SubspaceFunctor (basis)

Constructing a subspace of an ambient free module, given by a basis.

NOTE:

This construction functor keeps track of the basis. It can only be applied to free modules into which this basis coerces.

EXAMPLES:

```python
sage: M = ZZ^3
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.

EXAMPLES:
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 size 5
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.

EXAMPLES:

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 over Finite Field of size 2 (using GF2X)

merge(other)

Two constructors of free modules merge, if the module ranks and the inner products coincide. If both have explicitly given inner product matrices, they must coincide as well.

EXAMPLES:

Two modules without explicitly given inner product allow coercion:

sage: M1 = QQ^3
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

2.4. Coercion via Construction Functors
inner product was *explicitly* requested for $M_4$. It is therefore not possible to coerce with a different inner product:

```python
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 over Rational Field
Inner product matrix:
[1 0 0]
[0 1 0]
[0 0 1]' and 'Ambient free quadratic module of rank 3 over the integral domain Univariate Polynomial Ring in t over Integer Ring
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

`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.

**EXAMPLES:**

```python
sage: from sage.categories.pushout import construction_tower
sage: construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field), (Poly[x, y, z], Rational Field)]
```

`sage.categories.pushout.expand_tower(tower)`
An auxiliary function that is used in `pushout()`.

**INPUT:**

A construction tower as returned by `construction_tower()`.

**OUTPUT:**

A new construction tower with all the construction functors expanded.

**EXAMPLES:**

```python
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),]
```

(continues on next page)
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 \(R = \text{Complete}_7(\text{Frac}(\mathbb{Z}))\) and \(\text{Frac}(\text{Poly}_x(\mathbb{Z}))\), which give us \(\text{Frac}(\text{Poly}_x(\text{Complete}_7(\text{Frac}(\mathbb{Z}))))\):

```
sage: 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 precision 20
```

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 precision 20
```
Note that polynomial variable ordering must be unambiguously determined.

```
sage: pushout(Zp(7)['x'], Frac(QQ['x']))
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped relative precision 20
```

Some other examples:

```
sage: pushout(ZZ['x,y,z'], QQ['w,z,t'])
Traceback (most recent call last):
... CoercionException: ('Ambiguous Base Extension', Multivariate Polynomial Ring in x, y, z over Integer Ring, Multivariate Polynomial Ring in w, z, t over Rational Field)
sage: pushout(ZZ['x,y,z'], QQ['w,x,z,t'])
Multivariate Polynomial Ring in w, x, y, z, t 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.has_coerce_map_from(A)
True
```

Some more tests with `coercion_reversed = True`:
```python
sage: from sage.categories.pushout import ConstructionFunctor
sage: class EvenPolynomialRing(type(QQ['x'])):
    ...:     def __init__(self, base, var):
    ...:         super(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 _apply_functor(self, R):
    ...:         return EvenPolynomialRing(R.base(), R.variable_name())

sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR)
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 Ring in x over Real Field with 53 bits of precision
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 over Real Field with 53 bits of precision
```

Some more tests related to univariate/multivariate constructions. We consider a generalization of polynomial rings, where in addition to the coefficient ring $C$ we also specify an additive monoid $E$ for the exponents of the indeterminate. In particular, the elements of such a parent are given by

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

with $c_i \in C$ and $e_i \in E$. We define

```python
sage: class GPolynomialRing(Parent):
    ...:     def __init__(self, coefficients, var, exponents):
```

(continues on next page)

2.4. Coercion via Construction Functors 141
self.coefficients = coefficients
self.var = var
self.exponents = exponents
super(GPolynomialRing, self).__init__(category=Rings())
def _repr_(self):
    return 'Generalized Polynomial Ring in %s^(%s) over %s' % (self.var, self.exponents, self.coefficients)
def construction(self):
    return GPolynomialFunctor(self.var, self.exponents), self.coefficients
def _coerce_map_from_(self, R):
    return self.coefficients.has_coerce_map_from(R)

and

class GPolynomialFunctor(ConstructionFunctor):
    rank = 10
def __init__(self, var, exponents):
    self.var = var
    self.exponents = exponents
    ConstructionFunctor.__init__(self, Rings(), Rings())
def _repr_(self):
    return 'GPoly[%s^(%s)]' % (self.var, self.exponents)
def _apply_functor(self, coefficients):
    return GPolynomialRing(coefficients, self.var, self.exponents)
def merge(self, other):
    if isinstance(other, GPolynomialFunctor) and self.var == other.var:
        exponents = pushout(self.exponents, other.exponents)
        return GPolynomialFunctor(self.var, exponents)
def _coerce_map_from_(self, R):
    if isinstance(other, GPolynomialFunctor) and self.var == other.var:
        exponents = pushout(self.exponents, other.exponents)
        return GPolynomialFunctor(self.var, exponents)

We can construct a parent now in two different ways:

sage: GPolynomialRing(QQ, 'X', ZZ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: GP_ZZ = GPolynomialFunctor('X', ZZ); GP_ZZ
GPoly[X^(Integer Ring)]
sage: GP_ZZ(QQ)
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field

Since the construction

sage: GP_ZZ(QQ).construction()
(GPoly[X^(Integer Ring)], Rational Field)

uses the coefficient ring, we have the usual coercion with respect to this parameter:

sage: pushout(GP_ZZ(ZZ), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Rational Field
sage: pushout(GP_ZZ(ZZ['t']), GP_ZZ(QQ))
Generalized Polynomial Ring in X^(Integer Ring) over Univariate Polynomial Ring over Integer Ring
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['b,c']))
Generalized Polynomial Ring in a, b, c over Integer Ring
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(QQ['b,c']))
Generalized Polynomial Ring in X^(Integer Ring)
over Multivariate Polynomial Ring in a, b, c over Rational Field

```python
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

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

```python
sage: pushout(GP_QQ(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^((Rational Field) over Rational Field
```

```python
sage: GP_ZZt = GPolynomialFunctor('X', ZZ['t'])
sage: pushout(GP_ZZt(ZZ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^((Univariate Polynomial Ring in t over Rational Field) over Integer Ring
```

```python
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

```python
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['b,c']))
Generalized Polynomial Ring in X^((Univariate Polynomial Ring in t over Rational Field) over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

Some tests with Cartesian products:

```python
sage: from sage.sets.cartesian_product import CartesianProduct
sage: A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z']), Sets().CartesianProducts())
sage: B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z']), Sets().CartesianProducts())
sage: A.construction()  
(The cartesian_product functorial construction,  
(Univariate Polynomial Ring in x over Integer Ring,  
Univariate Polynomial Ring in y over Rational Field,  
Univariate Polynomial Ring in z over Rational Field))
sage: pushout(A, B)
The Cartesian product of  
(Univariate Polynomial Ring in x over Integer Ring,  
Univariate Polynomial Ring in y over Rational Field,  
Univariate Polynomial Ring in z over Univariate Polynomial Ring in t over Rational Field)  
over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

(continues on previous page)
sage: from sage.categories.pushout import PolynomialFunctor
sage: from sage.sets.cartesian_product import CartesianProduct
sage: class CartesianProductPoly(CartesianProduct):
        ....:     def __init__(self, polynomial_rings):
        ....:         sort = sorted(polynomial_rings, key=lambda P: P.variable_name())
        ....:         super(CartesianProductPoly, self).__init__(sort, Sets().\→CartesianProducts())
        ....:     def vars(self):
        ....:         return tuple(P.variable_name() for P in self.cartesian_factors())
        ....:     def _pushout_(self, other):
        ....:         if isinstance(other, CartesianProductPoly):
        ....:             s_vars = self.vars()
        ....:             o_vars = other.vars()
        ....:             if s_vars == o_vars:
        ....:                 return
        ....:             return pushout(CartesianProductPoly(self.cartesian_factors() +
                        tuple(f for f in other.cartesian_factors() if f.variable_name() not in s_vars)),
                        CartesianProductPoly(other.cartesian_factors() +
                        tuple(f for f in self.cartesian_factors() if f.variable_name() not in o_vars)))
        ....:             C = other.construction()
        ....:             if C is None:
        ....:                 return
        ....:             elif isinstance(C[0], PolynomialFunctor):
        ....:                 return pushout(self, CartesianProductPoly((other,))
        ....: sage: pushout(CartesianProductPoly((ZZ['x'],)),
                        CartesianProductPoly((ZZ['y'],)))
The Cartesian product of
  (Univariate Polynomial Ring in x over Integer Ring,
   Univariate Polynomial Ring in y over Integer Ring)
sage: pushout(CartesianProductPoly((ZZ['x'], ZZ['y'])),
                        CartesianProductPoly((ZZ['x'], ZZ['z'])))
The Cartesian product of
  (Univariate Polynomial Ring in x over Integer Ring,
   Univariate Polynomial Ring in y over Integer Ring,
   Univariate Polynomial Ring in z over Integer Ring)
sage: pushout(CartesianProductPoly((QQ['a,b']['x'], QQ['y'])),
                        CartesianProductPoly((ZZ['b,c']['x'], SR['z'])))
The Cartesian product of
  (Univariate Polynomial Ring in x over
   Multivariate Polynomial Ring in a, b, c over Rational Field,
   Univariate Polynomial Ring in y over Rational Field,
   Univariate Polynomial Ring in z over Symbolic Ring)
sage: pushout(CartesianProductPoly((ZZ['x'],)), ZZ['y'])
The Cartesian product of
  (Univariate Polynomial Ring in x over Integer Ring,
AUTHORS:

- Robert Bradshaw
- Peter Bruin
- Simon King
- Daniel Krenn
- David Roe

**sage.categories.pushout.pushout_lattice** *(R, S)*

Given a pair of objects *R* and *S*, try to construct a reasonable object *Y* and return maps such that canonically

\[ R \leftarrow Y \rightarrow S. \]

**ALGORITHM:**

This is based on the model that arose from much discussion at Sage Days 4. Going up the tower of constructions of *R* and *S* (e.g., the reals come from the rationals come from the integers), try to find a common parent, and then try to fill in a lattice with these two towers as sides with the top as the common ancestor and the bottom will be the desired ring.

See the code for a specific worked-out example.

**EXAMPLES:**

```python
sage: from sage.categories.pushout import pushout_lattice
sage: A, B = pushout_lattice(Qp(7), Frac(ZZ['x']))
```

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
### 3.1 Group, ring, etc. actions on objects.

The terminology and notation used is suggestive of groups acting on sets, but this framework can be used for modules, algebras, etc.

A group action \( G \times S \to S \) is a functor from \( G \) to \( \text{Sets} \).

**Warning:** An \texttt{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
sage: from sage.categories.action import Action
sage: class P:
    pass
sage: A = Action(P(),P())
sage: import gc
sage: _ = gc.collect()
```

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

```sage
sage: _ = gc.collect()
sage: from sage.categories.action import Action
sage: class P:
    pass
sage: q = P()
```

**AUTHOR:**

- Robert Bradshaw: initial version

```python
class sage.categories.action.Action
    Bases: sage.categories.functor.Functor

    The action of \( G \) on \( S \).

    INPUT:
```
• G – a parent or Python type
• S – a parent or Python type
• is_left – (boolean, default: True) whether elements of G are on the left
• op – (default: None) operation. This is not used by Action itself, but other classes may use it

G

act (g, x)

This is a consistent interface for acting on x by g, regardless of whether it’s a left or right action.
If needed, g and x are converted to the correct parent.

EXAMPLES:

```python
sage: R.<x> = ZZ []
sage: from sage.structure.coerce_actions import IntegerMulAction
sage: A = IntegerMulAction(ZZ, R, True)  # Left action
sage: A.act(5, x)
5*x
sage: A.act(int(5), x)
5*x
sage: A = IntegerMulAction(ZZ, R, False)  # Right action
sage: A.act(5, x)
5*x
sage: A.act(int(5), x)
5*x
```

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

class sage.categories.action.ActionEndomorphism
Bases: sage.categories.morphism.Morphism

The endomorphism defined by the action of one element.

EXAMPLES:

```python
sage: A = ZZ['x'].get_action(QQ, self_on_left=False, op=operator.mul)
sage: A
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Ring
sage: A(1/2)
Action of 1/2 on Univariate Polynomial Ring in x over Integer Ring under Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Ring.
```

class sage.categories.action.InverseAction
Bases: sage.categories.action.Action
An action that acts as the inverse of the given action.

EXAMPLES:

```python
 sage: V = QQ^3
g sage: v = V((1, 2, 3))
sage: cm = get_coercion_model()
sage: a = cm.get_action(V, QQ, operator.mul)
sage: a
Right scalar multiplication by Rational Field on Vector space of dimension 3 over Rational Field
 sage: ~(a)
Right inverse action by Rational Field on Vector space of dimension 3 over Rational Field
 sage: (~a)(v, 1/3)
(3, 6, 9)
sage: b = cm.get_action(QQ, V, operator.mul)
sage: b
Left scalar multiplication by Rational Field on Vector space of dimension 3 over Rational Field
 sage: ~(b)
Left inverse action by Rational Field on Vector space of dimension 3 over Rational Field
 sage: (~b)(1/3, v)
(3, 6, 9)
sage: c = cm.get_action(ZZ, list, operator.mul)
sage: c
Left action by Integer Ring on <... 'list'>
sage: ~(c)
Traceback (most recent call last):
... TypeError: no inverse defined for Left action by Integer Ring on <... 'list'>
```

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:

```python
 sage: E = ModularSymbols(11).2
g sage: s = E.modular_symbol_rep()
g sage: del E, s
g sage: import gc
g sage: _ = gc.collect()
g sage: E = ModularSymbols(11).2
g sage: v = E.manin_symbol_rep()
g sage: c, x = v[0]
g sage: y = x.modular_symbol_rep()
g sage: coercion_model.get_action(QQ, parent(y), op=operator.mul)
Left scalar multiplication by Rational Field on Abelian Group of all Formal Finite Sums over Rational Field
with precomposition on right by Coercion map:
```

(continues on next page)
From: Abelian Group of all Formal Finite Sums over Integer Ring
To: Abelian Group of all Formal Finite Sums over Rational Field

codomain()
domain()

**left_precomposition**
The left map to precompose with, or None if there is no left precomposition map.

**right_precomposition**
The right map to precompose with, or None if there is no right precomposition map.

## 3.2 Additive groups

```python
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:
```

```python
sage: from sage.categories.additive_groups import AdditiveGroups
dsage: from sage.categories.additive_monoids import AdditiveMonoids
dsage: AdditiveGroups()
Category of additive groups
sage: AdditiveGroups().super_categories()
[Category of additive inverse additive unital additive magmas,
 Category of additive monoids]
sage: AdditiveGroups().all_super_categories()
[Category of additive groups,
 Category of additive inverse additive unital additive magmas,
 Category of additive monoids,
 Category of additive unital additive magmas,
 Category of additive semigroups,
 Category of additive magmas,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
sage: AdditiveGroups().axioms()
frozenset({'AdditiveAssociative', 'AdditiveInverse', 'AdditiveUnital'})
sage: AdditiveGroups() is AdditiveMonoids().AdditiveInverse()
True
```

```
AdditiveCommutative
    alias of sage.categories.commutative_additive_groups.CommutativeAdditiveGroups

class Algebras(category, *args):
    Bases: sage.categories.algebra_functor.AlgebrasCategory

    class ParentMethods
```
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

class Finite(base_category)

   Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class Algebras(category, *args)

   Bases: sage.categories.algebra_functor.AlgebrasCategory

class ParentMethods

   extra_super_categories()

       Implement Maschke’s theorem.

       In characteristic 0 all finite group algebras are semisimple.

       EXAMPLES:

       sage: FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
       True
       sage: FiniteGroups().Algebras(FiniteField(7)).is_subcategory(Algebras(FiniteField(7)).Semisimple())
       False
       sage: FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
       False
       sage: FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
       False
       sage: Cat = CommutativeAdditiveGroups().Finite()
       sage: Cat.Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
       True
       sage: Cat.Algebras(GF(7)).is_subcategory(Algebras(GF(7)).Semisimple())
       False
       sage: Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
       False
       sage: Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
       False

3.3 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:
The following axioms are defined by this category:

```python
sage: AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
sage: AdditiveMagmas().AdditiveUnital()
Category of additive unital additive magmas
sage: AdditiveMagmas().AdditiveCommutative()
Category of additive commutative additive magmas
sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative()
Category of commutative additive semigroups
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().
    AdditiveUnital()
Category of commutative additive monoids
sage: AdditiveMagmas().AdditiveAssociative().AdditiveCommutative().
    AdditiveUnital().AdditiveInverse()
Category of commutative additive groups
```

**AdditiveAssociative**

alias of `sage.categories.additive_semigroups.AdditiveSemigroups`

**class AdditiveCommutative(base_category)**

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

**class Algebras(category, *args)**

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

```python
extra_super_categories()
Implement the fact that the algebra of a commutative additive magmas is commutative.
EXAMPLES:
```

```python
sage: AdditiveMagmas().AdditiveCommutative().Algebras(QQ).extra_super_
    _categories()
[Category of commutative magmas]
```

```python
sage: AdditiveMagmas().AdditiveCommutative().Algebras(QQ).super_
    _categories()
[Category of additive magma algebras over Rational Field,
  Category of commutative additive groups]
```

**class CartesianProducts(category, *args)**

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

```python
extra_super_categories()
Implement the fact that a Cartesian product of commutative additive magmas is a commutative
additive magma.
EXAMPLES:
```

```python
sage: C = AdditiveMagmas().AdditiveCommutative().CartesianProducts()
sage: C.extra_super_categories()
[Category of additive commutative additive magmas]
sage: C.axioms()
frozenset({'AdditiveCommutative'})
```

class AdditiveUnital (base_category)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

class AdditiveInverse (base_category)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

class CartesianProducts (category, *args)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

class ElementMethods

extra_super_categories ()

Implement the fact that a Cartesian product of additive magmas with inverses is an additive magma with inverse.

EXAMPLES:

```python
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 Algebras (category, *args)

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

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:

```python
sage: S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid: the free commutative monoid
   generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(ZZ)
sage: A.one_basis()
0
sage: A.one()
B[0]
sage: A(3)
3*B[0]
```

extra_super_categories ()

EXAMPLES:

```python
sage: C = AdditiveMagmas().AdditiveUnital().Algebras(QQ)
sage: C.extra_super_categories()
```

(continues on next page)

3.3. Additive Magmas 153
class CartesianProducts(category, *args)
    Bases: sage.categories.cartesian_product.CartesianProductsCategory

class ParentMethods

    zero()
    Returns the zero of this group

    EXAMPLES:

    sage: GF(8,'x').cartesian_product(GF(5)).zero()
    (0, 0)

extra_super_categories()
Implement the fact that a Cartesian product of unital additive magmas is a unital additive magma.

    EXAMPLES:

    sage: C = AdditiveMagmas().AdditiveUnital().CartesianProducts()
    sage: C.extra_super_categories()
    [Category of additive unital additive magmas]
    sage: C.axioms()
    frozenset({'AdditiveUnital'})

class ElementMethods

class Homsets(category, *args)
    Bases: sage.categories.homsets.HomsetsCategory

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

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

    EXAMPLES:
class 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:

```python
sage: A = AdditiveAbelianGroup([3,3])
sage: A in AdditiveMagmas()
True
sage: A.is_empty()
False
```

zero()

Return the zero of this additive magma, that is the unique neutral element for +.

The default implementation is to coerce 0 into self.

It is recommended to override this method because the coercion from the integers:

• is not always meaningful (except for 0), and
• often uses self.zero() otherwise.

EXAMPLES:

```python
sage: S = CommutativeAdditiveMonoids().example()
sage: S.zero()
0
```

class 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:

```python
sage: AdditiveMagmas().AdditiveUnital().AdditiveInverse()
Category of additive inverse additive unital additive magmas
sage: from sage.categories.additive_monoids import AdditiveMonoids
```
class WithRealizations (category, *args)
Bases: sage.categories.with_realizations.WithRealizationsCategory

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

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 Algebras (category, *args)
Bases: sage.categories.algebra_functor.AlgebrasCategory

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', 'c', 'd')
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\( (g_1, g_2) \)
Product, on basis elements, as per \texttt{MagmaticAlgebras.WithBasis.ParentMethods.product_on_basis()}.

The product of two basis elements is induced by the addition of the corresponding elements of the group.

EXAMPLES:

```python
sage: S = CommutativeAdditiveSemigroups().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(QQ)
sage: a,b,c,d = A.algebra_generators()
sage: a * b + b * d * c
B[b + c + d] + B[a + b]
```

extra_super_categories()

EXAMPLES:

```python
sage: AdditiveMagmas().Algebras(QQ).extra_super_categories()
[Category of magmatic algebras with basis over Rational Field]
sage: AdditiveMagmas().Algebras(QQ).super_categories()
[Category of magmatic algebras with basis over Rational Field, Category of set algebras over Rational Field]
```

class CartesianProducts\( (\text{category}, \astargs) \)

Bases: \texttt{sage.categories.cartesian_product.CartesianProductsCategory}

class ElementMethods

extra_super_categories()
Implement the fact that a Cartesian product of additive magmas is an additive magma.

EXAMPLES:

```python
sage: C = AdditiveMagmas().CartesianProducts() sagesage: C.extra_super_categories()
[Category of additive magmas]
sage: C.super_categories()
[Category of additive magmas, Category of Cartesian products of sets]
sage: C.axioms()
frozenset()
```

class ElementMethods

class Homsets\( (\text{category}, \astargs) \)

Bases: \texttt{sage.categories.homsets.HomsetsCategory}

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

EXAMPLES:

```python
sage: AdditiveMagmas().Homsets().extra_super_categories()
[Category of additive magmas]
sage: AdditiveMagmas().Homsets().super_categories()
[Category of additive magmas, Category of homsets]
```

3.3. Additive Magmas
class 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
```

(continues on next page)
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
```

3.3. Additive Magmas

159
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| 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: Ssummation(a, b)
a + b
```

A parent in `AdditiveMagmas()` must either implement `summation()` in the parent class or `_add_` in the element class. By default, the addition method on elements *x*._add_(y) calls *S*. `summation(x, y)`, and reciprocally.

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

```
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:**
A parent in `AdditiveMagmas()` must either implement `summation()` in the parent class or `_add_` in the element class. By default, the addition method on elements `x._add_(y)` calls `S.summation(x, y)`, and reciprocally.

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

```python
sage: bin = 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.

```python
class 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 ∈ M`,

\[ x + (y + z) = (x + y) + z \]

See also:

Wikipedia article Associative_property

EXAMPLES:

```python
sage: AdditiveMagmas().AdditiveAssociative()
Category of additive semigroups
```

AdditiveCommutative()

Return the full subcategory of the commutative objects of `self`.

An additive magma `M` is commutative if, for all `x, y ∈ M`,

\[ x + y = y + x \]

See also:

Wikipedia article Commutative_property

EXAMPLES:

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

(continues on next page)
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:

```python
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
```

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

### 3.4 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 additive semigroup, that is a set endowed with a binary operation + which is associative and admits a zero (see Wikipedia article Monoid).

EXAMPLES:

```python
sage: from sage.categories.additive_monoids import AdditiveMonoids
sage: C = AdditiveMonoids(); C
Category of additive monoids
sage: C.super_categories()
[Category of additive unital additive magmas, Category of additive semigroups]
sage: sorted(C.axioms())
['AdditiveAssociative', 'AdditiveUnital']
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: C is AdditiveSemigroups().AdditiveUnital()
True
```
AdditiveCommutative
alias of sage.categories.commutative_additive_monoids.CommutativeAdditiveMonoids

AdditiveInverse
alias of sage.categories.additive_groups.AdditiveGroups

class Homsets(category, *args)
Bases: sage.categories.homsets.HomsetsCategory

extra_super_categories()
Implement the fact that a homset between two monoids is associative.

EXAMPLES:

Todo: This could be deduced from AdditiveSemigroups.Homsets.
extra_super_categories(). See comment in Objects.SubcategoryMethods.
Homsets().

class 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:

3.5 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:
```python
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
```

**AdditiveCommutative**

alias of `sage.categories.commutative_additive_semigroups.CommutativeAdditiveSemigroups`

**AdditiveUnital**

alias of `sage.categories.additive_monoids.AdditiveMonoids`

class `Algebras(category, *args)`

Bases: `sage.categories.algebra_functor.AlgebrasCategory`

class `ParentMethods`

```
algebra_generators()
Return the generators of this algebra, as per `MagmaticAlgebras.ParentMethods.algebra_generators()`.

They correspond to the generators of the additive semigroup.

EXAMPLES:
```
```
sage: S = CommutativeAdditiveSemigroups().example(); S
An example of a commutative monoid: the free commutative monoid
→ generated by ('a', 'b', 'c', 'd')
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_basis()`.

The product of two basis elements is induced by the addition of the corresponding elements of the group.

EXAMPLES:
```
```
extra_super_categories()

EXAMPLES:

```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
def addive_semigroups() .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 CartesianProducts(category, *args)

Bases: sage.categories.cartesian_product.CartesianProductsCategory

declared_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
def addive_semigroups() .CartesianProducts()
def extra_super_categories()
[Category of additive semigroups]
sage: C.axioms()
frozenset({'AdditiveAssociative'})
```

class Homsets(category, *args)

Bases: sage.categories.homsets.HomsetsCategory

declared_super_categories()

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

EXAMPLES:

```
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
def addive_semigroups() .Homsets().extra_super_categories()
[Category of additive semigroups]
sage: AdditiveSemigroups().Homsets().super_categories()
[Category of homsets of additive magmas, Category of additive semigroups]
```

class ParentMethods

3.6 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 irreducible affine weyl groups
```

```
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([0,2,1,0])
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 :meth:`affine_grassmannian_to_core()`.

**EXAMPLES:**

```python
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
```

.. automethod:: is_affine_grassmannian

Tests whether :attr:`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 :attr:`self` end with 0.
- :attr:`self` is the identity, or 0 is its single right descent.
- :attr:`self` is a minimal coset representative for :math:`W / \text{cl } W`.

**EXAMPLES:**

```python
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** ParentMethods

.. automethod:: affine_grassmannian_elements_of_given_length

Returns the affine Grassmannian elements of length :math:`k`, as a list.

**EXAMPLES:**

```python
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:

:func:`AffineWeylGroups.ElementMethods.is_affine_grassmannian`

**Todo:** should return an enumerated set, with iterator, ...

.. automethod:: special_node

Returns the distinguished special node of the underlying Dynkin diagram

**EXAMPLES:**

```python
```
additional_structure()
    Return None.

    Indeed, the category of affine Weyl groups defines no additional structure: affine Weyl groups are a special
    class of Weyl groups.

    See also:
    Category.additional_structure()

    Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:
    sage: AffineWeylGroups().additional_structure()

super_categories()
    EXAMPLES:
    sage: AffineWeylGroups().super_categories()
    [Category of infinite weyl groups]

3.7 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:

```python
sage: AlgebraIdeals(QQ['x']).super_categories()
[Category of algebra modules over Univariate Polynomial Ring in x over Rational Field]
```

3.8 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:

```python
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:

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

classmethod an_instance()

Returns an instance of this class

EXAMPLES:

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

super_categories()

EXAMPLES:

```python
sage: AlgebraModules(QQ['x']).super_categories()
[Category of modules over Univariate Polynomial Ring in x over Rational Field]
```
3.9 Algebras

AUTHORS:

- David Kohel & William Stein (2005): initial revision

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]
```

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/35a72b1d02fc77a/348f42ae77a66d16#348f42ae77a66d16
- Wikipedia article 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]
```
Commutative
alias of sage.categories.commutative_algebras.CommutativeAlgebras

class DualObjects(category, *args)
Bases: sage.categories.dual.DualObjectsCategory

extra_super_categories()
Returns the dual category

EXAMPLES:
The category of algebras over the Rational Field is dual to the category of coalgebras over the same field:

```
sage: C = 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 ElementMethods

Filtered
alias of sage.categories.filtered_algebras.FilteredAlgebras

Graded
alias of sage.categories.graded_algebras.GradedAlgebras

class Quotients(category, *args)
Bases: sage.categories.quotients.QuotientsCategory

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 {'x': B['x'], 'y': B['y'], 'a': 0, 'b': 0}
```

Todo: this could possibly remove the elements that retract to zero.

Semisimple
alias of sage.categories.semisimple_algebras.SemisimpleAlgebras

class SubcategoryMethods

3.9. Algebras
**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
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
```

**Super**

alias of `sage.categories.super_algebras.SuperAlgebras`

**class TensorProducts** *(category, *args)*

Bases: `sage.categories.tensor.TensorProductsCategory`

class `ElementMethods`

class `ParentMethods`

**extra_super_categories()**

**EXAMPLES:**

```sage
sage: Algebras(QQ).TensorProducts().extra_super_categories()
[Category of algebras over Rational Field]

sage: Algebras(QQ).TensorProducts().super_categories()
[Category of algebras over Rational Field, 
Category of tensor products of vector spaces over Rational Field]
```

Meaning: a tensor product of algebras is an algebra

**WithBasis**

alias of `sage.categories.algebras_with_basis.AlgebrasWithBasis`

### 3.10 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
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
sage: A = C.example(); A
An example of an algebra with basis: the free algebra on the generators ('a', 'b', 'c') over Rational Field
```
sage: A.category()
Category of algebras with basis over Rational Field

sage: A.one_basis()
word:
sage: A.one()
B[word: ]

sage: A.base_ring()
Rational Field
sage: A.basis().keys()
Finite words over ('a', 'b', 'c')

sage: (a,b,c) = A.algebra_generators()
sage: a^3, b^2
(B[word: aaa], B[word: bb])
sage: a*c*b
B[word: acb]

sage: A.product
<bound method 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 Rational Field>
sage: A.product(a*b,b)
B[word: abb]

sage: TestSuite(A).run(verbosetrue)
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_new() ... pass
running ._test_nonzero_equal() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_elements_eq_reflexive() ... pass
running ._test_elements_eq_symmetric() ... pass
running ._test_elements_eq_transitive() ... pass
running ._test_neq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_one() ... pass
running ._test_pickling() ... pass
running ._test_prod() ... pass
running ._test_some_elements() ... pass
running ._test_zero() ... pass
sage: A.__class__
Please see the source code of $A$ (with $A??$) for how to implement other algebras with basis.

**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()**

**one_from_cartesian_product_of_one_basis()**

Returns the one of this Cartesian product of algebras, as per `Monoids.ParentMethods.one`

It is constructed as the Cartesian product of the ones of the summands, using their `one_basis()` methods.

This implementation does not require multiplication by scalars nor calling `cartesian_product`. This might help keeping things as lazy as possible upon initialization.

**EXAMPLES:**

```
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra on the
  generators ('a', 'b', 'c') over Rational Field
sage: A.one_basis()
word:

sage: B = cartesian_product((A, A, A))
sage: B.one_from_cartesian_product_of_one_basis()
B[(0, word: )] + B[(1, word: )] + B[(2, word: )]
sage: B.one()
B[(0, word: )] + B[(1, word: )] + B[(2, word: )]
sage: cartesian_product([SymmetricGroupAlgebra(QQ, 3),
  SymmetricGroupAlgebra(QQ, 4)]).one()
B[(0, [1, 2, 3])] + B[(1, [1, 2, 3, 4])]
```

**extra_super_categories()**

A Cartesian product of algebras with basis is endowed with a natural algebra with basis structure.

**EXAMPLES:**

```
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 ElementMethods

Filtered
alias of sage.categories.filtered_algebras_with_basis.
FilteredAlgebrasWithBasis

FiniteDimensional
alias of sage.categories.finite_dimensional_algebras_with_basis.
FiniteDimensionalAlgebrasWithBasis

Graded
alias of sage.categories.graded_algebras_with_basis.
GradedAlgebrasWithBasis

class ParentMethods

hochschild_complex(M)
Return the Hochschild complex of self with coefficients in M.

See also:
HochschildComplex

EXAMPLES:

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

one()
Return the multiplicative unit element.

EXAMPLES:

```
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one_basis()
word:
sage: A.one()
B[word: ]
```

Super
alias of sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis

class 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 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

3.10. Algebras With Basis 175
It is the tuple whose operands are the indices of the ones of the operands, as returned by their `one_basis()` methods.

**EXAMPLES:**

```
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra on the
→ generators ('a', 'b', 'c') over Rational Field
sage: A.one_basis()
word:
sage: B = tensor((A, A, A))

sage: B.one()
```

**product_on_basis (i1,i2)**

The product of the algebra on the basis, as per `AlgebrasWithBasis.ParentMethods.product_on_basis`.

**EXAMPLES:**

```
sage: A = AlgebrasWithBasis(QQ).example(); A
An example of an algebra with basis: the free algebra on the
→ generators ('a', 'b', 'c') over Rational Field
sage: (a,b,c) = A.algebra_generators()
sage: x = tensor( (a, b, c) ); x
sage: y = tensor( (c, b, a) ); y
sage: x*y
sage: x = tensor( ((a+2*b), c) ) ; x
sage: y = tensor( (c, a) ) + 1; y
sage: x*y
```

TODO: optimize this implementation!

**extra_super_categories ()**

**EXAMPLES:**

```
sage: AlgebrasWithBasis(QQ).TensorProducts().extra_super_categories()
[Category of algebras with basis over Rational Field]
sage: AlgebrasWithBasis(QQ).TensorProducts().super_categories()
[Category of algebras with basis over Rational Field, 
Category of tensor products of algebras over Rational Field, 
Category of tensor products of vector spaces with basis over Rational Field]
```

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

Return an example of algebra with basis.

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

An other set of generators can be specified as optional argument:

An example of an algebra with basis: the free algebra on the generators (1, 2, 3) over Rational Field

### 3.11 Aperiodic semigroups

```python
sage: AlgebrasWithBasis(QQ).example()
```

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

An other set of generators can be specified as optional argument:

```python
sage: AlgebrasWithBasis(QQ).example((1,2,3))
```

An example of an algebra with basis: the free algebra on the generators (1, 2, 3) over Rational Field

#### 3.11.2 Aperiodic semigroups

The category of aperiodic semigroups.

```python
class sage.categories.aperiodic_semigroups.AperiodicSemigroups(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom
```

```python
extra_super_categories()
```

Implement the fact that an aperiodic semigroup is $H$-trivial.

#### EXAMPLES:

```python
sage: Semigroups().Aperiodic().extra_super_categories()
```

```
[Category of h trivial semigroups]
```

### 3.12 Associative algebras

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.

#### EXAMPLES:

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

```
Unital
alias of sage.categories.algebras.Algebras
```

### 3.13 Bialgebras

The category of bialgebras over a given base ring.

#### EXAMPLES:

```python
class sage.categories.bialgebras.Bialgebras(base, name=None)
Bases: sage.categories.category_types.Category_over_base_ring
```

3.11. Aperiodic semigroups 177
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]
```

class Super(base_category)
Bases: `sage.categories.super_modules.SuperModulesCategory`

WithBasis
alias of `sage.categories.bialgebras_with_basis.BialgebrasWithBasis`

additional_structure()
Return None.

Indeed, the category of bialgebras defines no additional structure: a morphism of coalgebras and of algebras between two bialgebras is a bialgebra morphism.

See also:
`Category.additional_structure()`

Todo: This category should be a `CategoryWithAxiom`.

EXAMPLES:

```
sage: Bialgebras(QQ).additional_structure()
```

super_categories()
EXAMPLES:

```
sage: Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]
```

### 3.14 Bialgebras with basis

class sage.categories.bialgebras_with_basis.BialgebrasWithBasis(base_category)
Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

The category of bialgebras with a distinguished basis.

EXAMPLES:

```
sage: C = BialgebrasWithBasis(QQ); C
Category of bialgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of algebras with basis over Rational Field, Category of bialgebras over Rational Field, Category of coalgebras with basis over Rational Field]
```

class ElementMethods
**adams_operator** \((n)\)

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

**INPUT:**
- \(n\) – a nonnegative integer

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

**Note:** In the literature, this is also called a Hopf power or Sweedler power, cf. [AL2015].

**See also:**
- `sage.categories.bialgebras.ElementMethods.convolution_product()`

**Todo:** Remove dependency on `modules_with_basis` methods.

**EXAMPLES:**

```python
sage: h = SymmetricFunctions(QQ).h()
sage: h[5].adams_operator(2)
sage: h[[],].adams_operator(0)
h[]
sage: h[[],].adams_operator(1)
h[]
sage: h[3, 2].adams_operator(0)
0
sage: h[3, 2].adams_operator(1)
h[3, 2]
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: S[4].adams_operator(5)
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: m[[1,3],[2]].adams_operator(-2)
3*m({1}, {2, 3}) + 3*m({1, 2}, {3}) + 6*m({1, 2, 3}) - 2*m({1, 3}, {2})
```

**convolution_product** \((*\text{maps})\)

Return the image of \(\text{self}\) under the convolution product (map) of the maps.

Let \(A\) and \(B\) be bialgebras over a commutative ring \(R\). Given maps \(f_i : A \to B\) for \(1 \leq i < n\), define the convolution product

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

where \(\Delta^{(k)} := (\Delta \otimes \text{Id}^{(k-1)}) \circ \Delta^{(k-1)}\), with \(\Delta^{(1)} = \Delta\) (the ordinary coproduct in \(A\)) and \(\Delta^{(0)} = \text{Id}\); and with \(\mu^{(k)} := \mu \circ (\mu^{(k-1)} \otimes \text{Id})\) and \(\mu^{(1)} = \mu\) (the ordinary product in \(B\)). See [Swe1969].

(In the literature, one finds, e.g., \(\Delta^{(2)}\) for what we denote above as \(\Delta^{(1)}\). See [KMN2012].)

**INPUT:**
- \(\text{maps}\) – any number \(n \geq 0\) of linear maps \(f_1, f_2, \ldots, f_n\) on \(\text{self}.parent()\); or a single list or tuple of such maps
OUTPUT:

• the convolution product of maps applied to self

AUTHORS:

• Amy Pang - 12 June 2015 - Sage Days 65

Todo: Remove dependency on modules_with_basis methods.

EXAMPLES:

We compute convolution products of the identity and antipode maps on Schur functions:

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

The method accepts multiple arguments, or a single argument consisting of a list of maps:

```
sage: s[3,2].convolution_product(Id, Id)
2*s[2, 1, 1, 1] + 6*s[2, 2, 1] + 6*s[3, 1, 1] + 12*s[3, 2] + 6*s[4, 1] +
   → 2*s[5]
sage: s[3,2].convolution_product([Id, Id])
2*s[2, 1, 1, 1] + 6*s[2, 2, 1] + 6*s[3, 1, 1] + 12*s[3, 2] + 6*s[4, 1] +
   → 2*s[5]
```

We test the defining property of the antipode morphism; namely, that the antipode is the inverse of the identity map in the convolution algebra whose identity element is the composition of the counit and unit:

```
sage: s[3,2].convolution_product() == s[3,2].convolution_product(Antipode, \rightarrow Id) == s[3,2].convolution_product(Id, Antipode)
True
```

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

```
(continues on next page)
class ParentMethods

convolution_product(*maps)
Return the convolution product (a map) of the given maps.

Let $A$ and $B$ be bialgebras over a commutative ring $R$. Given maps $f_i : A \to B$ for $1 \leq i < n$, define the convolution product

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

where $\Delta^{(k)} := (\Delta \otimes \text{Id}^{(k-1)}) \circ \Delta^{(k-1)}$, with $\Delta^{(1)} = \Delta$ (the ordinary coproduct in $A$) and $\Delta^{(0)} = \text{Id}$; and with $\mu^{(k)} := \mu \circ (\mu^{(k-1)} \otimes \text{Id})$ and $\mu^{(1)} = \mu$ (the ordinary product in $B$). See [Swe1969].

(In the literature, one finds, e.g., $\Delta^{(2)}$ for what we denote above as $\Delta^{(1)}$. See [KMN2012].)
INPUT:
• maps – any number $n \geq 0$ of linear maps $f_1, f_2, \ldots, f_n$ on self; or a single list or tuple of such maps

OUTPUT:
• the new map $f_1 * f_2 * \cdots * f_n$ representing their convolution product

See also:
sage.categories.bialgebras.ElementMethods.convolution_product()

AUTHORS:
• Aaron Lauve - 12 June 2015 - Sage Days 65

Todo: Remove dependency on modules_with_basis methods.

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

```python
sage: Id = lambda x: x
sage: Antipode = lambda x: x.antipode()
sage: Proj2 = lambda x: x.parent().sum_of_terms([[(m, c) for (m, c) in x˓→if m.size() == 2]])
```

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

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

Compute the convolution product of no maps on the Hopf algebra of symmetric functions in non-commuting variables. This is the composition of the counit with the unit:

```python
sage: m = SymmetricFunctionsNonCommutingVariables(QQ).m()
sage: T = m.convolution_product()
sage: [T(m(lam)) for lam in SetPartitions(0).list() + SetPartitions(2).˓→list()]
```

Compute the convolution product of the projection Proj2 with the identity on the Hopf algebra of symmetric functions in non-commuting variables:

```python
sage: T = m.convolution_product(Proj2, Id)
sage: [T(m(lam)) for lam in SetPartitions(3)]
```
Compute the convolution product of the antipode with itself and the identity map on group algebra of the symmetric group:

```
sage: G = SymmetricGroup(3)
sage: QQ = GroupAlgebra(G, QQ)
sage: x = QQ.sum_of_terms([p.number_of_peaks() + p.number_of_inversions() for p in Permutations(3)]); x
2*[1, 3, 2] + [2, 1, 3] + 3*[2, 3, 1] + 2*[3, 1, 2] + 3*[3, 2, 1]
sage: T = QQ.convolution_product(Antipode, Antipode, Id)
sage: T(x)
2*[1, 3, 2] + [2, 1, 3] + 2*[2, 3, 1] + 3*[3, 1, 2] + 3*[3, 2, 1]
```

### 3.15 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: \(r \cdot (x \cdot s) = (r \cdot x) \cdot s\).

**EXAMPLES:**

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

**class** `ElementMethods`

**class** `ParentMethods`

**additional_structure()**

Return None.

Indeed, the category of bimodules defines no additional structure: a left and right module morphism between two bimodules is a bimodule morphism.

**See also:**

`Category.additional_structure()`

**Todo:** Should this category be a `CategoryWithAxiom`?

**EXAMPLES:**

```
sage: Bimodules(QQ, ZZ).additional_structure()
```

**classmethod** `an_instance()`

Return an instance of this class.

**EXAMPLES:**

```
sage: Bimodules(QQ, ZZ).an_instance()
```
```python
sage: Bimodules.an_instance()
Category of bimodules over Rational Field on the left and Real Field with 53
→bits of precision on the right
```

```python
left_base_ring()
Return the left base ring over which elements of this category are defined.
EXAMPLES:
```
```python
sage: Bimodules(QQ, ZZ).left_base_ring()
Rational Field
```
```
right_base_ring()
Return the right base ring over which elements of this category are defined.
EXAMPLES:
```
```python
sage: Bimodules(QQ, ZZ).right_base_ring()
Integer Ring
```
```
super_categories()
EXAMPLES:
```
```python
sage: Bimodules(QQ, ZZ).super_categories()
[Category of left modules over Rational Field, Category of right modules over
→Integer Ring]
```

### 3.16 Classical Crystals

```python
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:
```
```python
sage: C = ClassicalCrystals()
sage: C
Category of classical crystals
sage: C.super_categories()
[Category of regular crystals,
Category of finite crystals,
Category of highest weight crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```
```
class ElementMethods
```
```
lusztig_involution()
Return the Lusztig involution on the classical highest weight crystal self.
The Lusztig involution on a finite-dimensional highest weight crystal $B(\lambda)$ of highest weight $\lambda$ maps the highest weight vector to the lowest weight vector and the Kashiwara operator $f_i$ to $e_i^{*-}$, where $i^{*-}$ is defined as $\alpha_{i^{*-}} = -w_0(\alpha_i)$. Here $w_0$ is the longest element of the Weyl group acting on the $i$-th simple root $\alpha_i$.
```
EXAMPLES:

```python
sage: B = crystals.Tableaux(['A',3],shape=[2,1])
sage: b = B(rows=[[1,2],[4]])
sage: b.lusztig_involution()
[[1, 4], [3]]
sage: b.to_tableau().schuetzenberger_involution(n=4)
[[1, 4], [3]]
sage: all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involution(n=4) for b in B)
True
sage: B = crystals.Tableaux(['D',4],shape=[1])
sage: [[b,b.lusztig_involution()] for b in B]

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

\begin{verbatim}
sage: R = WeylCharacterRing("A2", style="coroots")
sage: chiT = T.character(R); chiT
A2(0,0) + 2*A2(1,1) + A2(0,3) + A2(3,0) + A2(2,2)
sage: chiT in R
True
\end{verbatim}

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

\begin{verbatim}
sage: R = WeylCharacterRing("A3", style="coroots")
sage: T.character(R)
Traceback (most recent call last):
... ValueError: Weyl character ring does not have the right Cartan type
\end{verbatim}

demazure_character \((w, f=None)\)

Return the Demazure character associated to \(w\).

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 \texttt{sage.categories.regular_crystals.RegularCrystals.PARENTMethods.demazure_operator()}) to the highest weight element of the classical crystal. The simple Demazure operators \(D_i\) (see \texttt{sage.categories.regular_crystals.RegularCrystals.EMBEDMENTS.demazure_operator_simple()}) do not braid on the level of crystals, but on the level of characters they do. That is why it makes sense to input \(w\) either as a weight, a reduced word, or as an element of the underlying Weyl group.

EXAMPLES:

\begin{verbatim}
sage: T = crystals.Tableaux(['A',2], shape = [2,1])
sage: e = T.weight_lattice_realization().basis()
sage: weight = e[0] + 2*e[2]
sage: weight.reduced_word()
[2, 1]
sage: T.demazure_character(weight)
x1^2*x2 + x1*x2^2 + x1^2*x3 + x1*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)
\end{verbatim}

(continues on next page)
\[ x_1^2 x_2 + x_1 x_2^2 + x_1^2 x_3 + x_1 x_2 x_3 + x_2^2 x_3 \]

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)
\[ x_1^2 + x_1 x_2 + x_2^2 + x_1 + x_2 + \frac{x_1}{x_2} + \frac{1}{x_2} + \frac{1}{x_2^2} + 1 \]

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()))
\[ b_2(-1/2,1/2) + b_2(1/2,-1/2) + b_2(1/2,1/2) \]

REFERENCES:

• [De1974]
• [Ma2009]

class 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]

additional_structure ()

Return None.

Indeed, the category of classical crystals defines no additional structure: it only states that its objects are \( U_q(g) \)-crystals, where \( g \) is of finite type.

See also:

Category.additional_structure ()

EXAMPLES:

sage: ClassicalCrystals().additional_structure()

example (n=3)

Returns an example of highest weight crystals, as per Category.example () .

EXAMPLES:

sage: B = ClassicalCrystals().example(); B
Highest weight crystal of type A_3 of highest weight \( \omega_1 \)

super_categories ()

EXAMPLES:

sage: ClassicalCrystals().super_categories()
[Category of regular crystals,
 Category of finite crystals,
 Category of highest weight crystals]
3.17 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]

class DualObjects(category, *args)
Bases: sage.categories.dual.DualObjectsCategory

extra_super_categories()

Return the dual category.

EXAMPLES:

The category of coalgebras over the Rational Field is dual to the category of algebras over the same field:

sage: C = Coalgebras(QQ)
sage: C.dual()
Category of duals of coalgebras over Rational Field
sage: C.dual().super_categories()
[Category of algebras over Rational Field, Category of duals of vector spaces over Rational Field]

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

class 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 6 as a permutation group over Rational Field
sage: [a,b] = A.algebra_generators()
sage: a, a.coproduct()
(B[(1,2,3)], B[(1,2,3)] # B[(1,2,3)])
sage: b, b.coproduct()
(B[(1,3)], B[(1,3)] # B[(1,3)])

counit()

Returns the counit of self

EXAMPLES:
**class 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:**

```python
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral
→group of order 6 as a permutation group over Rational Field
sage: [a,b] = A.algebra_generators()
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:**

```python
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral
→group of order 6 as a permutation group over Rational Field
sage: [a,b] = A.algebra_generators()
sage: a, A.counit(a)
(B[(1,2,3)], 1)
sage: b, A.counit(b)
(B[(1,3)], 1)
```

**TODO**: implement some tests of the axioms of coalgebras, bialgebras and Hopf algebras using the counit.

**class Realizations**(category, *args)
Bases: `sage.categories.realizations.RealizationsCategory`

**class ParentMethods**

**coproduct_by_coercion**(x)
Return 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()

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()

counit_by_coercion(x)
Return the counit of x if counit_by_basis is not implemented.

EXAMPLES:

sage: sp = SymmetricFunctions(QQ).sp()
sage: sp.an_element()
2*sp[] + 2*sp[1] + 3*sp[2]
sage: sp.counit(sp.an_element())
2

sage: o = SymmetricFunctions(QQ).o()
sage: o.an_element()
2*o[] + 2*o[1] + 3*o[2]
sage: o.counit(o.an_element())
-1

class Super(base_category)
Bases: sage.categories.super_modules.SuperModulesCategory

extra_super_categories()

EXAMPLES:

sage: Coalgebras(ZZ).Super().extra_super_categories()
[Join of Category of graded modules over Integer Ring
 and Category of coalgebras over Integer Ring]
sage: Coalgebras(ZZ).Super().super_categories()
[Category of super modules over Integer Ring,
 Category of coalgebras over Integer Ring]

Compare this with the situation for bialgebras:

sage: Bialgebras(ZZ).Super().extra_super_categories()
[]
The category of bialgebras does not occur in these results, since super bialgebras are not bialgebras.

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

Meaning: a tensor product of coalgebras is a coalgebra

```python
WithBasis
alias of sage.categories.coalgebras_with_basis.CoalgebrasWithBasis
```

```python
class WithRealizations (category, *args)
Bases: sage.categories.with_realizations.WithRealizationsCategory
```

```python
coproduct (x)
Returns the coproduct of x.

EXAMPLES:

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

```python
counit (x)
Return the counit of x.

EXAMPLES:

```python
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
```

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

## 3.18 Coalgebras with basis

### class sage.categories.coalgebras_with_basis.CalgebrasWithBasis(base_category)

The category of coalgebras with a distinguished basis.

**EXAMPLES:**

```
sage: CoalgebrasWithBasis(ZZ)
Category of coalgebras with basis over Integer Ring
sage: sorted(CalgebrasWithBasis(ZZ).super_categories(), key=str)
[Category of coalgebras over Integer Ring, 
 Category of modules with basis over Integer Ring]
```

### class ElementMethods

```
coproduct_iterated(n=1)
```

Apply \( n \) coproducts to \( \text{self} \).

**Todo:** Remove dependency on \text{modules_with_basis} methods.

**EXAMPLES:**

```
sage: Psi = NonCommutativeSymmetricFunctions(QQ).Psi()
sage: Psi[2,2].coproduct_iterated(0)
Psi[2, 2]
sage: Psi[2,2].coproduct_iterated(3)

(continues on next page)
```
class ParentMethods

\textbf{coproduct()} \\
If \texttt{coproduct\_on\_basis()} is available, construct the coproduct morphism from \texttt{self} to \texttt{self} $\otimes$ \texttt{self} by extending it by linearity. Otherwise, use \texttt{coproduct\_by\_coercion()}, if available.

\textbf{EXAMPLES:}

```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral\_group of order 6 as a permutation group over Rational Field
sage: (a, b) = A.algebra\_generators()
sage: a, A.coproduct(a)
(B[(1,2,3)], B[(1,2,3)] $\otimes$ B[(1,2,3)])
sage: b, A.coproduct(b)
(B[(1,3)], B[(1,3)] $\otimes$ B[(1,3)])
```

\textbf{coproduct\_on\_basis}(\texttt{i}) \\
The coproduct of the algebra on the basis (optional).

\textbf{INPUT:}

- \texttt{i} – the indices of an element of the basis of \texttt{self}

Returns the coproduct of the corresponding basis elements If implemented, the coproduct of the algebra is defined from it by linearity.

\textbf{EXAMPLES:}

```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral\_group of order 6 as a permutation group over Rational Field
sage: (a, b) = A._group.gens()
sage: A.coproduct\_on\_basis(a)
B[(1,2,3)] $\otimes$ B[(1,2,3)]
```

\textbf{counit()} \\
If \texttt{counit\_on\_basis()} is available, construct the counit morphism from \texttt{self} to \texttt{self} $\otimes$ \texttt{self} by extending it by linearity

\textbf{EXAMPLES:}
```
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral
→
group of order 6 as a permutation group over Rational Field
sage: [a,b] = A.algebra_generators()
sage: a, A.counit(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 6 as a permutation group over Rational Field
sage: (a, b) = A._group.gens()
sage: A.counit_on_basis(a)
1
```

class Super ($base\_category$)
Bases: `sage.categories.super_modules.SuperModulesCategory`

3.19 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.
3.20 Commutative additive monoids

```python
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.
3.21 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:

```python
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.

3.22 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:

```python
sage: C = CommutativeAlgebraIdeals(QQ['x'])

sage: C
Category of commutative algebra ideals in Univariate Polynomial Ring in x over Rational Field

sage: C.algebra()
Univariate Polynomial Ring in x over Rational Field

sage: C.super_categories()
[Category of algebra ideals in Univariate Polynomial Ring in x over Rational Field]
```

algebra()

EXAMPLES:

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

sage: CommutativeAlgebraIdeals(QQ['x']).super_categories()
[Category of algebra ideals in Univariate Polynomial Ring in x over Rational Field]
```
3.23 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:

\begin{verbatim}
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]
\end{verbatim}

This is just a shortcut for:

\begin{verbatim}
sage: Algebras(QQ).Commutative()
Category of commutative algebras over Rational Field
\end{verbatim}

3.24 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:

\begin{verbatim}
sage: C = CommutativeRingIdeals(IntegerRing())
sage: C
Category of commutative ring ideals in Integer Ring
sage: CommutativeRingIdeals(ZZ).super_categories()
[Category of ring ideals in Integer Ring]
\end{verbatim}

3.25 Commutative rings

class sage.categories.commutative_rings.CommutativeRings(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of commutative rings

\begin{verbatim}
commutative rings with unity, i.e. rings with commutative * and a multiplicative identity
\end{verbatim}

EXAMPLES:

\begin{verbatim}
sage: C = CommutativeRings(); C
Category of commutative rings
sage: C.super_categories()
[Category of rings, Category of commutative monoids]
\end{verbatim}
class CartesianProducts (category, *args)
Bases: sage.categories.cartesian_product.CartesianProductsCategory

extra_super_categories()

Let Sage knows that Cartesian products of commutative rings is a commutative ring.

EXAMPLES:

```
sage: CommutativeRings().Commutative().CartesianProducts().extra_super_categories()
[Category of commutative rings]
sage: cartesian_product([ZZ, Zmod(34), QQ, GF(5)]) in CommutativeRings()
True
```

class ElementMethods

class Finite (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

Check that Sage knows that Cartesian products of finite commutative rings is a finite commutative ring.

EXAMPLES:

```
sage: cartesian_product([Zmod(34), GF(5)]) in Rings().Commutative().Finite()
True
```

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 \( R = \mathbb{Z}/n\mathbb{Z} \) the smallest element of each coset is sometimes called a coset leader. This function returns sorted lists so that the coset leader will always be the first element of the coset.

**INPUT:**
- \( q \) – an invertible element of the ring
- \( \text{cosets} \) – an optional lists of elements of self. If provided, the function only return the list of cosets that contain some element from \( \text{cosets} \).

**OUTPUT:**
A list of lists.

**EXAMPLES:**
Since the group of invertible elements of a finite field is cyclic, the set of squares is a particular case of cyclotomic coset:

```sage
sage: K = GF(25,'z')
sage: a = K.multiplicative_generator()
sage: R.<X> = PolynomialRing(K, 'X')
sage: a.minimal_polynomial('X')
X^3 + 2*X + 1
```

We compute some examples of minimal polynomials:

```sage
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
```

Cyclotomic cosets of fields are useful in combinatorial design theory to provide so called difference families (see Wikipedia article Difference_set and difference_family). This is illustrated on the following examples:

```sage
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
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, 5, 5, ..., 33, 34, 34, 34, 35, 35, 36, 36]
```

The method `cyclotomic_cosets` works on any finite commutative ring:
```python
sage: R = cartesian_product([GF(7), Zmod(14)])
sage: a = R((3,5))
sage: R.cyclotomic_cosets((3,5), [(1,1)])
[[[1, 1], (2, 11), (3, 5), (4, 9), (5, 3), (6, 13)]]
```

### 3.26 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:**

```python
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

**denominator()**

Return the denominator of this element normalized as a power of the uniformizer

**EXAMPLES:**

```python
sage: K = Qp(7)
sage: x = K(1/21)
sage: x.denominator()
7 + O(7^21)
sage: x = K(7)
sage: x.denominator()
1 + O(7^20)
```

Note that the denominator lives in the ring of integers:

```python
sage: x.denominator().parent()
7-adic Ring with capped relative precision 20
```

An error is raised when the input is indistinguishable from 0:

```python
sage: x = K(0,5); x
O(7^5)
sage: x.denominator()
Traceback (most recent call last):
  ...
ValueError: Cannot determine the denominator of an element indistinguishable from 0
```
valuation()

Return the valuation of this element.

EXAMPLES:

```python
sage: K = Qp(7)
sage: x = K(7); x
7 + O(7^21)
sage: x.valuation()
1
```

super_categories()

EXAMPLES:

```python
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:

```python
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
denominator()

Return the denominator of this element normalized as a power of the uniformizer

EXAMPLES:

```python
sage: K = Qp(7)
sage: x = K(1/21)
sage: x.denominator()
7 + O(7^21)
sage: x = K(7)
sage: x.denominator()
1 + O(7^20)
```

Note that the denominator lives in the ring of integers:

```python
sage: x.denominator().parent()
7-adic Ring with capped relative precision 20
```

An error is raised when the input is indistinguishable from 0:
lift_to_precision(absprec=None)

Return another element of the same parent with absolute precision at least absprec, congruent to this element modulo the precision of this element.

INPUT:
• absprec – an integer or None (default: None), the absolute precision of the result. If None, lifts to the maximum precision allowed.

Note: If setting absprec that high would violate the precision cap, raises a precision error. Note that the new digits will not necessarily be zero.

EXAMPLES:

sage: R = ZpCA(17)
sage: R(-1,2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1,15).lift_to_precision(10)
1 + O(17^15)
sage: R(1,15).lift_to_precision(30)
Traceback (most recent call last):
  ... 
PrecisionError: Precision higher than allowed by the precision cap.

sage: R(-1,2).lift_to_precision().precision_absolute() == R.precision_cap()
True

sage: R = Zp(5); c = R(17,3); c.lift_to_precision(8)
2 + 3*5 + O(5^8)
sage: c.lift_to_precision().precision_relative() == R.precision_cap()
True

valuation()

Return the valuation of this element.

EXAMPLES:

sage: R = Zp(7)
sage: x = R(7); x
7 + O(7^21)
sage: x.valuation()
1

super_categories()

EXAMPLES:

sage: CompleteDiscreteValuationRings().super_categories()
[Category of discrete valuation rings]
### 3.27 Complex reflection groups

**class** `sage.categories.complex_reflection_groups.ComplexReflectionGroups(s=None)`

Bases: `sage.categories.category_singleton.Category_singleton`

The category of complex reflection groups.

Let $V$ be a complex vector space. A complex reflection is an element of $GL(V)$ fixing an hyperplane pointwise and acting by multiplication by a root of unity on a complementary line.

A complex reflection group is a group $W$ that is (isomorphic to) a subgroup of some general linear group $GL(V)$ generated by a distinguished set of complex reflections.

The dimension of $V$ is the rank of $W$.

For a comprehensive treatment of complex reflection groups and many definitions and theorems used here, we refer to [LT2009]. See also Wikipedia article Reflection_group.

See also:

- `ReflectionGroup()` for usage examples of this category.

**EXAMPLES:**

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups()
Category of complex reflection groups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]
sage: ComplexReflectionGroups().all_super_categories()
[Category of complex reflection groups,
 Category of complex reflection or generalized coxeter groups,
 Category of groups,
 Category of monoids,
 Category of finitely generated semigroups,
 Category of semigroups,
 Category of finitely generated magmas,
 Category of inverse unital magmas,
 Category of unital magmas,
 Category of magmas,
 Category of enumerated sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

An example of a reflection group:

```python
sage: W = ComplexReflectionGroups().example(); W
5-colored permutations of size 3
W is in the category of complex reflection groups:

```python
sage: W in ComplexReflectionGroups()
True
```

**Finite**

alias of `sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups`

3.27. Complex reflection groups 203
class ParentMethods

rank()

Return the rank of self.

The rank of self is the dimension of the smallest faithfull reflection representation of self.

EXAMPLES:

```
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: W.rank()
3
```

additional_structure()

Return None.

Indeed, all the structure complex reflection groups have in addition to groups (simple reflections, ...) is already defined in the super category.

See also:

Category.additional_structure()

EXAMPLES:

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().additional_structure()
```

eexample()

Return an example of a complex reflection group.

EXAMPLES:

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().example()
5-colored permutations of size 3
```

super_categories()

Return the super categories of self.

EXAMPLES:

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]
```

3.28 Common category for Generalized Coxeter Groups or Complex Reflection Groups

class sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups(s=None)

Bases: sage.categories.category_singleton.Category_singleton

The category of complex reflection groups or generalized Coxeter groups.
Finite Coxeter groups can be defined equivalently as groups generated by reflections, or by presentations. Over the last decades, the theory has been generalized in both directions, leading to the study of (finite) complex reflection groups on the one hand, and (finite) generalized Coxeter groups on the other hand. Many of the features remain similar, yet, in the current state of the art, there is no general theory covering both directions.

This is reflected by the name of this category which is about factoring out the common code, tests, and declarations.

A group in this category has:

- A distinguished finite set of generators \((s_i)_{I}\), called simple reflections. The set \(I\) is called the index set. The name “reflection” is somewhat of an abuse as they can have higher order; still, they are all of finite order: \(s_i^k = 1\) for some \(k\).
- A collection of distinguished reflections which are the conjugates of the simple reflections. For complex reflection groups, they are in one-to-one correspondence with the reflection hyperplanes and share the same index set.
- A collection of reflections which are the conjugates of all the non trivial powers of the simple reflections.

The usual notions of reduced words, length, irreducibility, etc can be canonically defined from the above.

The following methods must be implemented:

- `ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods.index_set()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods.simple_reflection()`

Optionally one can define analog methods for distinguished reflections and reflections (see below).

At least one of the following methods must be implemented:

- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection_left()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods.apply_simple_reflection_right()`
- `ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods._mul_()`

It’s recommended to implement either `_mul_` or both `apply_simple_reflection_left` and `apply_simple_reflection_right`.

See also:

- `complex_reflection_groups.ComplexReflectionGroups`
- `generalized_coxeter_groups.GeneralizedCoxeterGroups`

EXAMPLES:

```
sage: from sage.categories.complex_reflection_or_generalized_coxeter_groups import ComplexReflectionOrGeneralizedCoxeterGroups

sage: C = ComplexReflectionOrGeneralizedCoxeterGroups(); C
Category of complex reflection or generalized coxeter groups
sage: C.super_categories()
[Category of finitely generated enumerated groups]

sage: C.required_methods()
```
class ElementMethods

apply_conjugation_by_simple_reflection(i)

Conjugate self by the i-th simple reflection.

EXAMPLES:

sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.apply_conjugation_by_simple_reflection(1).reduced_word()
[3, 2]

apply_reflections(word, side='right', word_type='all')

Return the result of the (left/right) multiplication of self by word.

INPUT:
• word — a sequence of indices of reflections
• side — (default: 'right') indicates multiplying from left or right
• word_type — (optional, default: 'all'): either 'simple', 'distinguished', or 'all'

EXAMPLES:

sage: W = ReflectionGroup((1,1,3))
# optional - gap3
sage: W.one().apply_reflections([1])
# optional - gap3
(1,4)(2,3)(5,6)
sage: W.one().apply_reflections([2])
# optional - gap3
(1,3)(2,5)(4,6)
sage: W.one().apply_reflections([2,1])
# optional - gap3
(1,2,6)(3,4,5)

sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_reflections([0,1], word_type='simple')
(2, 3, 1, 0)
sage: w
(1, 2, 3, 0)
sage: w.apply_reflections([0,1], side='left', word_type='simple')
(0, 1, 3, 2)

sage: W = WeylGroup("A3", prefix='s')
sage: w = W.an_element(); w
s1*s2*s3
sage: AS = W.domain()
sage: r1 = AS.roots()[4]
sage: r1
(0, 1, 0, -1)
apply_simple_reflection \(i, \text{side='right'}\)

Return \(self\) multiplied by the simple reflection \(s[i]\).

**INPUT:**

- \(i\) – an element of the index set
- \(\text{side} – \text{(default: "right") "left" or "right"}

This default implementation simply calls \(\text{apply_simple_reflection\_left()}\) or \(\text{apply_simple_reflection\_right()}\).

**EXAMPLES:**

```python
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, \(\text{side} = \text{"right"}\):
Some tests with a complex reflection group:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: W = ComplexReflectionGroups().example(); W
5-colored permutations of size 3
sage: w = W.an_element(); w
[[1, 0, 0], [3, 1, 2]]
sage: w.apply_simple_reflection(1, side="left")
[[0, 1, 0], [1, 3, 2]]
sage: w.apply_simple_reflection(2, side="left")
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection(3, side="left")
[[1, 0, 1], [3, 1, 2]]
sage: w.apply_simple_reflection(1, side="right")
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection(2, side="right")
[[1, 0, 0], [2, 1, 3]]
sage: w.apply_simple_reflection(3, side="right")
[[2, 0, 0], [3, 1, 2]]
```

**apply_simple_reflection_left** (*i*)

Return `self` multiplied by the simple reflection `s[i]` on the left.

This low level method is used intensively. Coxeter groups are encouraged to override this straightforward implementation whenever a faster approach exists.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection_left(0)
(0, 2, 3, 1)
sage: w.apply_simple_reflection_left(1)
(2, 1, 3, 0)
sage: w.apply_simple_reflection_left(2)
(1, 3, 2, 0)
```

**apply_simple_reflection_right** (*i*)

Return `self` multiplied by the simple reflection `s[i]` on the right.

**EXAMPLES:**

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: W = ComplexReflectionGroups().example()
sage: w = W.an_element(); w
[[1, 0, 0], [3, 1, 2]]
sage: w.apply_simple_reflection_left(1)
[[0, 1, 0], [1, 3, 2]]
sage: w.apply_simple_reflection_left(2)
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection_left(3)
[[1, 0, 1], [3, 1, 2]]
```
This low level method is used intensively. Coxeter groups are encouraged to override this straightforward implementation whenever a faster approach exists.

**EXAMPLES:**

```python
sage: W=CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflection_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)
```

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: W = ComplexReflectionGroups().example()
sage: w = W.an_element(); w
[[1, 0, 0], [3, 1, 2]]
sage: w.apply_simple_reflection_right(1)
[[1, 0, 0], [3, 2, 1]]
sage: w.apply_simple_reflection_right(2)
[[1, 0, 0], [2, 1, 3]]
sage: w.apply_simple_reflection_right(3)
[[2, 0, 0], [3, 1, 2]]
```

**apply_simple_reflections** *(word, side='right', type='simple')*

Return the result of the (left/right) multiplication of `self` by `word`.

**INPUT:**

- `word` – a sequence of indices of simple reflections
- `side` – (default: 'right') indicates multiplying from left or right

This is a specialized implementation of `apply_reflections()` for the simple reflections. The rationale for its existence are:

- It can take advantage of `apply_simple_reflection`, which often is less expensive than computing a product.
- It reduced burden on implementations that would want to provide an optimized version of this method.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1])
(2, 3, 1, 0)
sage: w
(1, 2, 3, 0)
sage: w.apply_simple_reflections([0,1],side='left')
(0, 1, 3, 2)
```

**inverse()**

Return the inverse of `self`.

**EXAMPLES:**

```python
sage: W = WeylGroup(['B',7])
sage: w = W.an_element()
```
sage: w = 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]

is_reflection()
Return whether self is a reflection.

EXAMPLES:

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: [t.is_reflection() for t in W.reflections()]  # optional - gap3
[True, True, True, True, True]

reflection_length()
Return the reflection length of self.
This is the minimal length of a factorization of self into reflections.

EXAMPLES:

sage: W = ReflectionGroup((1,1,2))  # optional - gap3
sage: sorted([t.reflection_length() for t in W])  # optional - gap3
[0, 1]

class Irreducible(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom
class ParentMethods

irreducible_components()

Return a list containing all irreducible components of self as finite reflection groups.

EXAMPLES:

```sage
W = ColoredPermutations(4, 3)
sage: W.irreducible_components()
[4-colored permutations of size 3]
```

class ParentMethods
distinguished_reflection(i)

Return the \(i\)-th distinguished reflection of self.

INPUT:

- \(i\) – an element of the index set of the distinguished reflections.

See also:

- distinguished_reflections()
- hyperplane_index_set()

EXAMPLES:

```sage
W = ReflectionGroup((1,1,4), hyperplane_index_set=('a','b','c','d', 'e','f'))
sage: for i in W.hyperplane_index_set():
    print('%s %s'%(i, W.distinguished_reflection(i)))
  a (1,7)(2,4)(5,6)(8,10)(11,12)
  b (1,4)(2,8)(3,5)(7,10)(9,11)
  c (2,5)(3,9)(4,6)(8,11)(10,12)
  d (1,8)(2,7)(3,6)(4,10)(9,12)
  e (1,6)(2,9)(3,8)(5,11)(7,12)
  f (1,11)(3,10)(4,9)(5,7)(6,12)
```

distinguished_reflections()

Return a finite family containing the distinguished reflections of self, indexed by hyperplane_index_set().

A distinguished reflection is a conjugate of a simple reflection. For a Coxeter group, reflections and distinguished reflections coincide. For a Complex reflection groups this is a reflection acting on the complement of the fixed hyperplane \(H\) as \(\exp(2\pi i/n)\), where \(n\) is the order of the reflection subgroup fixing \(H\).

See also:

- distinguished_reflection()
- hyperplane_index_set()

EXAMPLES:

```sage
W = ReflectionGroup((1,1,3))
sage: distinguished_reflections = W.distinguished_reflections()
sage: for index in sorted(distinguished_reflections.keys()):
    print('%s %s'%(index, W.distinguished_reflection(index)))
  a (1,3)(2,4)(5,6)(8,10)(11,12)
  b (1,4)(2,8)(3,5)(7,10)(9,11)
  c (2,5)(3,9)(4,6)(8,11)(10,12)
  d (1,8)(2,7)(3,6)(4,10)(9,12)
  e (1,6)(2,9)(3,8)(5,11)(7,12)
  f (1,11)(3,10)(4,9)(5,7)(6,12)
```
from_reduced_word\(\text{word, word_type='simple'}\)

Return an element of \(\text{self}\) from its (reduced) word.

INPUT:

- **word** – a list (or iterable) of elements of the index set of \(\text{self}\) (resp. of the distinguished or of all reflections)
- **word_type** – (optional, default: 'simple'): either 'simple', 'distinguished', or 'all'

If \(\text{word}\) is \([i_1, i_2, \ldots, i_k]\), then this returns the corresponding product of simple reflections \(s_{i_1} s_{i_2} \cdots s_{i_k}\).
If `word_type` is 'distinguished' (resp. 'all'), then the product of the distinguished reflections (resp. all reflections) is returned.

**Note:** The main use case is for constructing elements from reduced words, hence the name of this method. However, the input word need not be reduced.

**See also:**
- `index_set()`
- `reflection_index_set()`
- `hyperplane_index_set()`
- `apply_simple_reflections()`
- `reduced_word()`
- `_test_reduced_word()`

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: W.from_reduced_word([0,2,0,1])
(0, 3, 1, 2)
sage: W.from_reduced_word((0,2,0,1))
(0, 3, 1, 2)
sage: s[0]*s[2]*s[0]*s[1]
(0, 3, 1, 2)
```

We now experiment with the different values for `word_type` for the colored symmetric group:

```python
sage: W = ColoredPermutations(1,4)
sage: W.from_reduced_word((1,2,1,2,1,2))
[[0, 0, 0, 0], [1, 2, 3, 4]]
sage: W.from_reduced_word((1, 2, 3)).reduced_word()
[1, 2, 3]
sage: W = WeylGroup("A3", prefix='s')
sage: AS = W.domain()
sage: r1 = AS.roots()[4]
sage: r1
(0, 1, 0, -1)
sage: r2 = AS.roots()[5]
sage: r2
(0, 0, 1, -1)
sage: W.from_reduced_word([r1, r2], word_type='all')
s3*s2
```

```python
sage: W = WeylGroup("G2", prefix='s')
sage: W.from_reduced_word(W.domain().positive_roots(), word_type='all')
s1*s2
```

```
sage: W = ReflectionGroup((1,1,4)) # optional - gap3
sage: W.from_reduced_word([1,2,3], word_type='all').reduced_word() # optional - gap3
[1, 2, 3]
```

(continues on next page)

(continued from previous page)

```python
sage: W.from_reduced_word([1,2,3], word_type='all').reduced_word_in_reflections()  # optional - gap3
[1, 2, 3]
sage: W.from_reduced_word([1,2,3]).reduced_word_in_reflections()  # optional - gap3
[1, 2, 3]
```

**group_generators()**

Return the simple reflections of *self*, as distinguished group generators.

See also:

- `simple_reflections()`
- `Groups.ParentMethods.group_generators()`
- `Semigroups.ParentMethods.semigroup_generators()`

**EXAMPLES:**

```python
sage: D10 = FiniteCoxeterGroups().example(10)
sage: D10.group_generators()  
Finite family {1: (1,), 2: (2,)}
sage: SymmetricGroup(5).group_generators()  
Finite family {1: (1,2), 2: (2,3), 3: (3,4), 4: (4,5)}
sage: W = ColoredPermutations(3,2)
sage: W.group_generators()  
Finite family {1: [[0, 0], [2, 1]], 2: [[0, 1], [1, 2]]}
```

The simple reflections are also semigroup generators, even for an infinite group:

```python
sage: W = WeylGroup(['A', 2, 1])
sage: W.semigroup_generators()  
Finite family {0: [-1  1  1]  
[ 0  1  0]  
[ 0  0  1], 1: [ 1  0  0]  
[ 1 -1  1]  
[ 0  0  1], 2: [ 1  0  0]  
[ 0  1  0]  
[ 1  1 -1]}
```

**hyperplane_index_set()**

Return the index set of the distinguished reflections of *self*.

This is also the index set of the reflection hyperplanes of *self*, hence the name. This name is slightly abusive since the concept of reflection hyperplanes is not defined for all generalized Coxeter groups. However for all practical purposes this is only used for complex reflection groups, and there this is the desirable name.

See also:

- `distinguished_reflection()`
- `distinguished_reflections()`
EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,4)) # optional - gap3
sage: W.hyperplane_index_set() # optional - gap3
(1, 2, 3, 4, 5, 6)
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=[1,3,'asdf',7,9, 11]) # optional - gap3
sage: W.hyperplane_index_set() # optional - gap3
(1, 3, 'asdf', 7, 9, 11)
sage: W = ReflectionGroup((1,1,4), hyperplane_index_set=('a','b','c','d', 'e','f')) # optional - gap3
sage: W.hyperplane_index_set() # optional - gap3
('a', 'b', 'c', 'd', 'e', 'f')
```

`index_set()`

Return the index set of the simple reflections of `self`, as a list (or iterable).

See also:

- `simple_reflection()`
- `simple_reflections()`

EXAMPLES:

```python
sage: W = CoxeterGroups().Finite().example(); W
The 5-th dihedral group of order 10
sage: W.index_set()
(1, 2)
sage: W = ColoredPermutations(1, 4)
(1, 2, 3)
sage: W = ReflectionGroup((1,1,4), index_set=[1,3,'asdf']) # optional - gap3
sage: W.index_set() # optional - gap3
(1, 3, 'asdf')
sage: W = ReflectionGroup((1,1,4), index_set=('a','b','c')) # optional - gap3
sage: W.index_set() # optional - gap3
('a', 'b', 'c')
```

`irreducible_component_index_sets()`

Return a list containing the index sets of the irreducible components of `self` as finite reflection groups.

EXAMPLES:

```python
sage: W = ReflectionGroup([1,1,3], [3,1,3], 4); W # optional - gap3
Reducible complex reflection group of rank 7 and type A2 x G(3,1,3) x ST4
sage: sorted(W.irreducible_component_index_sets()) # optional - gap3
[[[1, 2], [3, 4, 5], [6, 7]]
```

ALGORITHM:

Take the connected components of the graph on the index set with edges \((i,j)\), where \(s[i]\) and \(s[j]\) do not commute.

`irreducible_components()`

Return the irreducible components of `self` as finite reflection groups.

EXAMPLES:
sage: W = ReflectionGroup([1,1,3], [3,1,3], 4)  # optional - gap3
sage: W.irreducible_components()  # optional - gap3
[Irreducible real reflection group of rank 2 and type A2,
 Irreducible complex reflection group of rank 3 and type G(3,1,3),
 Irreducible complex reflection group of rank 2 and type ST4]

is_irreducible()  
Return True if self is irreducible.
EXAMPLES:

sage: W = ColoredPermutations(1,3); W
1-colored permutations of size 3
sage: W.is_irreducible()
True

sage: W = ReflectionGroup((1,1,3), (2,1,3)); W  # optional - gap3
Reducible real reflection group of rank 5 and type A2 x B3
sage: W.is_irreducible()  # optional - gap3
False

is_reducible()  
Return True if self is not irreducible.
EXAMPLES:

sage: W = ColoredPermutations(1,3); W
1-colored permutations of size 3
sage: W.is_reducible()
False

sage: W = ReflectionGroup((1,1,3), (2,1,3)); W  # optional - gap3
Reducible real reflection group of rank 5 and type A2 x B3
sage: W.is_reducible()  # optional - gap3
True

number_of_irreducible_components()  
Return the number of irreducible components of self.
EXAMPLES:

sage: SymmetricGroup(3).number_of_irreducible_components()
1

sage: ColoredPermutations(1,3).number_of_irreducible_components()
1

sage: ReflectionGroup((1,1,3), (2,1,3)).number_of_irreducible_components()  # optional - gap3
2

number_of_simple_reflections()  
Return the number of simple reflections of self.
EXAMPLES:

sage: W = ColoredPermutations(1,3)
sage: W.number_of_simple_reflections()
reflection\( (i) \)

Return the \( i \)-th reflection of \self.

For \( i \) in \( 1, \ldots, N \), this gives the \( i \)-th reflection of \self.

See also:

• \texttt{reflections\_index\_set()}  
• \texttt{reflections()}  

EXAMPLES:

\begin{verbatim}
sage: W = ReflectionGroup((1,1,4)) # optional - gap3 sage: for i in W.reflection_index_set(): # optional - gap3 ....: print('%s %s'%(i, W.reflection(i))) # optional - gap3 1 (1,7)(2,4)(5,6)(8,10)(11,12) 2 (1,4)(2,8)(3,5)(7,10)(9,11) 3 (2,5)(3,9)(4,6)(8,11)(10,12) 4 (1,8)(2,7)(3,6)(4,10)(9,12) 5 (1,6)(2,9)(3,8)(5,11)(7,12) 6 (1,11)(3,10)(4,9)(5,7)(6,12)
\end{verbatim}

reflection\_index\_set()

Return the index set of the reflections of \self.

See also:

• \texttt{reflection()}  
• \texttt{reflections()}  

EXAMPLES:

\begin{verbatim}
sage: W = ReflectionGroup((1,1,4)) # optional - gap3 sage: W.reflection_index_set() # optional - gap3 (1, 2, 3, 4, 5, 6)  # optional - gap3 sage: W = ReflectionGroup((1,1,4), reflection_index_set=[1,3,'asdf',7,9,-11]) # optional - gap3 sage: W.reflection_index_set() # optional - gap3 (1, 3, 'asdf', 7, 9, 11) sage: W.reflection_index_set() # optional - gap3 ('a', 'b', 'c', 'd', 'e', 'f') sage: W.reflection_index_set() # optional - gap3 ('a', 'b', 'c', 'd', 'e', 'f')
\end{verbatim}

reflections()

Return a finite family containing the reflections of \self, indexed by \texttt{reflection\_index\_set()}.

See also:
• `reflection()`  
• `reflection_index_set()`  

EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):
    ....: print('%s %s'%(index, reflections[index]))  # optional - gap3
  1 (1,4)(2,3)(5,6)
  2 (1,3)(2,5)(4,6)
  3 (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((1,1,3), reflection_index_set=['a','b','c'])  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):
    ....: print('%s %s'%(index, reflections[index]))  # optional - gap3
  a (1,4)(2,3)(5,6)
  b (1,3)(2,5)(4,6)
  c (1,5)(2,4)(3,6)

sage: W = ReflectionGroup((3,1,1))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):
    ....: print('%s %s'%(index, reflections[index]))  # optional - gap3
  1 (1,2,3)
  2 (1,3,2)

sage: W = ReflectionGroup((1,1,3), (3,1,2))  # optional - gap3
sage: reflections = W.reflections()  # optional - gap3
sage: for index in sorted(reflections.keys()):
    ....: print('%s %s'%(index, reflections[index]))  # optional - gap3
  1 (1,6)(2,5)(7,8)
  2 (1,5)(2,7)(6,8)
  3 (3,9,15)(4,10,16)(12,17,23)(14,18,24)(20,25,29)(21,22,26)(27,28,30)
  5 (1,7)(2,6)(5,8)
```

**semigroup_generators()**

Return the simple reflections of `self`, as distinguished group generators.

See also:

• `simple_reflections()`  
• `Groups.ParentMethods.group_generators()`  
• `Semigroups.ParentMethods.semigroup_generators()`  

EXAMPLES:
The simple reflections are also semigroup generators, even for an infinite group:

```
sage: W = WeylGroup("A",2,1)
sage: W.semigroup_generators()
Finite family {0: [-1 1 1]
[ 0 1 0]
[ 0 0 1],
1: [ 1 0 0]
[ 1 -1 1]
[ 0 0 1],
2: [ 1 0 0]
[ 0 1 0]
[ 1 1 -1]}
```

```
sage: simple_reflection(i)
Return the i-th simple reflection $s_i$ of self.

INPUT:
• i—an element from the index set

See also:
• index_set()
• simple_reflections()

EXAMPLES:
```
sage: W = CoxeterGroups().example()
sage: W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: s[1]
(0, 2, 1, 3)
```

```
sage: W = ReflectionGroup((1,1,4), index_set=[1,3,'asdf'])  # optional - gap3
sage: for i in W.index_set():  # optional - gap3
    print('%s %s'%(i, W.simple_reflection(i)))  # optional - gap3
1 (1,7) (2,4) (5,6) (8,10) (11,12)
3 (1,4) (2,8) (3,5) (7,10) (9,11)
asdf (2,5) (3,9) (4,6) (8,11) (10,12)
```

```
sage: simple_reflection_orders()
Return the orders of the simple reflections.
```
EXAMPLES:

```python
sage: W = WeylGroup(['B',3])
sage: W.simple_reflection_orders()
[2, 2, 2]
sage: W = CoxeterGroup(['C',4])
sage: W.simple_reflection_orders()
[2, 2, 2, 2]
sage: SymmetricGroup(5).simple_reflection_orders()
[2, 2, 2, 2]
sage: C = ColoredPermutations(4, 3)
sage: C.simple_reflection_orders()
[2, 2, 4]
```

**simple_reflections()**

Return the simple reflections \((s_i)_{i \in I}\) of `self` as a family indexed by `index_set()`.

See also:

- `simple_reflection()`
- `index_set()`

**EXAMPLES:**

For the symmetric group, we recognize the simple transpositions:

```python
sage: W = SymmetricGroup(4); W
Symmetric group of order 4! as a permutation group
sage: s = W.simple_reflections()
sage: s
Finite family {1: (1,2), 2: (2,3), 3: (3,4)}
sage: s[1]
(1,2)
sage: s[2]
(2,3)
sage: s[3]
(3,4)
```

Here are the simple reflections for a colored symmetric group and a reflection group:

```python
sage: W = ColoredPermutations(1,3)
sage: W.simple_reflections()
Finite family {1: [[0, 0, 0], [2, 1, 3]], 2: [[0, 0, 0], [1, 3, 2]]}

sage: W = ReflectionGroup((1,1,3), index_set=['a','b']) # optional - gap3
sage: W.simple_reflections() # optional - gap3
Finite family {'a': (1,4)(2,3)(5,6), 'b': (1,3)(2,5)(4,6)}
```

This default implementation uses `index_set()` and `simple_reflection()`.

**some_elements()**

Implement `Sets.ParentMethods.some_elements()` by returning some typical elements of `self`.

The result is currently composed of the simple reflections together with the unit and the result of `an_element()`.

**EXAMPLES:**

class SubcategoryMethods

Irreducible()
Return the full subcategory of irreducible objects of self.

A complex reflection group, or generalized coxeter group is reducible if its simple reflections can be
split in two sets $X$ and $Y$ such that the elements of $X$ commute with that of $Y$. In particular, the
group is then direct product of $\langle X \rangle$ and $\langle Y \rangle$. It’s irreducible otherwise.

EXAMPLES:

sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Irreducible()
Category of irreducible complex reflection groups
sage: CoxeterGroups().Irreducible()
Category of irreducible coxeter groups

super_categories()
Return the super categories of self.

EXAMPLES:

sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().super_categories()
[Category of complex reflection or generalized coxeter groups]

3.29 Coxeter Group Algebras

class sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras(category, *args)

Bases: sage.categories.algebra_functor.AlgebrasCategory

class ParentMethods

demazure_lusztig_eigenvectors(q1, q2)
Return the family of eigenvectors for the Cherednik operators.
INPUT:
• `self` – a finite Coxeter group $W$
• $q_1,q_2$ – two elements of the ground ring $K$

The affine Hecke algebra $H_{q_1,q_2}(\tilde{W})$ acts on the group algebra of $W$ through the Demazure-Lusztig operators $T_i$. Its Cherednik operators $Y^\lambda$ can be simultaneously diagonalized as long as $q_1/q_2$ is not a small root of unity [HST2008].

This method returns the family of joint eigenvectors, indexed by $W$.

See also:
• `demazure_lusztig_operators()`
• `sage.combinat.root_system.hecke_algebra_representation.CherednikOperatorsEigenvectors`

EXAMPLES:
```
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))*2121 + ((-q2)/(-q1+q2))*121 - 212 + 12
```

demazure_lusztig_operator_on_basis ($w$, $i$, $q_1$, $q_2$, side='right')
Return the result of applying the $i$-th Demazure Lusztig operator on $w$.

INPUT:
• $w$ – an element of the Coxeter group
• $i$ – an element of the index set
• $q_1,q_2$ – two elements of the ground ring
• $\text{bar}$ – a boolean (default False)

See `demazure_lusztig_operators()` for details.

EXAMPLES:
```
sage: W = WeylGroup(['B',3])
sage: W.element_class._repr_=lambda x: ''.join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: w = W.an_element()
sage: KW.demazure_lusztig_operator_on_basis(w, 0, q1, q2)  # (-q2)*323123 + (q1+q2)*123
sage: KW.demazure_lusztig_operator_on_basis(w, 1, q1, q2)  # q1*1231
sage: KW.demazure_lusztig_operator_on_basis(w, 2, q1, q2)  # q1*1232
sage: KW.demazure_lusztig_operator_on_basis(w, 3, q1, q2)  # (q1+q2)*123 + (-q2)*12
```

At $q_1 = 1$ and $q_2 = 0$ we recover the action of the isobaric divided differences $\pi_i$:  

222 Chapter 3. Individual Categories
demazure_lusztig_operators \((q1, q2, \text{side}=\text{'right'}, \text{affine=}True)\)

Return the Demazure Lusztig operators acting on \(self\).

INPUT:

- \(q1, q2\) — two elements of the ground ring \(K\)
- \text{side} — "left" or "right" (default: "right"); which side to act upon
- \text{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 \(\pi_i\) (see \text{CoxeterGroups.ElementMethods.simple_projection()}\)) and the simple reflection \(s_i\) so that \(T_i\) has eigenvalues \(q1\) and \(q2\):

\[(q1 + q2)\pi_i - q2s_i.\]

The Demazure-Lusztig operators give the usual representation of the operators \(T_i\) of the \(q1, q2\) Hecke algebra associated to the Coxeter group.

For a finite Coxeter group, and if \text{affine=}True, the Demazure-Lusztig operators \(T_1, \ldots, T_n\) are completed by \(T_0\) to implement the level 0 action of the affine Hecke algebra.

EXAMPLES:

```python
sage: W = WeylGroup(["B",3])
sage: W.element_class._repr_=lambda x: ".join(str(i) for i in x.reduced_word())
sage: K = QQ['q1,q2']
sage: q1, q2 = K.gens()
sage: KW = W.algebra(K)
sage: T = KW.demazure_lusztig_operators(q1, q2, affine=True)
sage: x = KW.monomial(W.an_element()); x
123
sage: T[0](x)
(-q2)*323123 + (q1+q2)*123
sage: T[1](x)
q1*1231
sage: T[2](x)
q1*1232
sage: T[3](x)
(q1+q2)*123 + (-q2)*12
```

(continues on next page)
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.

### 3.30 Coxeter Groups

```python
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:
```

```python
c Sage: C = CoxeterGroups(); C
Category of coxeter groups

c Sage: C.super_categories()
[Category of generalized coxeter groups]

c Sage: W = C.example(); W
The symmetric group on {0, ..., 3}

c Sage: W.simple_reflections()
Finite family {0: (1, 0, 2, 3), 1: (0, 2, 1, 3), 2: (0, 1, 3, 2)}
```

Here are some further examples:

```python
c Sage: FiniteCoxeterGroups().example()
The 5-th dihedral group of order 10

c Sage: FiniteWeylGroups().example()
The symmetric group on {0, ..., 3}

c Sage: WeylGroup(['B', 3])
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)

c Sage: S4 = SymmetricGroup(4); S4
Symmetric group of order 4! as a permutation group

c Sage: S4 in CoxeterGroups().Finite()
True
```

Those will eventually be also in this category:

```python
c Sage: DihedralGroup(5)
Dihedral group of order 10 as a permutation group
```
Todo: add a demo of usual computations on Coxeter groups.

See also:

- `sage.combinat.root_system`
- `WeylGroups`
- `GeneralizedCoxeterGroups`

Warning: It is assumed that morphisms in this category preserve the distinguished choice of simple reflections. In particular, subobjects in this category are parabolic subgroups. In this sense, this category might be better named Coxeter Systems. In the long run we might want to have two distinct categories, one for Coxeter groups (with morphisms being just group morphisms) and one for Coxeter systems:

```python
sage: CoxeterGroups().is_full_subcategory(Groups())
False
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: CoxeterGroups().is_full_subcategory(GeneralizedCoxeterGroups())
True
```

Algebras

alias of `sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras`

class `ElementMethods`

`absolute_covers()`

Return the list of covers of `self` in absolute order.

See also:

`absolute_length()`

EXAMPLES:

```python
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_covers()
[0 0 1 0]
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[0 1 0 0]
[0 1 0 0]
[0 1 0 0]
[0 0 1 0]

sage: w1.absolute_covers()
[0 1 0 0]
[1 0 0 0]
[0 0 0 1]
[0 1 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
[1 0 0 0]
```

`absolute_le(other)`

Return whether `self` is smaller than `other` in the absolute order.

A general reflection is an element of the form $w s_i w^{-1}$, where $s_i$ is a simple reflection. The absolute order is defined analogously to the weak order but using general reflections rather than just simple reflections.

This partial order can be used to define noncrossing partitions associated with this Coxeter group.
See also:

`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.

For permutations in the symmetric groups, the absolute length is the size minus the number of its disjoint cycles.

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
sage: W = SymmetricGroup(4)
sage: s = W.simple_reflections()
sage: (s[3]*s[2]*s[1]).absolute_length()
3
```

`apply_demazure_product (element, side='right', length_increasing=True)`

Returns the Demazure or 0-Hecke product of `self` with another Coxeter group element.


INPUT:

- `element` – either an element of the same Coxeter group as `self` or a tuple or a list (such as a reduced word) of elements from the index set of the Coxeter group.
- `side` – ‘left’ or ‘right’ (default: ‘right’); the side of `self` on which the element should be applied. If `side` is ‘left’ then the operation is applied on the left.
- `length_increasing` – a boolean (default True) whether to act length increasingly or decreasingly

EXAMPLES:

```
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')
```

(continues on next page)
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 \(\pi_i\) (resp. \(\pi_i'\)) on \(self\).

See `CoxeterGroups.ParentMethods.simple_projections()` for the definition of the simple projections.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: w = W.an_element()
sage: w
(1, 2, 3, 0)
sage: w.apply_simple_projection(2)
(1, 2, 3, 0)
sage: w.apply_simple_projection(2, length_increasing=False)
(1, 2, 0, 3)
sage: W = WeylGroup(['C',4],prefix="s")
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
```

binary_factorizations \((predicate=The constant function \(\ldots\rightarrow True\))\)

Return the set of all the factorizations \(self = uv\) such that \(l(self) = l(u) + l(v)\).

Iterating through this set is Constant Amortized Time (counting arithmetic operations in the Coxeter group as constant time) complexity, and memory linear in the length of \(self\).

One can pass as optional argument a predicate \(p\) such that \(p(u)\) implies \(p(u')\) for any \(u\) left factor of \(self\) and \(u'\) left factor of \(u\). Then this returns only the factorizations \(self = uv\) such \(p(u)\) holds.

**EXAMPLES:**

We construct the set of all factorizations of the maximal element of the group:

```python
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:
The number of factorizations of the elements just below the maximal element:

```python
sage: [[s[i]*w0).binary_factorizations().cardinality() for i in [1,2,3]]
[12, 12, 12]
sage: w0.binary_factorizations(lambda u: False).cardinality()
0
```

**bruhat_le** *(other)*

Bruhat comparison

**INPUT:**
- other – an element of the same Coxeter group

**OUTPUT:** a boolean

Returns whether *self* \(\leq\) *other* in the Bruhat order.

**EXAMPLES:**

```python
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 Möbius function for the Bruhat order. J. Algebraic Combin. 25 (2007), no. 2, 141–148, Proposition 1.1.

Complexity: \(O(l \ast c)\), where \(l\) is the minimum of the lengths of *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.

**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 \(\{w s_i, u_1 s_i, u_2 s_i, \ldots, u_j s_i\}\), where the \(u_k\) are elements that \(w s_i\) covers that also do not have a descent at *i*.

**EXAMPLES:**

```python
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]]
```

(continues on next page)
We now show how to construct the Bruhat poset:

```python
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:

```python
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 \(\text{self}\) and \(r\) is the reflection such that \(\text{self} = vr\).

**ALGORITHM:**

See **bruhat_lower_covers()**

**EXAMPLES:**

```python
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 \(\text{self}\) 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 \(\text{self}\). If \(w\) has no right descent \(i\), then \(ws_i\) is a cover; if \(w\) has a descent at \(i\), then \(u_js_i\) is a cover of \(w\) where \(u_j\) is a cover of \(ws_i\).

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',3,1], prefix="s")
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]
```

(continues on next page)
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 \(self = vr\).

ALGORITHM:
See `bruhat_upper_covers()`

EXAMPLES:

```python
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, s1*s2*s3*s4*s3*s2*s1)]
```

canonical_matrix()
Return the matrix of \(self\) in the canonical faithful representation.

This is an \(n\)-dimension real faithful essential representation, where \(n\) is the number of generators of the Coxeter group. Note that this is not always the most natural matrix representation, for instance in type \(A_n\).

EXAMPLES:

```python
sage: W = WeylGroup(['A',3])
sage: s = W.simple_reflections()
sage: (s[1]*s[2]*s[3]).canonical_matrix()
[ 0 0 -1]
[ 1 0 -1]
[ 0 1 -1]
```

coset_representation(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:

```python
sage: W = CoxeterGroups().example(5)
sage: s = W.simple_reflections()
sage: w = s[2]*s[1]*s[3]
sage: w.coset_representation([[]].reduced_word())
(continues on next page)```
cover_reflections (side='right')

Return the set of reflections t such that self t covers self.

If side is ‘left’, t self covers self.

EXAMPLES:

```python
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^\infty.

INPUT:

- c - a Coxeter element.

OUTPUT:

the c-sorting word of self as a list of integers.

EXAMPLES:

```python
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
```
**deodhar_factor_element**(*w, index_set*)

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 \( v = \text{self} \) 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 \( v' \leq f(v, w) \times w' \) in \( W' \) where \( \times \) is the Demazure product.

**EXAMPLS:**

```
sage: W = WeylGroup(['A',5],prefix="s")
sage: v = W.from_reduced_word([5])
sage: w = W.from_reduced_word([4,5,2,3,1,2])
sage: v.deodhar_factor_element(w,[1,3,4])
s3*s1
```

```
sage: W = WeylGroup(['C',2])
sage: w = W.from_reduced_word([2,1])
sage: w.deodhar_factor_element(W.from_reduced_word([2]),[1])
Traceback (most recent call last):
...  ValueError: [2, 1] is not of minimum length in its coset for the
˓→parabolic subgroup with index set [1]
```

**REFERENCES:**
- [Deo1987a]

**deodhar_lift_down**(*w, index_set*)

Letting \( v = \text{self} \), given a Bruhat relation \( v W' \geq w W' \) among cosets with respect to the subgroup \( W' \) given by the Dynkin node subset *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 \( v \geq x \).

**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([1,2,3,2])
sage: v.deodhar_lift_down(w, [3])
s2*s3*s2
```

**deodhar_lift_up**(*w, index_set*)

Letting \( v = \text{self} \), given a Bruhat relation \( v W' \leq w W' \) among cosets with respect to the subgroup \( W' \) given by the Dynkin node subset *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 \( \mathfrak{w} \) as \( \text{self} \).
• \( \text{index\_set} \) is a subset of Dynkin nodes defining a parabolic subgroup \( \mathfrak{w}' \).

OUTPUT:
The unique Bruhat-minimum element \( x \) in \( \mathfrak{w} \) such that \( x \mathfrak{w}' = w \mathfrak{w}' \) and \( v \leq x \).

See also:

\texttt{sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.deodhar\_lift\_down()}

\textbf{EXAMPLES:}

```python
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:
• \( \text{index\_set} \) - a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
• \( \text{side} \) - ‘left’ or ‘right’ (default: ‘right’)
• \( \text{positive} \) - a boolean (default: False)

Returns the descents of \( \text{self} \), as a list of elements of the \( \text{index\_set} \).

The \( \text{index\_set} \) option can be used to restrict to the parabolic subgroup indexed by \( \text{index\_set} \).

If positive is True, then returns the non-descents instead

\textbf{Todo:} find a better name for \( \text{positive}: \) complement? non_descent?

Caveat: the return type may change to some other iterable (tuple, …) in the future. Please use keyword arguments also, as the order of the arguments may change as well.

\textbf{EXAMPLES:}

```python
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]
```

\textbf{Todo:} \( \text{side, index\_set, positive} \)

\textbf{first\_descent (**side**='right', **index\_set=**None, **positive=**False)

Returns the first left (resp. right) descent of \( \text{self} \), as an element of \( \text{index\_set} \), or \text{None} if there is none.

See \texttt{descents()} for a description of the options.

\textbf{EXAMPLES:}

---

\textbf{3.30. Coxeter Groups}  \hspace{1cm} 233
```
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
```

(continues on next page)
has_right_descent\( (i) \)

Returns whether \( i \) is a right descent of self.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on \{0, ..., 3\}
sage: w = W.an_element(); w
(1, 2, 3, 0)
sage: w.has_right_descent(0)
False
sage: w.has_right_descent(1)
False
sage: w.has_right_descent(2)
True
```

inversions_as_reflections()

Returns the set of reflections \( r \) such that self\( r ) < \text{self} \).

**EXAMPLES:**

```python
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\( (c, \text{sorting_word}=\text{None}) \)

Return whether self is \( c \)-sortable.

Given a Coxeter element \( c \), an element \( w \) is \( c \)-sortable if its \( c \)-sorting word decomposes into a sequence of weakly decreasing subwords of \( c \).

**INPUT:**
- \( c \) – a Coxeter element.
- \( \text{sorting_word} \) – sorting word (default: None) used to not recomputed the \( c \)-sorting word if already computed.

**OUTPUT:**
- \( \text{is self c-sortable} \)

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()
sage: c = W.from_reduced_word([0,2,1])
sage: w = W.from_reduced_word([1,2,1,0,1])
sage: w.coxeter_sorting_word(c)
[2, 1, 2, 0, 1]
sage: w.is_coxeter_sortable(c)
False
sage: w = W.from_reduced_word([0,2,1,0,2])
```

(continues on next page)
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 A_3 (Catalan number)
14

is_grassmannian (side='right')
Return whether self is Grassmannian.

INPUT:
• side = “left” or “right” (default: “right”)
An element is Grassmannian if it has at most one descent on the right (resp. on the left).

EXAMPLES:

sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: W.one().is_grassmannian()
True
sage: s[1].is_grassmannian()
True
sage: (s[1]*s[2]).is_grassmannian()
True
sage: (s[0]*s[1]).is_grassmannian()
True
sage: (s[1]*s[2]*s[1]).is_grassmannian()
False
sage: (s[0]*s[2]*s[1]).is_grassmannian(side="left")
False
sage: (s[0]*s[2]*s[1]).is_grassmannian(side="right")
True
sage: (s[0]*s[2]*s[1]).is_grassmannian()
True

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 ()
Return the length of self.
This is the minimal length of a product of simple reflections giving self.

EXAMPLES:
sage: W = CoxeterGroups().example()
sage: s1 = W.simple_reflection(1)
sage: s2 = W.simple_reflection(2)
sage: s1.length()
1
sage: (s1*s2).length()
2
sage: W = CoxeterGroups().example()
sage: s = W.simple_reflections()
sage: w = s[0]*s[1]*s[0]
sage: w.length()
3
sage: W = CoxeterGroups().example()
sage: sum((x^w.length())
    for w in W) - expand(prod(sum(x^i
    for i in
    \rightarrow range(j+1))
    for j in range(4))) # This is scandalously slow!!!
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 \( \text{self} \) covers \( \text{self} t \).

If side is ‘left’, \( \text{self} \) covers \( t \text{self} \).

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)

Return all elements that \( \text{self} \) covers in weak order.

INPUT:
  • side – ‘left’ or ‘right’ (default: ‘right’)
  • index_set – a list of indices or None

OUTPUT: a list

EXAMPLES:

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])

(continues on next page)

(continued from previous page)

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)
Find 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:
sage:
sage:
sage:
s4*s2
sage:
s4*s2
sage:
s4*s2

W = WeylGroup(['A',4],prefix="s")
v = W.from_reduced_word([2,3,4,1,2])
u = W.from_reduced_word([2,3,2,1])
v.min_demazure_product_greater(u)
v.min_demazure_product_greater([2,3,2,1])
v.min_demazure_product_greater((2,3,2,1))

reduced_word()
Return a reduced word for self.
This is a word [𝑖1 , 𝑖2 , . . . , 𝑖𝑘 ] of minimal length such that 𝑠𝑖1 𝑠𝑖2 · · · 𝑠𝑖𝑘 = self, where the 𝑠𝑖 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 𝑚colored edge between two reduced words 𝑥 and 𝑦 whenever 𝑥 and 𝑦 differ by exactly one length-𝑚
braid move (with 𝑚 ≥ 2).

238

Chapter 3. Individual Categories


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
```

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.

The algorithm uses the Matsumoto property that any two reduced expressions are related by braid relations, see [Theorem 3.3.1(ii), BB2005].

REFERENCES:

See also:
- `braid_orbit()`

EXAMPLES:
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()

reflection_length()  
Return the reflection length of self.

The reflection length is the length of the shortest expression of the element as a product of reflections.

See also:

absolute_length()  

EXAMPLES:

```
    sage: W = WeylGroup(['A',3])
    sage: s = W.simple_reflections()
    sage: (s[1]*s[2]*s[3]).reflection_length()
    3
```
support()  
Return the support of self, that is the simple reflections that appear in the reduced expressions of self.

OUTPUT:
The support of self as a set of integers

EXAMPLES:

```
sage: W = CoxeterGroups().example()
sage: w = W.from_reduced_word([1, 2, 1])
sage: w.support()
{1, 2}
```

upper_covers(side='right', index_set=None)  
Return all elements that cover self in weak order.

INPUT:
• side – ‘left’ or ‘right’ (default: ‘right’)
• index_set – a list of indices or None

OUTPUT: a list

EXAMPLES:

```
sage: W = WeylGroup(['A', 3])
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])]
[[2, 3, 1]]
```

weak_covers(side='right', index_set=None, positive=False)  
Return all elements that self covers in weak order.

INPUT:
• side – ‘left’ or ‘right’ (default: ‘right’)
• positive – a boolean (default: False)
• index_set – a list of indices or None

OUTPUT: a list

EXAMPLES:
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: $O(l \cdot c)$, where $l$ is the minimum of the lengths of $u$ and of $v$, and $c$ is the cost of the low level methods `first_descent()`, `has_descent()`, `apply_simple_reflection()`, etc. Those are typically $O(n)$, where $n$ is the rank of the Coxeter group.

We now run consistency tests with permutations:

```python
sage: W = WeylGroup(['A',3])

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), side='left')
```

**Finite**

alias of `sage.categories.finite_coxeter_groups.FiniteCoxeterGroups`

**class ParentMethods**

**braid_group_as_finitely_presented_group()**

Return the associated braid group.

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A',2])

sage: W.braid_group_as_finitely_presented_group()
Finitely presented group < S1, S2 | S1*S2*S1*S2^-1*S1^-1*S2^-1 >

sage: W = WeylGroup(['B',2])

sage: W.braid_group_as_finitely_presented_group()
Finitely presented group < S1, S2 | (S1*S2)^2*(S1^-1*S2^-1)^2 >

sage: W = ReflectionGroup(['B',3], index_set=['AA','BB',5])  # optional -- gap3

sage: W.braid_group_as_finitely_presented_group()  # optional -- gap3
Finitely presented group < SAA, SBB, S5 | SAA*SBB*SAA*SBB^-1*SAA^-1*SBB^-1, SAA*S5*SAA^-1*S5^-1, (SBB*S5)^2*(SBB^-1*S5^-1)^2 >
```

**braid_orbit(word)**

Return the braid orbit of a word `word` of indices.

The input word does not need to be a reduced expression of an element.

**INPUT:**

- `word`: a list (or iterable) of indices in `self.index_set()`

**OUTPUT**: a list of all lists that can be obtained from `word` by replacements of braid relations

See `braid_relations()` for the definition of braid relations.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example()

sage: s = W.simple_reflections()


sage: word = w.reduced_word(); word
```
Todo: The result should be full featured finite enumerated set (e.g., counting can be done much faster than iterating).

See also:

- reduced_words()

braid_relations()

Return the braid relations of self as a list of reduced words of the braid relations.

EXAMPLES:

```python
sage: W = WeylGroup(['A',2])
sage: W.braid_relations()
[[[1, 2, 1], [2, 1, 2]]]
```

bruhat_graph (x=None, y=None, edge_labels=False)

Return the Bruhat graph as a directed graph, with an edge $u \to v$ if and only if $u < v$ in the Bruhat order, and $u = v$.

The Bruhat graph $\Gamma(x, y)$, defined if $x \leq y$ in the Bruhat order, has as its vertices the Bruhat interval...
\{ t \mid x \leq t \leq y \}, \text{ and as its edges are the pairs } (u, v) \text{ such that } u = r \cdot v \text{ where } r \text{ is a reflection, that is, a conjugate of a simple reflection.}

REFERENCES:


EXAMPLES:

```python
sage: W = CoxeterGroup(['H',3])
sage: G = W.bruhat_graph(); G
Digraph on 120 vertices

sage: W = CoxeterGroup(['A',2,1])
sage: s1, s2, s3 = W.simple_reflections()
sage: W.bruhat_graph(s1, s1*s3*s2*s3)
Digraph on 6 vertices

sage: W = CoxeterGroup(['A',2,1])
sage: W.bruhat_graph(s1, s3*s2*s3)
Digraph on 0 vertices

sage: W = WeylGroup("A3", prefix="s")
sage: s1, s2, s3 = W.simple_reflections()
sage: G = W.bruhat_graph(s1*s3, s1*s2*s3*s2*s1); G
Digraph on 10 vertices

Check that the graph has the correct number of edges (see trac ticket #17744):

```python
sage: len(G.edges())
16
```

\textbf{bruhat_interval} \((x, y)\)

Return the list of \( t \) such that \( x \leq t \leq y \).

EXAMPLES:

```python
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, s2*s3*s2, s1*s2*s1, s3*s1*s2, s1*s2*s3, s2*s1, s3*s2, s2*s3, s1*s2, s2]

sage: W = WeylGroup(['A',2,1], prefix="s")

sage: [s0,s1,s2] = W.simple_reflections()
sage: W.bruhat_interval(1, s0*s1*s2)
[s0*s1*s2, s1*s2, s0*s2, s0*s1, s2, s1, s0, 1]
```

\textbf{bruhat_interval_poset} \((x, y, \text{facade}=False)\)

Return the poset of the Bruhat interval between \( x \) and \( y \) in Bruhat order.

EXAMPLES:

```python
sage: W = WeylGroup("A3", prefix="s")
sage: s1,s2,s3 = W.simple_reflections()
sage: W.bruhat_interval_poset(s2, s1*s3*s2*s1*s3)
Finite poset containing 16 elements
```

(continues on next page)
sage: W = WeylGroup(['A',2,1], prefix="s")
sage: s0,s1,s2 = W.simple_reflections()
sage: W.bruhat_interval_poset(1, s0*s1*s2)
Finite poset containing 8 elements

canonical_representation()
Return the canonical faithful representation of self.

EXAMPLES:

sage: W = WeylGroup("A3")
sage: W.canonical_representation()
Finite Coxeter group over Integer Ring with Coxeter matrix:
[1 3 2]
[3 1 3]
[2 3 1]

coxeter_diagram()
Return the Coxeter diagram of self.

EXAMPLES:

sage: W = CoxeterGroup(['H',3], implementation="reflection")
sage: G = W.coxeter_diagram(); G
Graph on 3 vertices
sage: G.edges()
[(1, 2, 3), (2, 3, 5)]
sage: CoxeterGroup(G) is W
True
sage: G = Graph([(0, 1, 3), (1, 2, oo)])
sage: W = CoxeterGroup(G)
sage: W.coxeter_diagram() == G
True
sage: CoxeterGroup(W.coxeter_diagram()) is W
True

coxeter_element()
Return a Coxeter element.

The result is the product of the simple reflections, in some order.

Note: This implementation is shared with well generated complex reflection groups. It would be
nicer to put it in some joint super category; however, in the current state of the art, there is none where
it is clear that this is the right construction for obtaining a Coxeter element.

In this context, this is an element having a regular eigenvector (a vector not contained in any reflection
hyperplane of self).

EXAMPLES:

sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()
\[ [1, 2, 4, 3] \]
\[
\text{sage: } \text{CoxeterGroup(['F', 4]).coxeter_element().reduced_word()}
\]
\[ [1, 2, 3, 4] \]
\[
\text{sage: } \text{CoxeterGroup(['E', 8]).coxeter_element().reduced_word()}
\]
\[ [1, 3, 2, 4, 5, 6, 7, 8] \]
\[
\text{sage: } \text{CoxeterGroup(['H', 3]).coxeter_element().reduced_word()}
\]
\[ [1, 2, 3] \]

This method is also used for well generated finite complex reflection groups:

\[
\text{sage: } \text{W = ReflectionGroup((1,1,4))} \quad \# \text{optional - gap3}
\]
\[
\text{sage: } \text{W.coxeter_element().reduced_word()} \quad \# \text{optional - gap3}
\]
\[ [1, 2, 3] \]
\[
\text{sage: } \text{W = ReflectionGroup((2,1,4))} \quad \# \text{optional - gap3}
\]
\[
\text{sage: } \text{W.coxeter_element().reduced_word()} \quad \# \text{optional - gap3}
\]
\[ [1, 2, 3, 4] \]
\[
\text{sage: } \text{W = ReflectionGroup((4,1,4))} \quad \# \text{optional - gap3}
\]
\[
\text{sage: } \text{W.coxeter_element().reduced_word()} \quad \# \text{optional - gap3}
\]
\[ [1, 2, 3, 4] \]
\[
\text{sage: } \text{W = ReflectionGroup((4,4,4))} \quad \# \text{optional - gap3}
\]
\[
\text{sage: } \text{W.coxeter_element().reduced_word()} \quad \# \text{optional - gap3}
\]
\[ [1, 2, 3, 4] \]

**coxeter_matrix()**

Return the Coxeter matrix associated to self.

**EXAMPLES:**

\[
\text{sage: } \text{G = WeylGroup(['A',3])}
\]
\[
\text{sage: } \text{G.coxeter_matrix()}
\]
\[
\begin{bmatrix}
1 & 3 & 2 \\
3 & 1 & 3 \\
2 & 3 & 1 \\
\end{bmatrix}
\]

**coxeter_type()**

Return the Coxeter type of self.

**EXAMPLES:**

\[
\text{sage: } \text{W = CoxeterGroup(['H',3])}
\]
\[
\text{sage: } \text{W.coxeter_type()}
\]

Coxeter type of ['H', 3]

**demazure_product**(*Q*)

Return the Demazure product of the list *Q* in self.

**INPUT:**

- *Q* is a list of elements from the index set of self.

This returns the Coxeter group element that represents the composition of 0-Hecke or Demazure operators.

See **CoxeterGroups.ParentMethods.simple_projections()**.

**EXAMPLES:**

3.30. Coxeter Groups
sage: W = WeylGroup(\['A',2\])
sage: w = W.demazure_product([2,2,1])
sage: w.reduced_word()
[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:**

```python
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]]

**grassmannian_elements**(side='right')
Return the left or right Grassmannian elements of \(\text{self}\) as an enumerated set.

**INPUT:**
- **side** — (default: "right") "left" or "right"

**EXAMPLES:**

```python
sage: S = CoxeterGroups().example()
sage: G = S.grassmannian_elements()
sage: G.cardinality()
12

sage: G.list()
[(0, 1, 2, 3), (1, 0, 2, 3), (0, 2, 1, 3), (0, 1, 3, 2),
 (2, 0, 1, 3), (1, 2, 0, 3), (0, 3, 1, 2), (0, 2, 3, 1),
 (3, 0, 1, 2), (1, 3, 0, 2), (1, 2, 3, 0), (2, 3, 0, 1)]

sage: sorted(tuple(w.descents()) for w in G)
[(0,), (0,), (0,), (1,), (0,), (1,), (1,), (1,), (2,), (2,), (2,)]

sage: G = S.grassmannian_elements(side = "left")
sage: G.cardinality()
12

sage: sorted(tuple(w.descents(side = "left")) for w in G)
[(0,), (0,), (0,), (1,), (1,), (1,), (1,), (2,), (2,), (2,)]
```

**index_set**()
Return the index set of \(\text{self}\).

**EXAMPLES:**
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.index_set()
(1, 2)
sage: W = CoxeterGroup([[1,3],[3,1]], index_set=['x', 'y'])
sage: W.index_set()
('x', 'y')
sage: W = CoxeterGroup(['H',3])
sage: W.index_set()
(1, 2, 3)

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

simple_projection(i, side='right', length_increasing=True)
Return the simple projection $\pi_i$ (or $\pi_i$ if $\text{length}_{\text{increasing}}$ is False).

INPUT:

• i - an element of the index set of self
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)

(continues on next page)
simple_projections\((side='right', length\_increasing=True)\)

Return the family of simple projections, also known as 0-Hecke or Demazure operators.

**INPUT:**

- `self` – a Coxeter group \(W\)
- `side` – `'left'` or `'right'` (default: `'right'`)
- `length\_increasing` – a boolean (default: `True`) specifying whether the operator increases or decreases length

Returns the simple projections of \(W\), as a family.

To each simple reflection \(s_i\) of \(W\), corresponds a simple projection \(\pi_i\) from \(W\) to \(W\) defined by:

\[
\pi_i(w) = ws_i\text{ if } i \text{ is not a descent of } w.
\]

\[
\pi_i(w) = w\text{ otherwise}.
\]

The simple projections \((\pi_i)_{i \in I}\) move elements down the right permutohedron, toward the maximal element. They satisfy the same braid relations as the simple reflections, but are idempotents \(\pi_i^2 = \pi\) and 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):

\[
\overline{\pi}_i(w) = ws_i\text{ if } i \text{ is a descent of } w.
\]

\[
\overline{\pi}_i(w) = w\text{ otherwise}.
\]

as well as the analogues acting on the left (when the option `side` is `left`).

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: sigma = W.an_element(); sigma
(1, 2, 3, 0)
sage: pi = W.simple_projections(); pi
Finite family {0: <function ...<lambda> at ...>, 1: <function ...<lambda> ...>→, 2: <function ...<lambda> ...>}
sage: pi[1](sigma)
(1, 3, 2, 0)
sage: W.simple_projection(1)(sigma)
(1, 3, 2, 0)
```

standard_coxeter_elements()

Return all standard Coxeter elements in `self`.

This is the set of all elements in `self` obtained from any product of the simple reflections in `self`.

**Note:**

- `self` is assumed to be well-generated.
- This works even beyond real reflection groups, but the conjugacy class is not unique and we only obtain one such class.

**EXAMPLES:**

```python
sage: W = CoxeterGroups().example(); W
The symmetric group on {0, ..., 3}
sage: s = W.simple_reflections()
sage: sigma = W.an_element(); sigma
(1, 2, 3, 0)
sage: pi = W.simple_projections(); pi
Finite family {0: <function ...<lambda> at ...>, 1: <function ...<lambda> ...>→, 2: <function ...<lambda> ...>}
sage: pi[1](sigma)
(1, 3, 2, 0)
sage: W.standard_coxeter_elements()
```

250 Chapter 3. Individual Categories
\begin{verbatim}
sage: W = ReflectionGroup(4)  # optional - gap3
sage: sorted(W.standard_coxeter_elements())  # optional - gap3
[(1,7,6,12,23,20)(2,8,17,24,9,5) (3,16,10,19,15,21) (4,14,11,22,18,13),
 (1,10,4,12,21,22) (2,11,19,24,13,3) (5,15,7,17,16,23) (6,18,8,20,14,9)]
\end{verbatim}

\textbf{weak_order_ideal\ (predicate, side='right', category=None)}

Return a weak order ideal defined by a predicate

\textbf{INPUT:}

- \texttt{predicate}: a predicate on the elements of \texttt{self} defining a weak order ideal in \texttt{self}
- \texttt{side}: “left” or “right” (default: “right”)

\textbf{OUTPUT:} an enumerated set

\textbf{EXAMPLES:}

\begin{verbatim}
sage: D6 = FiniteCoxeterGroups().example(5)
sage: I = D6.weak_order_ideal(predicate = lambda w: w.length() <= 3)
sage: I.cardinality()
7
sage: list(I)
[(), (1,), (2,), (1, 2), (2, 1), (1, 2, 1), (2, 1, 2)]
\end{verbatim}

We now consider an infinite Coxeter group:

\begin{verbatim}
sage: W = WeylGroup(['A',1,1])
sage: I = W.weak_order_ideal(predicate = lambda w: w.length() <= 2)
sage: list(iter(I))
[([1 0] [-1 2] [ 1 0] [ 3 -2] [-1 2], [0 1], [ 2 -1], [ 2 -1], [-2 3])]
\end{verbatim}

Even when the result is finite, some features of \texttt{FiniteEnumeratedSets} are not available:

\begin{verbatim}
sage: I.cardinality()  # todo: not implemented
5
sage: list(I)  # todo: not implemented
\end{verbatim}

unless this finiteness is explicitly specified:

\begin{verbatim}
sage: I = W.weak_order_ideal(predicate = lambda w: w.length() <= 2,
...: category = FiniteEnumeratedSets())
sage: I.cardinality()
5
sage: list(I)
[([1 0] [-1 2] [ 1 0] [ 3 -2] [-1 2], [0 1], [ 2 -1], [ 2 -1], [-2 3])]
\end{verbatim}

\textbf{Background}

The weak order is returned as a \texttt{SearchForest}. This is achieved by assigning to each element \(u_1\) of the ideal a single ancestor \(u = u_1s_i\), where \(i\) is the smallest descent of \(u\).

This allows for iterating through the elements in roughly Constant Amortized Time and constant memory (taking the operations and size of the generated objects as constants).
additional_structure()

Return None.

Indeed, all the structure Coxeter groups have in addition to groups (simple reflections, ...) is already defined in the super category.

See also:

Category.additional_structure()

EXAMPLES:

```
sage: CoxeterGroups().additional_structure()
```

super_categories()

EXAMPLES:

```
sage: CoxeterGroups().super_categories()
[Category of generalized coxeter groups]
```

## 3.31 Crystals

```python
class sage.categories.crystals.CrystalHomset (X, Y, category=None)
    Bases: sage.categories.homset.Homset

The set of crystal morphisms from one crystal to another.

An $U_q(g)$ $I$-crystal morphism $\Psi : B \rightarrow C$ is a map $\Psi : B \cup \{0\} \rightarrow C \cup \{0\}$ such that:

- $\Psi(0) = 0$.
- If $b \in B$ and $\Psi(b) \in C$, then $\text{wt}(\Psi(b)) = \text{wt}(b)$, $\varepsilon_i(\Psi(b)) = \varepsilon_i(b)$, and $\varphi_i(\Psi(b)) = \varphi_i(b)$ for all $i \in I$.
- If $b, b' \in B$, $\Psi(b), \Psi(b') \in C$ and $f_i b = b'$, then $f_i \Psi(b) = \Psi(b')$ and $\Psi(b) = e_i \Psi(b')$ for all $i \in I$.

If the Cartan type is unambiguous, it is suppressed from the notation.

We can also generalize the definition of a crystal morphism by considering a map of $\sigma$ of the (now possibly different) Dynkin diagrams corresponding to $B$ and $C$ along with scaling factors $\gamma_i \in \mathbb{Z}$ for $i \in I$. Let $\sigma_i$ denote the orbit of $i$ under $\sigma$. We write objects for $B$ as $X$ with corresponding objects of $C$ as $\tilde{X}$. Then a virtual crystal morphism $\Psi$ is a map such that the following holds:

- $\Psi(0) = 0$.
- If $b \in B$ and $\Psi(b) \in C$, then for all $j \in \sigma_i$:
  \[ \varepsilon_i(b) = \frac{1}{\gamma_j} \tilde{\varepsilon}_j(\Psi(b)), \quad \varphi_i(b) = \frac{1}{\gamma_j} \tilde{\varphi}_j(\Psi(b)), \quad \text{wt}(\Psi(b)) = \sum_i c_i \sum_{j \in \sigma_i} \gamma_j \tilde{\Lambda}_j, \]
  where $\text{wt}(b) = \sum_i c_i \Lambda_i$.
- If $b, b' \in B$, $\Psi(b), \Psi(b') \in C$ and $f_i b = b'$, then independent of the ordering of $\sigma_i$ we have:
  \[ \Psi(b') = e_i \Psi(b) = \prod_{j \in \sigma_i} \tilde{e}_j^\gamma(\Psi(b)), \quad \Psi(b') = f_i \Psi(b) = \prod_{j \in \sigma_i} \tilde{f}_j^\gamma(\Psi(b)). \]

If $\gamma_i = 1$ for all $i \in I$ and the Dynkin diagrams are the same, then we call $\Psi$ a twisted crystal morphism.

INPUT:

- $X$ – the domain
• Y – the codomain
• category – (optional) the category of the crystal morphisms

See also:

For the construction of an element of the homset, see `CrystalMorphismByGenerators` and `crystal_morphism()`.

EXAMPLES:

We begin with the natural embedding of $B(2\Lambda_1)$ into $B(\Lambda_1) \otimes B(\Lambda_1)$ in type $A_1$:

```
sage: B = crystals.Tableaux(['A',1], shape=[2])
sage: F = crystals.Tableaux(['A',1], shape=[1])
sage: T = crystals.TensorProduct(F, F)
sage: v = T.highest_weight_vectors()[0]; v
[[[1]], [[1]]]
sage: H = Hom(B, T)
sage: psi = H([v])
sage: psi(b)
[[[1]], [[1]]]
sage: psi(b).
[[[1]], [[1]]]
sage: psi(b.f(1))
[[[1]], [[2]]]
sage: psi(b.f(1))
[[[1]], [[2]]]
sage: psi(b.f(1))
[[[1]], [[2]]]
```

We now look at the decomposition of $B(\Lambda_1) \otimes B(\Lambda_1)$ into $B(2\Lambda_1) \oplus B(0)$:

```
sage: B0 = crystals.Tableaux(['A',1], shape=[1])
sage: D = crystals.DirectSum([B, B0])
sage: H = Hom(T, D)
sage: psi = H(D.module_generators)
sage: psi
['A', 1] Crystal morphism:
  From: Full tensor product of the crystals
      [The crystal of tableaux of type ['A', 1] and shape(s) [[1]],
       The crystal of tableaux of type ['A', 1] and shape(s) [[1]]]
  To: Direct sum of the crystals Family
      (The crystal of tableaux of type ['A', 1] and shape(s) [[2]],
       The crystal of tableaux of type ['A', 1] and shape(s) [[1]])
  Defn: [[1]], [[1]] |--> [[1, 1]]
         [[2]], [[1]] |--> []
```

We can always construct the trivial morphism which sends everything to 0:

```
sage: Binf = crystals.infinity.Tableaux(['B', 2])
sage: B = crystals.Tableaux(['B',2], shape=[1])
sage: H = Hom(Binf, B)
sage: psi = H(lambda x: None)
sage: psi(Binf.highest_weight_vector())
```

For Kirillov-Reshetikhin crystals, we consider the map to the corresponding classical crystal:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: B = K.classical_decomposition()
```

(continues on next page)
Next we consider a type $D_4$ crystal morphism where we twist by $3 \leftrightarrow 4$:

```sage
sage: B = crystals.Tableaux(['D',4], shape=[1])
sage: H = Hom(B, B)
sage: d = {1:1, 2:2, 3:4, 4:3}
sage: psi = H(B.module_generators, automorphism=d)
sage: b = B.highest_weight_vector()
sage: b.f_string([1,2,3])
[[4]]
sage: b.f_string([1,2,4])
[[-4]]
sage: psi(b.f_string([1,2,3]))
[[-4]]
sage: psi(b.f_string([1,2,4]))
[[4]]
```

We construct the natural virtual embedding of a type $B_3$ into a type $D_4$ crystal:

```sage
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: H = Hom(B, C)
sage: psi = H(C.module_generators)
sage: psi
['B', 3] -> ['D', 4] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['B', 3] and shape(s) [[1]]
  To:  The crystal of tableaux of type ['D', 4] and shape(s) [[2]]
  Defn: [[1]] |--> [[1, 1]]

Element alias of CrystalMorphismByGenerators

class sage.categories.crystals.CrystalMorphism

Bases: sage.categories.morphism.Morphism

A crystal morphism.

INPUT:

- parent – a homset
- cartan_type – (optional) a Cartan type; the default is the Cartan type of the domain
• **virtualization** – (optional) a dictionary whose keys are in the index set of the domain and whose values are lists of entries in the index set of the codomain

• **scaling_factors** – (optional) a dictionary whose keys are in the index set of the domain and whose values are scaling factors for the weight, $\varepsilon$ and $\varphi$

### cartan_type()

Return the Cartan type of self.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: psi = Hom(B, B).an_element()
sage: psi.cartan_type()
['A', 2]
```

### is_injective()

Return if self is an injective crystal morphism.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: psi = Hom(B, B).an_element()
sage: psi.is_injective()
False
```

### is_surjective()

Check if self is a surjective crystal morphism.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_surjective()
False
```

### scaling_factors()

Return the scaling factors $\gamma_i$.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: psi = B.crystal_morphism(C.module_generators)
```
virtualization()
Return the virtualization sets $\sigma_i$.

EXAMPLES:

```
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: psi = B.crystal_morphism(C.module_generators)
sage: psi.virtualization()
Finite family {1: (1,), 2: (2,), 3: (3, 4)}
```

#### class sage.categories.crystals.CrystalMorphismByGenerators

```
class sage.categories.crystals.CrystalMorphismByGenerators(parent, on_gens, cartan_type=None, virtualization=None, scaling_factors=None, gens=None, check=True)
```

Bases: `sage.categories.crystals.CrystalMorphism`

A crystal morphism defined by a set of generators which create a virtual crystal inside the codomain.

INPUT:

- `parent` – a homset
- `on_gens` – a function or list that determines the image of the generators (if given a list, then this uses the order of the generators of the domain) of the domain under `self`
- `cartan_type` – (optional) a Cartan type; the default is the Cartan type of the domain
- `virtualization` – (optional) a dictionary whose keys are in the index set of the domain and whose values are lists of entries in the index set of the codomain
- `scaling_factors` – (optional) a dictionary whose keys are in the index set of the domain and whose values are scaling factors for the weight, $\varepsilon$ and $\varphi$
- `gens` – (optional) a finite list of generators to define the morphism; the default is to use the highest weight vectors of the crystal
- `check` – (default: True) check if the crystal morphism is valid

See also:

`sage.categories.crystals.Crystals.ParentMethods.crystal_morphism()`

#### im_gens()

Return the image of the generators of `self` as a tuple.

EXAMPLES:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: F = crystals.Tableaux(['A',2], shape=[1])
sage: T = crystals.TensorProduct(F, F, F)
sage: H = Hom(T, B)
sage: b = B.highest_weight_vector()
sage: psi = H((None, b, b, None), generators=T.highest_weight_vectors())
sage: psi.im_gens()
(Non...([[1, 1], [2]], [[1, 1], [2]], None)
```

256 Chapter 3. Individual Categories
image()  
Return the image of self in the codomain as a Subcrystal.

**Warning:** This assumes that self is a strict crystal morphism.

**EXAMPLES:**
```
sage: B = crystals.Tableaux(['B',3], shape=[1])
sage: C = crystals.Tableaux(['D',4], shape=[2])
sage: H = Hom(B, C)
sage: psi = H(C.module_generators)
sage: psi.image()
Virtual crystal of The crystal of tableaux of type ['D', 4] and shape(s) \rightarrow [[2]] of type ['B', 3]
```

to_module_generator(x)  
Return a generator mg and a path of \(e_i\) and \(f_i\) operations to mg.

**OUTPUT:**
A tuple consisting of:
- a module generator,
- a list of 'e' and 'f' to denote which operation, and
- a list of matching indices.

**EXAMPLES:**
```
sage: B = crystals.elementary.Elementary(['A',2], 2)
sage: psi = B.crystal_morphism(B.module_generators)
sage: psi.to_module_generator(B(4))
(0, ['f', 'f', 'f', 'f'], [2, 2, 2, 2])
sage: psi.to_module_generator(B(-2))
(0, ['e', 'e'], [2, 2])
```

class sage.categories.crystals.Crystals(s=None)

Bases: 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']}
```

class ElementMethods

Epsilon()

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: C(0).Epsilon()
(0, 0, 0, 0, 0)
sage: C(1).Epsilon()
(0, 0, 0, 0, 0)
sage: C(2).Epsilon()
(1, 0, 0, 0, 0)
```

Phi()

EXAMPLES:

```
sage: C = crystals.Letters(['A', 5])
sage: C(0).Phi()
(0, 0, 0, 0, 0)
sage: C(1).Phi()
(1, 0, 0, 0, 0)
sage: C(2).Phi()
(1, 1, 0, 0, 0)
```

all_paths_to_highest_weight(index_set=None)

Iterate over all paths to the highest weight from self with respect to \(\text{index}_{\text{set}}\).

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]]
```

(continues on next page)
```python
sage: B = crystals.Tableaux("A3", shape=[4,2,1])
sage: b0 = B.highest_weight_vector()
sage: b = b0.f_string([1, 1, 2, 3])

list(b.all_paths_to_highest_weight())
```

```
cartan_type()
Returns the Cartan type associated to self

EXAMPLES:
```
```python
sage: C = crystals.Letters(['A', 5])
sage: C(1).cartan_type() ['A', 5]
```

```
e(i)
Return \( e_i \) of self if it exists or None otherwise.

This method should be implemented by the element class of the crystal.

EXAMPLES:
```
```python
sage: C = Crystals().example(5)
sage: x = C[2]; x
3
sage: x.e(1), x.e(2), x.e(3) (None, 2, None)
```

```
e_string(list)
Applies \( e_{i_r} \cdots e_{i_1} \) to self for list as \([i_1, \ldots, i_r]\)

EXAMPLES:
```
```python
sage: C = crystals.Letters(['A', 3])
sage: b = C(3)
sage: b.e_string([2, 1])
1
sage: b.e_string([1, 2])
```

```
epsilon(i)

EXAMPLES:
```
```python
sage: C = crystals.Letters(['A', 5])
sage: C(1).epsilon(1) 0
sage: C(2).epsilon(1) 1
```

```
f(i)
Return \( f_i \) of self if it exists or None otherwise.

This method should be implemented by the element class of the crystal.

EXAMPLES:
```
\begin{verbatim}
sage: C = Crystals().example(5)
sage: x = C[1]; x
2
sage: x.f(1), x.f(2), x.f(3)
(0, 3, 0)

f_string(list)
Applies $f_{i_r} \cdots f_{i_1}$ to self for list as $[i_1, \ldots, i_r]$

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])

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])

phi(i)

EXAMPLES:

sage: C = crystals.Letters(['A',5])
sage: C(1).phi(1)
1
\end{verbatim}
\texttt{sage}: C(2).phi(1)
0

\texttt{sage}: C(2).phi(-1)
0

\texttt{phi_minus_epsilon}(i)

Return $\varphi_i - \varepsilon_i$ of \texttt{self}.

There are sometimes better implementations using the weight for this. It is used for reflections along a string.

EXAMPLES:

\begin{verbatim}
sage: C = crystals.Letters(['A',5])
sage: C(1).phi_minus_epsilon(1)
1
\end{verbatim}

\texttt{s}(i)

Return the reflection of \texttt{self} along its $i$-string.

EXAMPLES:

\begin{verbatim}
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]]
\end{verbatim}

\texttt{subcrystal}(index_set=None, max_depth=inf, direction='both', contained=None, cartan_type=None, category=None)

Construct the subcrystal generated by \texttt{self} using $e_i$ and/or $f_i$ for all $i$ in \texttt{index_set}.

INPUT:

- \texttt{index_set} = (default: None) the index set; if None then use the index set of the crystal
- \texttt{max_depth} = (default: infinity) the maximum depth to build
- \texttt{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 $f_i$
- \texttt{contained} = (optional) a set (or function) defining the containment in the subcrystal
- \texttt{cartan_type} = (optional) specify the Cartan type of the subcrystal
- \texttt{category} = (optional) specify the category of the subcrystal

See also:

- \texttt{Crystals.ParentMethods.subcrystal()}

EXAMPLES:

\begin{verbatim}
sage: C = crystals.KirillovReshetikhin(['A',3,1], 1, 2)
sage: elt = C(1,4)
sage: list(elt.subcrystal(index_set=[1,3]))
[[[1, 4]], [[1, 3]], [[2, 4]], [[2, 3]]]
\end{verbatim}
 sage: list(elt.subcrystal(index_set=[1,3], max_depth=1))
[[[1, 4]], [[1, 3]], [[2, 4]]]
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]]]

tensor (*elts)

Return the tensor product of self with the crystal elements elts.

EXAMPLES:

 sage: C = crystals.Letters(['A', 3])
sage: B = crystals.infinity.Tableaux(['A', 3])
sage: c = C[0]
sage: b = B.highest_weight_vector()
sage: t = c.tensor(c, b)
sage: ascii_art(t)  
   1  1  1
1 # 1 # 2  2
   3
sage: tensor([c, c, b]) == t
True
sage: ascii_art(tensor([b, b, c]))
  1  1  1  1  1  1
  2  2 # 2  2 # 1
  3  3

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)

Finite
alias of sage.categories.finite_crystals.FiniteCrystals
class MorphismMethods

is_embedding()
Check if self is an injective crystal morphism.

EXAMPLES:

sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_embedding()
True

sage: C = crystals.Tableaux(['A',2], shape=[2,1])
sage: B = crystals.infinity.Tableaux(['A',2])
sage: La = RootSystem(['A',2]).weight_lattice().fundamental_weights()
sage: W = crystals.elementary.T(['A',2], La[1]+La[2])
sage: T = W.tensor(B)
sage: mg = T(W.module_generators[0], B.module_generators[0])
sage: psi = Hom(C,T)([mg])
sage: psi.is_embedding()
True

is_isomorphism()
Check if \texttt{self} is a crystal isomorphism.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_isomorphism()
False
\end{verbatim}

\texttt{is\_strict()}\newline
Check if \texttt{self} is a strict crystal morphism.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = crystals.Tableaux(['C',2], shape=[1,1])
sage: C = crystals.Tableaux(['C',2], ([2,1], [1,1]))
sage: psi = B.crystal_morphism(C.module_generators[1:], codomain=C)
sage: psi.is_strict()
True
\end{verbatim}

class ParentMethods

\texttt{Lambda()}\newline
Returns the fundamental weights in the weight lattice realization for the root system associated with the crystal

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A', 5])
sage: C.Lambda()
Finite family {1: (1, 0, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0, 0), 3: (1, 1, 1, 0, 0, 0), 4: (1, 1, 1, 1, 0, 0), 5: (1, 1, 1, 1, 1, 0)}
\end{verbatim}

\texttt{an\_element()}\newline
Returns an element of \texttt{self}

\begin{verbatim}
sage: C = crystals.Letters(['A', 5])
sage: C.an_element()
sage: 1
\end{verbatim}

\texttt{cartan\_type()}\newline
Returns the Cartan type of the crystal

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C = crystals.Letters(['A',2])
sage: C.cartan_type()
['A', 2]
\end{verbatim}

\texttt{connected\_components()}\newline
Return the connected components of \texttt{self} as subcrystals.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: T.connected_components()
[Subcrystal of Full tensor product of the crystals
 [The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
  The crystal of letters for type ['A', 2]],
  ...] (continues on next page)
Subcrystal of Full tensor product of the crystals
[The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2]],
Subcrystal of Full tensor product of the crystals
[The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2]]

connected_components_generators()
Return a tuple of generators for each of the connected components of self.

EXAMPLES:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: T.connected_components_generators()
([[[1, 1], [2]], 1], [[[1, 2], [2]], 1], [[[1, 2], [3]], 1])
```

crystal_morphism(on_gens, codomain=None, cartan_type=None, index_set=None,
generators=None, automorphism=None, virtualization=None, scaling_factors=None,
category=None, check=True)
Construct a crystal morphism from self to another crystal codomain.

INPUT:
  • on_gens – a function or list that determines the image of the generators (if given a list, then this
    uses the order of the generators of the domain) of self under the crystal morphism
  • codomain – (default: self) the codomain of the morphism
  • cartan_type – (optional) the Cartan type of the morphism; the default is the Cartan type of self
  • index_set – (optional) the index set of the morphism; the default is the index set of the Cartan
type
  • generators – (optional) the generators to define the morphism; the default is the generators of self
  • automorphism – (optional) the automorphism to perform the twisting
  • virtualization – (optional) a dictionary whose keys are in the index set of the domain and
    whose values are lists of entries in the index set of the codomain; the default is the identity
dictionary
  • scaling_factors – (optional) a dictionary whose keys are in the index set of the domain and
    whose values are scaling factors for the weight, \( \varepsilon \) and \( \varphi \); the default are all scaling factors to be one
  • category – (optional) the category for the crystal morphism; the default is the category of Crystals.
  • check – (default: True) check if the crystal morphism is valid

See also:
For more examples, see sage.categories.crystals.CrystalHomset.

EXAMPLES:

We construct the natural embedding of a crystal using tableaux into the tensor product of single boxes via the reading word:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: F = crystals.Tableaux(['A',2], shape=[1])
sage: T = crystals.TensorProduct(F,F,F)
sage: mg = T.highest_weight_vectors()[2]; mg
```

We now take the other isomorphic highest weight component in the tensor product:

```
sage: mg = T.highest_weight_vectors()[1]; mg
[[[2]], [[1]], [[1]]]
sage: psi = B.crystal_morphism([mg], codomain=T)
sage: psi(lw) == mg.to_lowest_weight()[0]
True
```

We construct a crystal morphism of classical crystals using a Kirillov-Reshetikhin crystal:

```
sage: B = crystals.Tableaux(['D', 4], shape=[1,1])
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,2)
sage: v = K.module_generators[1]
sage: psi = B.crystal_morphism([v], codomain=K, category=FiniteCrystals())
sage: psi
['D', 4] -> ['D', 4, 1] Virtual Crystal morphism:
  From: The crystal of tableaux of type ['D', 4] and shape(s) [[1, 1]]
  To: Kirillov-Reshetikhin crystal of type ['D', 4, 1] with (r,s)=(2,2)
  Defn: [1, 2] |--> [3, 3]
```

We can define crystal morphisms using a different set of generators. For example, we construct an example using the lowest weight vector:
We can also use a dictionary to specify the generators and their images:

```
sage: psi = Bp.crystal_morphism({Bp.lowest_weight_vectors()[0]: x})
sage: psi(Bp.highest_weight_vector())
[[1, 1], [2]]
```

We construct a twisted crystal morphism induced from the diagram automorphism of type $A_3^{(1)}$:

```
sage: La = RootSystem(['A', 3, 1]).weight_lattice(extended=True).fundamental_weights()
sage: B0 = crystals.GeneralizedYoungWalls(3, La[0])
sage: B1 = crystals.GeneralizedYoungWalls(3, La[1])
sage: phi = B0.crystal_morphism(B1.module_generators, automorphism={0:1, 1:2, 2:3, 3:0})
sage: phi
['A', 3, 1] Twisted Crystal morphism:
  From: Highest weight crystal of generalized Young walls of Cartan type ['A', 3, 1] and highest weight Lambda[0]
  To: Highest weight crystal of generalized Young walls of Cartan type ['A', 3, 1] and highest weight Lambda[1]
  Defn: [] |--> []
sage: x = B0.module_generators[0].f_string([0, 1, 2, 3]); x
[[0, 3], [1], [2]]
sage: phi(x)
[[], [1, 0], [2], [3]]
```

We construct a virtual crystal morphism from type $G_2$ into type $D_4$:

```
sage: D = crystals.Tableaux(['D', 4], shape=[1,1])
sage: G = crystals.Tableaux(['G', 2], shape=[1])
sage: psi = G.crystal_morphism(D.module_generators,
...:   virtualization={1:[2], 2:[1,3,4]},
...:   scaling_factors={1:1, 2:1})
sage: for x in G:
...:     ascii_art(x, psi(x), sep=' |--> ')
...:     print("")
1 |--> 2
1
2 |--> 3
2
3 |--> -3
3
```

(continues on next page)
digraph \((subset=None, index_set=None)\)

Return the DiGraph associated to self.

INPUT:

• \(\text{subset} \) – (optional) a subset of vertices for which the digraph should be constructed

• \(\text{index_set} \) – (optional) the index set to draw arrows

EXAMPLES:

```python
sage: C = Crystals().example(5)
sage: C.digraph()
Digraph on 6 vertices
```

The edges of the crystal graph are by default colored using blue for edge 1, red for edge 2, and green for edge 3:

```python
sage: C = Crystals().example(3)
sage: G = C.digraph()
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external ->window)
```

One may also overwrite the colors:

```python
sage: C = Crystals().example(3)
sage: G = C.digraph()
sage: G.set_latex_options(color_by_label = {1:"red", 2:"purple", 3:"blue"})
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external ->window)
```

Or one may add colors to yet unspecified edges:

```python
sage: C = Crystals().example(4)
sage: G = C.digraph()
sage: C.cartan_type()._index_set_coloring[4]="purple"
sage: view(G)  # optional - dot2tex graphviz, not tested (opens external ->window)
```

Here is an example of how to take the top part up to a given depth of an infinite dimensional crystal:

```python
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
```
Here is a way to construct a picture of a Demazure crystal using the \texttt{subset} option:

\begin{verbatim}
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = B.highest_weight_vector()
sage: D = B.demazure_subcrystal(t, [2,1])
sage: list(D)
[[[1, 1], [2]], [[1, 1], [3]], [[1, 2], [2]],
 [[[1, 3], [2]], [[1, 3], [3]]]
sage: view(D)  # optional - dot2tex graphviz, not tested (opens external\window)
\end{verbatim}

We can also choose to display particular arrows using the \texttt{index_set} option:

\begin{verbatim}
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)  # optional - dot2tex graphviz, not tested (opens external\window)
\end{verbatim}

\textbf{Todo:} Add more tests.

\section{direct\_sum\(X\)}

Return the direct sum of \texttt{self} with \(X\).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: B.direct_sum(C)
Direct sum of the crystals Family
(The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2])
\end{verbatim}

As a shorthand, we can use +:

\begin{verbatim}
sage: B + C
Direct sum of the crystals Family
(The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]],
The crystal of letters for type ['A', 2])
\end{verbatim}

\section{dot\_tex()} 

Return a dot\_tex string representation of \texttt{self}.

\textbf{EXAMPLES:}
```python
sage: C = crystals.Letters(['A',2])
sage: C.dot_tex()
digraph G {
  node [ shape=plaintext ];
  N_0 [ label = " ", texlbl = "$1$" ];
  N_1 [ label = " ", texlbl = "$2$" ];
  N_2 [ label = " ", texlbl = "$3$" ];
  N_0 -> N_1 [ label = " ", texlbl = "1" ];
  N_1 -> N_2 [ label = " ", texlbl = "2" ];
}
```

**index_set()**

Returns the index set of the Dynkin diagram underlying the crystal.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: C.index_set()
(1, 2, 3, 4, 5)
```

**is_connected()**

Return True if self is a connected crystal.

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: B.is_connected()
True
sage: T.is_connected()
False
```

**latex(** **options**)**

Returns the crystal graph as a latex string. This can be exported to a file with self.latex_file('filename').

**EXAMPLES:**

```python
sage: T = crystals.Tableaux(['A',2],shape=[1])
sage: T._latex_()
'...tikzpicture...'
sage: view(T) # optional - dot2tex graphviz, not tested (opens external window)
```

One can for example also color the edges using the following options:

```python
sage: T = crystals.Tableaux(['A',2],shape=[1])
sage: T._latex_(color_by_label={0:"black", 1:"red", 2:"blue"})
'...tikzpicture...'
```

**latex_file( **filename**)**

Export a file, suitable for pdflatex, to ‘filename’.

This requires a proper installation of dot2tex in sage-python. For more information see the documentation for self.latex().

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: fn = tmp_filename(ext='.tex')
sage: C.latex_file(fn)
```

metapost \((\text{filename, thicklines=False, labels=True, scaling_factor=1.0, tallness=1.0})\)

Use `C.metapost(“filename.mp”, [options])`, where options can be:

- `thicklines = True` (for thicker edges)
- `labels = False` (to suppress labeling of the vertices)
- `scaling_factor=value`, where value is a floating point number, 1.0 by default. Increasing or decreasing the scaling factor changes the size of the image.
- `tallness=1.0`. Increasing makes the image taller without increasing the width.

Root operators e(1) or f(1) move along red lines, e(2) or f(2) along green. The highest weight is in the lower left. Vertices with the same weight are kept close together. The concise labels on the nodes are strings introduced by Berenstein and Zelevinsky and Littelmann; see Littelmann’s paper Cones, Crystals, Patterns, sections 5 and 6.

For Cartan types B2 or C2, the pattern has the form

\[ a_2 \, a_3 \, a_4 \, a_1 \]

where \(c*a_2 = a_3 = 2*a_4 = 0\) and \(a_1=0\), with \(c=2\) for B2, \(c=1\) for C2. Applying \(e(2)\) \(a_1\) times, \(e(1)\) \(a_2\) times, \(e(2)\) \(a_3\) times, \(e(1)\) \(a_4\) times returns to the highest weight. (Observe that Littelmann writes the roots in opposite of the usual order, so our e(1) is his e(2) for these Cartan types.) For type A2, the pattern has the form

\[ a_3 \, a_2 \, a_1 \]

where applying \(e(1)\) \(a_1\) times, \(e(2)\) \(a_2\) times then \(e(3)\) \(a_1\) times returns to the highest weight. These data determine the vertex and may be translated into a Gelfand-Tsetlin pattern or tableau.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 2])
sage: C.metapost(tmp_filename())
```

```python
sage: C = crystals.Letters(['A', 5])
sage: C.metapost(tmp_filename())
```

Traceback (most recent call last):
...
NotImplementedError

number_of_connected_components()

Return the number of connected components of \(\text{self}\).

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(B,C)
sage: T.number_of_connected_components()
3
```

plot (**options)**

Return the plot of \(\text{self}\) as a directed graph.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: print(C.plot())
```

Graphics object consisting of 17 graphics primitives

plot3d (**options)**

Return the 3-dimensional plot of \(\text{self}\) as a directed graph.
EXAMPLES:

```
sage: C = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: print(C.plot3d())
Graphics3d Object
```

```
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: G = C.subcrystal(index_set=[1,2,3]).digraph()
sage: GA = crystals.Tableaux('A3', shape=[2]).digraph()
sage: G.is_isomorphic(GA, edge_labels=True)
True
```

We construct the subcrystal which contains the necessary data to construct the corresponding dual equivalence graph:

```
sage: C = crystals.Tableaux(['A',5], shape=[3,3])
sage: is_wt0 = lambda x: all(x.epsilon(i) == x.phi(i) for i in x.parent().index_set())
sage: G = C.subcrystal(index_set=[1,2,3]).digraph()
sage: GA = crystals.Tableaux('A3', shape=[2]).digraph()
sage: G.is_isomorphic(GA, edge_labels=True)
True
```
sage: def check(x):
....:     if is_wt0(x):
....:         return True
....:     for i in x.parent().index_set()[:-1]:
....:         L = [x.e(i), x.e_string([i,i+1]), x.f(i), x.f_string([i, i+1])]
....:         if any(y is not None and is_wt0(y) for y in L):
....:             return True
....:     return False
sage: wt0 = [x for x in C if is_wt0(x)]
sage: S = C.subcrystal(contained=check, generators=wt0)
sage: S.module_generators[0]
[[1, 3, 5], [2, 4, 6]]
sage: S.module_generators[0].e(2).e(3).f(2).f(3)
[[1, 2, 5], [3, 4, 6]]

An example of a type $B_2$ virtual crystal inside of a type $A_3$ ambient crystal:

sage: A = crystals.Tableaux(['A',3], shape=[2,1,1])
sage: S = A.subcrystal(virtualization={1:[1,3], 2:[2]},
....:     scaling_factors={1:1,2:1}, cartan_type=['B',2])
sage: B = crystals.Tableaux(['B',2], shape=[1])
sage: S.digraph().is_isomorphic(B.digraph(), edge_labels=True)
True

\texttt{tensor}(\textit{\texttt{\*crystals, \*\textit{options}}})

Return the tensor product of \texttt{self} with the crystals \texttt{B}.

\textbf{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]) \texttt{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]]

\texttt{weight\_lattice\_realization}()

Return the weight lattice realization used to express weights in \texttt{self}.

This default implementation uses the ambient space of the root system for (non relabelled) finite types and the weight lattice otherwise. This is a legacy from when ambient spaces were partially implemented, and may be changed in the future.

For affine types, this returns the extended weight lattice by default.
EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

class SubcategoryMethods
Methods for all subcategories.

**TensorProducts()**
Return the full subcategory of objects of self constructed as tensor products.

See also:
- `tensor.TensorProductsCategory`
- `RegressiveCovariantFunctorialConstruction`.

EXAMPLES:

```python
sage: HighestWeightCrystals().TensorProducts()
Category of tensor products of highest weight crystals
```

class TensorProducts(category, *args)
Bases: `sage.categories.tensor.TensorProductsCategory`

The category of crystals constructed by tensor product of crystals.

**extra_super_categories()**

EXAMPLES:

```python
sage: Crystals().TensorProducts().extra_super_categories()
[Category of crystals]
```

def example(choice='highwt', **kwds)
Returns an example of a crystal, as per `Category.example()`.

**INPUT:**
- `choice` – str [default: ‘highwt’]. Can be either ‘highwt’ for the highest weight crystal of type A, or ‘naive’ for an example of a broken crystal.
- `**kwds` – keyword arguments passed onto the constructor for the chosen crystal.

**EXAMPLES:**

```python
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.
```

def super_categories()

**EXAMPLES:**

```python
sage: Crystals().super_categories()
[Category of enumerated sets]
```
class sage.categories.cw_complexes.CWComplexes(s=None)
    Bases: sage.categories.category_singleton.Category_singleton

The category of CW complexes.

A CW complex is a Closure-finite cell complex in the Weak topology.

REFERENCES:

- Wikipedia article CW_complex

Note: The notion of “finite” is that the number of cells is finite.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes(); C
Category of CW complexes
```

```
Compact_extra_super_categories()

Return extraneous super categories for CWComplexes().Compact().

A compact CW complex is finite, see Proposition A.1 in [Hat2002].

Todo: Fix the name of finite CW complexes.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().Compact() # indirect doctest
Category of finite finite dimensional CW complexes
sage: CWComplexes().Compact() is CWComplexes().Finite()
True
```

class Connected(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of connected CW complexes.

class Finite(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

Category of finite CW complexes.
A finite CW complex is a CW complex with a finite number of cells.

**class ParentMethods**

**dimension()**
Return the dimension of self.

**EXAMPLES:**

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.dimension()
2
```

**extra_super_categories()**
Return the extra super categories of self.

A finite CW complex is a compact finite-dimensional CW complex.

**EXAMPLES:**

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes().Finite()
sage: C.extra_super_categories()
[Category of finite dimensional CW complexes, 
Category of compact topological spaces]
```

**class FiniteDimensional(base_category)**
Bases: `sage.categories.category_with_axiom.CategoryWithAxiom`

Category of finite dimensional CW complexes.

**class ParentMethods**

**cells()**
Return the cells of self.

**EXAMPLES:**

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: C = X.cells()
sage: sorted((d, C[d]) for d in C.keys())
[(0, (0-cell v,)), 
 (1, (0-cell e1, 0-cell e2)), 
 (2, (2-cell f,))]
```

**dimension()**
Return the dimension of self.

**EXAMPLES:**

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
sage: X.dimension()
2
```
Connected()

Return the full subcategory of the connected objects of self.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().Connected()
Category of connected CW complexes
```

FiniteDimensional()

Return the full subcategory of the finite dimensional objects of self.

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: C = CWComplexes().FiniteDimensional(); C
Category of finite dimensional CW complexes
```

super_categories()

EXAMPLES:

```
sage: from sage.categories.cw_complexes import CWComplexes
sage: CWComplexes().super_categories()
[Category of topological spaces]
```

## 3.33 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 ParentMethods

residue_field()

Return the residue field of the ring of integers of this discrete valuation field.

EXAMPLES:
uniformizer()
Return a uniformizer of this ring.

EXAMPLES:

\[
\text{sage: } \text{Qp(5).uniformizer()}
\]
\[
5 + 0(5^{21})
\]

super_categories()
EXAMPLES:

\[
\text{sage: } \text{DiscreteValuationFields().super_categories()}
\]
[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:

\[
\text{sage: } \text{GF(7)['x'] in DiscreteValuationRings()}
\]
True
\[
\text{sage: } \text{TestSuite(DiscreteValuationRings()).run()}
\]

class ElementMethods

euclidean_degree()
Return the Euclidean degree of this element.

gcd(other)
Return the greatest common divisor of self and other, normalized so that it is a power of the distinguished uniformizer.

is_unit()
Return True if self is invertible.

EXAMPLES:

\[
\text{sage: } \text{x = Zp(5)(50)}
\]
\[
\text{sage: } \text{x.is_unit()}
\]
False
\[
\text{sage: } \text{x = Zp(7)(50)}
\]
\[
\text{sage: } \text{x.is_unit()}
\]
True

lcm(other)
Return the least common multiple of self and other, normalized so that it is a power of the distinguished uniformizer.
**quo_rem(other)**
Return the quotient and remainder for Euclidean division of `self` by `other`.

**valuation()**
Return the valuation of this element.

**EXAMPLES:**
```
sage: x = Zp(5)(50)
sage: x.valuation()
2
```

**class 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[[[]]
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[[[]]
K.uniformizer()
u
```

**super_categories()**

**EXAMPLES:**
```
sage: DiscreteValuationRings().super_categories()
[Category of euclidean domains]
```

### 3.34 Distributive Magmas and Additive Magmas

**class sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas**

**Bases:** `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of sets \((S, +, \cdot)\) with \(\cdot\) distributing on \(+\).

This is similar to a ring, but \(+\) and \(\cdot\) are only required to be (additive) magmas.

**EXAMPLES:**
from sage.categories.distributive_magmas_and_additive_magmas import DistributiveMagmasAndAdditiveMagmas

C = DistributiveMagmasAndAdditiveMagmas(); C
Category of distributive magmas and additive magmas

C.super_categories()
[Category of magmas and additive magmas]

class AdditiveAssociative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class AdditiveCommutative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class AdditiveUnital(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class Associative(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

    class AdditiveInverse
        alias of sage.categories.rngs.Rngs

    class Unital
        alias of sage.categories.semirings.Semirings

class CartesianProducts(category, *args)
    Bases: sage.categories.cartesian_product.CartesianProductsCategory

easy_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.easy_super_categories()
        [Category of distributive magmas and additive magmas]
        sage: C.axioms()
        frozenset({"Distributive")

class ParentMethods

3.35 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]

Commutative alias of sage.categories.fields.Fields
class ElementMethods

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 enumerated fields

Warning: This is not implemented in DivisionRings.Finite.
extra_super_categories because the categories of finite division rings and of finite
fields coincide. See the section Deduction rules in the documentation of axioms.

class ParentMethods

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

3.36 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
```

3.37 Enumerated Sets

class sage.categories.enumerated_setsEnumeratedSets(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of enumerated sets

An enumerated set is a finite or countable set or multiset \( S \) together with a canonical enumeration of its elements; conceptually, this is very similar to an immutable list. The main difference lies in the names and the return type of the methods, and of course the fact that the list of elements is not supposed to be expanded in memory. Whenever possible one should use one of the two sub-categories FiniteEnumeratedSets or InfiniteEnumeratedSets.

The purpose of this category is threefold:

- to fix a common interface for all these sets;
- to provide a bunch of default implementations;
- to provide consistency tests.

The standard methods for an enumerated set \( S \) are:

- \( S.cardinality() \): the number of elements of the set. This is the equivalent for \( \text{len} \) on a list except that the return value is specified to be a Sage Integer or infinity, instead of a Python int.
- \( \text{iter}(S) \): an iterator for the elements of the set;
- \( S.list() \): 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 equivalent for \( l[n] \) on a list.
- \( S.rank(e) \): the position of the element \( e \) in the set; This is equivalent to \( l.index(e) \) for a list except that the return value is specified to be a Sage Integer, instead of a Python int.
• \( S\text{.first()} \): the first object of the set; it is equivalent to \( S\text{.unrank}(0) \).

• \( S\text{.next}(e) \): the object of the set which follows \( e \); It is equivalent to \( S\text{.unrank}(S\text{.rank}(e)+1) \).

• \( S\text{.random_element()} \): a random generator for an element of the set. Unless otherwise stated, and for finite enumerated sets, the probability is uniform.

For examples and tests see:

• \( \text{FiniteEnumeratedSets()\text{.example()}} \)

• \( \text{InfiniteEnumeratedSets()\text{.example()}} \)

EXAMPLES:

```python
sage: EnumeratedSets()
Category of enumerated sets
sage: EnumeratedSets().super_categories()
[Category of sets]
```

```python
class CartesianProducts(category, *args)
    Bases: sage.categories.cartesian_product.CartesianProductsCategory
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 ElementMethods
    rank()
    Return the rank of \( \text{self} \) in its parent.
    See also \( \text{EnumeratedSets.ElementMethods.rank()} \)
    EXAMPLES:
    ```
    sage: F = FiniteSemigroups().example(('a','b','c'))
sage: L = list(F)
sage: L[7].rank()
7
```

Finite
    alias of \( \text{sage.categories.finite_enumerated_sets.FiniteEnumeratedSets} \)

Infinite
    alias of \( \text{sage.categories.infinite_enumerated_sets.InfiniteEnumeratedSets} \)

class ParentMethods
The “first” element of self.

```
sage: 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()  # doctest: +NORMALIZE_WHITESPACE
False
sage: F = FiniteEnumeratedSet([])
sage: F.is_empty()  # doctest: +NORMALIZE_WHITESPACE
True
```

**iterator_range**(start=None, stop=None, step=None)

Iterate over the range of elements of self starting at start, ending at stop, and stepping by step.

See also:

unrank(), unrank_range()

**EXAMPLES:**

```
sage: P = Partitions()
sage: list(P.iterator_range(stop=5))
[[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(0, 5))
[[], [1], [2], [1, 1], [3]]
sage: list(P.iterator_range(3, 5))
[[1, 1], [3]]
sage: list(P.iterator_range(3, 10))
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
sage: list(P.iterator_range(3, 10, 2))
[[1, 1], [2, 1], [4], [2, 2]]
sage: it = P.iterator_range(3)
sage: [next(it) for x in range(10)]
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1], [5]]
sage: it = P.iterator_range(3, step=2)
sage: [next(it) for x in range(5)]
[[1, 1], [2, 1], [4], [2, 2], [1, 1, 1, 1]]
sage: next(P.iterator_range(stop=-3))
Traceback (most recent call last):
...  
NotImplementedError: cannot list an infinite set
sage: next(P.iterator_range(start=-3))
```

(continues on next page)
list()

Return a list of the elements of self.

The elements of set \( x \) are created and cached on the first call of \( x.list() \). Then each call of \( x.list() \) returns a new list from the cached result. Thus in looping, it may be better to do for \( e \) in \( x \); not for \( e \) in \( x.list() \).

If \( x \) is not known to be finite, then an exception is raised.

EXAMPLES:

```python
sage: (GF(3)^2).list()
[(0, 0), (1, 0), (2, 0), (0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)]
sage: R = Integers(11)
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: 1 = R.list(); 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: 1.remove(0); 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: R.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

map(\( f, name=\text{None} \))

Return 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:

```python
sage: R = Compositions(4).map(attrcall('partial_sums')); R
Image of Compositions of 4 by *.partial_sums()
sage: R.cardinality()
8
sage: R.list()
[[1, 2, 3, 4], [1, 2, 4], [1, 3, 4], [1, 4], [2, 3, 4], [2, 4], [3, 4], [4]]
sage: [ r for r in R]
[[1, 2, 3, 4], [1, 2, 4], [1, 3, 4], [1, 4], [2, 3, 4], [2, 4], [3, 4], [4]]
```

**Warning:** If the function is not injective, then there may be repeated elements:

```python
sage: P = Compositions(4)
sage: P.list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: P.map(attrcall('major_index')).list()
[6, 3, 4, 1, 5, 2, 3, 0]
```
Warning: `MapCombinatorialClass` needs to be refactored to use categories:

```python
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:

```python
sage: C = InfiniteEnumeratedSets().example()
sage: C._next_from_iterator(10) # indirect doctest
11
```

TODO: specify the behavior when obj is not in self.

**random_element()**

Return a random element in self.

Unless otherwise stated, and for finite enumerated sets, the probability is uniform.

This is a generic implementation from the category `EnumeratedSets()`. It raise a `NotImplementedError` since one does not know whether the set is finite.

EXAMPLES:

```python
sage: class broken(UniqueRepresentation, Parent):
    ....:     def __init__(self):
    ....:         Parent.__init__(self, category = EnumeratedSets())
sage: broken().random_element()
Traceback (most recent call last):
...
NotImplementedError: unknown cardinality
```

**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:

```python
sage: C = FiniteEnumeratedSets().example()
sage: list(C)
[1, 2, 3]
sage: C.rank(3) # indirect doctest
```

(continues on next page)
some_elements()

Return some elements in self.

See TestSuite for a typical use case.

This is a generic implementation from the category `EnumeratedSets()` which can be used when the method `__iter__` is provided. It returns an iterator for up to the first 100 elements of self.

**EXAMPLES:**

```python
sage: C = FiniteEnumeratedSets().example()
sage: list(C.some_elements())  # indirect doctest
[1, 2, 3]
```

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:**

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.unrank(2)  # indirect doctest
3
sage: C._unrank_from_iterator(5)
Traceback (most recent call last):
... ValueError: the value must be between 0 and 2 inclusive
```

unrank_range(start=None, stop=None, step=None)

Return the range of elements of self starting at start, ending at stop, and stepping by step.

See also:

unrank(), iterator_range()

**EXAMPLES:**

```python
sage: P = Partitions()
sage: P.unrank_range(stop=5)
[[], [1], [2], [1, 1], [3]]
sage: P.unrank_range(0, 5)
[[], [1], [2], [1, 1], [3]]
sage: P.unrank_range(3, 5)
[[1, 1], [3]]
sage: P.unrank_range(3, 10)
[[1, 1], [3], [2, 1], [1, 1, 1], [4], [3, 1], [2, 2]]
sage: P.unrank_range(3, 10, 2)
[[1, 1], [2, 1], [4], [2, 2]]
sage: P.unrank_range(3)
Traceback (most recent call last):
...
```
additional_structure()
Return None.

Indeed, morphisms of enumerated sets are not required to preserve the enumeration.

See also:
Category.additional_structure()

EXAMPLES:
sage: EnumeratedSets().additional_structure()

super_categories()
EXAMPLES:
sage: EnumeratedSets().super_categories()

[Category of sets]

3.38 Euclidean domains

AUTHORS:
• Teresa Gomez-Diaz (2008): initial version
• Julian Rueth (2013-09-13): added euclidean degree, quotient remainder, and their tests
class sage.categories.euclidean_domains.EuclideanDomains(s=None)
   Bases: 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]

class ElementMethods

euclidean_degree()
Return the degree of this element as an element of an Euclidean domain, i.e., for elements $a, b$ the
euclidean degree $f$ satisfies the usual properties:
1. if $b$ is not zero, then there are elements $q$ and $r$ such that $a = bq + r$ with $r = 0$ or $f(r) < f(b)$
2. if \( a, b \) are not zero, then \( f(a) \leq f(ab) \)

**Note:** The name `euclidean_degree` was chosen because the euclidean function has different names in different contexts, e.g., absolute value for integers, degree for polynomials.

**OUTPUT:**

For non-zero elements, a natural number. For the zero element, this might raise an exception or produce some other output, depending on the implementation.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: x.euclidean_degree()
1
sage: ZZ.one().euclidean_degree()
1
```

**`gcd` (other)**

Return the greatest common divisor of this element and `other`.

**INPUT:**

- `other` – an element in the same ring as `self`

**ALGORITHM:**

Algorithm 3.2.1 in [Coh1993].

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(QQ, sparse=True)
sage: EuclideanDomains().element_class.gcd(x,x+1)
-1
```

**`quo_rem` (other)**

Return the quotient and remainder of the division of this element by the non-zero element `other`.

**INPUT:**

- `other` – an element in the same euclidean domain

**OUTPUT:**

a pair of elements

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: x.quo_rem(x)
(1, 0)
```

**class ParentMethods**

**`gcd_free_basis` (elts)**

Compute a set of coprime elements that can be used to express the elements of `elts`.

**INPUT:**

- `elts` - A sequence of elements of `self`

**OUTPUT:**

A GCD-free basis (also called a coprime base) of `elts`; that is, a set of pairwise relatively prime elements of `self` such that any element of `elts` can be written as a product of elements of the set.
Algorithm:

Naive implementation of the algorithm described in Section 4.8 of Bach & Shallit [BS1996].

Examples:

```python
sage: ZZ.gcd_free_basis([1])
[]
sage: ZZ.gcd_free_basis([4, 30, 14, 49])
[2, 15, 7]
sage: Pol.<x> = QQ[]
sage: sorted(Pol.gcd_free_basis([
..... (x+1)^3*(x+2)^3*(x+3), (x+1)*(x+2)*(x+3),
..... (x+1)*(x+2)*(x+4)]))
[x + 3, x + 4, x^2 + 3*x + 2]
```

`is_euclidean_domain()`

Return True, since this in an object of the category of Euclidean domains.

Examples:

```python
sage: Parent(QQ,category=EuclideanDomains()).is_euclidean_domain()
True
```

`super_categories()`

Examples:

```python
sage: EuclideanDomains().super_categories()
[Category of principal ideal domains]
```

### 3.39 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:

```python
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
```

Class `ElementMethods`
euclidean_degree()
Return the degree of this element as an element of an Euclidean domain.
In a field, this returns 0 for all but the zero element (for which it is undefined).

EXAMPLES:
```python
sage: QQ.one().euclidean_degree()
0
```

factor()
Return a factorization of self.
Since self is either a unit or zero, this function is trivial.

EXAMPLES:
```python
sage: x = GF(7)(5)
sage: x.factor()
5
sage: RR(0).factor()
Traceback (most recent call last):
  ...  
ArithmeticError: factorization of 0.000000000000000 is not defined
```

gcd(other)
Greatest common divisor.

**Note:** Since we are in a field and the greatest common divisor is only determined up to a unit, it is correct to either return zero or one. Note that fraction fields of unique factorization domains provide a more sophisticated gcd.

EXAMPLES:
```python
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:
```python
sage: gcd(15.0,12.0)
3.00000000000000
```

But for others floating point numbers, the gcd is just 0.0 or 1.0:
```python
sage: gcd(3.2, 2.18)
1.00000000000000
sage: gcd(0.0, 0.0)
0.00000000000000
```

AUTHOR:
\textbf{inverse_of_unit()}  
Return the inverse of this element.

**EXAMPLES:**

```
sage: NumberField(x^7+2,'a')(2).inverse_of_unit()
1/2
```

Trying to invert the zero element typically raises a \texttt{ZeroDivisionError}:

```
sage: QQ(0).inverse_of_unit()
Traceback (most recent call last):
...  
ZeroDivisionError: rational division by zero
```

To catch that exception in a way that also works for non-units in more general rings, use something like:

```
sage: try:
....:     QQ(0).inverse_of_unit()
....: except ArithmeticError:
....:     pass
```

Also note that some “fields” allow one to invert the zero element:

```
sage: RR(0).inverse_of_unit()
+infinity
```

\textbf{is_unit()}  
Returns True if \texttt{self} has a multiplicative inverse.

**EXAMPLES:**

```
sage: QQ(2).is_unit()
True
sage: QQ(0).is_unit()
False
```

\textbf{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:
But for others floating point numbers, it is just 0.0 or 1.0:

\[
\begin{align*}
\text{sage: } \text{lcm}(3.2, 2.18) & \quad 1.000000000000000 \\
\text{sage: } \text{lcm}(0.0, 0.0) & \quad 0.000000000000000
\end{align*}
\]

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:**

\[
\begin{align*}
\text{sage: } f, g = \text{QQ}(1), \text{QQ}(2) \\
\text{sage: } f.\text{quo_rem}(g) & \quad (1/2, 0)
\end{align*}
\]

**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 \(r = s \times \text{self} + t \times \text{other}\). Since the computations are done over a field, \(r\) is zero if self and other are zero, and one otherwise.

**AUTHORS:**

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } K = \text{GF}(5) \\
\text{sage: } K(2).\text{xgcd}(K(1)) & \quad (1, 3, 0) \\
\text{sage: } K(0).\text{xgcd}(K(4)) & \quad (1, 0, 4) \\
\text{sage: } K(1).\text{xgcd}(K(1)) & \quad (1, 1, 0) \\
\text{sage: } \text{GF}(5)(0).\text{xgcd}(\text{GF}(5)(0)) & \quad (0, 0, 0)
\end{align*}
\]

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 \(\mathbb{Z}\):

\[
\begin{align*}
\text{sage: } \text{xgcd}(12.0, 8.0) & \quad (4.000000000000000, 1.000000000000000, -1.000000000000000)
\end{align*}
\]

(continues on next page)
Finite alias of `sage.categories.finite_fields.FiniteFields`

class ParentMethods

```plaintext
fraction_field()

Returns the fraction field of self, which is self.

EXAMPLES:

```
sage: QQ.fraction_field() is QQ
True
```

```plaintext
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
```

```plaintext
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
```

```plaintext
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
```

extra_super_categories()

EXAMPLES:

```
3.40 Filtered Algebras

class sage.categories.filtered_algebras.FilteredAlgebras (base_category)
Bases: sage.categories.filtered_modules.FilteredModulesCategory

The category of filtered algebras.

An algebra \( A \) over a commutative ring \( R \) is filtered if \( A \) is endowed with a structure of a filtered \( R \)-module (whose underlying \( R \)-module structure is identical with that of the \( R \)-algebra \( A \) ) such that the indexing set \( I \) (typically \( I = \mathbb{N} \) ) is also an additive abelian monoid, the unity 1 of \( A \) belongs to \( F_0 \), and we have \( F_i \cdot F_j \subseteq F_{i+j} \) for all \( i, j \in I \).

EXAMPLES:

```
sage: Algebras(ZZ).Filtered()
Category of filtered algebras over Integer Ring
sage: Algebras(ZZ).Filtered().super_categories()
[Category of algebras over Integer Ring,
 Category of filtered modules over Integer Ring]
```

REFERENCES:

- Wikipedia article Filtered_algebra

class ParentMethods

graded_algebra ()

Return the associated graded algebra to self.

Todo: Implement a version of the associated graded algebra which does not require self to have a distinguished basis.

EXAMPLES:

```
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.graded_algebra()
Graded Algebra of An example of a filtered algebra with basis:
 the universal enveloping algebra of
 Lie algebra of RR^3 with cross product over Integer Ring
```

3.41 Filtered Algebras With Basis

A filtered algebra with basis over a commutative ring \( R \) is a filtered algebra over \( R \) endowed with the structure of a filtered module with basis (with the same underlying filtered-module structure). See FilteredAlgebras and FilteredModulesWithBasis for these two notions.

class sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis (base_category)
Bases: sage.categories.filtered_modules.FilteredModulesCategory
The category of filtered algebras with a distinguished homogeneous basis.

A filtered algebra with basis over a commutative ring $R$ is a filtered algebra over $R$ endowed with the structure of a filtered module with basis (with the same underlying filtered-module structure). See `FilteredAlgebras` and `FilteredModulesWithBasis` for these two notions.

**EXAMPLES:**

```python
sage: C = AlgebrasWithBasis(ZZ).Filtered(); C
Category of filtered algebras with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of algebras with basis over Integer Ring,
 Category of filtered algebras over Integer Ring,
 Category of filtered modules with basis over Integer Ring]
```

**class** `ElementMethods`

**class** `ParentMethods`

`from_graded_conversion()`  
Return the inverse of the canonical $R$-module isomorphism $A \to \text{gr } A$ induced by the basis of $A$ (where $A = \text{"}self\text{"}$). This inverse is an isomorphism $\text{gr } A \to A$.

This is an isomorphism of $R$-modules, not of algebras. See the class documentation `AssociatedGradedAlgebra`.

**See also:**

`to_graded_conversion()`

**EXAMPLES:**

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: p = A.an_element() + A.algebra_generators()['x'] + 2; p
U['x']^2*U['y']^2*U['z']^3 + 3*U['x'] + 3*U['y'] + 3
sage: q = A.to_graded_conversion()(p)
sage: A.from_graded_conversion()(q) == p
True
sage: q.parent() is A.graded_algebra()
True
```

`graded_algebra()`  
Return the associated graded algebra to `self`.

See `AssociatedGradedAlgebra` for the definition and the properties of this.

If the filtered algebra `self` with basis is called $A$, then this method returns $\text{gr } A$. The method `to_graded_conversion()` returns the canonical $R$-module isomorphism $A \to \text{gr } A$ induced by the basis of $A$, and the method `from_graded_conversion()` returns the inverse of this isomorphism. The method `projection()` projects elements of $A$ onto $\text{gr } A$ according to their place in the filtration on $A$.

**Warning:** When not overridden, this method returns the default implementation of an associated graded algebra – namely, `AssociatedGradedAlgebra(self)`, where `AssociatedGradedAlgebra` is `AssociatedGradedAlgebra`. But many instances of `FilteredAlgebrasWithBasis` override this method, as the associated graded algebra often is (isomorphic) to a simpler object (for instance, the associated graded algebra of a
graded algebra can be identified with the graded algebra itself). Generic code that uses associated graded algebras (such as the code of the `induced_graded_map()` method below) should make sure to only communicate with them via the `to_graded_conversion()`, `from_graded_conversion()`, and `projection()` methods (in particular, do not expect there to be a conversion from `self` to `self.graded_algebra()`; this currently does not work for Clifford algebras). Similarly, when overriding `graded_algebra()`, make sure to accordingly redefine these three methods, unless their definitions below still apply to your case (this will happen whenever the basis of your `graded_algebra()` has the same indexing set as `self`, and the partition of this indexing set according to degree is the same as for `self`).

**Todo:** Maybe the thing about the conversion from `self` to `self.graded_algebra()` on the Clifford at least could be made to work? (I would still warn the user against ASSUMING that it must work – as there is probably no way to guarantee it in all cases, and we shouldn’t require users to mess with element constructors.)

**EXAMPLES:**

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.graded_algebra()
Graded Algebra of An example of a filtered algebra with basis:
the universal enveloping algebra of
Lie algebra of RR^3 with cross product over Integer Ring
```

**induced_graded_map**(other, f)

Return the graded linear map between the associated graded algebras of `self` and `other` canonically induced by the filtration-preserving map `f : self -> other`.

Let `A` and `B` be two filtered algebras with basis, and let \((F_i)_{i \in I}\) and \((G_i)_{i \in I}\) be their filtrations. Let \(f : A \to B\) be a linear map which preserves the filtration (i.e., satisfies \(f(F_i) \subseteq G_i\) for all \(i \in I\)). Then, there is a canonically defined graded linear map \(grf : gr A \to gr B\) which satisfies

\[(gr f)(p_i(a)) = p_i(f(a)) \quad \text{for all } i \in I \text{ and } a \in F_i,
\]

where the \(p_i\) on the left hand side is the canonical projection from `F_i` onto the \(i\)-th graded component of `gr A`, while the \(p_i\) on the right hand side is the canonical projection from `G_i` onto the \(i\)-th graded component of `gr B`.

**INPUT:**

- `other` – a filtered algebra with basis
- `f` – a filtration-preserving linear map from `self` to `other` (can be given as a morphism or as a function)

**OUTPUT:**

The graded linear map `gr f`.

**EXAMPLES:**

**Example 1.**

We start with the universal enveloping algebra of the Lie algebra \(R^3\) (with the cross product serving as Lie bracket):

```python
sage: A = AlgebrasWithBasis(QQ).Filtered().example(); A
An example of a filtered algebra with basis: the
universal enveloping algebra of Lie algebra of RR^3
```

(continues on next page)
Let us define a stupid filtered map from $A$ to itself:

```python
sage: def map_on_basis(m):
    d = m.dict()
    i = d.get('x', 0); j = d.get('y', 0); k = d.get('z', 0)
    g = (y ** (i+j)) * (z ** k)
    if i > 0:
        g += i * (x ** (i-1)) * (y ** j) * (z ** k)
    return g

sage: f = A.module_morphism(on_basis=map_on_basis, codomain=A)
```

```python
sage: f(x)
U['y'] + 1
sage: f(x*y*z)
U['y']^2*U['z'] + U['y']*U['z']
sage: f(x*x*y*z)
U['y']^3*U['z'] + 2*U['x']*U['y']*U['z']
```

(There is nothing here that is peculiar to this universal enveloping algebra; we are only using its module structure, and we could just as well be using a polynomial algebra in its stead.)

We now compute $\text{gr } f$

```python
sage: grA = A.graded_algebra(); grA
Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Rational Field

sage: xx, yy, zz = [A.to_graded_conversion()(i) for i in [x, y, z]]

sage: grf = A.induced_graded_map(A, f); grf
Generic endomorphism of Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Rational Field

sage: grf(xx)
bar(U['y'])

sage: grf(xx*yy*zz)
bar(U['y'])*U['z']

sage: grf(xx*xx*yy*zz)
bar(U['y']^3*U['z'])

sage: grf(grA.one())
bar(1)

sage: grf(yy*zz)
bar(U['y']*U['z'])

sage: grf(yy*zz-2*yy)
bar(U['y']*U['z']) - 2*bar(U['y'])
```

Example 2.
We shall now construct $\text{gr } f$ for a different map $f$ out of the same $A$; the new map $f$ will lead into a graded algebra already, namely into the algebra of symmetric functions:

```python
sage: h = SymmetricFunctions(QQ).h()
sage: def map_on_basis(m): # redefining map_on_basis
...:     d = m.dict()
...:     i = d.get('x', 0); j = d.get('y', 0); k = d.get('z', 0)
...:     g = (h[1] ** i) * (h[2] ** (floor(j/2))) * (h[3] ** (floor(k/3)))
...:     g += i * (h[1] ** (i+j+k))
...:     return g
sage: f = A.module_morphism(on_basis=map_on_basis, codomain=h) # redefining f
sage: f(x) 2*h[1]
sage: f(y) h[]
sage: f(z) h[]
sage: f(y**2) h[2]
sage: f(x**2) 3*h[1, 1]
sage: f(x*y*z) h[1] + h[1, 1, 1]
sage: f(x*x*y*y*z) 2*h[1, 1, 1, 1] + h[2, 1, 1]
sage: f(A.one())
```

The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:

```python
sage: grh = h.graded_algebra(); grh is h
True
sage: grf = A.induced_graded_map(h, f); grf
Generic morphism:
  From: Graded Algebra of An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Rational Field
  To:  Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(xx) 2*h[1]
sage: grf(yy) 0
sage: grf(zz) 0
sage: grf(yy**2) h[2]
sage: grf(xx**2) 3*h[1, 1]
sage: grf(xx*yy*zz) h[1, 1, 1]
sage: grf(xx*xx*yy*yy*zz) 2*h[1, 1, 1, 1, 1]
sage: grf(grA.one())
```

(continues on next page)
Example 3.

After having had a graded algebra as the codomain, let us try to have one as the domain instead. Our new \( f \) will go from \( h \) to \( A \):

\begin{verbatim}
sage: def map_on_basis(lam):
    # redefining map_on_basis
    ....:     return x ** (sum(lam)) + y ** (len(lam))
sage: f = h.module_morphism(on_basis=map_on_basis,
    ....:                         codomain=A) # redefining f
sage: f(h[1])
U['x'] + U['y']
sage: f(h[2])
U['x']^2 + U['y']
sage: f(h[1, 1])
U['x']^2 + U['y']^2
sage: f(h[2, 2])
U['x']^4 + U['y']^2
sage: f(h[3, 2, 1])
U['x']^6 + U['y']^3
sage: f(h.one())
2
sage: grf = h.induced_graded_map(A, f); grf
Generic morphism:
   From: Symmetric Functions over Rational Field
      in the homogeneous basis
   To:   Graded Algebra of An example of a filtered
        algebra with basis: the universal enveloping
        algebra of Lie algebra of \( \mathbb{R}^3 \) with cross
        product over Rational Field
sage: grf(h[1])
bar(U['x']) + bar(U['y'])
sage: grf(h[2])
bar(U['x']^2)
sage: grf(h[1, 1])
bar(U['x']^2) + bar(U['y']^2)
sage: grf(h[2, 2])
bar(U['x']^4)
sage: grf(h[3, 2, 1])
bar(U['x']^6)
sage: grf(h.one())
2*bar(1)
\end{verbatim}

Example 4.

The construct \( \text{gr} f \) also makes sense when \( f \) is a filtration-preserving map between graded algebras.

\begin{verbatim}
sage: def map_on_basis(lam):
    # redefining map_on_basis
    ....:     return h[lam] + h[len(lam)]
sage: f = h.module_morphism(on_basis=map_on_basis,
    ....:                         codomain=h) # redefining f
sage: f(h[1])
2*hl[1]
sage: f(h[2])
h[1] + h[2]
sage: f(h[1, 1])
\end{verbatim}
Example 5.

For another example, let us compute $\text{gr} f$ for a map $f$ between two Clifford algebras:

```python
sage: Q = QuadraticForm(ZZ, 2, [1,2,3])
sage: B = CliffordAlgebra(Q, names=['u','v']); B
The Clifford algebra of the Quadratic form in 2 variables over Integer Ring with coefficients:
[ 1 2 ]
[ * 3 ]
sage: m = Matrix(ZZ, [[1, 2], [1, -1]])
sage: f = B.lift_module_morphism(m, names=['x','y'])
sage: A = f.domain(); A
The Clifford algebra of the Quadratic form in 2 variables over Integer Ring with coefficients:
[ 6 0 ]
[ * 3 ]
sage: x, y = A.gens()
sage: f(x)
u + v
sage: f(y)
2*u - v
sage: f(x*y)
6
sage: f(x*y)
-3*u*v + 3
sage: grA = A.graded_algebra(); grA
The exterior algebra of rank 2 over Integer Ring
sage: A.to_graded_conversion()(x)
x
sage: A.to_graded_conversion()(y)
y
sage: A.to_graded_conversion()(x*y)
x*y
sage: u = A.to_graded_conversion()(x*y+1); u
x*y + 1
sage: A.from_graded_conversion()(u)
x*y + 1
```
projection\((i)\)

Return the \(i\)-th projection \(p_i : \mathcal{F}_i \rightarrow \mathcal{G}_i\) (in the notations of the class documentation AssociatedGradedAlgebra, where \(A = \"self\)).

This method actually does not return the map \(p_i\) itself, but an extension of \(p_i\) to the whole \(R\)-module \(A\). This extension is the composition of the \(R\)-module isomorphism \(A \rightarrow \text{gr} A\) with the canonical projection of the graded \(R\)-module \(\text{gr} A\) onto its \(i\)-th graded component \(\mathcal{G}_i\). The codomain of this map is \(\text{gr} A\), although its actual image is \(\mathcal{G}_i\). The map \(p_i\) is obtained from this map by restricting its domain to \(\mathcal{F}_i\) and its image to \(\mathcal{G}_i\).

EXAMPLES:

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: p = A.an_element() + A.algebra_generators()\['x'\] + 2; p
U\['x'\]^2*U\['y'\]^2*U\['z'\]^3 + 3*U\['x'\] + 3*U\['y'\] + 3
sage: q = A.projection(7)(p); q
bar(U\['x'\]^2*U\['y'\]^2*U\['z'\]^3)
sage: q.parent() is A.graded_algebra()
True
sage: A.projection(8)(p)
0
```

to\_graded\_conversion()

Return the canonical \(R\)-module isomorphism \(A \rightarrow \text{gr} A\) induced by the basis of \(A\) (where \(A = \"self\)).

This is an isomorphism of \(R\)-modules, not of algebras. See the class documentation AssociatedGradedAlgebra.

See also:

\(\text{from}\_\text{graded}\_\text{conversion}\()\)

EXAMPLES:

```python
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: p = A.an_element() + A.algebra_generators()\['x'\] + 2; p
U\['x'\]^2*U\['y'\]^2*U\['z'\]^3 + 3*U\['x'\] + 3*U\['y'\] + 3
sage: q = A.to_graded_conversion()(p); q
```
3.42 Filtered Modules

A filtered module over a ring $R$ with a totally ordered indexing set $I$ (typically $I = \mathbb{N}$) is an $R$-module $M$ equipped with a family $(F_i)_{i \in I}$ of $R$-submodules satisfying $F_i \subseteq F_j$ for all $i, j \in I$ having $i \leq j$, and $M = \bigcup_{i \in I} F_i$. This family is called a filtration of the given module $M$.

Todo: Implement a notion for decreasing filtrations: where $F_j \subseteq F_i$ when $i \leq j$.

Todo: Implement filtrations for all concrete categories.

Todo: Implement gr as a functor.

```python
class sage.categories.filtered_modules.FilteredModules(base_category)
    Bases: sage.categories.filtered_modules.FilteredModulesCategory
    The category of filtered modules over a given ring $R$.

    A filtered module over a ring $R$ with a totally ordered indexing set $I$ (typically $I = \mathbb{N}$) is an $R$-module $M$ equipped with a family $(F_i)_{i \in I}$ of $R$-submodules satisfying $F_i \subseteq F_j$ for all $i, j \in I$ having $i \leq j$, and $M = \bigcup_{i \in I} F_i$. This family is called a filtration of the given module $M$.

    EXAMPLES:

sage: Modules(ZZ).Filtered()
Category of filtered modules over Integer Ring
sage: Modules(ZZ).Filtered().super_categories()
[Category of modules over Integer Ring]
```

REFERENCES:

- Wikipedia article Filtration_(mathematics)

```python
class Connected(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

class SubcategoryMethods

    Connected()
    Return the full subcategory of the connected objects of self.

    A filtered $R$-module $M$ with filtration $(F_0, F_1, F_2, \ldots)$ (indexed by $\mathbb{N}$) is said to be connected if $F_0$ is isomorphic to $R$.

    EXAMPLES:
```
sage: Modules(ZZ).Filtered().Connected()
Category of filtered connected modules over Integer Ring
sage: Coalgebras(QQ).Filtered().Connected()
Join of Category of filtered connected modules over Rational Field and Category of coalgebras over Rational Field
sage: AlgebrasWithBasis(QQ).Filtered().Connected()
Category of filtered connected algebras with basis over Rational Field

extra_super_categories()

Add `VectorSpaces` to the super categories of `self` if the base ring is a field.

EXAMPLES:

```python
sage: Modules(QQ).Filtered().extra_super_categories()
[Category of vector spaces over Rational Field]
sage: Modules(ZZ).Filtered().extra_super_categories()
[]
```

This makes sure that `Modules(QQ).Filtered()` returns an instance of `FilteredModules` and not a join category of an instance of this class and of `VectorSpaces(QQ)`:

```python
sage: type(Modules(QQ).Filtered())
<class 'sage.categories.filtered_modules.FilteredModules_with_category'>
```

Todo: Get rid of this workaround once there is a more systematic approach for the alias `Modules(QQ) -> VectorSpaces(QQ)`. Probably the latter should be a category with axiom, and covariant constructions should play well with axioms.

class sage.categories.filtered_modules.FilteredModulesCategory(base_category)

Bases: `sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory`, `sage.categories.category_types.Category_over_base_ring`

EXAMPLES:

```python
sage: C = Algebras(QQ).Filtered()
sage: C
Category of filtered 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 filtered modules over Rational Field]
sage: AlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field
sage: HopfAlgebrasWithBasis(QQ).Filtered().base_ring()
Rational Field
```

### 3.43 Filtered Modules With Basis

A **filtered module** with basis over a ring $R$ means (for the purpose of this code) a filtered $R$-module $M$ with filtration $(F_i)_{i \in I}$ (typically $I = \mathbb{N}$) endowed with a basis $(b_j)_{j \in J}$ of $M$ and a partition $J = \bigsqcup_{i \in I} J_i$ of the set $J$ (it is allowed
that some $J_i$ are empty) such that for every $n \in I$, the subfamily $(b_j)_{j \in U_n}$, where $U_n = \bigcup_{i \leq n} J_i$, is a basis of the $R$-submodule $F_n$.

For every $i \in I$, the $R$-submodule of $M$ spanned by $(b_j)_{j \in J_i}$ is called the $i$-th graded component (aka the $i$-th homogeneous component) of the filtered module with basis $M$; the elements of this submodule are referred to as homogeneous elements of degree $i$.

See the class documentation `FilteredModulesWithBasis` for further details.

class sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis(base_category)
Bases: sage.categories.filtered_modules.FilteredModulesCategory

The category of filtered modules with a distinguished basis.

A filtered module with basis over a ring $R$ means (for the purpose of this code) a filtered $R$-module $M$ with filtration $(F_i)_{i \in I}$ (typically $I = \mathbb{N}$) endowed with a basis $(b_j)_{j \in J}$ of $M$ and a partition $J = \bigcup_{i \in I} J_i$ of the set $J$ (it is allowed that some $J_i$ are empty) such that for every $n \in I$, the subfamily $(b_j)_{j \in U_n}$, where $U_n = \bigcup_{i \leq n} J_i$, is a basis of the $R$-submodule $F_n$.

For every $i \in I$, the $R$-submodule of $M$ spanned by $(b_j)_{j \in J_i}$ is called the $i$-th graded component (aka the $i$-th homogeneous component) of the filtered module with basis $M$; the elements of this submodule are referred to as homogeneous elements of degree $i$. The $R$-module $M$ is the direct sum of its $i$-th graded components over all $i \in I$, and thus becomes a graded $R$-module with basis. Conversely, any graded $R$-module with basis canonically becomes a filtered $R$-module with basis (by defining $F_n = \bigoplus_{i \leq n} G_i$, where $G_i$ is the $i$-th graded component, and defining $J_i$ as the indexing set of the basis of the $i$-th graded component). Hence, the notion of a filtered $R$-module with basis is equivalent to the notion of a graded $R$-module with basis.

However, the category of filtered $R$-modules with basis is not the category of graded $R$-modules with basis. Indeed, the morphisms of filtered $R$-modules with basis are defined to be morphisms of $R$-modules which send each $F_n$ of the domain to the corresponding $F_n$ of the target; in contrast, the morphisms of graded $R$-modules with basis must preserve each homogeneous component. Also, the notion of a filtered algebra with basis differs from that of a graded algebra with basis.

Note: Currently, to make use of the functionality of this class, an instance of `FilteredModulesWithBasis` should fulfill the contract of a `CombinatorialFreeModule` (most likely by inheriting from it). It should also have the indexing set $J$ encoded as its `_indices` attribute, and `_indices.subset(size=i)` should yield the subset $J_i$ (as an iterable). If the latter conditions are not satisfied, then `basis()` must be overridden.

Note: One should implement a `degree_on_basis` method in the parent class in order to fully utilize the methods of this category. This might become a required abstract method in the future.

EXAMPLES:

```python
sage: C = ModulesWithBasis(ZZ).Filtered(); C
Category of filtered modules with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of filtered modules over Integer Ring,
 Category of modules with basis over Integer Ring]
sage: C is ModulesWithBasis(ZZ).Filtered()
True
```

class ElementMethods

3.43. Filtered Modules With Basis 305
**degree()**

The degree of a nonzero homogeneous element self in the filtered module.

**Note:** This raises an error if the element is not homogeneous. To compute the maximum of the degrees of the homogeneous summands of a (not necessarily homogeneous) element, use `maximal_degree()` instead.

### EXAMPLES:

```python
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: x.degree()
6
sage: (x + 2*z).degree()
6
sage: (y - x).degree()
Traceback (most recent call last):
  ... ValueError: element is not homogeneous
```

An example in a graded algebra:

```python
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
```

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis):

```python
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).homogeneous_degree()
2
sage: (y*x).homogeneous_degree()
Traceback (most recent call last):
  ... ValueError: element is not homogeneous
sage: A.one().homogeneous_degree()
0
```

**degree_on_basis(m)**

Return the degree of the basis element indexed by m in self.

### EXAMPLES:

```python
sage: A = GradedModulesWithBasis(QQ).example()
sage: A.degree_on_basis(Partition((2,1)))
```
homogeneous_component \( (n) \)

Return the homogeneous component of degree \( n \) of the element \( \text{self} \).

Let \( m \) be an element of a filtered \( R \)-module \( M \) with basis. Then, \( m \) can be uniquely written in the form \( m = \sum_{i \in I} m_i \), where each \( m_i \) is a homogeneous element of degree \( i \). For \( n \in I \), we define the homogeneous component of degree \( n \) of the element \( m \) to be \( m_n \).

EXAMPLES:

```python
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A.an_element(); x
sage: x.homogeneous_component(-1)
0
sage: x.homogeneous_component(0)
2*P[]
sage: x.homogeneous_component(1)
2*P[1]
sage: x.homogeneous_component(2)
3*P[2]
sage: x.homogeneous_component(3)
0

sage: A = ModulesWithBasis(ZZ).Graded().example()
sage: x = A.an_element(); x
sage: x.homogeneous_component(-1)
0
sage: x.homogeneous_component(0)
2*P[]
sage: x.homogeneous_component(1)
2*P[1]
sage: x.homogeneous_component(2)
3*P[2]
sage: x.homogeneous_component(3)
0

sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: G = A.algebra_generators()
sage: g = A.an_element() - 2 * G['x'] * G['y']; g
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y'] + 2*U['x'] + 3*U['y'] + 1
sage: g.homogeneous_component(-1)
0
sage: g.homogeneous_component(0)
1
sage: g.homogeneous_component(2)
-2*U['x'] + U['y']
sage: g.homogeneous_component(5)
0
sage: g.homogeneous_component(7)
U['x']^2*U['y']^2*U['z']^3
sage: g.homogeneous_component(8)
```

(continues on next page)
h**omogeneous_degree()\

The degree of a nonzero homogeneous element self in the filtered module.

**Note:** This raises an error if the element is not homogeneous. To compute the maximum
of the degrees of the homogeneous summands of a (not necessarily homogeneous) element, use
maximal_degree() instead.

**EXAMPLES:**

```python
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: x.degree()
6
sage: (x + 2*z).degree()
6
sage: (y - x).degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

An example in a graded algebra:

```python
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
```

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the
choice of a basis):

```python
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x,y,z = A.algebra_generators()
sage: (x*y).homogeneous_degree()
2
sage: (y*x).homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
sage: A.one().homogeneous_degree()
0
```

**is_homogeneous()**

Return whether the element self is homogeneous.

**EXAMPLES:**
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: (3*x).is_homogeneous()  # True
sage: (x - y).is_homogeneous()  # False
sage: (x + 2*z).is_homogeneous()  # True

Here is an example with a graded algebra:

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

Let us now test a filtered algebra (but remember that the notion of homogeneity now depends on the choice of a basis, or at least on a definition of homogeneous components):

sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x, y, z = A.algebra_generators()
sage: (x*y).is_homogeneous()  # True
sage: (y*x).is_homogeneous()  # False
sage: A.one().is_homogeneous()  # True
sage: A.zero().is_homogeneous()  # True
sage: (A.one() + x).is_homogeneous()  # False

maximal_degree()

The maximum of the degrees of the homogeneous components of self.

This is also the smallest $i$ such that self belongs to $F_i$. Hence, it does not depend on the basis of the parent of self.

See also:

homogeneous_degree()

EXAMPLES:

sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A(Partition((3,2,1)))
sage: y = A(Partition((4,4,1)))
sage: z = A(Partition((2,2,2)))
sage: x.maximal_degree()  # 6
sage: (x + 2*z).maximal_degree()  # 6

(continues on next page)
Now, we test this on a graded algebra:

```
sage: S = NonCommutativeSymmetricFunctions(QQ).S()
sage: (x, y) = (S[2], S[3])
sage: x.maximal_degree()
2
sage: (x^3 + 4*y^2).maximal_degree()
6
sage: ((1 + x)^3).maximal_degree()
6
```

Let us now test a filtered algebra:

```
sage: A = AlgebrasWithBasis(QQ).Filtered().example()
sage: x, y, z = A.algebra_generators()
sage: (x*y).maximal_degree()
2
sage: (y*x).maximal_degree()
2
sage: A.one().maximal_degree()
0
sage: A.zero().maximal_degree()
Traceback (most recent call last):
  ... ValueError: the zero element does not have a well-defined degree
sage: (A.one()+x).maximal_degree()
1
```

**truncate** *(n)*

Return the sum of the homogeneous components of degree strictly less than *n* of *self*.

See **homogeneous_component()** for the notion of a homogeneous component.

**EXAMPLES:**

```
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: x = A.an_element(); x
sage: x.truncate(0)
0
sage: x.truncate(1)
2*P[]
sage: x.truncate(2)
2*P[] + 2*P[1]
sage: x.truncate(3)
sage: A = ModulesWithBasis(ZZ).Graded().example()
sage: x = A.an_element(); x
sage: x.truncate(0)
0
```

(continues on next page)
```python
sage: x.truncate(1)
2*P[]
sage: x.truncate(2)
2*P[] + 2*P[1]
sage: x.truncate(3)
```

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: G = A.algebra_generators()
sage: g = A.an_element() - 2 * G['x'] * G['y']; g
U['x']^2*U['y']^2*U['z']^3 - 2*U['x']*U['y'] + 2*U['x'] + 3*U['y'] + 1
```

```python
class ParentMethods

basis(d=None)
```

Return the basis for (the d-th homogeneous component of) self.

INPUT:

- d – (optional, default None) nonnegative integer or None

OUTPUT:

If d is None, returns the basis of the module. Otherwise, returns the basis of the homogeneous component of degree d (i.e., the subfamily of the basis of the whole module which consists only of the basis vectors lying in $F_d \setminus \bigcup_{i<d} F_i$).

The basis is always returned as a family.

EXAMPLES:

```python
sage: A = ModulesWithBasis(ZZ).Filtered().example()
```

```python
sage: A.basis(4)
Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions of the integer 4}
```

Without arguments, the full basis is returned:

```python
sage: A.basis()
Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions}
```

(continues on next page)
A basis method on a filtered module. Note that this will typically raise a NotImplementedError when this feature is not implemented.

```python
sage: A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Traceback (most recent call last):
  ...  
NotImplementedError: infinite set
```

Without arguments, the full basis is returned:

```python
sage: A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of RR^3 with cross product over Integer Ring(i))_{i in Free abelian monoid indexed by {'x', 'y', 'z'}}
```

An example with a graded algebra:

```python
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.basis()
Lazy family (Term map from Subsets of {0, 1} to The exterior algebra of rank 2 over Rational Field(i))_{i in Subsets of {0, 1}}
```

`from_graded_conversion()`

Return the inverse of the canonical $\mathbb{R}$-module isomorphism $A \to \text{gr} A$ induced by the basis of $A$ (where $A =$ “self”). This inverse is an isomorphism $\text{gr} A \to A$.

This is an isomorphism of $\mathbb{R}$-modules. See the class documentation AssociatedGradedAlgebra.

See also:

`to_graded_conversion()`

**EXAMPLES:**

```python
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: p = -2 * A.an_element(); p
sage: q = A.to_graded_conversion()(p); q
-4*Bbar[[0]] - 4*Bbar[[1]] - 6*Bbar[[2]]
sage: A.from_graded_conversion()(q) == p
True
sage: q.parent() is A.graded_algebra()
True
```

`graded_algebra()`

Return the associated graded module to self.

See AssociatedGradedAlgebra for the definition and the properties of this.
If the filtered module \( \text{self} \) with basis is called \( A \), then this method returns \( \text{gr}_A \). The method \( \text{to\_graded\_conversion()} \) returns the canonical \( R \)-module isomorphism \( A \to \text{gr}_A \) induced by the basis of \( A \), and the method \( \text{from\_graded\_conversion()} \) returns the inverse of this isomorphism. The method \( \text{projection()} \) projects elements of \( A \) onto \( \text{gr}_A \) according to their place in the filtration on \( A \).

**Warning:** When not overridden, this method returns the default implementation of an associated graded module – namely, \( \text{AssociatedGradedAlgebra}(\text{self}) \), where \( \text{AssociatedGradedAlgebra} \) is \( \text{AssociatedGradedAlgebra} \). But some instances of \( \text{FilteredModulesWithBasis} \) override this method, as the associated graded module often is (isomorphic) to a simpler object (for instance, the associated graded module of a graded module can be identified with the graded module itself). Generic code that uses associated graded modules (such as the code of the \( \text{induced\_graded\_map()} \) method below) should make sure to only communicate with them via the \( \text{to\_graded\_conversion()} \), \( \text{from\_graded\_conversion()} \) and \( \text{projection()} \) methods (in particular, do not expect there to be a conversion from \( \text{self} \) to \( \text{self.graded\_algebra()} \); this currently does not work for Clifford algebras). Similarly, when overriding \( \text{graded\_algebra()} \), make sure to accordingly redefine these three methods, unless their definitions below still apply to your case (this will happen whenever the basis of your \( \text{graded\_algebra()} \) has the same indexing set as \( \text{self} \), and the partition of this indexing set according to degree is the same as for \( \text{self} \)).

**EXAMPLES:**

```python
sage: A = ModulesWithBasis(ZZ).Filtered().example()
sage: A.graded_algebra()
Graded Module of An example of a filtered module with basis:
the free module on partitions over Integer Ring
```

\( \text{homogeneous\_component}(d) \)

Return the \( d \)-th homogeneous component of \( \text{self} \).

**EXAMPLES:**

```python
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.homogeneous_component(4)
Degree 4 homogeneous component of An example of a graded module
with basis: the free module on partitions over Integer Ring
```

\( \text{homogeneous\_component\_basis}(d) \)

Return a basis for the \( d \)-th homogeneous component of \( \text{self} \).

**EXAMPLES:**

```python
sage: A = GradedModulesWithBasis(ZZ).example()
sage: A.homogeneous_component_basis(4)
```

(continues on next page)
induced_graded_map\((other, f)\)

Return the graded linear map between the associated graded modules of self and other canonically induced by the filtration-preserving map\( f : \text{self} \rightarrow \text{other} \).

Let\( A \) and\( B \) be two filtered modules with basis, and let\( (F_i)_{i \in I} \) and\( (G_i)_{i \in I} \) be their filtrations. Let\( f : A \rightarrow B \) be a linear map which preserves the filtration (i.e., satisfies\( f(F_i) \subseteq G_i \) for all\( i \in I \)). Then, there is a canonically defined graded linear map\( gr f : gr A \rightarrow gr B \) which satisfies

\[
(gr f)(p_i(a)) = p_i(f(a)) \quad \text{for all} \ i \in I \ \text{and} \ a \in F_i,
\]

where the\( p_i \) on the left hand side is the canonical projection from\( F_i \) onto the\( i \)-th graded component of\( gr A \), while the\( p_i \) on the right hand side is the canonical projection from\( G_i \) onto the\( i \)-th graded component of\( gr B \).

INPUT:
-\( other \) – a filtered algebra with basis
-\( f \) – a filtration-preserving linear map from self to other (can be given as a morphism or as a function)

OUTPUT:
The graded linear map\( gr f \).

EXAMPLES:

Example 1.

We start with the free\( \mathbb{Q} \)-module with basis the set of all partitions:

\[
sage: A = Modules(QQ).WithBasis().Filtered().example(); A
An example of a filtered module with basis: the free module on partitions over Rational Field
\]

\[
sage: M = A.indices(); M
Partitions
\]

\[
sage: p1, p2, p21, p321 = [A.basis()[Partition(i)] for i in [[1], [2], [2, -1], [3,2,1]]]
\]

Let us define a map from\( A \) to itself which acts on the basis by sending every partition\( \lambda \) to the sum of the conjugates of all partitions\( \mu \) for which\( \lambda/\mu \) is a horizontal strip:

\[
sage: def\ \text{map\_on\_basis}(\lambda):
...: return A.sum\_of\_monomials([Partition(mu).conjugate() for k in range(sum(\lambda) + 1) for mu in \lambda.remove\_horizontal\_border\_strip(k)])
sage: f = A.module_morphism(on\_basis=\text{map\_on\_basis},
...: \text{codomain}=A)
\]

\[
sage: f(p1)
P[] + P[1]
\]

\[
sage: f(p2)
P[] + P[1] + P[1, 1]
\]

\[
sage: f(p21)
\]

\[
sage: f(p21 - p1)
\]

\[
sage: f(p321)
\]

(continues on next page)
We now compute $gr f$

```
sage: grA = A.graded_algebra(); grA
Graded Module of An example of a filtered module with basis:
the free module on partitions over Rational Field
sage: pp1, pp2, pp21, pp321 = [A.to_graded_conversion()(i) for i in [p1, p2, p321]]
sage: pp2 + 4 * pp21
Bbar[2] + 4*Bbar[2, 1]
sage: grf = A.induced_graded_map(A, f); grf
Generic endomorphism of Graded Module of An example of a
filtered module with basis:
the free module on partitions over Rational Field
sage: grf(pp1)
Bbar[1]
sage: grf(pp2 + 4 * pp21)
Bbar[1, 1] + 4*Bbar[2, 1]
```

Example 2.

We shall now construct $gr f$ for a different map $f$ out of the same $A$; the new map $f$ will lead into a graded algebra already, namely into the algebra of symmetric functions:

```
sage: h = SymmetricFunctions(QQ).h()
sage: def map_on_basis(lam):
    return h.sum_of_monomials([Partition(mu).conjugate() for k in range(sum(lam) + 1)
                              for mu in lam.remove_horizontal_border_strip(k)])
sage: f = A.module_morphism(on_basis=map_on_basis,
                          codomain=h) # redefining f
sage: f(p1)
h[] + h[1]
sage: f(p2)
h[] + h[1] + h[1, 1]
sage: f(A.zero())
0
sage: f(p2 - 3*p1)
-2*h[1] - 2*h[1, 1] + h[1, 1]
```

The algebra $h$ of symmetric functions in the $h$-basis is already graded, so its associated graded algebra is implemented as itself:

```
sage: grh = h.graded_algebra(); grh is h
True
sage: grf = A.induced_graded_map(h, f); grf
Generic morphism:
  From: Graded Module of An example of a filtered module with basis: the free module on partitions over Rational Field
  To:   Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(pp1)
h[1]
```

(continues on next page)
Example 3.

After having had a graded module as the codomain, let us try to have one as the domain instead. Our new \( f \) will go from \( h \) to \( A \):

```python
sage: def map_on_basis(lam):
    # redefining map_on_basis
    ....: return A.sum_of_monomials([Partition(mu).conjugate() for k in ...
    ....: for mu in lam.remove_horizontal_...
    ....: -border_strip(k)])

sage: f = h.module_morphism(on_basis=map_on_basis,
    ....: codomain=A)  # redefining f

sage: f(h[1])
P[0] + P[1]
sage: f(h[2])
P[0] + P[1] + P[1, 1]
sage: f(h[1, 1])
sage: f(h[3, 2])
sage: f(h.one())
P[0]
sage: grf = h.induced_graded_map(A, f); grf
Generic morphism:
    From: Symmetric Functions over Rational Field
    in the homogeneous basis
    To:  Graded Module of An example of a filtered
         module with basis: the free module on partitions
         over Rational Field

sage: grf(h[1])
Bbar[1]
sage: grf(h[2])
Bbar[1, 1]
sage: grf(h[3, 2, 1])
Bbar[3, 2, 1]
sage: grf(h.one())
Bbar[[]]
```

Example 4.
The construct $\text{gr} f$ also makes sense when $f$ is a filtration-preserving map between graded modules.

```
sage: def map_on_basis(lam):  # redefining map_on_basis
....:     return h.sum_of_monomials([Partition(mu).conjugate() for k in
˓→range(sum(lam) + 1)
....:         for mu in lam.remove_horizontal_  
˓→→ border_strip(k)])
sage: def f = h.module_morphism(on_basis=map_on_basis,
....:     codomain=h)  # redefining f
sage: f(h[1])
 h[1] + h[1]
sage: f(h[2])
 h[1] + h[1] + h[1, 1]
sage: f(h[1, 1])
 h[1] + h[2]
sage: f(h[2, 1])
sage: f(h.one())
 h[]
sage: grf = h.induced_graded_map(h, f); grf
Generic endomorphism of Symmetric Functions over Rational Field in the homogeneous basis
sage: grf(h[1])
 h[1]
sage: grf(h[2])
 h[1, 1]
sage: grf(h[1, 1])
 h[1]
sage: grf(h[2, 1])
 h[2, 1]
sage: grf(h.one())
 h[]
```

**projection($i$)**

Return the $i$-th projection $p_i : F_i \rightarrow G_i$ (in the notations of the class documentation $\text{AssociatedGradedAlgebra}$, where $A = \text{"self"}$).

This method actually does not return the map $p_i$ itself, but an extension of $p_i$ to the whole $R$-module $A$. This extension is the composition of the $R$-module isomorphism $A \rightarrow \text{gr} A$ with the canonical projection of the graded $R$-module $\text{gr} A$ onto its $i$-th graded component $G_i$. The codomain of this map is $\text{gr} A$, although its actual image is $G_i$. The map $p_i$ is obtained from this map by restricting its domain to $F_i$ and its image to $G_i$.

**EXAMPLES:**

```
sage: A = Modules(ZZ).WithBasis().Filtered().example()
sage: p = -2 * A.an_element(); p
sage: q = A.projection(2)(p); q
 -6*Bbar[[2]]
sage: q.parent() is A.graded_algebra()
True
sage: A.projection(3)(p)
 0
```

**to_graded_conversion()**

Return the canonical $R$-module isomorphism $A \rightarrow \text{gr} A$ induced by the basis of $A$ (where $A = \text{"self"}$).
This is an isomorphism of $R$-modules. See the class documentation AssociatedGradedAlgebra.

See also:

```
from_graded_conversion()
```

EXAMPLES:

```sage
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: p = -2 * A.an_element(); p
sage: q = A.to_graded_conversion()(p); q
sage: q.parent() is A.graded_algebra()
True
```

### 3.44 Finite Complex Reflection Groups

```python
class sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of finite complex reflection groups.

See ComplexReflectionGroups for the definition of complex reflection group. In the finite case, most of the information about the group can be recovered from its degrees and codegrees, and to a lesser extent to the explicit realization as subgroup of $GL(V)$. Hence the most important optional methods to implement are:

- ComplexReflectionGroups.Finite.ParentMethods.degrees()
- ComplexReflectionGroups.Finite.ParentMethods.codegrees()
- ComplexReflectionGroups.Finite.ElementMethods.to_matrix()

Finite complex reflection groups are completely classified. In particular, if the group is irreducible, then it’s uniquely determined by its degrees and codegrees and whether it’s reflection representation is primitive or not (see [LT2009] Chapter 2.1 for the definition of primitive).

See also:

Wikipedia article Complex_reflection_groups
```

EXAMPLES:

```sage
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite()
Category of finite complex reflection groups
sage: ComplexReflectionGroups().Finite().super_categories()

[Category of complex reflection groups, Category of finite groups, Category of finite finitely generated semigroups]
```

An example of a finite reflection group:

```sage
sage: W = ComplexReflectionGroups().Finite().example(); W  # optional - gap3
Reducible real reflection group of rank 4 and type A2 x B2
sage: W.reflections()  # optional - gap3
```
Finite family {1: (1,8)(2,5)(9,12), 2: (1,5)(2,9)(8,12),
3: (3,10)(4,7)(11,14), 4: (3,6)(4,11)(10,13),
5: (1,9)(2,8)(5,12), 6: (4,14)(6,13)(7,11),
7: (3,13)(6,10)(7,14)}

W is in the category of complex reflection groups:

```python
sage: W in ComplexReflectionGroups().Finite()  # optional - gap3
True
```

```python
class ElementMethods

    character_value()
    Return the value at self of the character of the reflection representation given by to_matrix().

    EXAMPLES:

    sage: W = ColoredPermutations(1,3); W
    1-colored permutations of size 3
    sage: [t.character_value() for t in W]
    [3, 1, 1, 0, 0, 1]

    Note that this could be a different (faithful) representation than that given by the corresponding root system:

    sage: W = ReflectionGroup((1,1,3)); W  # optional - gap3
    Irreducible real reflection group of rank 2 and type A2
    sage: [t.character_value() for t in W]  # optional - gap3
    [2, 0, 0, -1, -1, 0]

    sage: W = ColoredPermutations(2,2); W
    2-colored permutations of size 2
    sage: [t.character_value() for t in W]
    [2, 0, 0, -2, 0, 0, 0, 0]

    sage: W = ColoredPermutations(3,1); W
    3-colored permutations of size 1
    sage: [t.character_value() for t in W]
    [1, zeta3, -zeta3 - 1]
```

```python
    reflection_length(in_unitary_group=False)
    Return the reflection length of self.

    This is the minimal numbers of reflections needed to obtain self.

    INPUT:
    • in_unitary_group – (default: False) if True, the reflection length is computed in the
      unitary group which is the dimension of the move space of self

    EXAMPLES:

    sage: W = ReflectionGroup((1,1,3))  # optional - gap3
    sage: sorted([t.reflection_length() for t in W])  # optional - gap3
    [0, 1, 1, 1, 2, 2]

    sage: W = ReflectionGroup((2,1,2))  # optional - gap3
    sage: sorted([t.reflection_length() for t in W])  # optional - gap3
    [0, 1, 1, 1, 2, 2]
```

(continues on next page)
sage: W = ReflectionGroup((2,2,2))  # optional - gap3
sage: sorted([t.reflection_length() for t in W])  # optional - gap3
[0, 1, 1, 2]
sage: W = ReflectionGroup((3,1,2))  # optional - gap3
sage: sorted([t.reflection_length() for t in W])  # optional - gap3
[0, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]

to_matrix()
Return the matrix presentation of self acting on a vector space \( V \).

EXAMPLES:

sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: [t.to_matrix() for t in W]  # optional - gap3
[[1 0] [1 1] [-1 0] [-1 -1] [0 1] [0 -1]
 [0 1], [0 -1], [1 1], [1 0], [-1 -1], [-1 0]]

A different representation is given by the colored permutations:

sage: W = ColoredPermutations(3, 1)
sage: [t.to_matrix() for t in W]
[[1], [zeta3], [-zeta3 - 1]]

class Irreducible(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom
class ParentMethods

absolute_poset
Return the poset induced by the absolute order of self as a finite lattice.

INPUT:
• in_unitary_group – (default: False) if False, the relation is given by \( \sigma \leq \tau \) if \( l_R(\sigma) + l_R(\sigma^{-1} \tau) = l_R(\tau) \) If True, the relation is given by \( \sigma \leq \tau \) if \( \dim(\text{Fix}(\sigma)) + \dim(\text{Fix}(\sigma^{-1} \tau)) = \dim(\text{Fix}(\tau)) \)

See also:
noncrossing_partition_lattice()

EXAMPLES:

sage: P = ReflectionGroup((1,1,3)).absolute_poset(); P  # optional - gap3
Finite poset containing 6 elements
sage: sorted(w.reduced_word() for w in P)  # optional
[[], [1], [1, 2], [1, 2, 1], [2], [2, 1]]

sage: W = ReflectionGroup(4); W  # optional
Irreducible complex reflection group of rank 2 and type ST4
sage: W.absolute_poset()  # optional
Finite poset containing 24 elements

```
coxeter_number()
Return the Coxeter number of an irreducible reflection group.

This is defined as \( \frac{N + N^*}{n} \) where \( N \) is the number of reflections, \( N^* \) is the number of reflection hyperplanes, and \( n \) is the rank of \( \text{self} \).

EXAMPLES:
```
sage: W = ReflectionGroup(31)  # optional - gap3
sage: W.coxeter_number()  # optional - gap3
30

```
elements_below_coxeter_element(c=None)
Return all elements in \( \text{self} \) in the interval \([1, c] \) in the absolute order of \( \text{self} \).

This order is defined by

\[
\omega \leq_R \tau \iff \ell_R(\omega) + \ell_R(\omega^{-1}\tau) = \ell_R(\tau),
\]

where \( \ell_R \) denotes the reflection length.

**Note:** \( \text{self} \) is assumed to be well-generated.

INPUT:

- \( c \) (default: None) if an element \( c \) is given, it is used as the maximal element in the interval; if a list is given, the union of the various maximal elements is computed

EXAMPLES:
```
sage: W = ReflectionGroup((1,1,3))  # optional
sage: sorted( w.reduced_word() for w in W.elements_below_coxeter_element() )  # optional - gap3
[[], [1], [1, 2], [1, 2, 1], [2]]
sage: sorted( w.reduced_word() for w in W.elements_below_coxeter_element(W.from_reduced_word([2,1])) )  # optional - gap3
[[], [1], [1, 2, 1], [2], [2, 1]]
sage: sorted( w.reduced_word() for w in W.elements_below_coxeter_element(W.from_reduced_word([2])) )  # optional - gap3
[[], [2]]
```

3.44. Finite Complex Reflection Groups 321
**generalized_noncrossing_partitions** \((m, c=None, positive=False)\)

Return the set of all chains of length \(m\) in the noncrossing partition lattice of \(self\), see [noncrossing_partition_lattice](#).

**Note:** \(self\) is assumed to be well-generated.

**INPUT:**
- \(c\) – (default: None) if an element \(c\) in \(self\) is given, it is used as the maximal element in the interval.
- \(positive\) – (default: False) if True, only those generalized noncrossing partitions of full support are returned.

**EXAMPLES:**

```python
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: sorted([w.reduced_word() for w in chain]  # optional - gap3
           for chain in W.generalized_noncrossing_partitions(2))  # optional - gap3
[[[], [[], [1, 2]]],
 [[[], [1, [2]]],
 [[[], [1, 2], []]],
 [[[], [1, 2, 1], [1]]],
 [[[], [2], [1, 2, 1]]],
 [[1], [[2]],
 [[1], [2], []],
 [[1], [2], [1], []]],
 [[[[], [1, 2]], [],
 [[1], [1, 2], [1]],
 [[1], [2], []],
 [[1, 2], [1], []],
 [[1, 2, 1], [1], []],
 [[1, 2, 1], [1], [1]],
 [[2], [1, 2, 1], []]]
```

**noncrossing_partition_lattice** \((c=None, L=None, in_unitary_group=False)\)

Return the interval \([1, c]\) in the absolute order of \(self\) as a finite lattice.

**See also:**
- [elements_below_coxeter_element()](#)
• `in_unitary_group` – (default: `False`) if `False`, the relation is given by \( \sigma \leq \tau \) if 
\[ l_R(\sigma) + l_R(\sigma^{-1}) = l_R(\tau); \] if `True`, the relation is given by \( \sigma \leq \tau \) if 
\[ \dim(\text{Fix}(\sigma)) + \dim(\text{Fix}(\sigma^{-1})) = \dim(\text{Fix}(\tau)). \]

EXEMPLARY:

```python
sage: W = SymmetricGroup(4)
sage: W.noncrossing_partition_lattice()
Finite lattice containing 14 elements

sage: W = WeylGroup(['G', 2])
sage: W.noncrossing_partition_lattice()
Finite lattice containing 8 elements

sage: W = ReflectionGroup((1,1,3))  # optional
                                ← gap3
sage: sorted( w.reduced_word() for w in W.noncrossing_partition_lattice() )  # optional - gap3
[[], [1], [1, 2], [1, 2, 1], [2]]

sage: sorted( w.reduced_word() for w in W.noncrossing_partition_lattice(W.from_reduced_word([2,1])) )  # optional - gap3
[[], [2]]
```

example()

Return an example of an irreducible complex reflection group.

EXEMPLARY:

```python
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite().Irreducible().example()  # optional
Irreducible complex reflection group of rank 3 and type G(4,2,3)
```

class ParentMethods

**base_change_matrix()**

Return the base change from the standard basis of the vector space of `self` to the basis given by the independent roots of `self`.

**Todo:** For non-well-generated groups there is a conflict with construction of the matrix for an element.

EXEMPLARY:

```python
sage: W = ReflectionGroup((1,1,3))  # optional
                                ← gap3
sage: W.base_change_matrix()  # optional
[1 0]
```

(continues on next page)
cardinality()
Return the cardinality of self.

It is given by the product of the degrees of self.

EXAMPLES:

```
sage: W = ColoredPermutations(1,3)
sage: W.cardinality()
6
sage: W = ColoredPermutations(2,3)
sage: W.cardinality()
48
sage: W = ColoredPermutations(4,3)
sage: W.cardinality()
384
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.cardinality()  # optional - gap3
192
```

codegrees()
Return the codegrees of self.

OUTPUT: a tuple of Sage integers

EXAMPLES:

```
sage: W = ColoredPermutations(1,4)
sage: W.codegrees()
(2, 1, 0)
sage: W = ColoredPermutations(3,3)
sage: W.codegrees()
(6, 3, 0)
```
degrees()

Return the degrees of self.

OUTPUT: a tuple of Sage integers

EXAMPLES:

```python
sage: W = ColoredPermutations(1,4)
sage: W.degrees()
(2, 3, 4)
sage: W = ColoredPermutations(3,3)
sage: W.degrees()
(3, 6, 9)
sage: W = ReflectionGroup(31)
# optional - gap3
sage: W.degrees()
# optional - gap3
(8, 12, 20, 24)
```

is_real()

Return whether self is real.

A complex reflection group is real if it is isomorphic to a reflection group in $GL(V)$ over a real vector space $V$. Equivalently its character table has real entries.

This implementation uses the following statement: an irreducible complex reflection group is real if and only if 2 is a degree of self with multiplicity one. Hence, in general we just need to compare the number of occurrences of 2 as degree of self and the number of irreducible components.

EXAMPLES:

```python
sage: W = ColoredPermutations(1,3)
sage: W.is_real()
True
sage: W = ColoredPermutations(4,3)
sage: W.is_real()
False
```

Todo: Add an example of non real finite complex reflection group that is generated by order 2 reflections.

is_well_generated()

Return whether self is well-generated.

A finite complex reflection group is well generated if the number of its simple reflections coincides with its rank.

See also:

ComplexReflectionGroups.Finite.WellGenerated()
Note:
- All finite real reflection groups are well generated.
- The complex reflection groups of type $G(r, 1, n)$ and of type $G(r, r, n)$ are well generated.
- The complex reflection groups of type $G(r, p, n)$ with $1 < p < r$ are not well generated.
- The direct product of two well generated finite complex reflection group is still well generated.

EXAMPLES:

```sage
sage: W = ColoredPermutations(1,3)
sage: W.is_well_generated()  True
sage: W = ColoredPermutations(4,3)
sage: W.is_well_generated()  True
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.is_well_generated()  # optional - gap3
    False
sage: W = ReflectionGroup((4,4,3))  # optional - gap3
sage: W.is_well_generated()  # optional - gap3
    True
```

```

number_of_reflection_hyperplanes()

Return the number of reflection hyperplanes of self.

This is also the number of distinguished reflections. For real groups, this coincides with the number of reflections.

This implementation uses that it is given by the sum of the codegrees of self plus its rank.

See also:

`number_of_reflections()`

EXAMPLES:

```sage
sage: W = ColoredPermutations(1,3)
sage: W.number_of_reflection_hyperplanes()  3
sage: W = ColoredPermutations(2,3)
sage: W.number_of_reflection_hyperplanes()  9
sage: W = ColoredPermutations(4,3)
sage: W.number_of_reflection_hyperplanes()  15
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.number_of_reflection_hyperplanes()  # optional - gap3
    15
```

```

number_of_reflections()

Return the number of reflections of self.

For real groups, this coincides with the number of reflection hyperplanes.

This implementation uses that it is given by the sum of the degrees of self minus its rank.

See also:
number_of_reflection_hyperplanes()

EXAMPLES:

```
sage: [SymmetricGroup(i).number_of_reflections() for i in range(int(8))]
[0, 0, 1, 3, 6, 10, 15, 21]
sage: W = ColoredPermutations(1,3)
sage: W.number_of_reflections()
3
sage: W = ColoredPermutations(2,3)
sage: W.number_of_reflections()
9
sage: W = ColoredPermutations(4,3)
sage: W.number_of_reflections()
21
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.number_of_reflections()  # optional - gap3
15
```

rank()

Return the rank of self.

The rank of self is the dimension of the smallest faithful reflection representation of self.

This default implementation uses that the rank is the number of degrees().

See also:
ComplexReflectionGroups.rank()

EXAMPLES:

```
sage: W = ColoredPermutations(1,3)
sage: W.rank()
2
sage: W = ColoredPermutations(2,3)
sage: W.rank()
3
sage: W = ColoredPermutations(4,3)
sage: W.rank()
3
sage: W = ReflectionGroup((4,2,3))  # optional - gap3
sage: W.rank()  # optional - gap3
3
```

class SubcategoryMethods

WellGenerated()

Return the full subcategory of well-generated objects of self.

A finite complex generated group is well generated if it is isomorphic to a subgroup of the general linear group $GL_n$ generated by $n$ reflections.

See also:
ComplexReflectionGroups.Finite.ParentMethods.is_well_generated()

EXAMPLES:
```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: C = ComplexReflectionGroups().Finite().WellGenerated(); C
Category of well generated finite complex reflection groups
```

Here is an example of a finite well-generated complex reflection group:

```
sage: W = C.example(); W
Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)
```

All finite Coxeter groups are well generated:

```
sage: CoxeterGroups().Finite().is_subcategory(C)
True
sage: SymmetricGroup(3) in C
True
```

**Note:** The category of well generated finite complex reflection groups is currently implemented as an axiom. See discussion on trac ticket #11187. This may be a bit of overkill. Still it’s nice to have a full subcategory.

```python
class WellGenerated(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class Irreducible(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of finite irreducible well-generated finite complex reflection groups.
```

```python
class ParentMethods

    def catalan_number(self, positive=False, polynomial=False):
        Return the Catalan number associated to self.
        It is defined by
        \[
        \prod_{i=1}^{n} \frac{d_{i} + h}{d_{i}},
        \]
        where \(d_{1}, \ldots, d_{n}\) are the degrees and where \(h\) is the Coxeter number. See [Ar2006] for further information.

        INPUT:
        • positive – optional boolean (default False) if True, return instead the positive Catalan number
        • polynomial – optional boolean (default False) if True, return instead the \(q\)-analogue as a polynomial in \(q\)

        **Note:**
        • For the symmetric group \(S_{n}\), it reduces to the Catalan number \(\frac{1}{n+1} \binom{2n}{n}\).
        • The Catalan numbers for \(G(r,1,n)\) all coincide for \(r > 1\).
```

**EXAMPLES:**
sage: [ColoredPermutations(1,n).catalan_number() for n in [3,4,5]]
[5, 14, 42]

sage: [ColoredPermutations(2,n).catalan_number() for n in [3,4,5]]
[20, 70, 252]

sage: [ReflectionGroup((2,2,n)).catalan_number() for n in [3,4,5]]
→ # optional - gap3
[14, 50, 182]

coxeter_number()  
Return the Coxeter number of a well-generated, irreducible reflection group. This is defined to be the order of a regular element in self, and is equal to the highest degree of self.

See also:
ComplexReflectionGroups.Finite.Irreducible()

Note: This method overwrites the more general method for complex reflection groups since the expression given here is quicker to compute.

EXAMPLES:

sage: W = ColoredPermutations(1,3)
sage: W.coxeter_number()
3

sage: W = ColoredPermutations(4,3)
sage: W.coxeter_number()
12

sage: W = ReflectionGroup((4,4,3))  # optional - gap3
sage: W.coxeter_number()  # optional - gap3
8

fuss_catalan_number(m, positive=False, polynomial=False)  
Return the m-th Fuss-Catalan number associated to self.

This is defined by

\[ \prod_{i=1}^{n} \frac{d_i + mh}{d_i} , \]

where \(d_1,\ldots,d_n\) are the degrees and \(h\) is the Coxeter number.

INPUT:
• positive – optional boolean (default False) if True, return instead the positive Fuss-Catalan number
• polynomial – optional boolean (default False) if True, return instead the \(q\)-analogue as a polynomial in \(q\)

See [Ar2006] for further information.

Note:
• For the symmetric group \(S_n\), it reduces to the Fuss-Catalan number \(\frac{1}{mn+1} \binom{m+1}{n} \).  
• The Fuss-Catalan numbers for \(G(r,1,n)\) all coincide for \(r > 1\).
EXAMPLES:

```python
sage: W = ColoredPermutations(1,3)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[5, 12, 22]

sage: W = ColoredPermutations(1,4)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[14, 55, 140]

sage: W = ColoredPermutations(1,5)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[42, 273, 969]

sage: W = ColoredPermutations(2,2)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[6, 15, 28]

sage: W = ColoredPermutations(2,3)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[20, 84, 220]

sage: W = ColoredPermutations(2,4)
sage: [W.fuss_catalan_number(i) for i in [1,2,3]]
[70, 495, 1820]
```

`number_of_reflections_of_full_support()`

Return the number of reflections with full support.

EXAMPLES:

```python
sage: W = Permutations(4)
sage: W.number_of_reflections_of_full_support()
1

sage: W = ColoredPermutations(1,4)
sage: W.number_of_reflections_of_full_support()
1

sage: W = CoxeterGroup("B3")
sage: W.number_of_reflections_of_full_support()
3

sage: W = ColoredPermutations(3,3)
sage: W.number_of_reflections_of_full_support()
3
```

`rational_catalan_number(p, polynomial=False)`

Return the p-th rational Catalan number associated to self.

It is defined by

\[ \prod_{i=1}^{n} \frac{p + (p(d_i - 1))}{d_i} \mod h, \]

where \(d_1, \ldots, d_n\) are the degrees and \(h\) is the Coxeter number. See [STW2016] for this formula.

INPUT:
• polynomial – optional boolean (default False) if True, return instead the $q$-analogue as a polynomial in $q$

EXAMPLES:

```
sage: W = ColoredPermutations(1,3)
sage: [W.rational_catalan_number(p) for p in [5,7,8]]
[7, 12, 15]
sage: W = ColoredPermutations(2,2)
sage: [W.rational_catalan_number(p) for p in [7,9,11]]
[10, 15, 21]
```

```
example()
Return an example of an irreducible well-generated complex reflection group.

EXAMPLES:

```
sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite().WellGenerated().Irreducible().example()
4-colored permutations of size 3
```

```
class ParentMethods

coxeter_element()
Return a Coxeter element.

The result is the product of the simple reflections, in some order.

**Note:** This implementation is shared with well generated complex reflection groups. It would be nicer to put it in some joint super category; however, in the current state of the art, there is none where it is clear that this is the right construction for obtaining a Coxeter element.

In this context, this is an element having a regular eigenvector (a vector not contained in any reflection hyperplane of self).

EXAMPLES:

```
sage: CoxeterGroup(['A', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['B', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['D', 4]).coxeter_element().reduced_word()
[1, 2, 4, 3]
sage: CoxeterGroup(['F', 4]).coxeter_element().reduced_word()
[1, 2, 3, 4]
sage: CoxeterGroup(['E', 8]).coxeter_element().reduced_word()
[1, 2, 3, 4, 5, 6, 7, 8]
sage: CoxeterGroup(['H', 3]).coxeter_element().reduced_word()
[1, 2, 3]
```

This method is also used for well generated finite complex reflection groups:

```
sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: W.coxeter_element().reduced_word()  # optional - gap3
```

(continues on next page)


**coxeter_elements()**

Return the (unique) conjugacy class in `self` containing all Coxeter elements.

A Coxeter element is an element that has an eigenvalue $e^{2\pi i/h}$ where $h$ is the Coxeter number.

In case of finite Coxeter groups, these are exactly the elements that are conjugate to one (or, equivalently, all) standard Coxeter element, this is, to an element that is the product of the simple generators in some order.

See also:

**standard_coxeter_elements()**

EXAMPLES:

```python
sage: W = ReflectionGroup((1,1,3))  # optional - gap3
sage: sorted(c.reduced_word() for c in W.coxeter_elements())  # optional - gap3
[[[1, 2], [2, 1]]]

sage: W = ReflectionGroup((1,1,4))  # optional - gap3
sage: sorted(c.reduced_word() for c in W.coxeter_elements())  # optional - gap3
[[[1, 2, 1, 3, 2], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 1, 3, 2, 1], [3, 2, 1]]]
```

**is_well_generated()**

Return `True` as `self` is well-generated.

EXAMPLES:

```python
sage: W = ReflectionGroup((3,1,2))  # optional - gap3
sage: W.is_well_generated()  # optional - gap3
True
```

**standard_coxeter_elements()**

Return all standard Coxeter elements in `self`.

This is the set of all elements in `self` obtained from any product of the simple reflections in `self`.

Note:
- `self` is assumed to be well-generated.
- This works even beyond real reflection groups, but the conjugacy class is not unique and we only obtain one such class.
EXAMPLES:

```python
sage: W = ReflectionGroup(4)  # optional - gap3
sage: sorted(W.standard_coxeter_elements())  # optional - gap3
[(1, 7, 6, 12, 23, 20)(2, 8, 17, 24, 9, 5)(3, 16, 10, 19, 15, 21)(4, 14, 11, 22, 18, 13),
 (1, 10, 4, 12, 21, 22)(2, 11, 19, 24, 13, 3)(5, 15, 7, 17, 16, 23)(6, 18, 8, 20, 14, 9)]
```

```python
def example():
    """
    Return an example of a well-generated complex reflection group.
    """
    return ComplexReflectionGroups().Finite().WellGenerated().example()  # optional - gap3

sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite().WellGenerated().example()  # optional - gap3
Reducible complex reflection group of rank 4 and type A2 x G(3,1,2)
```

```python
def example():
    """
    Return an example of a complex reflection group.
    """
    return ComplexReflectionGroups().Finite().example()  # optional - gap3

sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
sage: ComplexReflectionGroups().Finite().example()  # optional - gap3
Reducible real reflection group of rank 4 and type A2 x B2
```

### 3.45 Finite Coxeter Groups

```python
class sage.categories.finite_coxeter_groups.FiniteCoxeterGroups(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of finite Coxeter groups.
```

```python
EXAMPLES:

sage: W = CoxeterGroups().Finite()
Category of finite coxeter groups
sage: W = CoxeterGroups().Finite().super_categories()  # optional - gap3
[Category of finite generalized coxeter groups,
 Category of coxeter groups]
sage: G = CoxeterGroups().Finite().example()  # optional - gap3
sage: G.cayley_graph(side = "right").plot()
Graphics object consisting of 40 graphics primitives
```

Here are some further examples:

```python
sage: W = WeylGroups().Finite().example()  # optional - gap3
The symmetric group on (0, ..., 3)
```

```python
sage: W = 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:
class ElementMethods

\texttt{bruhat\_upper\_covers()}

Returns all the elements that cover self in Bruhat order.

**EXAMPLES:**

```python
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]]
```

```python
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]]
```

Recursive algorithm: write \( w \) for self. If \( i \) is a non-descent of \( w \), then the covers of \( w \) are exactly \( \{ ws_i, u_1 s_i, u_2 s_i, \ldots, u_j s_i \} \), where the \( u_k \) are those covers of \( ws_i \) that have a descent at \( i \).

\texttt{covered\_reflections\_subgroup()}

Return the subgroup of \( W \) generated by the conjugates by \( w \) of the simple reflections indexed by right descents of \( w \).

This is used to compute the shard intersection order on \( W \).

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: len(W.long_element().covered_reflections_subgroup())
24
sage: s = W.simple_reflection(1)
sage: Gs = s.covered_reflections_subgroup()
sage: len(Gs)
2
sage: s in [u.lift() for u in Gs]
True
sage: len(W.one().covered_reflections_subgroup())
1
```
coxeter_knuth_graph()  
Return the Coxeter-Knuth graph of type $A$.

The Coxeter-Knuth graph of type $A$ is generated by the Coxeter-Knuth relations which are given by $aa + 1a \sim a + 1aa + 1$, $abc \sim acb$ if $b < a < c$ and $abc \sim bac$ if $a < c < b$.

EXAMPLES:

```python
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)]
```

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:

```python
sage: W = WeylGroup(['A',4], prefix='s')
sage: word = [1,2,1,3,2]
sage: w = W.from_reduced_word(word)
sage: w.coxeter_knuth_neighbor(word)
{(1, 2, 3, 1, 2), (2, 1, 2, 3, 2)}
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)}
```

is_coxeter_element()  
Return whether this is a Coxeter element.

This is, whether self has an eigenvalue $e^{2\pi i / h}$ where $h$ is the Coxeter number.

See also:

coxeter_elements()
EXAMPLES:

```python
sage: W = CoxeterGroup(['A', 2])
sage: c = prod(W.gens())
sage: c.is_coxeter_element()
True
sage: W.one().is_coxeter_element()
False

sage: W = WeylGroup(['G', 2])
sage: c = prod(W.gens())
sage: c.is_coxeter_element()
True
sage: W.one().is_coxeter_element()
False
```

```python
class 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.

`bhz_poset()`

Return the Bergeron-Hohlweg-Zabrocki partial order on the Coxeter group.

This is a partial order on the elements of a finite Coxeter group \( W \), which is distinct from the Bruhat order, the weak order and the shard intersection order. It was defined in [BHZ05].

This partial order is not a lattice, as there is no unique maximal element. It can be succinctly defined as follows.

Let \( u \) and \( v \) be two elements of the Coxeter group \( W \). Let \( S(u) \) be the support of \( u \). Then \( u \leq v \) if and only if \( v_{S(u)} = u \) (here \( v = v^I v_I \) denotes the usual parabolic decomposition with respect to the standard parabolic subgroup \( W_I \)).

See also:

`bruhat_poset()`, `shard_poset()``, `weak_poset()`

EXAMPLES:

```python
sage: W = CoxeterGroup(['A', 3], base_ring=ZZ)
sage: P = W.bhz_poset(); P
Finite poset containing 24 elements
sage: P.relations_number()
103
sage: P.chain_polynomial()
34*q^4 + 90*q^3 + 79*q^2 + 24*q + 1
sage: len(P.maximal_elements())
13
```

REFERENCE:

`bruhat_poset`(facade=False)

Return the Bruhat poset of self.

See also:

`bhz_poset()`, `shard_poset()`, `weak_poset()`

EXAMPLES:
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 \rightarrow\text{space})
```

See also:

- `Poset()` for more on posets and facade posets.

**Todo:**
- Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
- The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.

### `cambrian_lattice(c, on_roots=False)`

Return the $c$-Cambrian lattice on delta sequences.

See Arxiv 1503.00710 and Arxiv math/0611106.

Delta sequences are certain 2-colored minimal factorizations of $c$ into reflections.

**INPUT:**
- $c$ -- a standard Coxeter element in `self` (as a tuple, or as an element of `self`)
- `on_roots` (optional, default `False`) -- if `on_roots` is `True`, the lattice is realized on roots rather than on reflections. In order for this to work, the ElementMethod `reflection_to_root` must be available.

**EXAMPLES:**
```
sage: CoxeterGroup(['A', 2]).cambrian_lattice((1,2))
Finite lattice containing 5 elements

sage: CoxeterGroup(['B', 2]).cambrian_lattice((1,2))
Finite lattice containing 6 elements

sage: CoxeterGroup(['G', 2]).cambrian_lattice((1,2))
Finite lattice containing 8 elements
```

**codegrees()**

Return the codegrees of the Coxeter group.

These are just the degrees minus 2.

**EXAMPLES:**

```
sage: CoxeterGroup(['A', 4]).codegrees()
(0, 1, 2, 3)
sage: CoxeterGroup(['B', 4]).codegrees()
(0, 2, 4, 6)
sage: CoxeterGroup(['D', 4]).codegrees()
(0, 2, 4, 4)
sage: CoxeterGroup(['F', 4]).codegrees()
(0, 4, 6, 10)
sage: CoxeterGroup(['E', 8]).codegrees()
(0, 6, 10, 12, 16, 18, 22, 28)
sage: CoxeterGroup(['H', 3]).codegrees()
(0, 4, 8)
sage: WeylGroup([['A',3], ['A',3], ['B',2]]).codegrees()
(0, 1, 2, 0, 1, 2, 0, 2)
```

**degrees()**

Return the degrees of the Coxeter group.

The output is an increasing list of integers.

**EXAMPLES:**

```
sage: CoxeterGroup(['A', 4]).degrees()
(2, 3, 4, 5)
sage: CoxeterGroup(['B', 4]).degrees()
(2, 4, 6, 8)
sage: CoxeterGroup(['D', 4]).degrees()
(2, 4, 4, 6)
sage: CoxeterGroup(['F', 4]).degrees()
(2, 6, 8, 12)
sage: CoxeterGroup(['E', 8]).degrees()
(2, 8, 12, 14, 18, 20, 24, 30)
sage: CoxeterGroup(['H', 3]).degrees()
(2, 6, 10)
sage: WeylGroup([['A',3], ['A',3], ['B',2]]).degrees()
(2, 3, 4, 2, 3, 4, 2, 4)
```

**inversion_sequence(word)**

Return the inversion sequence corresponding to the word in indices of simple generators of self.

If word corresponds to \([w_0, w_1, \ldots, w_k]\), the output is \([w_0, w_0 w_1, \ldots, w_0 w_1 \cdots w_k, \ldots, w_1 w_0]\).
INPUT:
• word – a word in the indices of the simple generators of self.

EXAMPLES:

```
sage: CoxeterGroup(['A', 2]).inversion_sequence([1,2,1])
[-1 1] [ 0 -1] [ 1 0]
[ 0 1] [-1 0] [ 1 -1]
```

```
sage: [t.reduced_word() for t in CoxeterGroup(['A',3]).inversion_sequence([2,1,3,2,1,3])]
[[2], [1, 2, 1], [2, 3, 2], [1, 2, 3, 2, 1], [3], [1]]
```

```
is_real()
Return True since self is a real reflection group.

EXAMPLES:

```
sage: CoxeterGroup(['F',4]).is_real()
True
sage: CoxeterGroup(['H',4]).is_real()
True
```

```
long_element(index_set=None, as_word=False)
Return the longest element of self, or of the parabolic subgroup corresponding to the given index_set.

INPUT:
• index_set – a subset (as a list or iterable) of the nodes of the Dynkin diagram; (default: all of them)
• as_word – boolean (default False). If True, then return instead a reduced decomposition of the longest element.

Should this method be called maximal_element? longest_element?

EXAMPLES:

```
sage: D10 = FiniteCoxeterGroups().example(10)
sage: D10.long_element()
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
sage: D10.long_element([1])
(1,)
sage: D10.long_element([2])
(2,)
sage: D10.long_element([[]])
()
sage: D7 = FiniteCoxeterGroups().example(7)
sage: D7.long_element()
(1, 2, 1, 2, 1, 2, 1)
```

One can require instead a reduced word for w0:

```
sage: A3 = CoxeterGroup(['A', 3])
sage: A3.long_element(as_word=True)
[1, 2, 1, 3, 2, 1]
```

```
m_cambrian_lattice(c, m=1, on_roots=False)
Return the m-Cambrian lattice on m-delta sequences.
```

3.45. Finite Coxeter Groups 339
See Arxiv 1503.00710 and Arxiv math/0611106.

The $m$-delta sequences are certain $m$-colored minimal factorizations of $c$ into reflections.

**INPUT:**
- $c$ – a Coxeter element of self (as a tuple, or as an element of self)
- $m$ – a positive integer (optional, default 1)
- on_roots (optional, default False) – if on_roots is True, the lattice is realized on roots rather than on reflections. In order for this to work, the ElementMethod reflection_to_root must be available.

**EXAMPLES:**

```python
sage: CoxeterGroup(['A',2]).m_cambrian_lattice((1,2))
Finite lattice containing 5 elements
sage: CoxeterGroup(['A',2]).m_cambrian_lattice((1,2),2)
Finite lattice containing 12 elements
```

**permutahedron** (point=None, base_ring=None)

Return the permutahedron of self.

This is the convex hull of the point point in the weight basis under the action of self on the underlying vector space $V$.

**See also:**

permutahedron()

**INPUT:**
- point – optional, a point given by its coordinates in the weight basis (default is $(1,1,1,\ldots)$)
- base_ring – optional, the base ring of the polytope

**Note:** The result is expressed in the root basis coordinates.

**Note:** If function is too slow, switching the base ring to RDF will almost certainly speed things up.

**EXAMPLES:**

```python
sage: W = CoxeterGroup(['H',3], base_ring=RDF)
sage: W.permutahedron()
UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 120 vertices
sage: W = CoxeterGroup(['I',7])
sage: W.permutahedron()
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 14 vertices
sage: W.permutahedron(base_ring=RDF)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 14 vertices
```

(continues on next page)
sage: W = ReflectionGroup(['A',3])
       # optional - gap3
sage: W.permutahedron()
       # optional - gap3
A 3-dimensional polyhedron in QQ^3 defined as the convex hull
of 24 vertices

sage: W = ReflectionGroup(['A',3],['B',2])
       # optional - gap3
sage: W.permutahedron()
       # optional - gap3
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 192
vertices

reflections_from_w0()
Return the reflections of self using the inversion set of w_0.

EXAMPLES:

sage: WeylGroup(['A',2]).reflections_from_w0()
[0 1 0]
[1 0 0]
[0 0 1]
[0 0 1], [1 0 0], [0 1 0]

sage: WeylGroup(['A',3]).reflections_from_w0()
[0 1 0 0]
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1], [1 0 0 0], [0 1 0 0], [0 0 1 0], [0 0 0 1]

shard_poset (side='right')
Return the shard intersection order attached to W.

This is a lattice structure on W, introduced in [Reading]. It contains the noncrossing partition lattice,
as the induced lattice on the subset of c-sortable elements.

The partial order is given by simultaneous inclusion of inversion sets and subgroups attached to every
element.

The precise description used here can be found in [StThWi].

Another implementation for the symmetric groups is available as \texttt{shard\_poset()}. See also: \texttt{bhz\_poset()}, \texttt{bruhat\_poset()}, \texttt{weak\_poset()}

\begin{Verbatim}
EXAMPLES:

\begin{verbatim}
sage: W = CoxeterGroup(['A',3], base_ring=ZZ)
sage: SH = W.shard_poset(); SH
Finite lattice containing 24 elements
sage: SH.is_graded()
True
sage: SH.characteristic_polynomial()
q^3 - 11*q^2 + 23*q - 13
sage: SH.f_polynomial()
34*q^3 + 22*q^2 + q
\end{verbatim}
\end{Verbatim}

REFERENCES:

\begin{Verbatim}
\texttt{w0()}
\end{Verbatim}

Return the longest element of \texttt{self}.

This attribute is deprecated, use \texttt{long\_element()} instead.

\begin{Verbatim}
EXAMPLES:

\begin{verbatim}
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)
\end{verbatim}
\end{Verbatim}

\begin{Verbatim}
\texttt{weak\_lattice(side='right', facade=False)}
\end{Verbatim}

INPUT:

- \texttt{side} – “left”, “right”, or “twosided” (default: “right”)
- \texttt{facade} – a boolean (default: False)

Returns the left (resp. right) poset for weak order. In this poset, \(u\) is smaller than \(v\) if some reduced word of \(u\) is a right (resp. left) factor of some reduced word of \(v\).

See also: \texttt{bhz\_poset()}, \texttt{bruhat\_poset()}, \texttt{shard\_poset()}

\begin{Verbatim}
EXAMPLES:

\begin{verbatim}
sage: W = WeylGroup(['A', 2])
sage: P = W.weak_poset()
sage: P
Finite lattice containing 6 elements
sage: P.show()

This poset is in fact a lattice:

\begin{verbatim}
sage: W = WeylGroup(['B', 3])
sage: P = W.weak_poset(side = "left")
\end{verbatim}
\end{verbatim}
\end{Verbatim}
sage: P.is_lattice()
True

so this method has an alias `weak_lattice()`:

```python
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:

```python
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:

```python
sage: P.is_lattice()
False
```

By default, the elements of \( P \) are aware of that they belong to \( P \):

```python
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:

```python
sage: P = W.weak_poset(facade = True)
sage: P.an_element().parent()
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient
˓→space)
```

See also:
- `Poset()` for more on posets and facade posets.

Todo:
- Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
- The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.

```
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 \( v \).

See also:
- `bhz_poset()`, `bruhat_poset()`, `shard_poset()`

EXAMPLES:
sage: W = WeylGroup("A", 2)
sage: P = W.weak_poset()
sage: P
Finite lattice containing 6 elements
sage: P.show()

This poset is in fact a lattice:

sage: W = WeylGroup("B", 3)
sage: P = W.weak_poset(side = "left")
sage: P.is_lattice()
True

so this method has an alias \texttt{weak\_lattice()}:

sage: W.weak_lattice(side = "left") is W.weak_poset(side = "left")
True

As a bonus feature, one can create the left-right weak poset:

sage: W = WeylGroup("A", 2)
sage: P = W.weak_poset(side = "twosided")
sage: P.show()
sage: len(P.hasse_diagram().edges())
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 \texttt{facade} option:

sage: P = W.weak_poset(facade = True)
sage: P.an_element().parent()
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)

See also:
\texttt{Poset()} for more on posets and facade posets.

Todo:
• Use the symmetric group in the examples (for nicer output), and print the edges for a stronger test.
• The constructed poset should be lazy, in order to handle large / infinite Coxeter groups.
3.46 Finite Crystals

```python
sage: CoxeterGroups().Finite().super_categories()
[Category of finite generalized coxeter groups, 
  Category of coxeter groups]
```

class **FiniteCrystals** *(base_category)*

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of finite crystals.

**EXAMPLES:**

```python
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
```

class **TensorProducts** *(category, *args)*

Bases: `sage.categories.tensor.TensorProductsCategory`

The category of finite crystals constructed by tensor product of finite crystals.

**extra_super_categories** *(category, *args)*

**EXAMPLES:**

```python
sage: FiniteCrystals().TensorProducts().extra_super_categories()
[Category of finite crystals]
```

**example** *(n=3)*

Returns an example of highest weight crystals, as per `Category.example()`.

**EXAMPLES:**

```python
sage: B = FiniteCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1
```

**extra_super_categories** *(category, *args)*

**EXAMPLES:**

```python
sage: FiniteCrystals().extra_super_categories()
[Category of finite enumerated sets]
```

3.47 Finite dimensional algebras with basis

**Todo:** Quotients of polynomial rings.

Quotients in general.

Matrix rings.
REFERENCES:

• [CR1962]

```python
class sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis(base_category):
    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 magmatic algebras with basis over Rational Field]
```
```sage```
C.example()
```
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
```
```python
class Cellular(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

Cellular algebras.

Let $R$ be a commutative ring. A $R$-algebra $A$ is a cellular algebra if it has a cell datum, which is a tuple $(\Lambda, i, M, C)$, where $\Lambda$ is finite poset with order $\geq$, if $\mu \in \Lambda$ then $T(\mu)$ is a finite set and
\[
C : \prod_{\mu \in \Lambda} T(\mu) \times T(\mu) \to A; (\mu, s, t) \mapsto c^\mu_{st}
\]
is an injective map such that the following holds:

- The set \{ $c^\mu_{st}$ | $\mu \in \Lambda$, $s, t \in T(\mu)$ \} is a basis of $A$.
- If $a \in A$ and $\mu \in \Lambda$, $s, t \in T(\mu)$ then:
  \[
a c^\mu_{st} = \sum_{u \in T(\mu)} r_a(s, u) c^\mu_{ut} \pmod{A^{>\mu}},
  \]
  where $A^{>\mu}$ is spanned by
  \{ $c^\nu_{ab}$ | $\nu > \mu$ and $a, b \in T(\nu)$ \}.

Moreover, the scalar $r_a(s, u)$ depends only on $a$, $s$ and $u$ and, in particular, is independent of $t$.

- The map $\iota : A \to A ; c^\mu_{st} \mapsto c^\mu_{ts}$ is an algebra anti-isomorphism.

A cellular basis for $A$ is any basis of the form \{ $c^\mu_{st}$ | $\mu \in \Lambda$, $s, t \in T(\mu)$ \}.

Note that in particular, the scalars $r_a(u, s)$ in the second condition do not depend on $t$.

REFERENCES:

• [GrLe1996]
• [KX1998]
• [Mat1999]
• Wikipedia article Cellular_algebra
class ElementMethods

cellular_involution()

Return the cellular involution on self.

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: elt = S([3,1,2,4])
sage: ci = elt.cellular_involution(); ci
7/48*[1, 3, 2, 4] + 49/48*[2, 3, 1, 4] - 1/48*[3, 1, 2, 4] - 7/48*[3, 2, 1, 4]
sage: ci.cellular_involution()
[3, 1, 2, 4]
```

class ParentMethods

cell_module(mu, **kwds)

Return the cell module indexed by mu.

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: S.cell_module(Partition([2,1]))
Cell module indexed by [2, 1] of Cellular basis of Symmetric group algebra of order 3 over Rational Field
```

cell_module_indices(mu)

Return the indices of the cell module of self indexed by mu.

This is the finite set \( \mathcal{M}(\lambda) \).

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: S.cell_module_indices([2,1])
Standard tableaux of shape [2, 1]
```

cell_poset()

Return the cell poset of self.

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.cell_poset()
Finite poset containing 5 elements
```

cells()

Return the cells of self.

EXAMPLES:

```python
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: dict(S.cells())
{(1, 1, 1): Standard tableaux of shape [1, 1, 1],
 [2, 1]: Standard tableaux of shape [2, 1],
 [3]: Standard tableaux of shape [3]}
```
cellular_basis()
Return the cellular basis of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: S.cellular_basis()
Cellular basis of Symmetric group algebra of order 3
over Rational Field
```

cellular_involution(x)
Return the cellular involution of x in self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: for b in S.basis(): b, S.cellular_involution(b)
([1, 2, 3], [1, 2, 3])
([1, 3, 2], 49/48*[1, 3, 2] + 7/48*[2, 3, 1] - 7/48*[3, 1, 2] - 1/48*[3, 2, 1])
([2, 1, 3], [2, 1, 3])
([2, 3, 1], -7/48*[1, 3, 2] - 1/48*[2, 3, 1] + 49/48*[3, 1, 2] + 7/48*[3, 2, 1])
([3, 1, 2], 7/48*[1, 3, 2] + 49/48*[2, 3, 1] - 1/48*[3, 1, 2] - 7/48*[3, 2, 1])
([3, 2, 1], -1/48*[1, 3, 2] - 7/48*[2, 3, 1] + 7/48*[3, 1, 2] + 49/48*[3, 2, 1])
```

simple_module_parameterization()
Return a parameterization of the simple modules of self.

The set of simple modules are parameterized by \( \lambda \in \Lambda \) such that the cell module bilinear form \( \Phi_\lambda \neq 0 \).

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.simple_module_parameterization()
([4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1])
```

class TensorProducts(category, *args)
Bases: sage.categories.tensor.TensorProductsCategory
The category of cellular algebras constructed by tensor product of cellular algebras.

class ParentMethods

cell_module_indices(mu)
Return the indices of the cell module of self indexed by mu.

This is the finite set \( M(\lambda) \).

EXAMPLES:

```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: T = S2.tensor(S3)
sage: T.cell_module_indices(((1,1), [2,1]))
The Cartesian product of (Standard tableaux of shape [1, 1], Standard tableaux of shape [2, 1])
```
**cell_poset()**
Return the cell poset of self.

**EXAMPLES:**

```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: T = S2.tensor(S3)
sage: T.cell_poset()
Finite poset containing 6 elements
```

**cellular_involution()**
Return the image of the cellular involution of the basis element indexed by i.

**EXAMPLES:**

```
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S3 = SymmetricGroupAlgebra(QQ, 3)
sage: T = S2.tensor(S3)
sage: for b in T.basis(): b, T.cellular_involution(b)
(([1, 2] # [1, 2, 3], [1, 2] # [1, 2, 3])
([1, 2] # [1, 3, 2],
  49/48*([1, 2] # [1, 3, 2] + 7/48*([1, 2] # [2, 3, 1] - 7/48*([1, 2] # [3, 1, 2] - 1/48*([1, 2] # [3, 2, 1]))
([1, 2] # [2, 1, 3], [1, 2] # [2, 1, 3])
([1, 2] # [2, 3, 1],
  -7/48*([1, 2] # [1, 3, 2] - 1/48*([1, 2] # [2, 3, 1] + 49/48*([1, 2] # [3, 1, 2] - 1/48*([1, 2] # [3, 2, 1])
([1, 2] # [3, 1, 2],
  1/48*([1, 2] # [1, 3, 2] - 7/48*([1, 2] # [2, 3, 1]) + 7/48*([1, 2] # [3, 1, 2] - 7/48*([1, 2] # [3, 2, 1]))
([1, 2] # [3, 2, 1],
  -1/48*([1, 2] # [1, 3, 2] - 7/48*([1, 2] # [2, 3, 1]) + 7/48*([1, 2] # [3, 1, 2] - 7/48*([1, 2] # [3, 2, 1]))
([2, 1] # [1, 3, 2], [2, 1] # [1, 2, 3])
([2, 1] # [1, 3, 2],
([2, 1] # [2, 1, 3], [2, 1] # [2, 1, 3])
([2, 1] # [2, 3, 1],
([2, 1] # [3, 1, 2],
  1/48*([2, 1] # [1, 3, 2] - 7/48*([2, 1] # [2, 3, 1]) + 7/48*([2, 1] # [3, 3, 1])
([2, 1] # [3, 2, 1],
  -1/48*([2, 1] # [1, 3, 2] - 7/48*([2, 1] # [2, 3, 1]) + 7/48*([2, 1] # [3, 3, 1])
```

**extra_super_categories()**
Tensor products of cellular algebras are cellular.

**EXAMPLES:**

```
sage: cat = Algebras(QQ).FiniteDimensional().WithBasis()
sage: cat.Cellular().TensorProducts().extra_super_categories()
[Category of finite dimensional cellular algebras with basis over Rational Field]
```
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 1 0]
    [0 0 0 1 0 0]
    sage: a.to_matrix(base_ring=RDF, side="left")
    [0.0 0.0 1.0 0.0 0.0 0.0]
    [0.0 0.0 0.0 0.0 1.0 0.0]
    [1.0 0.0 0.0 0.0 0.0 0.0]
    [0.0 0.0 0.0 0.0 0.0 1.0]
    [0.0 1.0 0.0 0.0 0.0 0.0]
    [0.0 0.0 0.0 1.0 0.0 0.0]

    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]
    (continues on next page)
AUTHORS: Mike Hansen, ...

class 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 $(P^L_i)_{i \in I}$ and $(P^R_i)_{i \in I}$ be respectively representatives of the corresponding indecomposable projective left and right modules of $A$. In particular, we assume that the indexing is consistent so that $S_i^* = \text{top } P^L_i$ and $S_i = \text{top } P^R_i$.

The Cartan invariant matrix $(C_{i,j})_{i,j \in I}$ is a matrix of non negative integers that encodes much of the representation theory of $A$; namely:

\begin{itemize}
  \item $C_{i,j}$ counts how many times $S_i^* \otimes S_j$ appears as composition factor of $A$ seen as a bimodule over itself;
  \item $C_{i,j} = \dim \text{Hom}_A(P^R_j, P^R_i)$;
  \item $C_{i,j}$ counts how many times $S_j$ appears as composition factor of $P^R_i$;
  \item $C_{i,j} = \dim \text{Hom}_A(P^L_i, P^L_j)$;
  \item $C_{i,j}$ counts how many times $S_i^*$ appears as composition factor of $P^L_j$.
\end{itemize}

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.

\underline{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:

\begin{itemize}
  \item peirce_decomposition()
  \item isotypic_projective_modules()
\end{itemize}

EXAMPLES:
For a semisimple algebra, in particular for group algebras in characteristic zero, the Cartan invariants matrix is the identity:

```python
sage: A3 = SymmetricGroup(3).algebra(QQ)
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:

```python
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:

```python
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:

```python
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:

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
```
The center of a semisimple algebra is semisimple:

```
True
```

Todo:
- Pickling by construction, as `A.center()`?
- Lazy evaluation of `_repr_`

### center_basis()

Return a basis of the center of `self`.

**OUTPUT:**
- a list of elements of `self`.

**See also:**
- `center()`

**EXAMPLES:**

```
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver 
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: A.center_basis()
(x + y,)
```

### idempotent_lift(x)

Lift an idempotent of the semisimple quotient into an idempotent of `self`.

Let $A$ be this finite dimensional algebra and $\pi$ be the projection $A \to \overline{A}$ on its semisimple quotient. Let $\overline{e}$ be an idempotent of $\overline{A}$, and $x$ any lift thereof in $A$. This returns an idempotent $e$ of $A$ such that $\pi(e) = \pi(x)$ and $e$ is a polynomial in $x$.

**INPUT:**
- $x$ -- an element of $A$ that projects on an idempotent $\overline{e}$ of the semisimple quotient of $A$. Alternatively one may give as input the idempotent $\overline{e}$ itself.

**OUTPUT:** the idempotent $e$ of `self`

**ALGORITHM:**

3.47. Finite dimensional algebras with basis
Iterate the formula \( 1 - (1 - x^2)^2 \) until having an idempotent.

See [CR1962] for correctness and termination proofs.

**EXAMPLES:**

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example()
sage: S = A.semisimple_quotient()
x
sage: A.idempotent_lift(S.basis()['x'])
y
Todo: Add some non trivial example

**is_commutative()**

Return whether \( \text{self} \) is a commutative algebra.

**EXAMPLES:**

```python
sage: S4 = SymmetricGroupAlgebra(QQ, 4)
sage: S4.is_commutative()
False
sage: S2 = SymmetricGroupAlgebra(QQ, 2)
sage: S2.is_commutative()
True
```

**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 \( \text{self} \)

**EXAMPLES:**

```python
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver (containing the arrows a:x->y and b:x->y) over Rational Field
sage: 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:

```python
sage: A.is_identity_decomposition_into_orthogonal_idempotents([x])
False
```

Here \( 1 + x \) and \( -x \) are neither idempotent nor orthogonal:

```python
sage: A.is_identity_decomposition_into_orthogonal_idempotents([1+x, -x])
False
```
With the algebra of the 0-Hecke monoid:

```python
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: A = HeckeMonoid(SymmetricGroup(4)).algebra(QQ)
sage: idempotents = A.orthogonal_idempotents_central_radical()
```

Here are some more counterexamples:

1. Some orthogonal elements summing to 1 but not being idempotent:

```python
sage: class PQAlgebra(CombinatorialFreeModule):
....:     def __init__(self, F, p):
....:         # Construct the quotient algebra F[x] / p,
....:         # where p is a univariate polynomial.
....:         R = parent(p); x = R.gen()
....:         I = R.ideal(p)
....:         self._xbar = R.quotient(I).gen()
....:         basis_keys = [self._xbar**i for i in range(p.degree())]
....:         CombinatorialFreeModule.__init__(self, F, basis_keys,
....:         category=Algebras(F).FiniteDimensional().WithBasis())
....:     def x(self):
....:         return self(self._xbar)
....:     def one(self):
....:         return self.basis()[self.base_ring().one()]
....:     def product_on_basis(self, w1, w2):
....:         return self.from_vector(vector(w1 * w2))
```

```python
sage: R.<x> = PolynomialRing(QQ)
sage: A = PQAlgebra(QQ, x**3 - x**2 + x + 1); y = A.x()
sage: a, b = y, 1-y
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
False
```

For comparison:

```python
sage: A = PQAlgebra(QQ, x**2 - x); y = A.x()
sage: a, b = y, 1-y
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
True
```

2. Some idempotents summing to 1 but not orthogonal:

```python
sage: R.<x> = PolynomialRing(GF(2))
sage: A = PQAlgebra(GF(2), x)
sage: a = A.one()
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a,))
True
sage: A = PQAlgebra(GF(2), x**3 - x**2 + x - 1); y = A.x()
sage: a = (y**2 + 1) / 2
sage: b = 1 - a
sage: A.is_identity_decomposition_into_orthogonal_idempotents((a, b))
True
```

(continues on next page)
3. Some orthogonal idempotents not summing to the identity:

    sage: A.is_identity_decomposition_into_orthogonal_idempotents((a,a))
    False
    sage: A.is_identity_decomposition_into_orthogonal_idempotents(())
    False

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 \( (e_i) \) 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 $(f_i)$ 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 $e_k$ and $f = \sum_{i<k} f_i$.

See [CR1962] for correctness and termination proofs.

See also:

  central_orthogonal_idempotents()
• 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[8] + 1/2*B[4],
 1/2*B[9] - 1/2*B[3],
 B[0],
sage: sum(idempotents) == 1
True
sage: all(e*e == e for e in idempotents)
True
sage: all(e*f == 0 and f*e == 0 for e in idempotents for f in idempotents
 ...if e != f)
True
```

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)
```

(continues on next page)
peirce_decomposition (idempotents=None, check=True)

Return a Peirce decomposition of self.

Let \((e_i)\) be a collection of orthogonal idempotents of \(A\) with sum 1. The Peirce decomposition of \(A\) is the decomposition of \(A\) into the direct sum of the subspaces \(e_i A e_j\).

With the default collection of orthogonal idempotents, one has

\[
\dim e_i A e_j = C_{i,j} \dim S_i \dim S_j
\]

where \((S_i)\) are the simple modules of \(A\) and \((C_{i,j})_{i,j}\) is the Cartan invariants matrix.

**INPUT:**
- `idempotents` – a list of orthogonal idempotents \((e_i)_{i=0,\ldots,n}\) 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 and idempotent and sum to 1

**OUTPUT:**
A list of lists \(l\) such that \(l[i][j]\) is the subspace \(e_i A e_j\).

**EXAMPLES:**

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver (containing the arrows a:x->y and b:x->y) over Rational Field
sage: A.orthogonal_idempotents_central_mod_radical()
(x, y)
sage: decomposition = A.peirce_decomposition(); decomposition
[[Free module generated by {0} over Rational Field, Free module generated by {0, 1} over Rational Field]], [[Free module generated by {} over Rational Field, Free module generated by {0} over Rational Field]]
sage: [ [x.lift() for x in decomposition[i][j].basis()] for i in range(2) for j in range(2) ]
[[[x], [a, b]], [[], [y]]]
```

We recover that the group algebra of the symmetric group \(S_4\) is a block matrix algebra:

```python
sage: A = SymmetricGroup(4).algebra(QQ)
sage: decomposition = A.peirce_decomposition() # long time
sage: [ [decomposition[i][j].dimension() for j in range(len(decomposition))] for i in range(len(decomposition)) ]
[[9, 0, 0, 0, 0],]
```

(continues on next page)
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 $(e_i, e_j)$

Return the Peirce decomposition summand $e_i A e_j$.

**INPUT:**
- `self` – an algebra $A$
- $e_i, e_j$ – two idempotents of $A$

**OUTPUT:** $e_i A e_j$, as a subspace of $A$.

**See also:**
- `peirce_decomposition()`
- `principal_ideal()`

**EXAMPLES:**

```sage
A = Algebras(QQ).FiniteDimensional().WithBasis().example()
idemp = A.orthogonal_idempotents_central_mod_radical()
A.peirce_summand(idemp[0], idemp[1])
Free module generated by {0, 1} over Rational Field
A.peirce_summand(idemp[1], idemp[0])
Free module generated by {} over Rational Field
```

We recover the $2 \times 2$ block of $Q[S_4]$ corresponding to the unique simple module of dimension 2 of the symmetric group $S_4$:

```sage
A4 = SymmetricGroup(4).algebra(QQ)
e = A4.central_orthogonal_idempotents()[2]
A4.peirce_summand(e, e)
Free module generated by {0, 1, 2, 3} over Rational Field
```

principal_ideal $(a, \text{side}='\text{left}')$

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 $x$:  

```
3.47. Finite dimensional algebras with basis 359
```
so the left ideal generated by $x$ is one-dimensional:

```python
sage: A = A.principal_ideal(x, side='left'); A
Free module generated by {0} over Rational Field
sage: [B.lift() for B in A.basis()]
[x]
```

Multiplication on the left by $x$ annihilates only $x$ and fixes the other basis elements:

```python
sage: x*x, x*y, x*a, x*b
(x, 0, a, b)
```

so the right ideal generated by $x$ is 3-dimensional:

```python
sage: xA = A.principal_ideal(x, side='right'); xA
Free module generated by {0, 1, 2} over Rational Field
sage: [B.lift() for B in xA.basis()]
[x, a, b]
```

See also:
- `peirce_summand()`

`radical()`

Return the Jacobson radical of `self`.

This uses `radical_basis()`, whose default implementation handles algebras over fields of characteristic zero or fields of characteristic $p$ in which we can compute $x^{1/p}$.

See also:
- `radical_basis()`, `semisimple_quotient()`

**EXAMPLES:**

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
sage: radical = A.radical(); radical
Radical of An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field
```

The radical is an ideal of $A$, and thus a finite dimensional non unital associative algebra:

```python
sage: from sage.categories.associative_algebras import AssociativeAlgebras
sage: radical in AssociativeAlgebras(QQ).WithBasis().FiniteDimensional()
True
sage: radical in Algebras(QQ)
False
sage: radical.dimension()
2
sage: radical.basis()
Finite family {0: B[0], 1: B[1]}
```
Todo:
• Tell Sage that the radical is in fact an ideal;
• Pickling by construction, as A.center();
• Lazy evaluation of _repr_.

radical_basis()
Return a basis of the Jacobson radical of this algebra.

Note: This implementation handles algebras over fields of characteristic zero (using Dixon’s lemma) or fields of characteristic $p$ in which we can compute $x^{1/p}$ [FR1985], [Eb1989].

OUTPUT:
• a list of elements of self.
See also:
radical(), Algebras.Semisimple

EXAMPLES:

```sage
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x→y and b:x→y) over Rational Field
sage: A.radical_basis()
(a, b)
```

We construct the group algebra of the Klein Four-Group over the rationals:

```sage
sage: A = KleinFourGroup().algebra(QQ)
```

This algebra belongs to the category of finite dimensional algebras over the rationals:

```sage
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
sage: A in Algebras(QQ).Semisimple()
True
sage: A.radical_basis()
()
```

Let’s work instead over a field of characteristic 2:

```sage
sage: A = KleinFourGroup().algebra(GF(2))
sage: A in Algebras(GF(2)).Semisimple()
False
```
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 \texttt{ModulesWithBasis}:

```python
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()

sage: AnAlgebra(GF(16,'a')).radical_basis()

sage: AnAlgebra(GF(49,'a')).radical_basis()
```

\texttt{semisimple_quotient()}\n
Return the semisimple quotient of \texttt{self}.

This is the quotient of \texttt{self} by its radical.

See also:

\texttt{radical()}\n
**EXAMPLES:**

```python
sage: A = Algebras(QQ).FiniteDimensional().WithBasis().example(); A
An example of a finite dimensional algebra with basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field

sage: a,b,x,y = sorted(A.basis())

sage: S = A.semisimple_quotient(); S
Semisimple quotient of An example of a finite dimensional algebra with
basis:
the path algebra of the Kronecker quiver
(containing the arrows a:x->y and b:x->y) over Rational Field

sage: S in Algebras(QQ).Semisimple()
True

sage: S.basis()
Finite family \{'x': B['x'], \'y': B['y']\}

sage: xs,ys = sorted(S.basis())

sage: (xs + ys) * xs
B['x']
```
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_`

```
class SubcategoryMethods

Cellular()
Return the full subcategory of the cellular objects of self.

See also:
Wikipedia article Cellular_algebra

EXAMPLES:
```
sage: Algebras(QQ).FiniteDimensional().WithBasis().Cellular()
```

3.48 Finite dimensional bialgebras with basis

```
sage.categories.finite_dimensional_bialgebras_with_basis.FiniteDimensionalBialgebrasWithBasis
The category of finite dimensional bialgebras with a distinguished basis

EXAMPLES:
```
sage: C = FiniteDimensionalBialgebrasWithBasis(QQ); C
Category of finite dimensional bialgebras with basis over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of bialgebras with basis over Rational Field,
 Category of finite dimensional algebras with basis over Rational Field]
sage: C is Bialgebras(QQ).WithBasis().FiniteDimensional()
True
```

3.49 Finite dimensional coalgebras with basis

```
sage.categories.finite_dimensional_coalgebras_with_basis.FiniteDimensionalCoalgebrasWithBasis
The category of finite dimensional coalgebras with a distinguished basis

EXAMPLES:
```
3.50 Finite Dimensional Graded Lie Algebras With Basis

AUTHORS:

- Eero Hakavuori (2018-08-16): initial version

```python
class sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGraded
    Base Category: Category of finite dimensional graded Lie algebras with basis.

A grading of a Lie algebra \( g \) is a direct sum decomposition \( g = \bigoplus_i V_i \) such that \([V_i, V_j] \subset V_{i+j}\).

EXAMPLES:

```
sage: C = LieAlgebras(QQ).WithBasis().Graded(); C
Category of finite dimensional graded Lie algebras with basis over Rational Field

sage: C is LieAlgebras(QQ).WithBasis().Graded()
True
```
A finite-dimensional stratified Lie algebra is nilpotent:

```python
sage: C = LieAlgebras(QQ).WithBasis().Graded().Stratified().FiniteDimensional()
sage: C
Category of finite dimensional stratified lie algebras with basis over Rational Field

sage: C is C.Nilpotent()
True
```

class ParentMethods

def degree_on_basis(m)

Return the degree of the basis element indexed by m.

If the degrees of the basis elements are not defined, they will be computed. By assumption the stratification $L_1 \oplus \cdots \oplus L_s$ of self is such that each component $L_k$ is spanned by some subset of the basis.

The degree of a basis element $X$ is therefore the largest index $k$ such that $X \in L_k \oplus \cdots \oplus L_s$. The space $L_k \oplus \cdots \oplus L_s$ is by assumption the $k$-th term of the lower central series.

EXAMPLES:

```python
sage: C = LieAlgebras(QQ).WithBasis().Graded()
sage: C = C.FiniteDimensional().Stratified().Nilpotent()
sage: sc = {('X','Y'): {'Z': 1}}
sage: L.<X,Y,Z> = LieAlgebra(QQ, sc, nilpotent=True, category=C)
sage: L.degree_on_basis(X.leading_support())
1
sage: X.degree()
1
sage: Y.degree()
1
sage: L[X, Y]
Z
sage: Z.degree()
2
```

3.51 Finite dimensional Hopf algebras with basis

class sage.categories.finite_dimensional_hopf_algebras_with_basis.FiniteDimensionalHopfAlgebrasWithBasis(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of finite dimensional Hopf algebras with a distinguished basis.

EXAMPLES:

```python
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]
```
class ElementMethods

class ParentMethods

## 3.52 Finite Dimensional Lie Algebras With Basis

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

```python
class sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis(BaseCategory):
    pass
```

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

Category of finite dimensional Lie algebras with a basis.

**Todo:** Many of these tests should use non-abelian Lie algebras and need to be added after trac ticket #16820.

### class ElementMethods

#### adjoint_matrix()

Return the matrix of the adjoint action of \texttt{self}.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.an_element().adjoint_matrix()
```

```
[0 0 0]
[0 0 0]
[0 0 0]
```

```python
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: x.adjoint_matrix()
```

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

```python
sage: y.adjoint_matrix()
```

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

#### to_vector()

Return the vector in \texttt{g.module()} corresponding to the element \texttt{self} of \texttt{g} (where \texttt{g} is the parent of \texttt{self}).

Implement this if you implement \texttt{g.module()}. See \texttt{sage.categories.lie_algebras.LieAlgebras.module()} for how this is to be done.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.an_element().to_vector()
```

```
(0, 0, 0)
```

```python
sage: D = DescentAlgebra(QQ, 4).D()
sage: L = LieAlgebra(associative=D)
sage: L.an_element().to_vector()
```

```
(1, 1, 1, 1, 1, 1, 1, 1)
```
Nilpotent

alias of sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.

FiniteDimensionalNilpotentLieAlgebrasWithBasis

class ParentMethods

as_finite_dimensional_algebra()

Return self as a FiniteDimensionalAlgebra.

EXAMPLES:

```sage
sage: L = lie_algebras.cross_product(QQ)
sage: x,y,z = L.basis()
sage: F = L.as_finite_dimensional_algebra()
sage: X,Y,Z = F.basis()
sage: x.bracket(y)
  Z
sage: X * Y
  Z
```

center()

Return the center of self.

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: Z = L.center(); Z
An example of a finite dimensional Lie algebra with basis: the 3-dimensional abelian Lie algebra over Rational Field
sage: Z.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

centralizer(S)

Return the centralizer of S in self.

INPUT:

• S – a subalgebra of self or a list of elements that represent generators for a subalgebra

See also:

centralizer_basis()

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a,b,c = L.lie_algebra_generators()
sage: S = L.centralizer([a + b, 2*a + c]); S
An example of a finite dimensional Lie algebra with basis: the 3-dimensional abelian Lie algebra over Rational Field
sage: S.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

centralizer_basis(S)

Return a basis of the centralizer of S in self.

INPUT:

• S – a subalgebra of self or a list of elements that represent generators for a subalgebra
See also:

```
centralizer()
```

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a,b,c = L.lie_algebra_generators()
sage: L.centralizer_basis([a + b, 2*a + c])
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: H.centralizer_basis(H)
[z]
sage: D = DescentAlgebra(QQ, 4).D()
sage: L = LieAlgebra(associative=D)
sage: L.centralizer_basis(L)
[D{},
 D{1} + D{1, 2} + D{2, 3} + D{3},
 D{1, 2, 3} + D{1, 3} + D{2}]
sage: D.center_basis()
(D{},
 D{1} + D{1, 2} + D{2, 3} + D{3},
 D{1, 2, 3} + D{1, 3} + D{2})
```

**chevalley_eilenberg_complex** *(M=None, dual=False, sparse=True, ncpus=None)*

Return the Chevalley-Eilenberg complex of self.

Let \(g\) be a Lie algebra and \(M\) be a right \(g\)-module. The *Chevalley-Eilenberg complex* is the chain complex on

\[
C_\bullet (g, M) = M \otimes \bigwedge^\bullet g,
\]

where the differential is given by

\[
\begin{align*}
\text{for } m \otimes g_1 \wedge \cdots \wedge g_p \\
\quad d(m \otimes g_1 \wedge \cdots \wedge g_p) &= \sum_{i=1}^p (-1)^{i+1} (mg_i) \otimes g_1 \wedge \cdots \wedge \hat{g}_i \wedge \cdots \wedge g_p + \sum_{1 \leq i < j \leq p} (-1)^{i+j} m \otimes [g_i, g_j] \wedge g_1 \wedge \cdots \wedge \hat{g}_i \wedge \cdots \wedge \hat{g}_j \wedge \cdots \wedge g_p
\end{align*}
\]

**INPUT:**
- \(M\) – (default: the trivial 1-dimensional module) the module \(M\)
- \(dual\) – (default: False) if True, causes the dual of the complex to be computed
- \(sparse\) – (default: True) whether to use sparse or dense matrices
- \(ncpus\) – (optional) how many cpus to use

**EXAMPLES:**

```
sage: L = lie_algebras.sl(ZZ, 2)
sage: C = L.chevalley_eilenberg_complex(); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
0 <-- C_0 <-------- C_1 <----------- C_2 <---- C_3 <-- 0
sage: L = LieAlgebra(QQ, cartan_type=['C',2])
sage: C = L.chevalley_eilenberg_complex()  # long time
```

(continues on next page)
REFERENCES:

• Wikipedia article Lie_algebra_cohomology#Chevalley-Eilenberg_complex
• [Wei1994] Chapter 7

Todo: Currently this is only implemented for coefficients given by the trivial module $R$, where $R$ is the base ring and $gR = 0$ for all $g \in g$. Allow generic coefficient modules $M$.

**cohomology** (*deg=*
*None*, *M=*
*None*, *sparse=True*, *ncpus=None*)

Return the Lie algebra cohomology of *self*.

The Lie algebra cohomology is the cohomology of the Chevalley-Eilenberg cochain complex (which is the dual of the Chevalley-Eilenberg chain complex).

Let $g$ be a Lie algebra and $M$ a left $g$-module. It is known that $H^0(g; M)$ is the subspace of $g$-invariants of $M$:

\[ H^0(g; M) = M^g = \{ m \in M | gm = 0 \text{ for all } g \in g \}. \]

Additionally, $H^1(g; M)$ is the space of derivations $g \to M$ modulo the space of inner derivations, and $H^2(g; M)$ is the space of equivalence classes of Lie algebra extensions of $g$ by $M$.

**INPUT:**

• *deg* – the degree of the homology (optional)
• *M* – (default: the trivial module) a right module of *self*
• *sparse* – (default: True) whether to use sparse matrices for the Chevalley-Eilenberg chain complex
• *ncpus* – (optional) how many cpus to use when computing the Chevalley-Eilenberg chain complex

**EXAMPLES:**

```python
sage: L = lie_algebras.so(QQ, 4)
sage: L.cohomology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 2 over Rational Field,
 4: Vector space of dimension 0 over Rational Field,
 5: Vector space of dimension 0 over Rational Field,
 6: Vector space of dimension 1 over Rational Field}
```

```python
sage: L = lie_algebras.Heisenberg(QQ, 2)
sage: L.cohomology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 4 over Rational Field,
 2: Vector space of dimension 5 over Rational Field,
 3: Vector space of dimension 5 over Rational Field,
 4: Vector space of dimension 4 over Rational Field,
 5: Vector space of dimension 1 over Rational Field}
```

```python
d = {('x', 'y'): {'y': 2}}
sage: L.<x,y> = LieAlgebra(ZZ, d)
sage: L.cohomology()
{0: Z, 1: Z, 2: C2}
```
See also:

\texttt{chevalley_eilenberg_complex()}

REFERENCES:
- Wikipedia article Lie_algebra_cohomology

\texttt{derivations\_basis()}

Return a basis for the Lie algebra of derivations of \texttt{self} as matrices.

A derivation \(D\) of an algebra is an endomorphism of \(A\) such that

\[
D([a, b]) = [D(a), b] + [a, D(b)]
\]

for all \(a, b \in A\). The set of all derivations form a Lie algebra.

EXAMPLES:

We construct the derivations of the Heisenberg Lie algebra:

\begin{verbatim}
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: H.derivations_basis()
(  \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix})
\end{verbatim}

We construct the derivations of \texttt{sl2}:

\begin{verbatim}
sage: sl2 = lie_algebras.sl(QQ, 2)
sage: sl2.derivations_basis()
(  \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1/2 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & -2 & 0 \end{bmatrix})
\end{verbatim}

We verify these are derivations:

\begin{verbatim}
sage: D = [sl2.module_morphism(matrix=M, codomain=sl2) for M in sl2.derivations_basis()]
sage: all(d(a.bracket(b)) == d(a).bracket(b) + a.bracket(d(b)) for a in sl2.basis() for b in sl2.basis() for d in D)
True
\end{verbatim}

REFERENCES:
- Wikipedia article Derivation_(differential_algebra)

\texttt{derived\_series()}

Return the derived series \((\mathfrak{g}^{(i)})\), of \texttt{self} where the rightmost \(\mathfrak{g}^{(k)} = \mathfrak{g}^{(k+1)} = \cdots\).

We define the derived series of a Lie algebra \(\mathfrak{g}\) recursively by \(\mathfrak{g}^{(0)} := \mathfrak{g}\) and

\[
\mathfrak{g}^{(k+1)} = [\mathfrak{g}^{(k)}, \mathfrak{g}^{(k)}]
\]

and recall that \(\mathfrak{g}^{(k)} \supseteq \mathfrak{g}^{(k+1)}\). Alternatively we can express this as

\[
\mathfrak{g} \supseteq [\mathfrak{g}, \mathfrak{g}] \supseteq [[\mathfrak{g}, \mathfrak{g}], [\mathfrak{g}, \mathfrak{g}]] \supseteq \cdots.
\]

EXAMPLES:
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.derived_series()
(An example of a finite dimensional Lie algebra with basis:
 the 3-dimensional abelian Lie algebra over Rational Field,
An example of a finite dimensional Lie algebra with basis:
 the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix:
[])

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: L.derived_series() # todo: not implemented - #17416
(Lie algebra on 2 generators (x, y) over Rational Field,
 Subalgebra generated of Lie algebra on 2 generators (x, y) over RationalField
 with basis: (x,),
 Subalgebra generated of Lie algebra on 2 generators (x, y) over RationalField
 with basis: ()
)

\textbf{derived_subalgebra()}

Return the derived subalgebra of \texttt{self}.

\textbf{EXAMPLES:}

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.derived_subalgebra()
An example of a finite dimensional Lie algebra with basis:
 the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix:
[]

\textbf{from_vector(v)}

Return the element of \texttt{self} corresponding to the vector \texttt{v} in \texttt{self.module()}.

Implement this if you implement \texttt{module()}; see the documentation of \texttt{sage.categories.lie_algebras.LieAlgebras.module()} for how this is to be done.

\textbf{EXAMPLES:}

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
sage: parent(u) is L
True

\textbf{homology}(\textit{deg=}None, \textit{M=}None, \textit{sparse=}True, \textit{ncpus=}None)

Return the Lie algebra homology of \texttt{self}.

The Lie algebra homology is the homology of the Chevalley-Eilenberg chain complex.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{deg} – the degree of the homology (optional)
  \item \texttt{M} – (default: the trivial module) a right module of \texttt{self}
  \item \texttt{sparse} – (default: True) whether to use sparse matrices for the Chevalley-Eilenberg chain complex
  \item \texttt{ncpus} – (optional) how many cpus to use when computing the Chevalley-Eilenberg chain complex
\end{itemize}

\textbf{EXAMPLES:}
sage: L = lie_algebras.cross_product(QQ)
sage: L.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

sage: L = lie_algebras.pwitt(GF(5), 5)
sage: L.homology()
{0: Vector space of dimension 1 over Finite Field of size 5,
 1: Vector space of dimension 0 over Finite Field of size 5,
 2: Vector space of dimension 1 over Finite Field of size 5,
 3: Vector space of dimension 1 over Finite Field of size 5,
 4: Vector space of dimension 0 over Finite Field of size 5,
 5: Vector space of dimension 1 over Finite Field of size 5}

d = {('x', 'y'): {'y': 2}}
sage: L.<x,y> = LieAlgebra(ZZ, d)
sage: L.homology()
{0: Z, 1: Z x C2, 2: 0}

See also:
chevalley_ellenberg_complex()

ideal(*gens, **kwds)
Return the ideal of self generated by gens.

INPUT:

• gens – a list of generators of the ideal
• category – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

EXAMPLES:

sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: p1,p2,q1,q2,z = H.basis()
sage: I = H.ideal([p1-p2, q1-q2])
sage: I.basis().list()
[-p1 + p2, -q1 + q2, z]
sage: I.reduce(p1 + p2 + q1 + q2 + z)
2*p1 + 2*q1

Passing an extra category to an ideal:

sage: L.<x,y,z> = LieAlgebra(QQ, abelian=True)
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: C = C.Subobjects().Graded().Stratified()
sage: I = L.ideal(x, y, category=C)
sage: I.homogeneous_component_basis(1).list()
[x, y]

inner_derivations_basis()
Return a basis for the Lie algebra of inner derivations of self as matrices.

EXAMPLES:

sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: H.inner_derivations_basis()
is_abelian() Return if \texttt{self} is an abelian Lie algebra.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_abelian()
True
```

```
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x':1}})
sage: L.is_abelian()
False
```

is_ideal(\(A\)) Return if \texttt{self} is an ideal of \(A\).

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: I = L.ideal([2*a - c, b + c])
sage: I.is_ideal(L)
True
```

```
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: L.is_ideal(L)
True
```

```
sage: F = LieAlgebra(QQ, 'F', representation='polynomial')
sage: L.is_ideal(F)
Traceback (most recent call last):
...:
NotImplementedError: A must be a finite dimensional Lie algebra with basis
```

is_nilpotent() Return if \texttt{self} is a nilpotent Lie algebra.

A Lie algebra is nilpotent if the lower central series eventually becomes 0.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_nilpotent()
True
```

is_semisimple() Return if \texttt{self} is a semisimple Lie algebra.

A Lie algebra is semisimple if the solvable radical is zero. In characteristic 0, this is equivalent to saying the Killing form is non-degenerate.

**EXAMPLES:**
is_solvable()
Return if self is a solvable Lie algebra.

A Lie algebra is solvable if the derived series eventually becomes 0.

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_solvable()
True
```

killing_form(x, y)
Return the Killing form on x and y, where x and y are two elements of self.

The Killing form is defined as

\[ \langle x \mid y \rangle = \text{tr} (\text{ad}_x \circ \text{ad}_y) \]

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a,b,c = L.lie_algebra_generators()
sage: L.killing_form(a, b)
0
```

killing_form_matrix()
Return the matrix of the Killing form of self.

The rows and the columns of this matrix are indexed by the elements of the basis of self (in the order provided by basis()).

EXAMPLES:

```sage
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.killing_form_matrix()
[0 0 0]
[0 0 0]
[0 0 0]
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example(0)
sage: m = L.killing_form_matrix(); m
[]
sage: parent(m)
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
```

killing_matrix(x, y)
Return the Killing matrix of x and y, where x and y are two elements of self.

The Killing matrix is defined as the matrix corresponding to the action of \( \text{ad}_x \circ \text{ad}_y \) in the basis of self.

EXAMPLES:
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a,b,c = L.lie_algebra_generators()
sage: L.killing_matrix(a, b)
\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: L.killing_matrix(x, y)
\[
\begin{bmatrix}
0 & 0 \\
-1 & 0
\end{bmatrix}
\]

lower_central_series (submodule=False)
Return the lower central series \((g_i)_i\) of self where the rightmost
\(g_k = g_{k+1} = \cdots\).

INPUT:
• submodule -- (default: False) if True, then the result is given as
  submodules of self

We define the lower central series of a Lie algebra \(g\) recursively by
\(g_0 := g\) and
\[g_{k+1} = [g, g_k]\]
and recall that \(g_k \supseteq g_{k+1}\). Alternatively we can express this as
\[g \supseteq [g, g] \supseteq [[g, g], g] \supseteq \cdots.\]

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.derived_series()
(An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field,
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix:
[])

The lower central series as submodules:

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: L.lower_central_series(submodule=True)
(Sparse vector space of dimension 2 over Rational Field,
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0])

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: L.lower_central_series()
# todo: not implemented - #17416
(Lie algebra on 2 generators (x, y) over Rational Field,
Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational
Field with basis:
(x,))

module (R=None)
Return a dense free module associated to self over R.

EXAMPLES:
**morphism** (on\_generators, codomain=None, check=True)

Return a Lie algebra morphism defined by images of a Lie generating subset of self.

**INPUT:**
- on\_generators – dictionary \{X: Y\} of the images Y in codomain of elements X of domain
- codomain – a Lie algebra (optional); this is inferred from the values of on\_generators if not given
- check – (default: True) boolean; if False the values on the Lie brackets implied by on\_generators will not be checked for contradictory values

**Note:** The keys of on\_generators need to generate domain as a Lie algebra.

**See also:**
sage.algebras.lie\_algebras.morphism.LieAlgebraMorphism\_from\_generators

**EXAMPLES:**
A quotient type Lie algebra morphism

```python
sage: L.<X,Y,Z,W> = LieAlgebra(QQ, {('X','Y'): {'Z':1}, ('X','Z'): {'W':1} \rightarrow})
sage: K.<A,B> = LieAlgebra(QQ, abelian=True)
sage: L.morphism({X: A, Y: B})
```

Lie algebra morphism:

From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
To: Abelian Lie algebra on 2 generators (A, B) over Rational Field

```
Defn: X |--> A
        Y |--> B
        Z |--> 0
        W |--> 0
```

The reverse map \( A \mapsto X, B \mapsto Y \) does not define a Lie algebra morphism, since \([A, B] = 0\) but \([X, Y] \neq 0\):

```python
sage: K.morphism({A:X, B: Y})
Traceback (most recent call last):
...
ValueError: this does not define a Lie algebra morphism;
contradictory values for brackets of length 2
```

**product\_space** (L, submodule=False)

Return the product space \([\text{self}, L]\).

**INPUT:**
- L – a Lie subalgebra of self
- submodule – (default: False) if True, then the result is forced to be a submodule of self

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()  
sage: a,b,c = L.lie\_algebra\_generators()
```
sage: L.product_space(X)
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field
with basis matrix:
[]
sage: Y = L.subalgebra([a, 2*b-c])
sage: X.product_space(Y)
An example of a finite dimensional Lie algebra with basis:
the 0-dimensional abelian Lie algebra over Rational Field with basis matrix:
[]

sage: H = lie_algebras.Heisenberg(ZZ, 4)
sage: Hp = H.product_space(H, submodule=True).basis()
sage: [H.from_vector(v) for v in Hp]
z

sage: L.<x,y> = LieAlgebra(QQ, {('x','y'):{'x':1}})
sage: Lp = L.product_space(L) # todo: not implemented - #17416
sage: Lp # todo: not implemented - #17416
Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis:
(x,)
sage: Lp.product_space(L) # todo: not implemented - #17416
Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis:
(x,)
sage: L.product_space(Lp) # todo: not implemented - #17416
Subalgebra generated of Lie algebra on 2 generators (x, y) over Rational Field with basis:
()

\textbf{quotient (I, names=None, category=None)}

Return the quotient of self by the ideal \( I \).

A quotient Lie algebra.

INPUT:

- \( I \) – an ideal or a list of generators of the ideal
- \( \text{names} \) – (optional) a string or a list of strings; names for the basis elements of the quotient. If \( \text{names} \) is a string, the basis will be named \( \text{names}_1, \ldots, \text{names}_n \).

EXAMPLES:

The Engel Lie algebra as a quotient of the free nilpotent Lie algebra of step 3 with 2 generators:

sage: L.<X,Y,Z,W,U> = LieAlgebra(QQ, 2, step=3)
sage: E = L.quotient(U); E
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X, Y, Z, W, U) over Rational Field
I: Ideal (U)
sage: E.basis().list()
Quotients when the base ring is not a field are not implemented:

```
sage: L = lie_algebras.Heisenberg(ZZ, 1)
sage: L.quotient(L.an_element())
Traceback (most recent call last):
...  
NotImplementedError: quotients over non-fields not implemented
```

structure_coefficients (include_zeros=False)

Return the structure coefficients of self.

INPUT:

- include_zeros -- (default: False) if True, then include the \([x, y] = 0\) pairs in the output

OUTPUT:

A dictionary whose keys are pairs of basis indices \((i, j)\) with \(i < j\), and whose values are the corresponding elements \([b_i, b_j]\) in the Lie algebra.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.structure_coefficients()
Finite family {}
sage: L.structure_coefficients(True)
Finite family {(0, 1): (0, 0, 0), (0, 2): (0, 0, 0), (1, 2): (0, 0, 0)}
```

```
sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L = LieAlgebra(associative=S)
sage: L.structure_coefficients()
Finite family {(2,3), (1,2)}: (1,2,3) - (1,3,2),
            ((2,3), (1,3)): -(1,2,3) + (1,3,2),
            ((1,2,3), (2,3)): -(1,2) + (1,3),
            ((1,2,3), (1,2)): (2,3) - (1,3),
            ((1,2,3), (1,3)): -(2,3) + (1,2),
            ((1,3,2), (2,3)): (1,2) - (1,3),
            ((1,3,2), (1,2)): (2,3) - (1,2),
            ((1,3,2), (1,3)): -2 + (1,3),
            ((1,3,2), (1,2)): -(1,3) + (1,2),
            ((1,3,2), (1,3)): (1,3) - (1,2)}
```

subalgebra (*gens*, **kwds)

Return the subalgebra of self generated by gens.

INPUT:

- gens -- a list of generators of the subalgebra
- category -- (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

EXAMPLES:
```python
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: p1, p2, q1, q2, z = H.basis()
sage: S = H.subalgebra([p1, q1])
sage: S.basis().list()
[p1, q1, z]
sage: S.basis_matrix()
[1 0 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
```

Passing an extra category to a subalgebra:

```python
sage: L = LieAlgebra(QQ, 3, step=2)
sage: x, y, z = L.homogeneous_component_basis(1)
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: C = C.Subobjects().Graded().Stratified()
sage: S = L.subalgebra([x, y], category=C)
sage: S.homogeneous_component_basis(2).list()
[X12]
```

```python
class Subobjects(category, *args)
Bases: sage.categories.subobjects.SubobjectsCategory

A category for subalgebras of a finite dimensional Lie algebra with basis.

class ParentMethods

ambient()
Return the ambient Lie algebra of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.ambient() == L
True
```

basis_matrix()
Return the basis matrix of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.basis_matrix()
[ 1 0 -1/2]
[ 0 1 1]
```

element(n=3)
Return an example of a finite dimensional Lie algebra with basis as per Category.example.

EXAMPLES:

```python
sage: C = LieAlgebras(QQ).FiniteDimensional().WithBasis()
sage: C.example()
```
An example of a finite dimensional Lie algebra with basis:
the 3-dimensional abelian Lie algebra over Rational Field

Other dimensions can be specified as an optional argument:

```
sage: C.example(5)
An example of a finite dimensional Lie algebra with basis:
the 5-dimensional abelian Lie algebra over Rational Field
```

## 3.53 Finite dimensional modules with basis

```python
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
    True
```

class ElementMethods

dense_coefficient_list(order=None)
    Return a list of all coefficients of self.

    By default, this list is ordered in the same way as the indexing set of the basis of the parent of self.

    INPUT:
    • order – (optional) an ordering of the basis indexing set

    EXAMPLES:

    sage: v = vector([0, -1, -3])
    sage: v.dense_coefficient_list()
    [0, -1, -3]
    sage: v.dense_coefficient_list([2,1,0])
    [-3, -1, 0]
    sage: sorted(v.coefficients())
    [-3, -1]
```

class MorphismMethods

image()
    Return the image of self as a submodule of the codomain.

    EXAMPLES:
image_basis()

Return a basis for the image of self in echelon form.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: f = SGA.module_morphism(lambdax: SGA(x**2), codomain=SGA)
sage: f.image_basis()
[(1, 2, 3), [2, 3, 1], [3, 1, 2]]
```

kernel()

Return the kernel of self as a submodule of the domain.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: f = SGA.module_morphism(lambdax: SGA(x**2), codomain=SGA)
sage: K = f.kernel()
sage: K
Free module generated by {0, 1, 2} over Rational Field
sage: K.ambient()
Symmetric group algebra of order 3 over Rational Field
```

kernel_basis()

Return a basis of the kernel of self in echelon form.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: f = SGA.module_morphism(lambdax: SGA(x**2), codomain=SGA)
sage: f.kernel_basis()
[(1, 2, 3) - (3, 2, 1), (1, 3, 2) - (3, 2, 1), (2, 1, 3) - (3, 2, 1)]
```

matrix(base_ring=None, side='left')

Return the matrix of this morphism in the distinguished bases of the domain and codomain.

INPUT:

- `base_ring` – a ring (default: None, meaning the base ring of the codomain)
- `side` – “left” or “right” (default: “left”)

If `side` is “left”, this morphism is considered as acting on the left; i.e. each column of the matrix represents the image of an element of the basis of the domain.

The order of the rows and columns matches with the order in which the bases are enumerated.

See also:

Modules.WithBasis.ParentMethods.module_morphism()

EXAMPLES:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2]); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); y = Y.basis()
    (continues on next page)```
....:

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()
....: 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 \(\mathbb{Z}\) and over \(\mathbb{Q}\)):

sage: phi.parent().homset_category()
Category of commutative additive semigroups
sage: phi.parent().homset_category()  # todo: not implemented
Category of finite dimensional modules with basis over Integer Ring

```python
class ParentMethods

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 \( \text{action}(x, s) = 0 \) for all \( s \in S \). If side is ‘left’ replace the above equation by \( \text{action}(s, x) = 0 \).

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() 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:
An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver (containing the arrows $a:x\rightarrow y$ and $b:x\rightarrow y$) over Rational Field

```
sage: x, y, a, b = 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: 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()`.

**from_vector**(vector, order=None)

Build an element of self from a vector.

EXAMPLES:

```
sage: p_mult = matrix([[0,0,0],[0,0,-1],[0,0,0]])
sage: q_mult = matrix([[0,0,1],[0,0,0],[0,0,0]])
sage: A = algebras.FiniteDimensional(QQ, [p_mult, q_mult, matrix(QQ,3,3)],
....: 'p,q,z')
sage: A.from_vector(vector([1,0,2]))
p + 2*z
```

**gens()**

Return the generators of self.

OUTPUT:

A tuple containing the basis elements of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
sage: F.gens()
(B['a'], B['b'], B['c'])
```

**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)
```

(continues on next page)
sage: Y.print_options(prefix='y'); Y
Free module generated by {2} over Rational Field
sage: y = Y.basis()
y[2]
sage: y[2].lift()
x[2]
sage: Y.retract(x[0]+2*x[1])
3*y[2]

sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
sage: x = C.basis()
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
sage: Y = X.quotient_module(gens)

See also:

• Modules.WithBasis.ParentMethods.submodule()
• Rings.ParentMethods.quotient()
• sage.modules.with_basis.subquotient.QuotientModuleWithBasis

3.54 Finite Dimensional Nilpotent Lie Algebras With Basis

AUTHORS:

• Eero Hakavuori (2018-08-16): initial version

class sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.FiniteDimensionalNilpotentLieAlgebrasWithBasis(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

Category of finite dimensional nilpotent Lie algebras with basis.

class ParentMethods

is_nilpotent()
Return True since self is nilpotent.

EXAMPLES:

sage: L = LieAlgebra(QQ, {('x','y'): {'z': 1}}, nilpotent=True)
sage: L.is_nilpotent()
True

lie_group(name='G', **kwds)
Return the Lie group associated to self.

INPUT:

• name – string (default: 'G'); the name (symbol) given to the Lie group

EXAMPLES:

We define the Heisenberg group:

sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: G = L.lie_group('G'); G
Lie group G of Heisenberg algebra of rank 1 over Rational Field
We test multiplying elements of the group:

```
sage: p, q, z = L.basis()
sage: g = G.exp(p); g
exp(p1)
sage: h = G.exp(q); h
exp(q1)
sage: g*h
exp(p1 + q1 + 1/2*z)
```

We extend an element of the Lie algebra to a left-invariant vector field:

```
sage: X = G.left_invariant_extension(2*p + 3*q, name='X'); X
Vector field X on the Lie group G of Heisenberg algebra of rank 1 over
→Rational Field
sage: X.at(G.one()).display()
X = 2 d/dx_0 + 3 d/dx_1
sage: X.display()
X = 2 d/dx_0 + 3 d/dx_1 + (3/2*x_0 - x_1) d/dx_2
```

See also:

```
NilpotentLieGroup
```

```
step()

Return the nilpotency step of self.
```

EXAMPLES:

```
sage: L = LieAlgebra(QQ, {('X','Y'): {'Z': 1}}, nilpotent=True)
sage: L.step()
2
sage: sc = {('X','Y'): {'Z': 1}, ('X','Z'): {'W': 1}}
sage: LieAlgebra(QQ, sc, nilpotent=True).step()
3
```

3.55 Finite dimensional semisimple algebras with basis

```
class sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis(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 FiniteDimensionalSemisimpleAlgebrasWithBasis
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
```
```
3.55. Finite dimensional semisimple algebras with basis 387
class Commutative(base_category)
Bases:
sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

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
1/6*B[0] + 1/6*B[2] - 1/12*B[3],
```

Lifting those idempotents from the center, we recognize among them the sum and alternating sum of all permutations:

```
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) + 1/24*(1,2,3,4) + 1/24*(1,3,2) + 1/24*(1,3,4) + 1/24*(1,3,4,2) + 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) - 1/24*(1,2,3,4) - 1/24*(1,3,2) - 1/24*(1,3,4) - 1/24*(1,3,4,2) - 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 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).

EXAMPLES:

For the algebra of the (abelian) alternating group $A_3$, we recover three idempotents corresponding to the three one-dimensional representations $V_i$ on which $(1, 2, 3)$ acts on $V_i$ as multiplication by the $i$-th power of a cube root of unity:

```python
sage: R = CyclotomicField(3)
sage: A3 = AlternatingGroup(3).algebra(R)
sage: idempotents = A3.central_orthogonal_idempotents()
sage: idempotents
(1/3*() + 1/3*(1,2,3) + 1/3*(1,3,2),
 1/3*() - (1/3*zeta3+1/3)*(1,2,3) - (-1/3*zeta3)*(1,3,2),
 1/3*() - (-1/3*zeta3)*(1,2,3) - (1/3*zeta3+1/3)*(1,3,2))
sage: A3.is_identity_decomposition_into_orthogonal_idempotents(idempotents)
True
```

For the semisimple quotient of a quiver algebra, we recover the vertices of the quiver:

```python
sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver (containing the arrows a:x->y and b:x->y) over Rational Field
sage: Aquo = A.semisimple_quotient()
sage: Aquo.central_orthogonal_idempotents()
(B['x'], B['y'])
```

radical_basis(**keywords)

Return a basis of the Jacobson radical of this algebra.

• keywords – for compatibility; ignored.

OUTPUT: the empty list since this algebra is semisimple.

EXAMPLES:

```python
sage: A = SymmetricGroup(4).algebra(QQ)
sage: A.radical_basis()
()`
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

    class ParentMethods

    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)  
    # random
cardinalities = 
    ([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.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])

3.56. Finite Enumerated Sets
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:**

- \(\text{EnumeratedSets.ParentMethods.unrank()}\)
- \(\text{rank()}\)

**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
```

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 IsomorphicObjects\(\text{(category, *args)}\)

**Bases:** \(\text{sage.categories.isomorphic_objects.IsomorphicObjectsCategory}\)

**class ParentMethods**

cardinality()

Returns the cardinality of \(\text{self}\) which is the same as that of the ambient set \(\text{self}\) 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
```

def example()

Returns an example of isomorphic object of a finite enumerated set, as per \(\text{Category.example}\).

**EXAMPLES:**
class ParentMethods

cardinality(*ignored_args, **ignored_kwds)
Return the cardinality of self.
This brute force implementation of cardinality() iterates through the elements of self to count them.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
sage: C._cardinality_from_iterator()
3
```

iterator_range(start=None, stop=None, step=None)
Iterate over the range of elements of self starting at start, ending at stop, and stepping by step.
See also:
unrank(), unrank_range()

EXAMPLES:

```python
sage: F = FiniteEnumeratedSet([1,2,3])
sage: list(F.iterator_range(1))
[2, 3]
sage: list(F.iterator_range(stop=2))
[1, 2]
sage: list(F.iterator_range(stop=2, step=2))
[1]
sage: list(F.iterator_range(start=1, step=2))
[2]
sage: list(F.iterator_range(start=1, stop=2))
[2]
sage: list(F.iterator_range(start=0, stop=1))
[1]
sage: list(F.iterator_range(start=0, stop=3, step=2))
[1, 3]
sage: list(F.iterator_range(stop=-1))
[1, 2]
sage: F = FiniteEnumeratedSet([1,2,3,4])
sage: list(F.iterator_range(start=1, stop=3))
[2, 3]
sage: list(F.iterator_range(stop=10))
[1, 2, 3, 4]
```

last()
The last element of self.
self.last() returns the last element of self.
This is the default (brute force) implementation from the category FiniteEnumeratedSet() which can be used when the method __iter__ is provided. Its complexity is $O(n)$ where $n$ is the
size of self.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.last() 3
sage: C._last_from_iterator() 3
```

`list()`
Return a list of the elements of self.

The elements of set x is created and cached on the first call of x.list(). Then each call of x.list() returns a new list from the cached result. Thus in looping, it may be better to do `for e in x:, not for e in x.list():`

See also:

 `_list_from_iterator()`, `_cardinality_from_list()`, `_iterator_from_list()`, and `_unrank_from_list()`

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.list() [1, 2, 3]
```

`random_element()`
A random element in self.

self.random_element() returns a random element in self with uniform probability.

This is the default implementation from the category `EnumeratedSet()` which uses the method unrank.

EXAMPLES:

```python
sage: C = FiniteEnumeratedSets().example()
sage: C.random_element() 1
sage: C._random_element_from_unrank() 2
```

TODO: implement _test_random which checks uniformness

`unrank_range(start=None, stop=None, step=None)`
Return the range of elements of self starting at start, ending at stop, and stepping by step.

See also unrank().

EXAMPLES:

```python
sage: F = FiniteEnumeratedSet([1, 2, 3])
sage: F.unrank_range(1) [2, 3]
sage: F.unrank_range(stop=2) [1, 2]
sage: F.unrank_range(stop=2, step=2) [1]
sage: F.unrank_range(start=1, step=2) (continues on next page)```
3.57 Finite Fields

class sage.categories.finite_fields.FiniteFields(base_category)

The category of finite fields.

EXAMPLES:

```python
sage: K = FiniteFields(); K
Category of finite enumerated fields
```

A finite field is a finite monoid with the structure of a field; it is currently assumed to be enumerated:

```python
sage: K.super_categories()
[Category of fields, Category of finite commutative rings, Category of finite enumerated sets]
```

Some examples of membership testing and coercion:

```python
sage: FiniteField(17) in K
True
sage: RationalField() in K
False
sage: K(RationalField())
Traceback (most recent call last):
...
TypeError: unable to canonically associate a finite field to Rational Field
```

class ElementMethods
class ParentMethods

```python
extra_super_categories()
```

Any finite field is assumed to be endowed with an enumeration.

3.58 FiniteGroups

class sage.categories.finite_groups.FiniteGroups(base_category)

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

class Algebras(category, *args)
    Bases: sage.categories.algebra_functor.AlgebrasCategory

class ParentMethods

element_extra_super_categories()
    Implement Maschke's theorem.
    In characteristic 0 all finite group algebras are semisimple.

EXAMPLES:

sage: FiniteGroups().Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
sage: FiniteGroups().Algebras(FiniteField(7)).is_subcategory(Algebras(FiniteField(7)).Semisimple())
False
sage: FiniteGroups().Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: FiniteGroups().Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False
sage: Cat = CommutativeAdditiveGroups().Finite()
sage: Cat.Algebras(QQ).is_subcategory(Algebras(QQ).Semisimple())
True
sage: Cat.Algebras(GF(7)).is_subcategory(Algebras(GF(7)).Semisimple())
False
sage: Cat.Algebras(ZZ).is_subcategory(Algebras(ZZ).Semisimple())
False
sage: Cat.Algebras(Fields()).is_subcategory(Algebras(Fields()).Semisimple())
False

class ElementMethods

class 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: G = groups.misc.SemimonomialTransformation(GF(5), 3); G
Semimonomial transformation group over Finite Field of size 5 of degree 3
sage: G.cardinality.__module__
'sage.categories.finite_groups'
sage: G.cardinality()
384
cayley_graph_disabled

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 <sage.groups.˓
˓→group.FiniteGroup object at ...> 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))

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

3.59 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, FiniteLatticePoset

class ParentMethods

irreducibles_poset()
Return the poset of meet- or join-irreducibles of the lattice.

A join-irreducible element of a lattice is an element with exactly one lower cover. Dually a meet-
irreducible element has exactly one upper cover.

This is the smallest poset with completion by cuts being isomorphic to the lattice. As a special case
this returns one-element poset from one-element lattice.

See also:

completion_by_cuts().

EXAMPLES:
is_lattice_morphism ($f$, codomain)

Return whether $f$ is a morphism of posets from self to codomain.

A map $f : P \rightarrow Q$ is a poset morphism if

$$x \leq y \Rightarrow f(x) \leq f(y)$$

for all $x, y \in P$.

INPUT:

• $f$ – a function from self to codomain
• codomain – a lattice

EXAMPLES:

We build the boolean lattice of $\{2, 2, 3\}$ and the lattice of divisors of 60, and check that the map $b \mapsto 5 \prod x \in b x$ is a morphism of lattices:

```python
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 $B_2$:

```python
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 $3$:

```python
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]]
```

```python
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()
Return the join-irreducible elements of this finite lattice.

A join-irreducible element of self is an element \( x \) that is not minimal and that can not be written as the join of two elements different from \( x \).

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2],1:[3],2:[3,4],3:[5],4:[5]})
sage: L.join_irreducibles()
[1, 2, 4]
```

See also:
- Dual function: meet_irreducibles()
- Other: double_irreducibles(), join_irreducibles_poset()

join_irreducibles_poset()
Return the poset of join-irreducible elements of this finite lattice.

A join-irreducible element of self is an element \( x \) that is not minimal and can not be written as the join of two elements different from \( x \).

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.join_irreducibles_poset()  
Finite poset containing 3 elements
```

See also:
- Dual function: meet_irreducibles_poset()
- Other: join_irreducibles()

meet_irreducibles()
Return the meet-irreducible elements of this finite lattice.

A meet-irreducible element of self is an element \( x \) that is not maximal and that 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:
- Dual function: join_irreducibles()
- Other: double_irreducibles(), meet_irreducibles_poset()

meet_irreducibles_poset()
Return the poset of join-irreducible elements of this finite lattice.

A meet-irreducible element of self is an element \( x \) that is not maximal and 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.meet_irreducibles_poset()
Finite poset containing 3 elements
```
See also:
- Dual function: `join_irreducibles_poset()`
- Other: `meet_irreducibles()`

### 3.60 Finite Monoids

```python
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:**

```python
sage: FiniteMonoids()
Category of finite monoids
sage: FiniteMonoids().super_categories()
[Category of monoids, Category of finite semigroups]
```

```python
class ElementMethods

pseudo_order()

Returns the pair \([k, j]\) with \(k\) minimal and \(0 \leq j < k\) such that \(\text{self}^k == \text{self}^j\).

Note that \(j\) is uniquely determined.

**EXAMPLES:**

```python
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]
```

```python
sage: x.pseudo_order()
[4, 2]
```

```python
sage: x = M(3)
sage: [ x^i for i in range(7) ]
[1, 3, 9, 3, 9, 3, 9]
```

```python
sage: x.pseudo_order()
[3, 1]
```

```python
sage: x = M(4)
sage: [ x^i for i in range(7) ]
[1, 4, 4, 4, 4, 4, 4]
```

```python
sage: x.pseudo_order()
[2, 1]
```

```python
sage: x = M(5)
sage: [ x^i for i in range(7) ]
[1, 5, 1, 5, 1, 5, 1]
```

```python
sage: x.pseudo_order()
[2, 0]
```
```

TODO: more appropriate name? see, for example, Jean-Eric Pin’s lecture notes on semigroups.

---

3.60. Finite Monoids
class ParentMethods

nerve()

The nerve (classifying space) of this monoid.

OUTPUT: the nerve $BG$ (if $G$ denotes this monoid), as a simplicial set. The $k$-dimensional simplices of this object are indexed by products of $k$ elements in the monoid:

$$a_1 * a_2 * \cdots * a_k$$

The 0th face of this is obtained by deleting $a_1$, and the $k$-th face is obtained by deleting $a_k$. The other faces are obtained by multiplying elements: the 1st face is

$$(a_1 * a_2) * \cdots * a_k$$

and so on. See Wikipedia article Nerve_(category_theory), which describes the construction of the nerve as a simplicial set.

A simplex in this simplicial set will be degenerate if in the corresponding product of $k$ elements, one of those elements is the identity. So we only need to keep track of the products of non-identity elements. Similarly, if a product $a_{i-1}a_i$ is the identity element, then the corresponding face of the simplex will be a degenerate simplex.

EXAMPLES:

The nerve (classifying space) of the cyclic group of order 2 is infinite-dimensional real projective space.

```
sage: Sigma2 = groups.permutation.Cyclic(2)
sage: BSigma2 = Sigma2.nerve()
sage: BSigma2.cohomology(4, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

The $k$-simplices of the nerve are named after the chains of $k$ non-unit elements to be multiplied. The group $\Sigma_2$ has two elements, written $()$ (the identity element) and $(1,2)$ in Sage. So the 1-cells and 2-cells in $B\Sigma_2$ are:

```
sage: BSigma2.n_cells(1)
[(1,2)]
sage: BSigma2.n_cells(2)
[(1,2) * (1,2)]
```

Another construction of the group, with different names for its elements:

```
sage: C2 = groups.misc.MultiplicativeAbelian([2])
sage: BC2 = C2.nerve()
sage: BC2.n_cells(0)
[1]
sage: BC2.n_cells(1)
[f]
sage: BC2.n_cells(2)
[f * f]
```

With mod $p$ coefficients, $B\Sigma_p$ should have its first nonvanishing homology group in dimension $p$:

```
sage: Sigma3 = groups.permutation.Symmetric(3)
sage: BSigma3 = Sigma3.nerve()
```

(continues on next page)
Note that we can construct the $n$-skeleton for $B\Sigma_2$ for relatively large values of $n$, while for $B\Sigma_3$, the complexes get large pretty quickly:

```python
sage: Sigma2.nerve().n_skeleton(14)
Simplicial set with 15 non-degenerate simplices
sage: BSigma3 = Sigma3.nerve()
sage: BSigma3.n_skeleton(3)
Simplicial set with 156 non-degenerate simplices
sage: BSigma3.n_skeleton(4)
Simplicial set with 781 non-degenerate simplices
```

Finally, note that the classifying space of the order $p$ cyclic group is smaller than that of the symmetric group on $p$ letters, and its first homology group appears earlier:

```python
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: list(C3)
[1, f, f^2]
sage: BC3 = C3.nerve()
sage: BC3.n_cells(1)
[f, f^2]
sage: BC3.n_cells(2)
[f * f, f * f^2, f^2 * f, f^2 * f^2]
sage: len(BSigma3.n_cells(2))
25
sage: len(BC3.n_cells(3))
8
sage: len(BSigma3.n_cells(3))
125
```

```python
sage: BC3.homology(range(5), base_ring=GF(3))
{0: Vector space of dimension 0 over Finite Field of size 3,
 1: Vector space of dimension 1 over Finite Field of size 3,
 2: Vector space of dimension 1 over Finite Field of size 3,
 3: Vector space of dimension 1 over Finite Field of size 3,
 4: Vector space of dimension 1 over Finite Field of size 3}
```

```python
sage: BC5 = groups.permutation.Cyclic(5).nerve()
sage: BC5.homology(range(5), base_ring=GF(5))
{0: Vector space of dimension 0 over Finite Field of size 5,
 1: Vector space of dimension 1 over Finite Field of size 5,
 2: Vector space of dimension 1 over Finite Field of size 5,
 3: Vector space of dimension 1 over Finite Field of size 5,
 4: Vector space of dimension 1 over Finite Field of size 5}
```

```
rhodes_radical_congruence (base_ring=None)

Return the Rhodes radical congruence of the semigroup.

The Rhodes radical congruence is the congruence induced on $S$ by the map $S \rightarrow kS \rightarrow kS/radkS$
with $k$ a field.

INPUT:
```
• base_ring (default: \(Q\)) a field

OUTPUT:
• A list of couples \((m, n)\) with \(m \neq n\) in the lexicographic order for the enumeration of the monoid \texttt{self}.

EXAMPLES:

```
sage: M = Monoids().Finite().example()
sage: M.rhodes_radical_congruence()
[(0, 6), (2, 8), (4, 10)]
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: H3 = HeckeMonoid(SymmetricGroup(3))
sage: H3.repr_element_method(style="reduced")
sage: H3.rhodes_radical_congruence()
[(([1, 2], [2, 1]), ([1, 2], [1, 2, 1]), ([2, 1], [1, 2, 1]))]
```

By Maschke’s theorem, every group algebra over \(Q\) is semisimple hence the Rhodes radical of a group must be trivial:

```
sage: SymmetricGroup(3).rhodes_radical_congruence()
[]
sage: DihedralGroup(10).rhodes_radical_congruence()
[]
```

REFERENCES:
• [Rho69]

### 3.61 Finite Permutation Groups

```python
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 \texttt{group_generators}); this is the case for all the existing implementations in Sage.

    EXAMPLES:
```

```python
sage: C = PermutationGroups().Finite(); C
Category of finite enumerated permutation groups
sage: C.super_categories()
[Category of permutation groups,
   Category of finite groups,
   Category of finite finitely generated semigroups]
sage: C.example()
Dihedral group of order 6 as a permutation group
```

```python
class ElementMethods

class ParentMethods

    cycle_index (parent=None)
    Return the cycle index of \texttt{self}.

    INPUT:
```
• self - a permutation group $G$
• parent – a free module with basis indexed by partitions, or behave as such, with a term and sum method (default: the symmetric functions over the rational field in the $p$ basis)

The cycle index of a permutation group $G$ (Wikipedia article Cycle_index) is a gadget counting the elements of $G$ by cycle type, averaged over the group:

$$P = \frac{1}{|G|} \sum_{g \in G} p_{\text{cycle type}(g)}$$

EXAMPLES:

Among the permutations of the symmetric group $S_4$, there is the identity, 6 cycles of length 2, 3 products of two cycles of length 2, 8 cycles of length 3, and 6 cycles of length 4:

```
sage: S4 = SymmetricGroup(4)
sage: P = S4.cycle_index()
sage: 24 * P
```

If $l = (l_1, \ldots, l_k)$ is a partition, $|G| P[l]$ is the number of elements of $G$ with cycles of length $(p_1, \ldots, p_k)$:

```
sage: 24 * P[ Partition([3,1]) ]
8
```

The cycle index plays an important role in the enumeration of objects modulo the action of a group (Pólya enumeration), via the use of symmetric functions and plethysms. It is therefore encoded as a symmetric function, expressed in the powersum basis:

```
sage: P.parent()
Symmetric Functions over Rational Field in the powersum basis
```

This symmetric function can have some nice properties; for example, for the symmetric group $S_n$, we get the complete symmetric function $h_n$:

```
sage: S = SymmetricFunctions(QQ); h = S.h()
sage: h( P )
h[4]
```

Todo: Add some simple examples of Pólya enumeration, once it will be easy to expand symmetric functions on any alphabet.

Here are the cycle indices of some permutation groups:

```
sage: 6 * CyclicPermutationGroup(6).cycle_index()
```

```
sage: 60 * AlternatingGroup(5).cycle_index()
```

```
sage: for G in TransitiveGroups(5):  # long time
....:     G.cardinality() * G.cycle_index()
p[1, 1, 1, 1, 1] + 4*p[5]
p[1, 1, 1, 1, 1] + 5*p[2, 2, 1] + 4*p[5]
p[1, 1, 1, 1, 1] + 5*p[2, 2, 1] + 10*p[4, 1] + 4*p[5]
```

(continues on next page)
Permutation groups with arbitrary domains are supported (see trac ticket #22765):

```python
sage: G = PermutationGroup([['b','c','a']], domain=['a','b','c'])
sage: G.cycle_index()
1/3*p[1, 1, 1] + 2/3*p[3]
```

One may specify another parent for the result:

```python
sage: F = CombinatorialFreeModule(QQ, Partitions())
sage: P = CyclicPermutationGroup(6).cycle_index(parent = F)
sage: 6 * P
B[[1, 1, 1, 1, 1, 1]] + B[[2, 2, 2]] + 2*B[[3, 3]] + 2*B[[6]]
sage: P.parent() is F
True
```

This parent should be a module with basis indexed by partitions:

```python
sage: CyclicPermutationGroup(6).cycle_index(parent = QQ)
Traceback (most recent call last):
  ... ValueError: 'parent' should be a module with basis indexed by partitions
```

REFERENCES:

- [Ke1991]

AUTHORS:

- Nicolas Borie and Nicolas M. Thiéry

example() Returns an example of finite permutation group, as per `Category.example()`.

EXAMPLES:

```python
sage: G = FinitePermutationGroups().example(); G
Dihedral group of order 6 as a permutation group
```

extra_super_categories() Any permutation group is assumed to be endowed with a finite set of generators.

## 3.62 Finite posets

Here is some terminology used in this file:

- An order filter (or upper set) of a poset $P$ is a subset $S$ of $P$ such that if $x \leq y$ and $x \in S$ then $y \in S$.
- An order ideal (or lower set) of a poset $P$ is a subset $S$ of $P$ such that if $x \leq y$ and $y \in S$ then $x \in S$.

```python
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()

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, labels=None, min_label=None, max_label=None)

Return the birational free labelling of self.

Let us hold back defining this, and introduce birational toggles and birational rowmotion first. These notions have been introduced in [EP2013] as generalizations of the notions of toggles (order_ideal_toggle()) and rowmotion on order ideals of a finite poset. They have been studied further in [GR2013].

Let $K$ be a field, and $P$ be a finite poset. Let $\hat{P}$ 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 $K$-labelling of $P$ will mean any function from $\hat{P}$ to $K$. The image of an element $v$ of $\hat{P}$ under this labelling will be called the label of this labelling at $v$. The set of all $K$-labellings of $P$ is clearly $K^{\hat{P}}$.

For any $v \in P$, we now define a rational map $T_v : K^{\hat{P}} \rightarrow K^{\hat{P}}$ as follows: For every $f \in K^{\hat{P}}$, the image $T_v f$ should send every element $u \in \hat{P}$ distinct from $v$ to $f(u)$ (so the labels at all $u \neq v$ don’t change), while $v$ is sent to

$$
\frac{1}{f(v)} \cdot \frac{\sum_{u \leq v} f(u)}{\sum_{u \gg v} f(u)}
$$

(both sums are over all $u \in \hat{P}$ satisfying the respectively given conditions). Here, $\lt$ and $\gt$ mean (respectively) “covered by” and “covers”, interpreted with respect to the poset $\hat{P}$. This rational map $T_v$ is an involution and is called the (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 $K^{\hat{P}} \rightarrow K^{\hat{P}}$ 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 $K$ 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 [EP2013b] and future work, via the related notion of birational promotion) and rowmotion on order ideals of the poset ([EP2013]).

The birational free labelling is a special labelling defined for every finite poset $P$ and every linear extension $(v_1, v_2, \ldots, v_n)$ of $P$. It is given by sending every element $v_i$ in $P$ to $x_i$, sending the element 0 of $\hat{P}$ to $a$, and sending the element 1 of $\hat{P}$ to $b$, where the ground field $K$ is the field of rational functions in $n + 2$ indeterminates $a, x_1, x_2, \ldots, x_n, b$ over $\mathbb{Q}$.

In Sage, a labelling $f$ of a poset $P$ is encoded as a 4-tuple $(K, d, u, v)$, where $K$ is the ground field of the labelling (i.e., its target), $d$ is the dictionary containing the values of $f$ at the elements of $P$ (the keys being the respective elements of $P$), $u$ is the label of $f$ at 0, and $v$ is the label of $f$ at 1.

**Warning:** The dictionary $d$ is labelled by the elements of $P$. If $P$ is a poset with facade option set to False, these might not be what they seem to be! (For instance, if $P == Poset((1: [2, 3]), facade=False)$, 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 frogl, frog2, ..., frogn rather than x1, x2, ..., xn).
- `base_field` – (default: $\mathbb{Q}$) the base field to be used instead of $\mathbb{Q}$ to define the rational function field over; this is not going to be the base field of the labelling, because the latter will have indeterminates adjoined!
- `reduced` – (default: False) if set to True, the result will be the reduced birational free labelling, which differs from the regular one by having 0 and 1 both sent to 1 instead of $a$ and $b$ (the indeterminates $a$ and $b$ then also won’t appear in the ground field)
- `addvars` – (default: '') a string containing names of extra variables to be adjoined to the ground field (these don’t have an effect on the labels)
- `labels` – (default: 'x') Either a function that takes an element of the poset and returns a name for the indeterminate corresponding to that element, or a string containing a comma-separated list of indeterminates that will be assigned to elements in the order of `linear_extension`. If the list contains more indeterminates than needed, the excess will be ignored. If it contains too few, then the needed indeterminates will be constructed from `prefix`.
- `min_label` – (default: 'a') a string to be used as the label for the element 0 of $\hat{P}$
- `max_label` – (default: 'b') a string to be used as the label for the element 1 of $\hat{P}$

**OUTPUT:**

The birational free labelling of the poset `self` and the linear extension `linear_extension`. Or, if `reduced` is set to True, the reduced birational free labelling.

**EXAMPLES:**

We construct the birational free labelling on a simple poset:
Illustrating labelling with a function:

```
sage: P = posets.ChainPoset(2).product(posets.ChainPoset(2))
sage: l = P.birational_free_labelling(labels=lambda e : 'x_' + str(e[0]) + str(e[1]))
sage: sorted(l[1].items())
[(1, x1), (2, x2), (3, x3)]
```
The same, but with min_label and max_label provided:

```
sage: P = posets.ChainPoset(2).product(posets.ChainPoset(2))
sage: l = P.birational_free_labelling(labels=lambda e : 'x_' + str(e[0]) + str(e[1]), min_label="lambda", max_label="mu")
sage: sorted(l[1].items())
```

```
[((0, 0), x_00), ((0, 1), x_01), ((1, 0), x_10), ((1, 1), x_11)]
```

```
sage: lambda
```

```
sage: mu
```

Illustrating labelling with a comma separated list of labels:

```
sage: l = P.birational_free_labelling(labels='w,x,y,z')
sage: sorted(l[1].items())
```

```
[((0, 0), w), ((0, 1), x), ((1, 0), y), ((1, 1), z)]
```

```
sage: l = P.birational_free_labelling(labels='w,x,y,z,m')
sage: sorted(l[1].items())
```

```
[((0, 0), w), ((0, 1), x), ((1, 0), y), ((1, 1), z)]
```

```
sage: l = P.birational_free_labelling(labels='w')
sage: sorted(l[1].items())
```

```
[((0, 0), w), ((0, 1), x1), ((1, 0), x2), ((1, 1), 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="spam, eggs"); l
```

```
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, b, spam, eggs over Rational Field,
{"...}, 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, b, ohai over Rational Field,
{"...}, a,)
```

(continues on next page)
b) `sage: sorted(l[1].items())

```
([(1, 1), (2), x1), ([1, 1], [3], x2), ([1, 2], [2], x3), ([1, 2], [3], x4),
 ([1, 3], [2], x5), ([1, 3], [3], x6), ([2, 2], [3], x7), ([2, 3], [3], x8)]
```

See `birational_rowmotion()`, `birational_toggle()` and `birational_toggles()` for more substantial examples of what one can do with the birational free labelling.

**birational_rowmotion** *(labelling)*

Return the result of applying birational rowmotion to the K-labelling `labelling` of the poset `self`. See the documentation of `birational_free_labelling()` for a definition of birational rowmotion and K-labellings and for an explanation of how K-labellings are to be 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 Field,
..., a, b)
sage: sorted(t[1].items())
[(1, x1), (2, x2), (3, x3), (4, x4)]
sage: t = P.birational_rowmotion(t); t
(Fraction Field of Multivariate Polynomial Ring in a, x1, x2, x3, x4, b
→over Rational Field,
..., a, b)
sage: sorted(t[1].items())
[(1, a*b/x4), (2, (x1*x2*b + x1*x3*b)/(x2*x4)),
(3, (x1*x2*b + x1*x3*b)/(x3*x4)), (4, (x2*b + x3*b)/x4)]
```

A result of [GR2013] states that applying birational rowmotion \( n + m \) times to a K-labelling \( f \) of the poset \( [n] \times [m] \) gives back \( f \). Let us check this:

```
sage: def test_rectangle_periodicity(n, m, k):
    ....:     P = posets.ChainPoset(n).product(posets.ChainPoset(m))
    ....:     t0 = P.birational_free_labelling(P)
    ....:     t = t0
    ....:     for i in range(k):
```

(continues on next page)
While computations with the birational free labelling quickly run out of memory due to the complexity of the rational functions involved, it is computationally cheap to check properties of birational rowmotion on examples in the tropical semiring:

```python
sage: def test_rectangle_periodicity_tropical(n, m, k):
    ....:    P = posets.ChainPoset(n).product(posets.ChainPoset(m))
    ....:    TT = TropicalSemiring(ZZ)
    ....:    t0 = (TT, {v: TT(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 $\mathbb{Z}$ and $P$ is a finite poset, then we can define an embedding $\phi$ from the set $J(P)$ of all order ideals of $P$ into the set $T^P$ of all $T$-labellings of $P$ by sending every $I \in J(P)$ to the indicator function of $I$ extended by the value 1 at the element 0 and the value 0 at the element 1. This map $\phi$ has the property that $R \circ \phi = \phi \circ r$, where $R$ denotes birational rowmotion, and $r$ denotes classical rowmotion on $J(P)$. An example:

```python
sage: P = posets.IntegerPartitions(5)
sage: TT = TropicalSemiring(ZZ)
sage: def indicator_labelling(I):
    ....:    # 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
```

**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 **KLabelling**.
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:
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, {'123': TT(2), '132': TT(4), '213': TT(3), '231': TT(1), '312': TT(2), '321': TT(1)}, TT(7), TT(1))
sage: t1 = P.birational_toggle('123', t); t1
(Tropical semiring over Rational Field, {123, 132, 213, 231, 312, 321}, 7, 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 )
```
Furthermore, two toggles $T_v$ and $T_w$ commute unless one of $v$ or $w$ covers the other:

```python
sage: all( P.covers(v, w) or P.covers(w, v)
....:     or P.birational_toggle(v, P.birational_toggle(w, t))
....:     for v in P for w in P )
True
```

**`birational_toggles(vs, labelling)`**

Return the result of applying a sequence of birational toggles (specified by `vs`) to the $K$-labelling `labelling` of the poset `self`.

See the documentation of `birational_free_labelling()` for a definition of birational toggles and $K$-labellings and for an explanation of how $K$-labellings are to be encoded to be understood by Sage. This implementation allows $K$ to be a semifield, not just a field. The birational $v$-toggle is only a rational map, so an exception (most likely, `ZeroDivisionError`) will be thrown if the denominator is zero.

**INPUT:**
- `vs` – an iterable comprising elements of `self` (which must have `self` as parent if `self` is a `facade=False` poset)
- `labelling` – a $K$-labelling of `self` in the sense as defined in the documentation of `birational_free_labelling()`

**OUTPUT:**
The $K$-labelling $T_{v_n} T_{v_{n-1}} \cdots T_{v_1} f$ of `self`, where $f$ is `labelling` and $(v_1, v_2, \ldots, v_n)$ is `vs` (written as list).

**EXAMPLES:**

```python
sage: P = posets.SymmetricGroupBruhatOrderPoset(3)
sage: sorted(list(P))
['123', '132', '213', '231', '312', '321']
sage: TT = TropicalSemiring(ZZ)
sage: t = (TT, {'123': TT(4), '132': TT(2), '213': TT(3), '231': TT(1),
        '312': TT(2), '321': TT(1)});
(...)
sage: tA = P.birational_toggles(['123', '231', '312'], t); tA
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tA[1].items())
[('123', 6), ('132', 2), ('213', 3), ('231', 2), ('312', 1), ('321', 1)]
sage: tAB = P.birational_toggles(['132', '213', '321'], tA); tAB
(Tropical semiring over Integer Ring, {...}, 7, 1)
sage: sorted(tAB[1].items())
[('123', 6), ('132', 6), ('213', 5), ('231', 2), ('312', 1), ('321', 1)]
sage: P = Poset({1: [2, 3], 2: [4], 3: [4]})
sage: Qx = PolynomialRing(QQ, 'x').fraction_field()
sage: x = Qx.gen()
sage: t = (Qx, {1: 1, 2: x, 3: (x+1)/x, 4: x^2}, 1, 1)
sage: t1 = P.birational_toggles((i for i in range(1, 5)), t); t1
(Fraction Field of Univariate Polynomial Ring in x over Rational Field,
(...), 1, 1)
sage: sorted(t1[1].items())
```
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, 6, 12], [4, 12], [6, 12], [12]]
```

is_lattice()

Return whether the poset is a lattice.

A poset is a lattice if all pairs of elements have both a least upper bound (“join”) and a greatest lower bound (“meet”) in the poset.

EXAMPLES:

```
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

sage: P = Poset({1: [2, 3, 4], 2: [5, 6], 3: [5, 7], 4: [6, 7], 5: [8, 9], 6: [8, 10], 7: [9, 10], 8: [11], 9: [11], 10: [11]})
sage: P.is_lattice()
False
```

See also:
• Weaker properties: `is_join_semilattice()`, `is_meet_semilattice()`

`is_poset_isomorphism(f, codomain)`
Return whether $f$ is an isomorphism of posets from `self` to `codomain`.

**INPUT:**
- $f$ – a function from `self` to `codomain`
- `codomain` – a poset

**EXAMPLES:**

We build the poset $D$ of divisors of 30, and check that it is isomorphic to the boolean lattice $B$ of the subsets of $\{2, 3, 5\}$ ordered by inclusion, via the reverse function $f : B \to D, b \mapsto \prod_{x \in b} x$:

```
sage: D = Poset((divisors(30), attrcall("divides")))
sage: B = Poset(([frozenset(s) for s in Subsets([2,3,5])], attrcall("issubset")))
sage: def f(b): return D(prod(b))
sage: B.is_poset_isomorphism(f, D)
True
```

On the other hand, $f$ is not an isomorphism to the chain of divisors of 30, ordered by usual comparison:

```
sage: P = Poset((divisors(30), operator.le))
sage: def f(b): return P(prod(b))
sage: B.is_poset_isomorphism(f, P)
False
```

A non surjective case:

```
sage: B = Poset(([frozenset(s) for s in Subsets([2,3])], attrcall("issubset")))
sage: def f(b): return D(prod(b))
sage: B.is_poset_isomorphism(f, D)
False
```

A non injective case:

```
sage: B = Poset(([frozenset(s) for s in Subsets([2,3,5,6])], attrcall("issubset")))
sage: def f(b): return D(gcd(prod(b), 30))
sage: B.is_poset_isomorphism(f, D)
False
```

**Note:** since $D$ and $B$ are not facade posets, $f$ is responsible for the conversions between integers and subsets to elements of $D$ and $B$ and back.

**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

\[ x \leq y \implies f(x) \leq f(y) \]

for all $x$ and $y$ in `self`.

**INPUT:**
• \( f \) – a function from \texttt{self} to \texttt{codomain}
• \texttt{codomain} – a poset

EXAMPLES:

We build the boolean lattice of the subsets of \( \{2, 3, 5, 6\} \) and the lattice of divisors of 30, and check that the map \( b \mapsto \gcd(\prod_{x \in b} x, 30) \) is a morphism of posets:

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: B = Poset({[frozenset(s) for s in Subsets([2,3,5,6])], attrcall("issubset")})
sage: def f(b):
    return D(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, D)
True
```

Note: since \( D \) and \( B \) are not facade posets, \( f \) is responsible for the conversions between integers and subsets to elements of \( D \) and \( B \) and back.

\( f \) is also a morphism of posets to the chain of divisors of 30, ordered by usual comparison:

```python
sage: P = Poset((divisors(30), operator.le))
sage: def f(b):
    return P(gcd(prod(b), 30))
sage: B.is_poset_morphism(f, P)
True
```

FIXME: should this be \texttt{is_order_preserving_morphism}?

See also:

\texttt{is_poset_isomorphism()}

\textbf{is_self_dual()}

Return whether the poset is \textit{self-dual}.

A poset is self-dual if it is isomorphic to its dual poset.

EXAMPLES:

```python
sage: P = Poset({1: [3, 4], 2: [3, 4]})
sage: P.is_self_dual()
True
```

```python
sage: P = Poset({1: [2, 3]})
sage: P.is_self_dual()
False
```

See also:

• Stronger properties: \texttt{is_orthocomplemented()} (for lattices)
• Other: \texttt{dual()}

\textbf{order_filter_generators} \texttt{(filter)}

Generators for an order filter

INPUT:

• \texttt{filter} – an order filter of \texttt{self}, as a list (or iterable)

EXAMPLES:
sage: P = Poset((Subsets([1,2,3]), attrcall("issubset")))
sage: I = P.order_filter([Set([1,2]), Set([2,3]), Set([1])])
sage: sorted(sorted(p) for p in I)
[[1], [1, 2], [1, 2, 3], [1, 3], [2, 3]]
sage: gen = P.order_filter_generators(I)
sage: sorted(sorted(p) for p in gen)
[[1], [2, 3]]

See also:

order_ideal_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:

\[
\begin{align*}
\text{sage: } P &= \text{Poset}([\text{Subsets([1,2,3]), attrcall("issubset")}]) \\
\text{sage: } I &= \text{P.order_ideal}([\text{Set([1,2])}, \text{Set([2,3])}, \text{Set([1])}]) \\
\text{sage: } \text{sorted(sorted(p) for p in I)} \\
&= [\emptyset, [1], [1, 2], [2], [2, 3], [3]]
\end{align*}
\]

Then, we retrieve the generators of this ideal:

\[
\begin{align*}
\text{sage: } \text{gen} &= \text{P.order_ideal_generators(I)} \\
\text{sage: } \text{sorted(sorted(p) for p in gen)} \\
&= [[1, 2], [2, 3]]
\end{align*}
\]

If direction is ‘up’, then this instead computes the minimal generators for an order filter:

\[
\begin{align*}
\text{sage: } I &= \text{P.order_filter}([\text{Set([1,2])}, \text{Set([2,3])}, \text{Set([1])}]) \\
\text{sage: } \text{sorted(sorted(p) for p in I)} \\
&= [[1], [1, 2], [1, 2, 3], [1, 3], [2, 3]]
\end{align*}
\]

\[
\begin{align*}
\text{sage: } \text{gen} &= \text{P.order_ideal_generators(I, direction='up')} \\
\text{sage: } \text{sorted(sorted(p) for p in gen)} \\
&= [[1], [2, 3]]
\end{align*}
\]

Complexity: \(O(n + m)\) where \(n\) is the cardinality of \(I\), and \(m\) the number of upper covers of elements of \(I\).

order_ideals_lattice(as_ideals=True, facade=None)
Return the lattice of order ideals of a poset self, ordered by inclusion.

The lattice of order ideals of a poset \(P\) is usually denoted by \(J(P)\). Its underlying set is the set of order ideals of \(P\), and its partial order is given by inclusion.

The order ideals of \(P\) are in a canonical bijection with the antichains of \(P\). The bijection maps every order ideal to the antichain formed by its maximal elements. By setting the as_ideals keyword variable to False, one can make this method apply this bijection before returning the lattice.

INPUT:
• as_ideals – Boolean, if True (default) returns a poset on the set of order ideals, otherwise on the set of antichains
• facade – Boolean or None (default). Whether to return a facade lattice or not. By default return facade lattice if the poset is a facade poset.

EXAMPLES:
As a lattice on antichains:

```
sage: J2 = P.order_ideals_lattice(False); J2
Finite lattice containing 8 elements
sage: sorted(J2)
[(), (0,), (1,), (1, 2), (1, 3), (2,), (3,), (4,)]
```

`panyushev_complement` \((antichain, direction='up')\)

Return the Panyushev complement of the antichain `antichain`.

Given an antichain \(\mathcal{A}\) of a poset \(P\), the Panyushev complement of \(\mathcal{A}\) 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 \(\mathcal{A}\).

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 \(\mathcal{A}\) is the antichain whose Panyushev complement is \(\mathcal{A}\). 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 \(\mathcal{A}\).

`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()
```
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:

```python
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))]
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)
```

(continues on next page)
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:

```python
sage: P = Poset( ([1,2,3], [[1,3], [2,3]] ) )
sage: orb = P.panyushev_orbits()
sage: sorted(sorted(o) for o in orb)
[[set(), {1, 2}, {3}], [{2}, {1}]]
sage: orb = P.panyushev_orbits(element_constructor=list)
sage: sorted(sorted(o) for o in orb)
[[[], [1, 2], [3]], [[1], [2]]]
sage: orb = P.panyushev_orbits(element_constructor=frozenset)
sage: sorted(sorted(o) for o in orb)
[[frozenset(), frozenset({1, 2}), frozenset({3})],
 [frozenset({2}), frozenset({1})]]
sage: orb = P.panyushev_orbits(element_constructor=tuple)
sage: sorted(sorted(o) for o in orb)
[[(), (1, 2), (3,)], [(1,), (2,)]]
sage: P = Poset( {} )
sage: P.panyushev_orbits()
[[set()]]
```

rowmotion (order_ideal)

The image of the order ideal order_ideal under rowmotion in self.

Rowmotion on a finite poset P is an automorphism of the set J(P) of all order ideals of P. One way to define it is as follows: Given an order ideal I ∈ J(P), we let F be the set-theoretic complement of I in P. Furthermore we let A be the antichain consisting of all minimal elements of F. Then, the rowmotion of I is defined to be the order ideal of P generated by the antichain A (that is, the order ideal consisting of each element of P which has some element of A above it).

Rowmotion is related (actually, isomorphic) to Panyushev complementation (panyushev_complement()).

INPUT:

• order_ideal – an order ideal of self, as a set

OUTPUT:

• the image of order_ideal under rowmotion, as a set again
EXAMPLES:

```python
sage: P = Poset( {1: [2, 3], 2: [], 3: [], 4: [8], 5: [], 6: [5], 7: [1, 4], 8: [1] })
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)
{}
```

```python
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:

```python
sage: P = Poset( ( [1,2,3], [ [1,3], [2,3] ] ) )
sage: list(P.rowmotion_orbit_iter(set([1, 2])))
[(1, 2), (1, 2, 3), set()]
sage: list(P.rowmotion_orbit_iter([1, 2]))
[(1, 2), (1, 2, 3), set()]
sage: list(P.rowmotion_orbit_iter([2, 1]))
[(1, 2), (1, 2, 3), set()]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=list))
[[1, 2], [1, 2, 3], []]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=frozenset))
[frozenset({1, 2}), frozenset({1, 2, 3}), frozenset({})]
sage: list(P.rowmotion_orbit_iter(set([1, 2]), element_constructor=tuple))
[(1, 2), (1, 2, 3), ()]
sage: P = Poset( {} )
sage: list(P.rowmotion_orbit_iter([]))
[set()]
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [] })
sage: Piter = P.rowmotion_orbit_iter([1, 2, 3], stop=False)
sage: next(Piter)
```
```
(continues on next page)
```
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 \(L[i]\) 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: [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]],
 ```
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_1 T_2 I, T_1 T_2 T_1 I, ...) (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, ...).

**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), (1, 2, 3), set()]
sage: list(P.toggling_orbit_iter([1, 2, 3], 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: P = Poset( {} )
sage: list(P.toggling_orbit_iter([], []))
[set()]

(continues on next page)
**toggling_orbits** *(vs, element_constructor=\textless\texttt{type 'set'}\textgreater\*)

Return the orbits of order ideals in *self* under the operation of toggling the vertices *vs*[0], *vs*[1], ... in this order.

See *order_ideal_toggle()* for a definition of toggling.

**Warning:** The orbits are those under the composition of toggles, not under the single toggles themselves. Thus, for example, if *vs* == \([1,2]\), then the orbits have the form \((I, T_2 T_1 I, T_2 T_1 T_2 T_1 I, ...)\) (where \(I\) denotes an order ideal and \(T_i\) means toggling at \(i\)) rather than \((I, T_1 I, T_1 T_2 I, T_1 T_2 T_1 I, ...)\).

**INPUT:**
- *vs*: a list (or other iterable) of elements of *self* (but since the output depends on the order, sets should not be used as *vs*).

**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:**

```python
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [ ] })
sage: sorted(len(o) for o in P.toggling_orbits([1, 2, 3]))
[2, 3, 3]
sage: P = Poset({ 1: [3], 2: [1, 4], 3: [ ], 4: [ ] })
sage: sorted(len(o) for o in P.toggling_orbits([1, 2, 3]))
[3, 3]
```

**toggling_orbits_plots** *(vs)*

Return plots of the orbits of order ideals in *self* under the operation of toggling the vertices *vs*[0], *vs*[1], ... in this order.

See *toggling_orbits()* for more information.

**EXAMPLES:**

```python
sage: P = Poset({ 1: [2, 3], 2: [4], 3: [4], 4: [2] })
sage: P.toggling_orbits_plots([1,2,3,4])
Graphics Array of size 2 x 5
sage: P = Poset({})
sage: P.toggling_orbits_plots([])
Graphics Array of size 2 x 1
```
3.63 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 enumerated set structure; the default enumeration is then obtained by iteratively multiplying the semigroup generators. This forced any finite semigroup to either implement an enumeration, or provide semigroup generators; this was often inconvenient.

Instead, finite semigroups that provide a distinguished finite set of generators with \(\text{semigroup\_generators}()\) should now explicitly declare themselves in the category of finitely generated semigroups:

```python
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
```

This is a backward incompatible change.

**EXAMPLES:**

```python
sage: C = FiniteSemigroups(); C
Category of finite semigroups
sage: C.super_categories()
[Category of semigroups, Category of finite sets]
sage: sorted(C.axioms())
['Associative', 'Finite']
sage: C.example()
An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
```

**class ParentMethods**

**idempotents ()**

Returns the idempotents of the semigroup

**EXAMPLES:**

```python
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 \(J\)S-classes of self, as a list of lists.

**EXAMPLES:**
sage: S = FiniteSemigroups().example(alphabet=('a','b', 'c'))
sage: sorted(map(sorted, S.j_classes()))
[['a'], ['ab', 'ba'], ['abc', 'acb', 'bac', 'bca', 'cab', 'cba'], ['ac', 'ca'], ['b'], ['bc', 'cb'], ['c']]

j_classes_of_idempotents()
Returns all the idempotents of self, grouped by J-class.
OUTPUT:
a list of lists.
EXAMPLES:

sage: S = FiniteSemigroups().example(alphabet=('a','b', 'c'))

sage: sorted(map(sorted, S.j_classes_of_idempotents()))
[['a'], ['ab', 'ba'], ['abc', 'acb', 'bac', 'bca', 'cab', 'cba'], ['ac', 'ca'], ['b'], ['bc', 'cb'], ['c']]

j_transversal_of_idempotents()
Returns a list of one idempotent per regular J-class
EXAMPLES:

sage: S = FiniteSemigroups().example(alphabet=('a','b', 'c'))

sage: sorted(S.j_transversal_of_idempotents())
['a', 'acb', 'b', 'ba', 'bc', 'c', 'ca']

3.64 Finite sets

class sage.categories.finite_sets.FiniteSets(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiomSingleton

The category of finite sets.
EXAMPLES:

sage: C = FiniteSets(); C
Category of finite sets
sage: C.super_categories()
[Category of sets]
C.all_super_categories()
[Category of finite sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
sage: C.example()
NotImplemented

class Algebras(category, *args)
Bases: sage.categories.algebra_functor.AlgebrasCategory

extra_super_categories()
EXAMPLES:

sage: FiniteSets().Algebras(QQ).extra_super_categories()
[Category of finite dimensional vector spaces with basis over RationalField]
This implements the fact that the algebra of a finite set is finite dimensional:

```
sage: FiniteMonoids().Algebras(QQ).is_subcategory(AlgebrasWithBasis(QQ).FiniteDimensional())
True
```

```python
class ParentMethods
    is_finite()
    Return True since self is finite.

    EXAMPLES:
    sage: C = FiniteEnumeratedSets().example()
sage: C.is_finite()
    True
```

```python
class Subquotients(*args)
    Bases: sage.categories.subquotients.SubquotientsCategory
    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
```

### 3.65 Finite Weyl Groups

```python
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}
```

```python
class ElementMethods

class ParentMethods
```
### 3.66 Finitely generated magmas

**class** `sage.categories.finitely_generated_magmas.FinitelyGeneratedMagmas(base_category)`

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of finitely generated (multiplicative) magmas.

See `Magmas.SubcategoryMethods.FinitelyGeneratedAsMagma()` for details.

**EXAMPLES:**

```python
sage: C = Magmas().FinitelyGeneratedAsMagma(); C
Category of finitely generated magmas
sage: C.super_categories()
[Category of magmas]
sage: sorted(C.axioms())
['FinitelyGeneratedAsMagma']
```

**class ParentMethods**

**magma_generators()**

Return distinguished magma generators for `self`.

OUTPUT: a finite family

This method should be implemented by all finitely generated magmas.

**EXAMPLES:**

```python
sage: S = FiniteSemigroups().example()
sage: S.magma_generators()
Family ('a', 'b', 'c', 'd')
```

### 3.67 Finitely generated semigroups

**class** `sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups(base_category)`

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

The category of finitely generated (multiplicative) semigroups.

A finitely generated semigroup is a semigroup endowed with a distinguished finite set of generators (see `FinitelyGeneratedSemigroups.ParentMethods.semigroup_generators()`). This makes it into an enumerated set.

**EXAMPLES:**

```python
sage: C = Semigroups().FinitelyGenerated(); C
Category of finitely generated semigroups
sage: C.super_categories()
[Category of semigroups,
  Category of finitely generated magmas,
  Category of enumerated sets]
sage: sorted(C.axioms())
['Associative', 'Enumerated', 'FinitelyGeneratedAsMagma']
sage: C.example()
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')
```
class Finite(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class ParentMethods

    some_elements()
    Return an iterable containing some elements of the semigroup.

    OUTPUT: the ten first elements of the semigroup, if they exist.

    EXAMPLES:

        sage: S = FiniteSemigroups().example(alphabet=('x','y'))
        sage: sorted(S.some_elements())
        ['x', 'xy', 'y', 'yx']
        sage: S = FiniteSemigroups().example(alphabet=('x','y','z'))
        sage: S.some_elements()
        ['x', 'y', 'z', 'xz', 'yx', 'yz', 'zx', 'zy', 'xy', 'yxz']

class ParentMethods

    ideal(gens, side='twosided')
    Return the side-sided ideal generated by gens.

    This brute force implementation recursively multiplies the elements of gens by the distinguished generators of this semigroup.

    See also:

    semigroup_generators()

    INPUT:

    • gens – a list (or iterable) of elements of self
    • side – [default: “twosided”] “left”, “right” or “twosided”

    EXAMPLES:

        sage: S = FiniteSemigroups().example()
        sage: sorted(S.ideal([S('cab')], side="left"))
        ['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
         'adcb', 'bac', 'badc', 'bcad', 'bcda', 'bdac', 'bdca', 'cab', 'cabd', 'cbad', 'caba', 'cbda',
         'cdab', 'cdca', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']
        sage: list(S.ideal([S('cab')], side="right"))
        ['cab', 'cabd']
        sage: sorted(S.ideal([S('cab')], side="twosided"))
        ['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
         'adcb', 'bac', 'badc', 'bcad', 'bcda', 'bdac', 'bdca', 'cab', 'cabd', 'cbad', 'caba', 'cbda',
         'cdab', 'cdca', 'dabc', 'dacb', 'dbac', 'dbca', 'dcab', 'dcba']
        sage: sorted(S.ideal([S('cab')]))
        ['abc', 'abcd', 'abdc', 'acb', 'acbd', 'acdb', 'adbc',
         'adcb', 'bac', 'badc', 'bcad', 'bcda', 'bdac', 'bdca', 'cab', 'cabd', 'cbad', 'caba', 'cbda',
         'cdab', 'cdca', 'dabc', 'dacb', 'dbac', 'dbca',
         'dcab', 'dcba']
semigroup_generators()

Return distinguished semigroup generators for self.

OUTPUT: a finite family

This method should be implemented by all semigroups in FinitelyGeneratedSemigroups.

EXAMPLES:

```python
sage: S = FiniteSemigroups().example()
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

succ_generators(side='twosided')

Return the successor function of the \texttt{side}-sided Cayley graph of self.

This is a function that maps an element of self to all the products of \(x\) by a generator of this semigroup, where the product is taken on the left, right, or both sides.

INPUT:
• side: “left”, “right”, or “twosided”

Todo: Design choice:
• find a better name for this method
• should we return a set? a family?

EXAMPLES:

```python
sage: S = FiniteSemigroups().example()
sage: S.succ_generators("left") (S('ca'))
('ac', 'bca', 'ca', 'dca')
sage: S.succ_generators("right") (S('ca'))
('ca', 'cab', 'ca', 'cad')
sage: S.succ_generators("twosided") (S('ca'))
('ac', 'bca', 'ca', 'dca', 'ca', 'cab', 'ca', 'cad')
```

example()

EXAMPLES:

```python
sage: Semigroups().FinitelyGenerated().example()
An example of a semigroup: the free semigroup generated by \{'a', 'b', 'c', 'd'\}
```

extra_super_categories()

State that a finitely generated semigroup is endowed with a default enumeration.

EXAMPLES:

```python
sage: Semigroups().FinitelyGenerated().extra_super_categories()
[Category of enumerated sets]
```

3.68 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
class ElementMethods
class ParentMethods
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]
```

### 3.69 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]
```

### 3.70 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:

```python
sage: GcdDomains()
Category of gcd domains
sage: GcdDomains().super_categories()
[Category of integral domains]
```

class ElementMethods
class ParentMethods

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:

```python
sage: GcdDomains().additional_structure()
```

super_categories()

EXAMPLES:

```python
sage: GcdDomains().super_categories()
[Category of integral domains]
```

3.71 Generalized Coxeter Groups

class sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups(s=None)
Bases: sage.categories.category_singleton.Category_singleton

The category of generalized Coxeter groups.

A generalized Coxeter group is a group with a presentation of the following form:

\[ \langle s_i \mid s_i^{p_i}, s_i s_j \cdots = s_j s_i \cdots \rangle, \]

where \( p_i > 1 \), \( i \in I \), and the factors in the braid relation occur \( m_{ij} = m_{ji} \) times for all \( i \neq j \in I \).

EXAMPLES:

```python
sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
sage: C = GeneralizedCoxeterGroups(); C
Category of generalized coxeter groups
```

class Finite(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of finite generalized Coxeter groups.
extra_super_categories()
    Implement that a finite generalized Coxeter group is a well-generated complex reflection group.

    EXAMPLES:
    sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
    sage: from sage.categories.complex_reflection_groups import ComplexReflectionGroups
    sage: Cat = GeneralizedCoxeterGroups().Finite()
    sage: Cat.extra_super_categories()
    [Category of well generated finite complex reflection groups]
    sage: Cat.is_subcategory(ComplexReflectionGroups().Finite().WellGenerated())
    True

additional_structure()
    Return None.

    Indeed, all the structure generalized Coxeter groups have in addition to groups (simple reflections, ...) is
    already defined in the super category.

    See also:
    Category.additional_structure()

    EXAMPLES:
    sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
    sage: GeneralizedCoxeterGroups().additional_structure()

super_categories()
    EXAMPLES:
    sage: from sage.categories.generalized_coxeter_groups import GeneralizedCoxeterGroups
    sage: GeneralizedCoxeterGroups().super_categories()

3.72 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 filtered algebras over Integer Ring,
     Category of graded modules over Integer Ring]

class ElementMethods
class ParentMethods

graded_algebra()

Return the associated graded algebra to self.

Since self is already graded, this just returns self.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m.graded_algebra() is m
True```

3.73 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 filtered algebras with basis over Integer Ring,
 Category of graded algebras over Integer Ring,
 Category of graded modules with basis over Integer Ring]
```

class ElementMethods
class ParentMethods

graded_algebra()

Return the associated graded algebra to self.

This is self, because self is already graded. See graded_algebra() for the general behavior of this method, and see AssociatedGradedAlgebra for the definition and properties of associated graded algebras.

EXAMPLES:

```
sage: m = SymmetricFunctions(QQ).m()
sage: m.graded_algebra() is m
True```

3.74 Graded bialgebras

sage.categories.graded_bialgebras.GradedBialgebras(base_ring)

The category of graded bialgebras

EXAMPLES:
3.75 Graded bialgebras with basis

The category of graded bialgebras with a distinguished basis

EXAMPLES:

```python
sage: C = GradedBialgebrasWithBasis(QQ); C
Join of Category of graded modules over Rational Field
    and Category of coalgebras with basis over Rational Field
sage: C is Coalgebras(QQ).Graded()
True
```

3.76 Graded Coalgebras

The category of graded coalgebras

EXAMPLES:

```python
cell(�)
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
```

3.77 Graded coalgebras with basis

The category of graded coalgebras with a distinguished basis

EXAMPLES:

```python
cell(�)
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
```

3.78 Graded Hopf algebras

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
```

### 3.79 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.

```
sage: C = GradedHopfAlgebrasWithBasis(ZZ); C
Category of graded hopf algebras with basis over Integer Ring
sage: C.super_categories()
[Category of filtered hopf algebras with basis over Integer Ring,
  Category of graded algebras with basis over Integer Ring]
sage: C is HopfAlgebras(ZZ).WithBasis().Graded()
True
sage: C is HopfAlgebras(ZZ).Graded().WithBasis()
False
```

#### class Connected(base_category)

```
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring
```

#### class ElementMethods

#### class ParentMethods

**antipode_on_basis(index)**

The antipode on the basis element indexed by `index`.

**INPUT:**
- `index` – an element of the index set

For a graded connected Hopf algebra, we can define an antipode recursively by

\[
S(x) := - \sum_{x^L \neq x} S(x^L) \times x^R
\]

when \(|x| > 0\), and by \(S(x) = x\) when \(|x| = 0\).

**counit_on_basis(i)**

The default counit of a graded connected Hopf algebra.

**INPUT:**
- `i` – an element of the index set

**OUTPUT:**
- `an element of the base ring`
\[ c(i) := \begin{cases} 1 & \text{if } i \text{ indexes the 1 of the algebra} \\ 0 & \text{otherwise.} \end{cases} \]

**EXAMPLES:**

```python
sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
sage: H.monomial(4).counit()  # indirect doctest
0
sage: H.monomial(0).counit()  # indirect doctest
1
```

**example()**

Return an example of a graded connected Hopf algebra with a distinguished basis.

**class** `ElementMethods`

**class** `ParentMethods`

**class** `WithRealizations(category, *args)`

Bases: `sage.categories.with_realizations.WithRealizationsCategory`

**super_categories()**

**EXAMPLES:**

```python
sage: GradedHopfAlgebrasWithBasis(QQ).WithRealizations().super_categories()
[Join of Category of hopf algebras over Rational Field
and Category of graded algebras over Rational Field]
```

**example()**

Return an example of a graded Hopf algebra with a distinguished basis.

### 3.80 Graded Lie Algebras

**AUTHORS:**

- Eero Hakavuori (2018-08-16): initial version

**class** `sage.categories.graded_lie_algebras.GradedLieAlgebras(base_category)`

Bases: `sage.categories.graded_modules.GradedModulesCategory`

Category of graded Lie algebras.

**class** `Stratified(base_category)`

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

Category of stratified Lie algebras.

A graded Lie algebra \( L = \bigoplus_{k=1}^{\infty} L_k \) (where possibly \( M = \infty \)) is called *stratified* if it is generated by \( L_1 \); in other words, we have \( L_{k+1} = [L_1, L_k] \).

**class** `FiniteDimensional(base_category)`

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

Category of finite dimensional stratified Lie algebras.

**EXAMPLES:**

```python
sage: LieAlgebras(QQ).Graded().Stratified().FiniteDimensional()
Category of finite dimensional stratified Lie algebras over Rational Field
```
extra_super_categories()

Implements the fact that a finite dimensional stratified Lie algebra is nilpotent.

EXAMPLES:

```
sage: C = LieAlgebras(QQ).Graded().Stratified().FiniteDimensional()
sage: C.extra_super_categories()
[Category of nilpotent Lie algebras over Rational Field]
sage: C is C.Nilpotent()
True
sage: C.is_subcategory(LieAlgebras(QQ).Nilpotent())
True
```

class SubcategoryMethods

\textbf{Stratified()}

Return the full subcategory of stratified objects of \texttt{self}.

A Lie algebra is stratified if it is graded and generated as a Lie algebra by its component of degree one.

EXAMPLES:

```
sage: LieAlgebras(QQ).Graded().Stratified()
Category of stratified Lie algebras over Rational Field
```

3.81 Graded Lie Algebras With Basis

\textbf{class sage.categories.graded_lie_algebras_with_basis.GradedLieAlgebrasWithBasis}(base_category)

\textbf{Bases:} sage.categories.graded_modules.GradedModulesCategory

The category of graded Lie algebras with a distinguished basis.

EXAMPLES:

```
sage: C = LieAlgebras(ZZ).WithBasis().Graded(); C
Category of graded lie algebras with basis over Integer Ring
sage: C.is_subcategory(LieAlgebras(ZZ).Graded().WithBasis())
False
```

\textbf{FiniteDimensional}

\textit{alias of} sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis

3.82 Graded modules

\textbf{class sage.categories.graded_modules.GradedModules}(base_category)

\textbf{Bases:} sage.categories.graded_modules.GradedModulesCategory
The category of graded modules.

We consider every graded module $M = \bigoplus_i M_i$ as a filtered module under the (natural) filtration given by

$$F_i = \bigoplus_{j < i} M_j.$$

**EXAMPLES:**

```python
sage: GradedModules(ZZ)
Category of graded modules over Integer Ring
sage: GradedModules(ZZ).super_categories()
[Category of filtered modules over Integer Ring]
```

The category of graded modules defines the graded structure which shall be preserved by morphisms:

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

### class `ElementMethods`

### class `ParentMethods`

```python
class sage.categories.graded_modules.GradedModulesCategory(base_category):
    Bases:  
sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory,  
sage.categories.category_types.Category_over_base_ring

EXAMPLES:

```python
sage: C = GradedAlgebras(QQ)
sage: C
Category of graded algebras over Rational Field
sage: C.base_category()
Category of algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of filtered algebras over Rational Field,  
  Category of graded modules over Rational Field]
```

```python
sage: AlgebrasWithBasis(QQ).Graded().base_ring()
Rational Field
sage: GradedHopfAlgebrasWithBasis(QQ).base_ring()
Rational Field
```

```python
classmethod default_super_categories(category, *args):
    Return the default super categories of category.Graded().

    Mathematical meaning: every graded object (module, algebra, etc.) is a filtered object with the (implicit) filtration defined by $F_i = \bigoplus_{j \leq i} G_j$.

    INPUT:
    - cls -- the class GradedModulesCategory
    - category -- a category

    OUTPUT: a (join) category
```

In practice, this returns `category.Filtered()`, joined together with the result of the method `RegressiveCovariantConstructionCategory.default_super_categories()` (that is the join of `category.Filtered()` and `cat` for each `cat` in the super categories of `category`).
EXAMPLES:

Consider `category=Algebras()`, which has `cat=Modules()` as super category. Then, a grading of an algebra $G$ is also a filtration of $G$:

```
sage: Algebras(QQ).Graded().super_categories()
[Category of filtered algebras over Rational Field,
 Category of graded modules over Rational Field]
```

This resulted from the following call:

```
sage: sage.categories.graded_modules.GradedModulesCategory.default_super_categories(Algebras(QQ))
Join of Category of filtered algebras over Rational Field
 and Category of graded modules over Rational Field
```

3.83 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 filtered modules with basis over Integer Ring,
     Category of graded modules over Integer Ring]
    sage: C is ModulesWithBasis(ZZ).Graded()
    True
    
    class ElementMethods

        degree_negation()

        Return the image of self under the degree negation automorphism of the graded module to which self belongs.

        The degree negation is the module automorphism which scales every homogeneous element of degree $k$ by $(-1)^k$ (for all $k$). This assumes that the module to which self belongs (that is, the module self.parent()) is $\mathbb{Z}$-graded.

        EXAMPLES:

        sage: E.<a,b> = ExteriorAlgebra(QQ)
        sage: ((1 + a) * (1 + b)).degree_negation()
        a^b - a - b + 1
        sage: E.zero().degree_negation()
        0
        sage: P = GradedModulesWithBasis(ZZ).example(); P
        An example of a graded module with basis: the free module on partitions over Integer Ring
        sage: pbp = lambda x: P.basis()[Partition(list(x))]
        sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])
```

(continues on next page)
class ParentMethods

def degree_negation(element)
    Return the image of element under the degree negation automorphism of the graded module self.
    The degree negation is the module automorphism which scales every homogeneous element of degree
    \( k \) by \((-1)^k\) (for all \( k \)). This assumes that the module self is \( \mathbb{Z} \)-graded.

    INPUT:
    * element – element of the module self

    EXAMPLES:

    sage: E.<a,b> = ExteriorAlgebra(QQ)
    sage: E.degree_negation((1 + a) * (1 + b))
    a^b - a - b + 1
    sage: E.degree_negation(E.zero())
    0
    sage: P = GradedModulesWithBasis(ZZ).example(); P
    An example of a graded module with basis: the free module on partitions \( \leadsto \) over Integer Ring
    sage: pbp = lambda x: P.basis()[Partition(list(x))]
    sage: p = pbp([3,1]) - 2 * pbp([2]) + 4 * pbp([1])
    sage: P.degree_negation(p)
edges()
    Return the edges of self.

    EXAMPLES:
    sage: from sage.categories.graphs import Graphs
    sage: C = Graphs().example()
    sage: C.edges()
    [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]

faces()
    Return the faces of self.

    EXAMPLES:
    sage: from sage.categories.graphs import Graphs
    sage: C = Graphs().example()
    sage: sorted(C.faces(), key=lambda x: (x.dimension(), x.value))
    [0, 1, 2, 3, 4, (0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]

facets()
    Return the facets of self.

    EXAMPLES:
    sage: from sage.categories.graphs import Graphs
    sage: C = Graphs().example()
    sage: C.facets()
    [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]

vertices()
    Return the vertices of self.

    EXAMPLES:
    sage: from sage.categories.graphs import Graphs
    sage: C = Graphs().example()
    sage: C.vertices()
    [0, 1, 2, 3, 4]

super_categories()
    EXAMPLES:
    sage: from sage.categories.graphs import Graphs
    sage: Graphs().super_categories()
    [Category of simplicial complexes]

3.85 Group Algebras

This module implements the category of group algebras for arbitrary groups over arbitrary commutative rings. For details, see sage.categories.algebra_functor.

AUTHOR:

- David Loeffler (2008-08-24): initial version
- Martin Raum (2009-08): update to use new coercion model – see trac ticket #6670.
class sage.categories.group_algebras.GroupAlgebras(category, *args)

Bases: sage.categories.algebra_functor.AlgebrasCategory

The category of group algebras over a given base ring.

EXAMPLES:

```python
sage: C = Groups().Algebras(ZZ); C
Category of group algebras over Integer Ring
sage: C.super_categories()
[Category of hopf algebras with basis over Integer Ring,
 Category of monoid algebras over Integer Ring]
```

We can also construct this category with:

```python
sage: C is GroupAlgebras(ZZ)
True
```

Here is how to create the group algebra of a group \(G\):

```python
sage: G = DihedralGroup(5)
sage: QG = G.algebra(QQ); QG
Algebra of Dihedral group of order 10 as a permutation group over Rational Field
```

and an example of computation:

```python
sage: g = G.an_element(); g
(1,4)(2,3)
sage: (QG.term(g) + 1)**3
4*() + 4*(1,4)(2,3)
```

Todo:

- Check which methods would be better located in `Monoid.Algebras` or `Groups.Finite.Algebras`.

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:

```python
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)]
```

See also:
• `Groups.Algebras.ParentMethods.center_basis()`
• `Monoids.Algebras.ElementMethods.is_central()`

class ParentMethods

`antipode_on_basis(g)`

Return the antipode of the element `g` of the basis.

Each basis element `g` is group-like, and so has antipode `g^{-1}`. This method is used to compute the antipode of any element.

EXAMPLES:

```python
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
Algebra of Cyclic group of order 6 as a permutation group over Integer
        Ring
sage: g = CyclicPermutationGroup(6).an_element(); g
(1,2,3,4,5,6)
sage: A.antipode_on_basis(g)
(1,6,5,4,3,2)
```

`center_basis()`

Return a basis of the center of the group algebra.

3.85. Group Algebras
The canonical basis of the center of the group algebra is the family \((f_\sigma)_{\sigma \in C}\), where \(C\) is any collection of representatives of the conjugacy classes of the group, and \(f_\sigma\) is the sum of the elements in the conjugacy class of \(\sigma\).

**OUTPUT:**
- tuple of elements of self

**Warning:**
- This method requires the underlying group to have a method `conjugacy_classes` (every permutation group has one, thanks GAP!).

**EXAMPLES:**

```sage
sage: SymmetricGroup(3).algebra(QQ).center_basis()
(() , (2,3) + (1,2) + (1,3), (1,2,3) + (1,3,2))
```

**See also:**
- `Monoids.Algebras.ElementMethods.is_central()`

**coproduct_on_basis**\((g)\)

Return the coproduct of the element \(g\) of the basis.

Each basis element \(g\) is group-like. This method is used to compute the coproduct of any element.

**EXAMPLES:**

```sage
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
Algebra of Cyclic group of order 6 as a permutation group over Integer
˓→ Ring
sage: g = CyclicPermutationGroup(6).an_element(); g
(1,2,3,4,5,6)
sage: A.coproduct_on_basis(g)
(1,2,3,4,5,6) # (1,2,3,4,5,6)
sage: a = A.an_element(); a
() + (1,2,3,4,5,6) + 3* (1,3,5) (2,4,6) + 2* (1,5,3) (2,6,4)
sage: a.coproduct()
() # () + (1,2,3,4,5,6) # (1,2,3,4,5,6) + 3* (1,3,5) (2,4,6) # (1,3,5) (2,4,6) + 2* (1,5,3) (2,6,4) # (1,5,3) (2,6,4)
```

**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
sage: A = CyclicPermutationGroup(6).algebra(ZZ); A
Algebra of Cyclic group of order 6 as a permutation group over Integer
˓→ Ring
sage: a = A.an_element(); a
() + (1,2,3,4,5,6) + 3* (1,3,5) (2,4,6) + 2* (1,5,3) (2,6,4)
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:**

```python
sage: A=CyclicPermutationGroup(6).algebra(ZZ);A
Algebra of Cyclic group of order 6 as a permutation group over Integer
˓→Ring
sage: g=CyclicPermutationGroup(6).an_element();g
(1,2,3,4,5,6)
sage: A.counit_on_basis(g)
1
```

**group()**

Return the underlying group of the group algebra.

**EXAMPLES:**

```python
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
```

**is_integral_domain**(proof=True)

Return True if self is an integral domain.

This is false unless self.base_ring() is an integral domain, and even then it is false unless self.group() has no nontrivial elements of finite order. I don’t know if this condition suffices, but it obviously does if the group is abelian and finitely generated.

**EXAMPLES:**

```python
sage: GroupAlgebra(SymmetricGroup(2)).is_integral_domain()
False
sage: GroupAlgebra(SymmetricGroup(1)).is_integral_domain()
True
sage: GroupAlgebra(SymmetricGroup(1), IntegerModRing(4)).is_integral_˓→domain()
False
sage: GroupAlgebra(AbelianGroup(1)).is_integral_domain()
True
sage: GroupAlgebra(AbelianGroup(2, [0,2])).is_integral_domain()
False
sage: GroupAlgebra(GL(2, ZZ)).is_integral_domain() # not implemented
False
```

**example**(G=None)

Return an example of group algebra.

**EXAMPLES:**

```python
sage: GroupAlgebras(QQ['x']).example()
Algebra of Dihedral group of order 8 as a permutation group over Univariate
˓→Polynomial Ring in x over Rational Field
```

An other group can be specified as optional argument:

```python
sage: GroupAlgebras(QQ).example(AlternatingGroup(4))
Algebra of Alternating group of order 4!/2 as a permutation group over ˓→Rational Field
```
extra_super_categories()
Implement the fact that the algebra of a group is a Hopf algebra.

EXAMPLES:

```
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]
```

3.86 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 Wikipedia article Groupoid
- Should Groupoid inherit from Category_over_base?

EXAMPLES:

```
sage: Groupoid(DihedralGroup(3))
Groupoid with underlying set Dihedral group of order 6 as a permutation group
```

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]
```

3.87 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]

Algebras
alias of sage.categories.group_algebras.GroupAlgebras

class 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 ElementMethods

    multiplicative_order()
    Return the multiplicative order of this element.

    EXAMPLES:

    sage: G1 = SymmetricGroup(3)
sage: G2 = SL(2,3)
sage: G = cartesian_product([G1,G2])
sage: G((G1.gen(0), G2.gen(1))).multiplicative_order()
12

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 (...)}

3.87. Groups 451
order()
Return the cardinality of self.

EXAMPLES:

```python
sage: C = cartesian_product([SymmetricGroup(10), SL(2,GF(3))])
sage: C.order()
87091200
```

Todo: this method is just here to prevent FiniteGroups.ParentMethods to call _cardinality_from_iterator.

eextra_super_categories()
A Cartesian product of groups is endowed with a natural group structure.

EXAMPLES:

```python
sage: C = Groups().CartesianProducts()
sage: C.extra_super_categories()
[Category of groups]
sage: sorted(C.super_categories(), key=str)
[Category of Cartesian products of inverse unital magmas,
 Category of Cartesian products of monoids,
 Category of groups]
```

class Commutative(base_category)
Bases: 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:

```python
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 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

**Finite**

alias of **sage.categories.finite_groups.FiniteGroups**

**Lie**

alias of **sage.categories.lie_groups.LieGroups**

class 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()  
(() , (5,6,7), ... , (1,4,2,3)(5,7))
sage: T
+------------------------
a| a b c d e f g h i j k
b| b c a e f d i g h l j
u| c a b f d e h i g k l j
d| d e f a b c j k l g h
i| e f d b c a l j k i g h
j| f d e c a b k l j h i g
k| g h i j k l d e f a b c
l| h i g k l j f d e c a b
sage: M = SL(2, 2)
sage: M.cayley_table()  
+------------
a| c e a f b d
b| d f b e a c
c| a b c d e f
d| b a d c f e
e| f d e b c a
f| e c f a d b
sage: A = AbelianGroup([2, 3])
sage: A.cayley_table()  
+------------
a| a b c d e f
e| a b c d e f
b| b a d c f e
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.

```python
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
```

```python
sage: G=QuaternionGroup()
sage: names=['1', 'I', '-1', '-I', 'J', '-K', '-J', 'K']
sage: G.cayley_table(names=names)
+ 1 I -1 -I J -K -J K
|---|---|---|---|---|---|---|---|
1| 1 I -1 -I J -K -J K
I| I -1 -I 1 K J -K -J
-1| -1 -I 1 I -J K J -K
-1| -I 1 -1 -K -J K J
J| J -K -J K -1 -I 1 I
-1| -K -J K J I -1 -I 1
-1| -J K J -K 1 I -1 -I
K| K J -K -J -I 1 I -1
```

```python
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.”

```python
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.
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| a a a a a a a a a a a a
b| a a a a a a b b b b b
f| a a a a a a c c c c c
b| a b c b c b c a b a
i| a b c b c b c a b a
j| a b c b c b c a b a
k| a b c b c b c a b a
l| a b c b c b c a b a
```

```
sage: comm = [trans['a'], trans['b'], trans['c']]
sage: comm
[(), (5,6,7), (5,7,6)]
sage: P = G.cayley_table(elements=comm)
sage: P
* a b c
+-------
a| a b c
b| b c a
c| c a b
```

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 ticket #7555.

AUTHOR:
- Rob Beezer (2010-03-15)

`conjugacy_class(g)`
Return the conjugacy class of the element `g`.  

---

456 Chapter 3. Individual Categories
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()**

Return 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 \(G \rtimes_{id} \text{Aut}(G)\), where \(id\) is the identity function on \(\text{Aut}(G)\), 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 \(\rightarrow\) Rational Field not yet implemented
```

**monoid_generators()**

Return the generators of self as a monoid.

Let \(G\) be a group with generating set \(X\). In general, the generating set of \(G\) as a monoid is given by \(X \cup X^{-1}\), where \(X^{-1}\) is the set of inverses of \(X\). If \(G\) is a finite group, then the generating set as a monoid is \(X\).

EXAMPLES:

```
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))
(continues on next page)
```
class Topological(category, *args)
    Bases: sage.categories.topological_spaces.TopologicalSpacesCategory

Category of topological groups.

A topological group $G$ is a group which has a topology such that multiplication and taking inverses are continuous functions.

REFERENCES:

• Wikipedia article Topological_group

def example()  # doctest: +NORMALIZE_WHITESPACE

    EXAMPLES:

    sage: Groups().example()
    General Linear Group of degree 4 over Rational Field

def static free(index_set=None, names=None, **kwds)  # doctest: +NORMALIZE_WHITESPACE

    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()  # doctest: +NORMALIZE_WHITESPACE
    Free group indexed by Integer Ring
    sage: Groups().free(ZZ)  # doctest: +NORMALIZE_WHITESPACE
    Free group indexed by Integer Ring
    sage: Groups().free(5)  # doctest: +NORMALIZE_WHITESPACE
    Free Group on generators {x0, x1, x2, x3, x4}
    sage: F.<x,y,z> = Groups().free(); F  # doctest: +NORMALIZE_WHITESPACE
    Free Group on generators {x, y, z}

3.88 Hecke modules

class sage.categories.hecke_modules.HeckeModules(R)  # doctest: +NORMALIZE_WHITESPACE
    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 $\mathbb{Q}$:

```python
sage: C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
```

TODO: check that this is what we want:

```python
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:

```python
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:

```python
sage: HeckeModules(PolynomialRing(IntegerRing(), 'x'))
Category of Hecke modules over Univariate Polynomial Ring in x over Integer Ring
```

### 3.89 Highest Weight Crystals

```python
class sage.categories.highest_weight_crystals.HighestWeightCrystalHomset(X, Y, category=None)

Bases: sage.categories.crystals.CrystalHomset
```

3.89. Highest Weight Crystals 459
The set of crystal morphisms from a highest weight crystal to another crystal.

See also:

See `sage.categories.crystals.CrystalHomset` for more information.

**Element**

alias of `HighestWeightCrystalMorphism`

class `sage.categories.highest_weight_crystals.HighestWeightCrystalMorphism`(
    parent,
    on_gens,
    cartan_type=None,
    virtualization=None,
    scaling_factors=None,
    gens=None,
    check=True)

Bases: `sage.categories.crystals.CrystalMorphismByGenerators`

A virtual crystal morphism whose domain is a highest weight crystal.

INPUT:

- `parent` – a homset
- `on_gens` – a function or list that determines the image of the generators (if given a list, then this uses the order of the generators of the domain) of the domain under `self`
- `cartan_type` – (optional) a Cartan type; the default is the Cartan type of the domain
- `virtualization` – (optional) a dictionary whose keys are in the index set of the domain and whose values are lists of entries in the index set of the codomain
- `scaling_factors` – (optional) a dictionary whose keys are in the index set of the domain and whose values are scaling factors for the weight, 𝝀 and 𝜙
- `gens` – (optional) a list of generators to define the morphism; the default is to use the highest weight vectors of the crystal
- `check` – (default: True) check if the crystal morphism is valid

class `sage.categories.highest_weight_crystals.HighestWeightCrystals`(
s=None)

Bases: `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
```
class ElementMethods

**string_parameters**\(\text{word=\text{None}}\)

Return the string parameters of \texttt{self} corresponding to the reduced word \texttt{word}.

Given a reduced expression \(w = s_{i_1} \cdots s_{i_k}\), the string parameters of \(b \in B\) corresponding to \(w\) are \((a_1, \ldots, a_k)\) such that

\[
e_{i_m} \cdots e_{i_1} b \neq 0
\]
\[
e_{i_m+1} \cdots e_{i_1} b = 0
\]

for all \(1 \leq m \leq k\).

For connected components isomorphic to \(B(\lambda)\) or \(B(\infty)\), if \(w = w_0\) is the longest element of the Weyl group, then the path determined by the string parametrization terminates at the highest weight vector.

**INPUT:**
- \texttt{word} – a word in the alphabet of the index set; if not specified and we are in finite type, then this will be some reduced expression for the long element determined by the Weyl group

**EXAMPLES:**

```python
sage: B = crystals.infinity.NakajimaMonomials(['A',3])
sage: mg = B.highest_weight_vector()
sage: w0 = [1,2,1,3,2,1]
sage: mg.string_parameters(w0)
[0, 0, 0, 0, 0, 0]
sage: mg.f_string([1]).string_parameters(w0)
[1, 0, 0, 0, 0, 0]
sage: mg.f_string([1,1]).string_parameters(w0)
[3, 0, 0, 0, 0, 0]
sage: mg.f_string([1,1,2]).string_parameters(w0)
[1, 2, 2, 0, 0, 0]
sage: mg.f_string([1,1,2,1]) == mg.f_string([1,1,2,1])
True
sage: x = mg.f_string([1,1,1,2,2,1,3,3,2,1,1,1])
sage: x.string_parameters(w0)
[0, 0, 0, 0, 0, 0]
sage: x == mg.f_string([1]*7 + [2]*3 + [3]*2)
True
```

```python
sage: B = crystals.infinity.Tableaux("A5")
sage: b = B(rows=[1,1,1,1,1,1,1,1,1,1,1,1,1,3,6,6,6,6,6,6],
....: [2,2,2,2,2,2,2,2,2,4,5,5,5,6],
....: [3,3,3,3,3,3,3,5],
....: [4,4,4,6,6,6],
....: [5,6])
sage: b.string_parameters([1,2,1,3,2,1,4,3,2,1,5,4,3,2,1])
[0, 1, 1, 1, 1, 0, 4, 4, 3, 0, 11, 10, 7, 7, 6]
sage: B = crystals.infinity.Tableaux("G2")
sage: b = B(rows=[1,1,1,1,1,1,3,3,0,-3,-3,-2,-2,-1,-1,-1,-1,[2,3,3,3]])
sage: b.string_parameters([2,1,2,1,2,1])
[5, 13, 11, 15, 4, 4]
sage: b.string_parameters([1,2,1,2,1,2])
[7, 12, 15, 8, 10, 0]
```
class ParentMethods

connected_components_generators()

Returns the highest weight vectors of self

This default implementation selects among the module generators those that are highest weight, and caches the result. A crystal element $b$ is highest weight if $e_i(b) = 0$ for all $i$ in the index set.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C.highest_weight_vectors()
(1,)
```

```python
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])
```

digraph(subset=None, index_set=None, depth=None)

Return the DiGraph associated to self.

INPUT:

- subset – (optional) a subset of vertices for which the digraph should be constructed
- index_set – (optional) the index set to draw arrows
- depth – the depth to draw; optional only for finite crystals

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: T.digraph()
Digraph on 8 vertices
sage: S = T.subcrystal(max_depth=2)
sage: len(S)
5
sage: G = T.digraph(subset=list(S))
sage: G.is_isomorphic(T.digraph(depth=2), edge_labels=True)
True
```

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:
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:

```python
sage: C = crystals.Letters(['A',5])
sage: C.highest_weight_vectors()
(1,)
sage: C = crystals.Letters(['A',2])
sage: T = crystals.TensorProduct(C,C,C,generators=[[C(2),C(1),C(1)],[C(1),-C(2),C(1)]])
sage: T.highest_weight_vectors()
([2, 1, 1], [1, 2, 1])
```

lowest_weight_vectors()  
Return the lowest weight vectors of self.  
This default implementation selects among all elements of the crystal those that are lowest weight, and cache the result. A crystal element \( b \) is lowest weight if \( f_i(b) = 0 \) for all \( i \) in the index set.  
EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C.lowest_weight_vectors()
(6,)
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 \( \mu \)-weight space of \( B(\lambda) \) (under the principal gradation) is equal to \( \langle \rho^\vee, \lambda - \mu \rangle \) where \( \langle \rho^\vee, \alpha_i \rangle = 1 \) for all \( i \in I \) (in particular, take \( \rho^\vee = \sum_{i \in I} h_i \)).  
The \( q \)-dimension of a highest weight crystal \( B(\lambda) \) is defined as

\[
\dim_q B(\lambda) := \sum_{j \geq 0} \dim(B_j) q^j,
\]

where \( B_j \) denotes the degree \( j \) portion of \( B(\lambda) \). This can be expressed as the product

\[
\dim_q B(\lambda) = \prod_{\alpha^\vee \in \Delta^+_\Lambda} \left( \frac{1 - q^{\langle \lambda + \rho, \alpha^\vee \rangle}}{1 - q^{\langle \rho, \alpha^\vee \rangle}} \right)^{\text{mult} \alpha},
\]

where \( \Delta^+_\Lambda \) denotes the set of positive coroots. Taking the limit as \( q \to 1 \) gives the dimension of \( B(\lambda) \). For more information, see [Ka1990] Section 10.10.
INPUT:

- $q$ – the (generic) parameter $q$
- $\text{prec} – \text{(default: None)}$ The precision of the power series ring to use if the crystal is not known to be finite (i.e. the number of terms returned). If None, then the result is returned as a lazy power series.
- $\text{use\_product} – \text{(default: False)}$ 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: qdim = C.q_dimension(); qdim
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: qdim == C.q_dimension(use_product=True)
True
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: qdim = C.q_dimension(); qdim
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
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
```

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
```

(continues on next page)
The $q$-dimensions of infinite crystals are returned as formal power series:

```python
sage: C = crystals.LSPaths(['A',2,1], [1,0,0])
sage: C.q_dimension(prec=5)
1 + q + 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)
```

### Class TensorProducts

```python
class 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

        extra_super_categories()
        EXAMPLES:
            sage: HighestWeightCrystals().TensorProducts().extra_super_categories()
            [Category of highest weight crystals]

        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()
```

element()

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
```

super_categories()

EXAMPLES:

```
sage: HighestWeightCrystals().super_categories()
[Category of crystals]
```

### 3.90 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]
```

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 ElementMethods

antipode()

Return 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 6 as a permutation group over Rational Field
sage: [a,b] = A.algebra_generators()
```

(continues on next page)
class Morphism($s=None$
Bases: sage.categories.category.Category
The category of Hopf algebra morphisms
class ParentMethods
class Realizations(category, *args)
Bases: sage.categories.realizations.RealizationsCategory
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
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 Super(base_category)
Bases: sage.categories.super_modules.SuperModulesCategory
class TensorProducts(category, *args)
Bases: sage.categories.tensor.TensorProductsCategory
The category of Hopf algebras constructed by tensor product of Hopf algebras
class ElementMethods
class ParentMethods
extra_super_categories()
EXAMPLES:

```sage
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]
```

WithBasis
alias of sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis
dual()
Return the dual category
EXAMPLES:
The category of Hopf algebras over any field is self dual:

```
sage: C = HopfAlgebras(QQ)
sage: C.dual()
Category of hopf algebras over Rational Field
```

```
sage: HopfAlgebras(QQ).super_categories()
[Category of bialgebras over Rational Field]
```

### 3.91 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 bialgebras with basis over Rational Field]
```

We now show how to use a simple Hopf algebra, namely the group algebra of the dihedral group (see also `AlgebrasWithBasis`):

```
sage: A = C.example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral group of order 6 as a permutation group over Rational Field
sage: A.__custom_name = "A"
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
```

(continues on next page)
Let us look at the code for implementing A:

```python
sage: A?? # todo: not implemented

class ElementMethods
```

Let us look at the code for implementing A:
Filtered
alias of sage.categories.filtered_hopf_algebras_with_basis.
FilteredHopfAlgebrasWithBasis

FiniteDimensional
alias of sage.categories.finite_dimensional_hopf_algebras_with_basis.
FiniteDimensionalHopfAlgebrasWithBasis

Graded
alias of sage.categoriesgraded_hopf_algebras_with_basis.
GradedHopfAlgebrasWithBasis

class 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 6 as a permutation group over Integer Ring
sage: A = HopfAlgebrasWithBasis(QQ).example()

[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)])

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.gen(0); w
(1,2,3)
sage: A.antipode_on_basis(w)
B[(1,3,2)]

Super
alias of sage.categories.super_hopf_algebras_with_basis.
SuperHopfAlgebrasWithBasis

class 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 ParentMethods

extra_super_categories()

EXAMPLES:

```python
sage: C = HopfAlgebrasWithBasis(QQ).TensorProducts()
sage: C.extra_super_categories()
[Category of hopf algebras with basis over Rational Field]
sage: sorted(C.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]
```

element (G=None)

Returns an example of algebra with basis:

```python
sage: HopfAlgebrasWithBasis(QQ['x']).example()
An example of Hopf algebra with basis: the group algebra of the Dihedral
  group of order 6 as a permutation group over Univariate Polynomial Ring in
  x over Rational Field
```

An other group can be specified as optional argument:

```python
sage: HopfAlgebrasWithBasis(QQ).example(SymmetricGroup(4))
An example of Hopf algebra with basis: the group algebra of the Symmetric
  group of order 4! as a permutation group over Rational Field
```

3.92 H-trivial semigroups

class sage.categories.h_trivial_semigroups.HTrivialSemigroups (base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

Finite_extra_super_categories()

Implement the fact that a finite $H$-trivial is aperiodic

EXAMPLES:

```python
sage: Semigroups().HTrivial().Finite_extra_super_categories()
[Category of aperiodic semigroups]
sage: Semigroups().HTrivial().Finite() is Semigroups().Aperiodic().Finite()
True
```

Inverse_extra_super_categories()

Implement the fact that an $H$-trivial inverse semigroup is $J$-trivial.

Todo: Generalization for inverse semigroups.

Recall that there are two invertibility axioms for a semigroup $S$:

- One stating the existence, for all $x$, of a local inverse $y$ satisfying $x = xyx$ and $y = yxy$;
- One stating the existence, for all $x$, of a global inverse $y$ satisfying $xy = yx = 1$, where 1 is the unit of $S$ (which must of course exist).

It is sufficient to have local inverses for $H$-triviality to imply $J$-triviality. However, at this stage, only the second axiom is implemented in Sage (see `Magmas.Unital.SubcategoryMethods`).
Inverse()). Therefore this fact is only implemented for semigroups with global inverses, that is groups. However the trivial group is the unique $H$-trivial group, so this is rather boring.

EXAMPLES:

```
sage: Semigroups().HTrivial().Inverse_extra_super_categories()
[Category of j trivial semigroups]
sage: Monoids().HTrivial().Inverse()
Category of h trivial groups
```

### 3.93 Infinite Enumerated Sets

**AUTHORS:**


**class** `sage.categories.infiniteEnumeratedSets.InfiniteEnumeratedSets(base_category)`

```python
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]
```

**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):
  ... Not Implemented: cannot list an infinite set
```

**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?

3.94 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

class ElementMethods
class 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

3.95 J-trivial semigroups

class sage.categories.j_trivial_semigroups.JTrivialSemigroups (base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

    extra_super_categories()
      Implement the fact that a J-trivial semigroup is L and R-trivial.

    EXAMPLES:

sage: Semigroups().JTrivial().extra_super_categories()
[Category of l trivial semigroups, Category of r trivial semigroups]
3.96 Kac-Moody Algebras

AUTHORS:

- Travis Scrimshaw (07-15-2017): Initial implementation

class sage.categories.kac_moody_algebras.KacMoodyAlgebras(base, name=None)
Bases: sage.categories.category_types.Category_over_base_ring

Category of Kac-Moody algebras.

class ParentMethods

    cartan_type()
    Return the Cartan type of self.

    EXAMPLES:

    sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.cartan_type()
['A', 2]

    weyl_group()
    Return the Weyl group of self.

    EXAMPLES:

    sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.weyl_group()
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)

example(n=2)
    Return an example of a Kac-Moody algebra as per Category.example.

    EXAMPLES:

    sage: from sage.categories.kac_moody_algebras import KacMoodyAlgebras
dsage: KacMoodyAlgebras(QQ).example()
    Lie algebra of ['A', 2] in the Chevalley basis

    We can specify the rank of the example:

    sage: KacMoodyAlgebras(QQ).example(4)
    'Lie algebra of ['A', 4] in the Chevalley basis'

    super_categories()
    EXAMPLES:

    sage: from sage.categories.kac_moody_algebras import KacMoodyAlgebras
dsage: KacMoodyAlgebras(QQ).super_categories()                 [Category of Lie algebras over Rational Field]

3.97 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 \emph{join}) and a unique infimum (greatest lower bound; called their \emph{meet}).

**EXAMPLES:**

```python
sage: LatticePosets()
Category of lattice posets
sage: LatticePosets().super_categories()
[Category of posets]
sage: LatticePosets().example()
NotImplemented
```

See also:

\texttt{Posets, FiniteLatticePosets, LatticePoset()}

**Finite**

alias of \texttt{sage.categories.finite_lattice_posets.FiniteLatticePosets}

**class ParentMethods**

\texttt{join}(x, y)

Returns the join of \(x\) and \(y\) in this lattice

**INPUT:**

\begin{itemize}
  \item \(x, y\) – elements of \texttt{self}
\end{itemize}

**EXAMPLES:**

```python
sage: D = LatticePoset((divisors(60), attrcall("divides")))
sage: D.join(D(6), D(10))
30
```

\texttt{meet}(x, y)

Returns the meet of \(x\) and \(y\) in this lattice

**INPUT:**

\begin{itemize}
  \item \(x, y\) – elements of \texttt{self}
\end{itemize}

**EXAMPLES:**

```python
sage: D = LatticePoset((divisors(30), attrcall("divides")))
sage: D.meet(D(6), D(15))
3
```

\texttt{super_categories}()

Returns a list of the (immediate) super categories of \texttt{self}, as per \texttt{Category.super_categories()}.

**EXAMPLES:**

```python
sage: LatticePosets().super_categories()
[Category of posets]
```

### 3.98. Left modules

**class** \texttt{sage.categories.left_modules.LeftModules}(\texttt{base, name=None})

**Bases:** \texttt{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 multiplication by elements of the rng

EXAMPLES:

```python
sage: LeftModules(ZZ)
Category of left modules over Integer Ring
sage: LeftModules(ZZ).super_categories()
[Category of commutative additive groups]
```

```
class ElementMethods
class ParentMethods
super_categories()
```

```python
sage: LeftModules(QQ).super_categories()
[Category of commutative additive groups]
```

### 3.99 Lie Algebras

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

```python
class sage.categories.lie_algebras.LieAlgebras(base, name=None)
Bases: sage.categories.category_types.Category_over_base_ring
```

The category of Lie algebras.

EXAMPLES:

```python
sage: C = LieAlgebras(QQ); C
Category of Lie algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
[Category of vector spaces over Rational Field]
```

We construct a typical parent in this category, and do some computations with it:

```python
sage: A = C.example(); A
An example of a Lie algebra: the Lie algebra from the associative algebra Symmetric group algebra of order 3 over Rational Field generated by ([2, 1, 3], [2, 3, 1])

sage: A.category()
Category of Lie algebras over Rational Field

sage: A.base_ring()
Rational Field

sage: a,b = A.lie_algebra_generators()
sage: a.bracket(b)
-1*[2, 3, 1] + [1, 3, 2]
sage: b.bracket(2*a + b)
2*[2, 3, 1] - 2*[1, 3, 2]
```
Please see the source code of $A$ (with $A$?) for how to implement other Lie algebras.

**Todo:** Many of these tests should use Lie algebras that are not the minimal example and need to be added after trac ticket #16820 (and trac ticket #16823).

```python
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: x.bracket(y)
-[-1, 3, 2] + [3, 2, 1]
sage: x.bracket(0)
0
```

**class ElementMethods**

```python
bracket(rhs)
```

Return the Lie bracket $[\text{self}, \text{rhs}]$.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).example()
sage: x, y = L.lie_algebra_generators()
sage: x.bracket(y)
-[-1, 3, 2] + [3, 2, 1]
sage: x.bracket(0)
0
```

```python
exp(lie_group=None)
```

Return the exponential of $\text{self}$ in $\text{lie_group}$.

**INPUT:**

- `lie_group` – (optional) the Lie group to map into; If `lie_group` is not given, the Lie group associated to the parent Lie algebra of `self` is used.

**EXAMPLES:**

```python
sage: L.<X,Y,Z> = LieAlgebra(QQ, 2, step=2)
sage: g = (X + Y + Z).exp(); g
exp(X + Y + Z)
sage: h = X.exp(); h
exp(X)
sage: g.parent()  # Lie group G of Free Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: g.parent() == h.parent()  # True
True
```

The Lie group can be specified explicitly:

```python
sage: H = L.lie_group('H')
sage: k = Z.exp(lie_group=H); k
exp(Z)
sage: k.parent()  # Lie group H of Free Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: g.parent() == k.parent()  # False
False
```

```python
killing_form(x)
```

Return the Killing form of $\text{self}$ and $x$.

**EXAMPLES:**

3.99. Lie Algebras 477
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: a.killing_form(b)
0

**lift()**

Return the image of `self` under the canonical lift from the Lie algebra to its universal enveloping algebra.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 3*a + b - c
sage: elt.lift()
3*b0 + b1 - b2
```

```python
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: x.lift()
x
```

**to_vector()**

Return the vector in `g.module()` corresponding to the element `self` of `g` (where `g` is the parent of `self`).

Implement this if you implement `g.module()`. See `LieAlgebras.module()` for how this is to be done.

**EXAMPLES:**

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L((1, 0, 0)).to_vector(); u
(1, 0, 0)
sage: parent(u)
Vector space of dimension 3 over Rational Field
```

**class FiniteDimensional** *(base_category)*

**Bases:** `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

**WithBasis**

alias of `sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis`

**extra_super_categories()**

Implements the fact that a finite dimensional Lie algebra over a finite ring is finite.

**EXAMPLES:**

```python
sage: LieAlgebras(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
[Category of finite sets]
sage: LieAlgebras(ZZ).FiniteDimensional().extra_super_categories()
[]
sage: LieAlgebras(GF(5)).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: LieAlgebras(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
(continues on next page)```
Graded
alias of `sage.categories.graded_lie_algebras.GradedLieAlgebras`

class Nilpotent (base_category)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

Category of nilpotent Lie algebras.

class ParentMethods

    is_nilpotent()
    Return True since self is nilpotent.

    EXAMPLES:

        sage: h = lie_algebras.Heisenberg(ZZ, oo)
        sage: h.is_nilpotent()
        True

    step()
    Return the nilpotency step of self.

    EXAMPLES:

        sage: h = lie_algebras.Heisenberg(ZZ, oo)
        sage: h.step()
        2

class ParentMethods

    baker_campbell_hausdorff(X, Y, prec=None)
    Return the element log(exp(X) exp(Y)).

    The BCH formula is an expression for log(exp(X) exp(Y)) as a sum of Lie brackets of X and Y with rational coefficients. It is only defined if the base ring of self has a coercion from the rationals.

    INPUT:
    • X -- an element of self
    • Y -- an element of self
    • prec -- an integer; the maximum length of Lie brackets to be considered in the formula

    EXAMPLES:

    The BCH formula for the generators of a free nilpotent Lie algebra of step 4:

        sage: L = LieAlgebra(QQ, 2, step=4)
        sage: L.inject_variables()
        Defining X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222
        sage: L.bch(X_1, X_2)
        X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122

    An example of the BCH formula in a quotient:
The BCH formula for a non-nilpotent Lie algebra requires the precision to be explicitly stated:

```
sage: L.<X,Y> = LieAlgebra(QQ)
sage: L.bch(X, Y)
Traceback (most recent call last):
  ...  
ValueError: the Lie algebra is not known to be nilpotent, so you must specify the precision
sage: L.bch(X, Y, 4)
X + 1/12*[X, [X, Y]] + 1/24*[X, [[X, Y], Y]] + 1/2*[X, Y] + 1/12*[X, [Y, Y]] + Y
```

The BCH formula requires a coercion from the rationals:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2)
sage: L.bch(X, Y)
Traceback (most recent call last):
  ...  
TypeError: the BCH formula is not well defined since Integer Ring has no coercion from Rational Field
```

\[ \text{bch} \]  
\begin{align*}
\text{bch} \ (X, Y, \text{prec}=\text{None}) \\
\text{Return the element } \log(\exp(X) \exp(Y)).
\end{align*}

The BCH formula is an expression for \( \log(\exp(X) \exp(Y)) \) as a sum of Lie brackets of \( X \) and \( Y \) with rational coefficients. It is only defined if the base ring of \( \text{self} \) has a coercion from the rationals.

**INPUT:**
- \( X \) – an element of \( \text{self} \)
- \( Y \) – an element of \( \text{self} \)
- \( \text{prec} \) – an integer; the maximum length of Lie brackets to be considered in the formula

**EXAMPLES:**

The BCH formula for the generators of a free nilpotent Lie algebra of step 4:

```
sage: L = LieAlgebra(QQ, 2, step=4)
sage: L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222
sage: L.bch(X_1, X_2)
X_1 + X_2 + 1/2*X_12 + 1/12*X_112 + 1/12*X_122 + 1/24*X_1122
```

An example of the BCH formula in a quotient:

```
sage: Q = L.quotient(X_112 + X_122)
sage: x, y = Q.basis().list()[:2]
sage: Q.bch(x, y)
X_1 + X_2 + 1/2*X_12 - 1/24*X_1112
```

The BCH formula for a non-nilpotent Lie algebra requires the precision to be explicitly stated:

```
sage: L.<X,Y> = LieAlgebra(QQ)
sage: L.bch(X, Y)
(continues on next page)
Traceback (most recent call last):
...
ValueError: the Lie algebra is not known to be nilpotent, so you must specify the precision

\[
\text{sage: } L.\text{bch}(X, Y, 4) \\
X + 1/12*[X, [X, Y]] + 1/24*[X, [[X, Y], Y]] + 1/2*[X, Y] + 1/12*[X, Y]'
\]
\[
\rightarrow Y + Y
\]

The BCH formula requires a coercion from the rationals:

\[
\text{sage: } L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2) \\
\text{sage: } L.\text{bch}(X, Y) \\
\text{Traceback (most recent call last):} \\
... \\
\text{TypeError: the BCH formula is not well defined since Integer Ring has no coercion from Rational Field}
\]

\[\text{bracket}(\text{lhs}, \text{rhs})\]

Return the Lie bracket \([\text{lhs}, \text{rhs}]\) after coercing \text{lhs} and \text{rhs} into elements of \text{self}.

If \text{lhs} and \text{rhs} are Lie algebras, then this constructs the product space, and if only one of them is a Lie algebra, then it constructs the corresponding ideal.

\[\text{EXAMPLES:}\]

\[
\text{sage: } L = \text{lie_algebras.Heisenberg}(\text{QQ}, 1) \\
\text{sage: } Z = L.\text{bracket}(L, L); Z \\
\text{Ideal (z) of Heisenberg algebra of rank 1 over Rational Field} \\
\text{sage: } L.\text{bracket}(L, Z) \\
\text{Ideal () of Heisenberg algebra of rank 1 over Rational Field}
\]

Constructing the product space:

\[
\text{from_vector}(v) \\
\text{Return the element of self corresponding to the vector v in self.module().}
\]

Implement this if you implement \text{module()}; see the documentation of the latter for how this is to be done.

\[\text{EXAMPLES:}\]
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
sage: parent(u)  is L
True

ideal(*gens, **kwds)
Return the ideal of self generated by gens.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.ideal([2*a - c, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field
with basis matrix:
[ 1 0 -1/2]
[ 0 1  1]

is_abelian()
Return True if this Lie algebra is abelian.
A Lie algebra \( \mathfrak{g} \) is abelian if \([x, y] = 0\) for all \( x, y \in \mathfrak{g}\).

EXAMPLES:

sage: L = LieAlgebras(QQ).example()
sage: L.is_abelian()
False
sage: R = QQ['x,y']
sage: L = LieAlgebras(QQ).example(R.gens())
sage: L.is_abelian()
True
sage: L.<x> = LieAlgebra(QQ,1)  # todo: not implemented - #16823
sage: L.is_abelian()             # todo: not implemented - #16823
True
sage: L.<x,y> = LieAlgebra(QQ,2) # todo: not implemented - #16823
sage: L.is_abelian()             # todo: not implemented - #16823
False

is_commutative()
Return if self is commutative. This is equivalent to self being abelian.

EXAMPLES:

sage: L = LieAlgebras(QQ).example()
sage: L.is_commutative()
False
sage: L.<x> = LieAlgebra(QQ, 1) # todo: not implemented - #16823
sage: L.is_commutative() # todo: not implemented - #16823
True

is_ideal(A)
Return if self is an ideal of A.

EXAMPLES:

sage: L = LieAlgebras(QQ).example()
sage: L.is_ideal(L)
True

is_nilpotent()
Return if self is a nilpotent Lie algebra.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_nilpotent()
True

is_solvable()
Return if self is a solvable Lie algebra.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.is_solvable()
True

killing_form(x, y)
Return the Killing form of x and y.

EXAMPLES:

sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.killing_form(a, b+c)
0

lie_group(name='G', **kwds)
Return the simply connected Lie group related to self.

INPUT:
• name - string (default: 'G'); the name (symbol) given to the Lie group

EXAMPLES:

sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: G = L.lie_group('G'); G
Lie group G of Heisenberg algebra of rank 1 over Rational Field

lift()
Construct the lift morphism from self to the universal enveloping algebra of self (the latter is implemented as universal_enveloping_algebra()).

This is a Lie algebra homomorphism. It is injective if self is a free module over its base ring, or if the base ring is a \mathbb{Q}-algebra.

EXAMPLES:
module()

Return an $R$-module which is isomorphic to the underlying $R$-module of self.

The rationale behind this method is to enable linear algebraic functionality on self (such as computing the span of a list of vectors in self) via an isomorphism from self to an $R$-module (typically, although not always, an $R$-module of the form $R^n$ for an $n \in \mathbb{N}$) on which such functionality already exists. For this method to be of any use, it should return an $R$-module which has linear algebraic functionality that self does not have.

For instance, if self has ordered basis $(e, f, h)$, then self.module() will be the $R$-module $R^3$, and the elements $e, f$ and $h$ of self will correspond to the basis vectors $(1, 0, 0), (0, 1, 0)$ and $(0, 0, 1)$ of self.module().

This method module() needs to be set whenever a finite-dimensional Lie algebra with basis is intended to support linear algebra (which is, e.g., used in the computation of centralizers and lower central series). One then needs to also implement the $R$-module isomorphism from self to self.module() in both directions; that is, implement:

- a to_vector ElementMethod which sends every element of self to the corresponding element of self.module();
- a from_vector ParentMethod which sends every element of self.module() to an element of self.

The from_vector method will automatically serve as an element constructor of self (that is, self(v) for any v in self.module() will return self.from_vector(v)).

Todo: Ensure that this is actually so.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field
```

```
sage: L = LieAlgebras(QQ).example()
sage: x,y = L.lie_algebra_generators()
```

(subalgebra (gens, names=None, index_set=None, category=None)

Return the subalgebra of self generated by gens.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([2*a - c, b + c])
An example of a finite dimensional Lie algebra with basis: the 2-dimensional abelian Lie algebra over Rational Field with basis matrix:
   [ 1  0 -1/2]
   [ 0  1   1]
```

(continues on next page)
universal_enveloping_algebra()  
Return the universal enveloping algebra of self.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.universal_enveloping_algebra()
Noncommutative Multivariate Polynomial Ring in b0, b1, b2
over Rational Field, nc-relations: {}
```

```
sage: L = LieAlgebra(QQ, 3, 'x', abelian=True)
sage: L.universal_enveloping_algebra()
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

See also:

* lift()

class SubcategoryMethods

Nilpotent()  
Return the full subcategory of nilpotent objects of self.

A Lie algebra \( L \) is nilpotent if there exist an integer \( s \) such that all iterated brackets of \( L \) of length more than \( s \) vanish. The integer \( s \) is called the nilpotency step. For instance any abelian Lie algebra is nilpotent of step 1.

EXAMPLES:

```
sage: LieAlgebras(QQ).Nilpotent()
Category of nilpotent Lie algebras over Rational Field
```

```
sage: LieAlgebras(QQ).WithBasis().Nilpotent()
Category of nilpotent lie algebras with basis over Rational Field
```

WithBasis  
alias of `sage.categories.lie_algebras_with BASIS.LieAlgebrasWithBasis`

def example(gens=None)

Return an example of a Lie algebra as per `Category.example`.

EXAMPLES:

```
sage: LieAlgebras(QQ).example()
An example of a lie algebra: the Lie algebra from the associative algebra
Symmetric group algebra of order 3 over Rational Field
generated by [(2, 1, 3), (2, 3, 1)]
```

Another set of generators can be specified as an optional argument:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: LieAlgebras(QQ).example(F.gens())
An example of a lie algebra: the Lie algebra from the associative algebra
```

(continues on next page)
Free Algebra on 3 generators \((x, y, z)\) over Rational Field generated by \((x, y, z)\)

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

class `sage.categories.lie_algebras.LiftMorphism`(domain, codomain)
Bases: `sage.categories.morphism.Morphism`

The natural lifting morphism from a Lie algebra to its enveloping algebra.

### 3.100 Lie Algebras With Basis

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

```python
class sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

Category of Lie algebras with a basis.

class ElementMethods

`lift()`
Lift `self` to the universal enveloping algebra.

EXAMPLES:

```python
sage: S = SymmetricGroup(3).algebra(QQ)
sage: L = LieAlgebra(associative=S)
sage: x = L.gen(3)
sage: y = L.gen(1)
sage: x.lift()
b3
sage: y.lift()
b1
sage: x * y
b1*b3 + b4 - b5
```

`to_vector()`
Return the vector in `g.module()` corresponding to the element `self` of `g` (where `g` is the parent of `self`).

Implement this if you implement `g.module()`. See `sage.categories.lie_algebras.LieAlgebras.module()` for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.an_element().to_vector()
(0, 0, 0)
```
Todo: Doctest this implementation on an example not overshadowed.

Graded

alias of sage.categories.graded_lie_algebras_with_basis.
GradedLieAlgebrasWithBasis

class ParentMethods

bracket_on_basis(x, y)

Return the bracket of basis elements indexed by x and y where x < y. If this is not implemented, then the method _bracket_() for the elements must be overwritten.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.bracket_on_basis(Partition([3,1]), Partition([2,2,1,1]))
0
```

dimension()

Return the dimension of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.dimension()
3
```

from_vector(v)

Return the element of self corresponding to the vector v in self.module().

Implement this if you implement module(); see the documentation of sage.categories.lie_algebras.LieAlgebras.module() for how this is to be done.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
sage: parent(u) is L
True
```

module()

Return an $R$-module which is isomorphic to the underlying $R$-module of self.

See sage.categories.lie_algebras.LieAlgebras.module() for an explanation.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.module()
Free module generated by Partitions over Rational Field
```
**pbw_basis** *(basis_key=None, **kwds)*

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to *self*.

**EXAMPLES:**

```python
sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
```

**poincare_birkhoff_witt_basis** *(basis_key=None, **kwds)*

Return the Poincare-Birkhoff-Witt basis of the universal enveloping algebra corresponding to *self*.

**EXAMPLES:**

```python
sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
```

**example** *(gens=None)*

Return an example of a Lie algebra as per *Category.example*.

**EXAMPLES:**

```python
sage: LieAlgebras(QQ).WithBasis().example()
An example of a Lie algebra: the abelian Lie algebra on the generators indexed by Partitions over Rational Field

sage: LieAlgebras(QQ).WithBasis().example(Compositions())
An example of a Lie algebra: the abelian Lie algebra on the generators indexed by Compositions of non-negative integers over Rational Field
```

## 3.101 Lie Groups

**class** `sage.categories.lie_groups.LieGroups` *(base, name=None)*

The category of Lie groups.

A Lie group is a topological group with a smooth manifold structure.

**EXAMPLES:**

```python
sage: from sage.categories.lie_groups import LieGroups
sage: C = LieGroups(QQ); C
Category of Lie groups over Rational Field
```

**additional_structure** *

Return None.

Indeed, the category of Lie groups defines no new structure: a morphism of topological spaces and of smooth manifolds is a morphism as Lie groups.

**See also:**

*Category.additional_structure()*

**EXAMPLES:**
class sage.categories.loop_crystals.KirillovReshetikhinCrystals(s=None)
    Bases: sage.categories.category_singleton.Category_singleton

Category of Kirillov-Reshetikhin crystals.

class ElementMethods

energy_function()

Return the energy function of self.

Let $B$ be a KR crystal. Let $b^\varphi$ denote the unique element such that $\varphi(b^\varphi) = \ell \Lambda_0$ with $\ell = \min \{ \langle c, \varphi(b) \rangle \mid b \in B \}$. Let $u_B$ denote the maximal element of $B$. The energy of $b \in B$ is given by

$$D(b) = H(b \otimes b^\varphi) - H(u_B \otimes b^\varphi),$$

where $H$ is the local energy function.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: for x in K.classically_highest_weight_vectors():
    ....: x, x.energy_function()
([[]], 1)
([[1], [2]], 0)
sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,2)
sage: for x in K.classically_highest_weight_vectors():
    ....: x, x.energy_function()
([[]], 2)
([[1]], 1)
([[1, 1]], 0)
```

lusztig_involution()

Return the result of the classical Lusztig involution on self.

EXAMPLES:

```
sage: KRT = crystals.KirillovReshetikhin(['D',4,1], 2, 3, model='KR')
sage: mg = KRT.module_generators[1]
sage: mg.lusztig_involution()
[[[-2, -2, 1], [-1, -1, 2]]
sage: elt = mg.f_string([2,1,3,2]); elt
```

(continues on next page)
class ParentMethods

\texttt{R\_matrix}\,(K)

Return the combinatorial $R$-matrix of \texttt{self} to $K$.

The \textit{combinatorial $R$-matrix} is the affine crystal isomorphism $R : L \otimes K \to K \otimes L$ which maps $u_L \otimes u_K$ to $u_K \otimes u_L$, where $u_K$ is the unique element in $K = B^{r,s}$ of weight $s\Lambda_r - sc\Lambda_0$ (see \texttt{maximal\_vector()}).

INPUT:

- \texttt{self} -- a crystal $L$
- $K$ -- a Kirillov-Reshetikhin crystal of the same type as $L$

EXAMPLES:

\begin{verbatim}
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: L = crystals.KirillovReshetikhin(['A',2,1],1,2)
sage: f = K.R_matrix(L)
sage: [[b,f(b)] for b in crystals.TensorProduct(K,L)]
[[[[[1]], [[1, 1]]], [[[1, 1]], [[1]]]],
 [[[1]], [[1, 2]]], [[[1, 1]], [[2]]]],
 [[[1]], [[2, 2]]], [[[1, 2]], [[2]]]],
 [[[1]], [[1, 3]]], [[[1, 1]], [[3]]]],
 [[[1]], [[2, 3]]], [[[1, 2]], [[3]]]],
 [[[1]], [[3, 3]]], [[[1, 3]], [[3]]]],
 [[[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]]]],
 [[[1]], [[2, 3]]], [[[1, 2]], [[3]]]],
 [[[1]], [[3, 3]]], [[[1, 3]], [[3]]]]]

sage: K1 = crystals.KirillovReshetikhin(['D',4,1],1,1)
sage: K2 = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: T1 = crystals.TensorProduct(K1,K2)
sage: T2 = crystals.TensorProduct(K2,K1)
\end{verbatim}

Alternatively, one can compute the combinatorial $R$-matrix using the isomorphism method of digraphs:

\begin{verbatim}
sage: K1 = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: K2 = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: T1 = crystals.TensorProduct(K1,K2)
sage: T2 = crystals.TensorProduct(K2,K1)
\end{verbatim}
sage: T1.digraph().is_isomorphic(T2.digraph(), edge_labels=True, certificate=True) # todo: not implemented (see #10904 and #10549)
(True, [[[1]], [[2], [3]]]: [[[1], [3]], [[2]]], [[[3]], [[2], [3]]]: [[],
-[[[2], [3]], [[3]]],
-[[[3]], [[1], [3]]]: [[[1], [3]], [[3]]], [[[1]], [[1], [3]]]: [[1],
-[[3]], [[1]]], [[[1]],
-[[[1], [2]]]: [[[1], [2]], [[1]]], [[[2]], [[1], [2]]]: [[[1], [2]],
-[[[2]], [[3]]],
-[[[3]], [[2], [3]]]: [[], [2]], [[[2]], [[2], [3]]]: [[[2], [3]], [[2]]]})

affinization()

Return the corresponding affinization crystal of self.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.affinization()
Affinization of Kirillov-Reshetikhin crystal of type ['A', 2, 1] with (r, s)=(1,1)
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1, model='KR')
sage: K.affinization()
Affinization of Kirillov-Reshetikhin tableaux of type ['A', 2, 1] and shape (1, 1)

b_sharp()

Return the element $b^\sharp$ of self.

Let $B$ be a KR crystal. The element $b^\sharp$ is the unique element such that $\phi(b^\sharp) = \ell \Lambda_0$ with $\ell = \min\{c, \phi(b) | b \in B\}$.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,1)
sage: K.b_sharp()
[]
sage: K.b_sharp().Phi()
Lambda[0]
sage: K = crystals.KirillovReshetikhin(['C',3,1], 1,3)
sage: K.b_sharp()
[-1]
sage: K.b_sharp().Phi()
2*Lambda[0]
sage: K = crystals.KirillovReshetikhin(['D',6,2], 2,2)
sage: K.b_sharp() # long time
[]
sage: K.b_sharp().Phi() # long time
2*Lambda[0]

cardinality()

Return the cardinality of self.

EXAMPLES:
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1)
sage: K.cardinality()
27
sage: K = crystals.KirillovReshetikhin(['C',6,1], 4,3)
sage: K.cardinality()
4736732

classical_decomposition()
Return the classical decomposition of self.

EXAMPLES:
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[2, 2]]

classically_highest_weight_vectors()
Return the classically highest weight elements of self.

EXAMPLES:
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.classically_highest_weight_vectors()
([(1,)],)

is_perfect (ell=None)
Check if self is a perfect crystal of level ell.

A crystal $\mathcal{B}$ is perfect of level $\ell$ if:
1. $\mathcal{B}$ is isomorphic to the crystal graph of a finite-dimensional $U'_q(\mathfrak{g})$-module.
2. $\mathcal{B} \otimes \mathcal{B}$ is connected.
3. There exists a $\lambda \in X$ such that $\text{wt}(\mathcal{B}) \subset \lambda + \sum_{i \in I} Z \leq 0 \alpha_i$ and there is a unique element in $\mathcal{B}$ of classical weight $\lambda$.
4. For all $b \in \mathcal{B}$, $\text{level}(\varepsilon(b)) \geq \ell$.
5. For all $\Lambda$ dominant weights of level $\ell$, there exist unique elements $b_\Lambda, b^{\Lambda} \in \mathcal{B}$, such that $\varepsilon(b_\Lambda) = \Lambda = \varphi(b^{\Lambda})$.

Points (1)-(3) are known to hold. This method checks points (4) and (5).

If self is the Kirillov-Reshetikhin crystal $B_{r,s}$, then it was proven for non-exceptional types in [FOS2010] that it is perfect if and only if $s/c_r$ is an integer (where $c_r$ is a constant related to the type of the crystal).

It is conjectured this is true for all affine types.

INPUT:
• ell = (default: $s/c_r$) integer; the level

REFERENCES:
[FOS2010]

EXAMPLES:
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.is_perfect()
True
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: K.is_perfect()
False
Todo: Implement a version for tensor products of KR crystals.

level()

Return the level of self when self is a perfect crystal.

See also:

is_perfect()

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: K.level()
1
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: K.level()
Traceback (most recent call last):
  ... ValueError: this crystal is not perfect
```

local_energy_function(B)

Return the local energy function of self and B.

See LocalEnergyFunction for a definition.

EXAMPLES:

```
sage: K = crystals.KirillovReshetikhin(['A',6,2], 2,1)
sage: Kp = crystals.KirillovReshetikhin(['A',6,2], 1,1)
sage: H = K.local_energy_function(Kp); H
Local energy function of Kirillov-Reshetikhin crystal of type ['BC', 3, 2] with (r,s)=(2,1) tensor Kirillov-Reshetikhin crystal of type ['BC', 3, 2] with (r,s)=(1,1)
```

maximal_vector()

Return the unique element of classical weight \( s\Lambda_r \) in self.

EXAMPLES:
module_generator()

Return the unique module generator of classical weight \(s\Lambda_r\) of the Kirillov-Reshetikhin crystal \(B_{r,s}\).

EXAMPLES:

```python
sage: La = RootSystem(['G',2,1]).weight_space().fundamental_weights()
sage: K = crystals.ProjectedLevelZeroLSPaths(La[1])
sage: K.module_generator()
(-Lambda[0] + Lambda[1],)
```

q_dimension(q=None, prec=None, use_product=False)

Return the \(q\)-dimension of self.

The \(q\)-dimension of a KR crystal is defined as the \(q\)-dimension of the underlying classical crystal.

EXAMPLES:

```python
sage: KRC = crystals.KirillovReshetikhin(['A',2,1], 2,2)
sage: KRC.q_dimension()
q^4 + q^3 + 2*q^2 + q + 1
sage: KRC = crystals.KirillovReshetikhin(['D',4,1], 2,1)
sage: KRC.q_dimension()
q^10 + q^9 + 3*q^8 + 3*q^7 + 4*q^6 + 4*q^5 + 4*q^4 + 3*q^3 + 3*q^2 + q + 2
```

r()

Return the value \(r\) in self written as \(B_{r,s}\).

EXAMPLES:

```python
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,4)
sage: K.r()
2
```

s()

Return the value \(s\) in self written as \(B_{r,s}\).

EXAMPLES:

```python
sage: K = crystals.KirillovReshetikhin(['A',3,1], 2,4)
sage: K.s()
4
```

class TensorProducts(category, *args)

Bases: sage.categories.tensor.TensorProductsCategory

The category of tensor products of Kirillov-Reshetikhin crystals.
affine_grading()  
Return the affine grading of \texttt{self}.

The affine grading is calculated by finding a path from \texttt{self} to a ground state path (using the helper method \texttt{e_string_to_ground_state()}) and counting the number of affine Kashiwara operators \(e_0\) applied on the way.

**OUTPUT:** an integer

**EXAMPLES:**

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]
sage: t.affine_grading()  
1

sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:  
....:     print("{} "*len(b)).format(b, b.affine_grading()))
[[[1]], [[1]], [[1]]] 3
[[[2]], [[1]], [[1]]] 2
[[[1]], [[2]], [[1]]] 1
[[[3]], [[2]], [[1]]] 0
```

e_string_to_ground_state()  
Return a string of integers in the index set \((i_1, \ldots, i_k)\) such that \(e_{i_k} \cdots e_{i_1}\) of \texttt{self} is the ground state.

This method calculates a path from \texttt{self} to a ground state path using Demazure arrows as defined in Lemma 7.3 in [ST2011].

**OUTPUT:** a tuple of integers \((i_1, \ldots, i_k)\)

**EXAMPLES:**

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: t = T.module_generators[0]; t  
[[[1]], [[1]]]  
(continues on next page)
```
energy_function(algorithm=None)

Return the energy function of self.

ALGORITHM:

definition

Let $T$ be a tensor product of Kirillov-Reshetikhin crystals. Let $R_i$ and $H_i$ be the combinatorial $R$-matrix and local energy functions, respectively, acting on the $i$ and $i + 1$ factors. Let $D_B$ be the energy function of a single Kirillov-Reshetikhin crystal. The energy function is given by

$$D = \sum_{j>i} H_i R_{i+1} R_{i+2} \cdots R_{j-1} + \sum_{j} D_B R_1 R_2 \cdots R_{j-1},$$

where $D_B$ acts on the rightmost factor.

grading

If self is an element of $T$, a tensor product of perfect crystals of the same level, then use the affine grading to determine the energy. Specifically, let $g$ denote the affine grading of self and $d$ the affine grading of the maximal vector in $T$. Then the energy of self is given by $d - g$.

For more details, see Theorem 7.5 in [ST2011].

INPUT:

- algorithm – (default: None) use one of the following algorithms to determine the energy function:
  - 'definition' - use the definition of the energy function;
  - 'grading' - use the affine grading;

if not specified, then this uses 'grading' if all factors are perfect of the same level and otherwise this uses 'definition'

OUTPUT: an integer

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
.....:     print("{} {}\).format(b, b.energy_function()))
[[[1]], [[1]], [[1]]] 0
[[[2]], [[1]], [[1]]] 1
[[[1]], [[2]], [[1]]] 2
[[[3]], [[2]], [[1]]] 3

(continued from previous page)

sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 2)
sage: T = crystals.TensorProduct(K,K)
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
....:
print("{} {}".format(b, b.energy_function()))
[[], []] 4
[[[1, 1]], []] 3
[[], [[1, 1]]] 1
[[[1, 1]], [[1, 1]]] 0
[[[1, 2]], [[1, 1]]] 1
[[[2, 2]], [[1, 1]]] 2
[[[-1, -1]], [[1, 1]]] 2
[[[1, -1]], [[1, 1]]] 2
[[[2, -1]], [[1, 1]]] 2
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1, 1)
sage: T = crystals.TensorProduct(K)
sage: t = T.module_generators[0]
sage: t.energy_function('grading')
Traceback (most recent call last):
...
NotImplementedError: all crystals in the tensor product
need to be perfect of the same level

class ParentMethods
cardinality()
Return the cardinality of self.
EXAMPLES:
sage: RC = RiggedConfigurations(['A', 3, 1], [[3, 2], [1, 2]])
sage: RC.cardinality()
100
sage: len(RC.list())
100
sage: RC = RiggedConfigurations(['E', 7, 1], [[1,1]])
sage: RC.cardinality()
134
sage: len(RC.list())
134
sage: RC = RiggedConfigurations(['B', 3, 1], [[2,2],[1,2]])
sage: RC.cardinality()
5130

classically_highest_weight_vectors()
Return the classically highest weight elements of self.
This works by using a backtracking algorithm since if 𝑏2 ⊗ 𝑏1 is classically highest weight then
𝑏1 is classically highest weight.
EXAMPLES:

3.102. Loop Crystals

497


sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.classically_highest_weight_vectors()
([[[1]], [[1]], [[1]]],
 [[[2]], [[1]], [[1]]],
 [[[1]], [[2]], [[1]]],
 [[[3]], [[2]], [[1]]])

maximal_vector()

Return the maximal vector of self.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.maximal_vector()
([[[1]], [[1]], [[1]]])

one_dimensional_configuration_sum(q=None, group_components=True)

Compute the one-dimensional configuration sum of self.

INPUT:
• q - (default: None) a variable or None; if None, a variable q is set in the code
• group_components – (default: True) boolean; if True, then the terms are grouped by classical component

The one-dimensional configuration sum is the sum of the weights of all elements in the crystal weighted by the energy function.

EXAMPLES:

sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
B[-2Lambda[1] + 2Lambda[2]] + (q+1)*B[-Lambda[1]]
+ B[-2Lambda[2]] + (q+1)*B[Lambda[2]]
sage: R.<t> = ZZ[]
sage: T.one_dimensional_configuration_sum(t, False)
B[-2*Lambda[1] + 2*Lambda[2]] + (t+1)*B[-Lambda[1]]
+ B[-2*Lambda[2]] + (t+1)*B[Lambda[2]]
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: LS.one_dimensional_configuration_sum() == T.one_dimensional_configuration_sum()  # long time
True

extra_super_categories()

EXAMPLES:

sage: from sage.categories.loop_crystals import KirillovReshetikhinCrystals
sage: KirillovReshetikhinCrystals().TensorProducts().extra_super_categories()
[Category of finite regular loop crystals]
super_categories()

EXAMPLES:

```python
sage: from sage.categories.loop_crystals import KirillovReshetikhinCrystals
sage: KirillovReshetikhinCrystals().super_categories()
[Category of finite regular loop crystals]
```

```python
class sage.categories.loop_crystals.LocalEnergyFunction (B, Bp, normalization=0)
Bases: sage.categories.map.Map

The local energy function.

Let $B$ and $B'$ be Kirillov-Reshetikhin crystals with maximal vectors $u_B$ and $u_{B'}$ respectively. The local energy function $H : B \otimes B' \to \mathbb{Z}$ is the function which satisfies

$$H(e_0(b \otimes b')) = H(b \otimes b') + \begin{cases} 1 & \text{if } i = 0 \text{ and LL}, \\ -1 & \text{if } i = 0 \text{ and RR}, \\ 0 & \text{otherwise}, \end{cases}$$

where LL (resp. RR) denote $e_0$ acts on the left (resp. right) on both $b \otimes b'$ and $R(b \otimes b')$, and normalized by $H(u_B \otimes u_{B'}) = 0$.

INPUT:

- $B$ – a Kirillov-Reshetikhin crystal
- $Bp$ – a Kirillov-Reshetikhin crystal
- normalization – (default: 0) the normalization value

EXAMPLES:

```python
sage: K = crystals.KirillovReshetikhin(['C',2,1], 1,2)
sage: K2 = crystals.KirillovReshetikhin(['C',2,1], 2,1)
sage: H = K.local_energy_function(K2)
sage: T = tensor([K, K2])
sage: hw = T.classically_highest_weight_vectors()
sage: for b in hw:
    ....: b, H(b)
([[], [[1], [2]]], 1)
([[[1, 1]], [[1], [2]]], 0)
([[[2, -2]], [[1], [2]]], 1)
([[[1, -2]], [[1], [2]]], 1)
```

REFERENCES:

[KKMMNN1992]

```python
class sage.categories.loop_crystals.LoopCrystals (s=None)
Bases: sage.categories.category_singleton.Category_singleton

The category of $U'_q(g)$-crystals, where $g$ is of affine type.

The category is called loop crystals as we can also consider them as crystals corresponding to the loop algebra $g_0[t]$, where $g_0$ is the corresponding classical type.

EXAMPLES:

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: C = LoopCrystals()
sage: C
```
Category of loop crystals

```python
sage: C.super_categories()
[Category of crystals]
```

```python
sage: C.example()
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)
```

**class ParentMethods**

**digraph**(subset=None, index_set=None)

Return the DiGraph associated to self.

**INPUT:**

- subset – (optional) a subset of vertices for which the digraph should be constructed
- index_set – (optional) the index set to draw arrows

**See also:**

`sage.categories.crystals.Crystals.ParentMethods.digraph()`

**EXAMPLES:**

```python
sage: C = crystals.KirillovReshetikhin(['D',4,1], 2, 1)
sage: G = C.digraph()
sage: G.latex_options()  # optional - dot2tex
LaTeX options for Digraph on 29 vertices:
{...'edge_options': <function <lambda> at 0x...>,...}
sage: view(G, tightpage=True)  # optional - dot2tex graphviz, not tested
˓→(opens external window)
```

**weight_lattice_realization()**

Return the weight lattice realization used to express weights of elements in self.

The default is to use the non-extended affine weight lattice.

**EXAMPLES:**

```python
sage: C = crystals.Letters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
sage: K = crystals.KirillovReshetikhin(['A',2,1], 1, 1)
sage: K.weight_lattice_realization()
Weight lattice of the Root system of type ['A', 2, 1]
```

**example**(n=3)

Return an example of Kirillov-Reshetikhin crystals, as per `Category.example()`.

**EXAMPLES:**

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: B = LoopCrystals().example(); B
Kirillov-Reshetikhin crystal of type ['A', 3, 1] with (r,s)=(1,1)
```

**super_categories()**

**EXAMPLES:**

```python
sage: from sage.categories.loop_crystals import LoopCrystals
sage: LoopCrystals().super_categories()
[Category of crystals]
```
class sage.categories.loop_crystals.RegularLoopCrystals(s=None)

Bases: sage.categories.category_singleton.Category_singleton

The category of regular \( U'_q(g) \)-crystals, where \( g \) is of affine type.

class ElementMethods

classical_weight()

Return the classical weight of self.

EXAMPLES:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.ProjectedLevelZeroLSPaths(2*La[1])
sage: hw = LS.classically_highest_weight_vectors()
sage: [(v.weight(), v.classical_weight()) for v in hw]
[(-2*Lambda[0] + 2*Lambda[1], (2, 0, 0)),
 (-Lambda[0] + Lambda[2], (1, 1, 0))]
```

super_categories()

EXAMPLES:

```
sage: from sage.categories.loop_crystals import RegularLoopCrystals
sage: RegularLoopCrystals().super_categories()
[Category of regular crystals,
 Category of loop crystals]
```

3.103 L-trivial semigroups

class sage.categories.l_trivial_semigroups.LTrivialSemigroups(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

Commutative_extra_super_categories()

Implement the fact that a commutative \( R \)-trivial semigroup is \( J \)-trivial.

EXAMPLES:

```
sage: Semigroups().LTrivial().Commutative_extra_super_categories()
[Category of j trivial semigroups]
```

RTrivial_extra_super_categories()

Implement the fact that an \( L \)-trivial and \( R \)-trivial semigroup is \( J \)-trivial.

EXAMPLES:

```
sage: Semigroups().LTrivial().RTrivial_extra_super_categories()
[Category of j trivial magmas]
```

exa_super_categories()

Implement the fact that a \( L \)-trivial semigroup is \( H \)-trivial.

EXAMPLES:

```
sage: Semigroups().LTrivial().extra_super_categories()
[Category of h trivial semigroups]
```
### 3.104 Magmas

**class** `sage.categories.magas.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:**

```python
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:

```python
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Unital()
Category of unital magmas
sage: Magmas().Commutative()
Category of commutative magmas
sage: Magmas().Unital().Inverse()
Category of inverse unital magmas
sage: Magmas().Associative()
Category of semigroups
sage: Magmas().Associative().Unital()
Category of monoids
sage: Magmas().Associative().Unital().Inverse()
Category of groups
```

**class** `Algebras`(*category, *args*)

**Bases:** `sage.categories.algebra_functor.AlgebrasCategory`

**class** `ParentMethods`

**is_field**(proof=True)

Return `True` if self is a field.

For a magma algebra $RS$ this is always false unless $S$ is trivial and the base ring $R'$ is a field.

**EXAMPLES:**

```python
sage: SymmetricGroup(1).algebra(QQ).is_field()
True
sage: SymmetricGroup(1).algebra(ZZ).is_field()
False
sage: SymmetricGroup(2).algebra(QQ).is_field()
False
```

**extra_super_categories**()

**EXAMPLES:**

```python
```
This implements the fact that the algebra of a commutative magma is commutative:

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

In particular, commutative monoid algebras are commutative algebras:

```
sage: Monoids().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

### Associative

alias of `sage.categories.semigroups.Semigroups`

#### class CartesianProducts(category, *args)

Bases: `sage.categories.cartesian_product.CartesianProductsCategory`

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])]
```

**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()
Category of Cartesian products of commutative rings
sage: sorted(C.category().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveInverse', 'AdditiveUnital', 'Associative', 'Commutative', 'Distributive', 'Unital']
sage: TestSuite(C).run()
```
extra_super_categories()
This implements the fact that a subquotient (and therefore a quotient or subobject) of a finite set is finite.

EXAMPLES:

```
sage: Semigroups().CartesianProducts().extra_super_categories()
[Category of semigroups]
sage: Semigroups().CartesianProducts().super_categories()
[Category of semigroups, Category of Cartesian products of magmas]
```

class Commutative

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton
class Algebras(category, *args)

Bases: sage.categories.algebra_functor.AlgebrasCategory
eextra_super_categories()

EXAMPLES:

```
sage: Magmas().Commutative().Algebras(QQ).extra_super_categories()
[Category of commutative magmas]
sage: Magmas().Commutative().Algebras(QQ).super_categories()
[Category of magma algebras over Rational Field, Category of commutative magmas]
```

This implements the fact that the algebra of a commutative magma is commutative:

```
sage: Magmas().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

In particular, commutative monoid algebras are commutative algebras:

```
sage: Monoids().Commutative().Algebras(QQ).is_subcategory(Algebras(QQ).Commutative())
True
```

class CartesianProducts

Bases: sage.categories.cartesian_product.CartesianProductsCategory
eextra_super_categories()

Implement the fact that a Cartesian product of commutative additive magmas is still an commutative additive magmas.

EXAMPLES:

```
sage: C = Magmas().Commutative().CartesianProducts()
sage: C.extra_super_categories()
[Category of commutative magmas]
sage: C.axioms()
frozenset({'Commutative'})
```

class ParentMethods

is_commutative()
Return True, since commutative magmas are commutative.

EXAMPLES:

```
sage: Parent(QQ,category=CommutativeRings()).is_commutative()
True
```
class 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
```

FinitelyGeneratedAsMagma
alias of Sage.categories.finitely_generated_magmas.FinitelyGeneratedMagmas

class JTrivial(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class 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.

INPUT:
- names - the type of names used
  - 'letters' - lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by column_keys(), padded to a common width with leading 'a's.
  - 'digits' - base 10 representation of the elements' positions in the list given by column_keys(), padded to a common width with leading zeros.
  - 'elements' - the string representations of the elements themselves.
  - a list - a list of strings, where the length of the list equals the number of elements.
- elements - default = None. A list of elements of the magma, in forms that can be coerced into the structure, eg. their string representations. This may be used to impose a 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 b c d e |
| b c d e a |
| c d e a b |
| d e a b c |
| 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', 'ab', 'b', 'ba')
sage: T
+--------+
| 0 1 2 3 |
+--------+
| 0| 0 1 1 1 |
| 1| 1 1 1 1 |
| 2| 3 3 2 3 |
| 3| 3 3 3 3 |
```

Specifying the elements in an alternative order can provide more insight into how the operation behaves.

```
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 d d e e e |
| b| b b c c b b c c c c c c c |
| c| c c c c c c c c c c c c c |
| d| d e d e e e e e e e e e e e |
| e| e e e e e e e e e e e e e e |
| f| f g h h h f g h i j i j j j |
| g| g g h h h g 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 i j i j i j i j |
| j| j j j j j i j i j i j i j i j |
| k| l m l m n o o o o o k l m n o |
| l| l 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 |
| n| o o o o o o o o o o o o 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: T.translation()
{'a': (), 'b': (1,2,3), 'c': (1,3,2)}
sage: T.change_names(['x', 'y', 'z'])
sage: T.translation()
{'x': (), 'y': (1,2,3), 'z': (1,3,2)}
sage: T
 * x y z
 +-----
 x| x y z
 y| y z x
 z| z x y
```

**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:
When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich \texttt{S.product} with extra mathematical structure. This will typically be implemented using lazy attributes:

\begin{verbatim}
sage: bin # todo: not implemented
Generic binary morphism:
From: (S x S)
To: S
\end{verbatim}

\texttt{product\_from\_element\_class\_mul}(x, y)

The binary multiplication of the magma.

\textbf{INPUT:}
- \(x, y\) – elements of this magma

\textbf{OUTPUT:}
- an element of the magma (the product of \(x\) and \(y\))

\textbf{EXAMPLES:}

\begin{verbatim}
sage: S = Semigroups().example("free")
sage: x = S('a'); y = S('b')
sage: S.product(x, y)
'ab'
\end{verbatim}

A parent in \texttt{Magmas()} must either implement \texttt{product()} in the parent class or \_\texttt{mul\_} in the element class. By default, the addition method on elements \(x\._\texttt{mul\_}(y)\) calls \texttt{S.product}(x, y), and reciprocally.

As a bonus, \texttt{S.product} models the binary function from \(S\) to \(S\):

\begin{verbatim}
sage: bin = S.product
sage: bin(x,y)
'ab'
\end{verbatim}

Currently, \texttt{S.product} is just a bound method:

\begin{verbatim}
sage: bin # py2
<bound method FreeSemigroup\_with\_category.product of An example of a_
→semigroup: the free semigroup generated by ('a', 'b', 'c', 'd')>
sage: bin # py3, due to difference in how bound methods are repr'd
<bound method FreeSemigroup.product of An example of a semigroup: the_
→free semigroup generated by ('a', 'b', 'c', 'd')>
\end{verbatim}

When Sage will support multivariate morphisms, it will be possible, and in fact recommended, to enrich \texttt{S.product} with extra mathematical structure. This will typically be implemented using lazy attributes:

\begin{verbatim}
sage: bin # todo: not implemented
Generic binary morphism:
From: (S x S)
To: S
\end{verbatim}
class Realizations(category, *args)
    Bases: sage.categories.realizations.RealizationsCategory

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 algebra of {1, 2, 3} over Rational Field in the Out basis>
    sage: Out.product.__module__
    'sage.categories.magas'
    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 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 ∈ 𝑀,
    
    \[ 𝑥 \ast (𝑦 \ast 𝑧) = (𝑥 \ast 𝑦) \ast 𝑧 \]

    See also:
    Wikipedia article Associative_property

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

    Commutative()
    Return the full subcategory of the commutative objects of self.
    A (multiplicative) magma Magmas 𝑀 is commutative if, for all 𝑥, 𝑦 ∈ 𝑀,
    
    \[ 𝑥 \ast 𝑦 = 𝑦 \ast 𝑥 \]

    See also:
    Wikipedia article Commutative_property

    EXAMPLES:
    sage: Magmas().Commutative()
    Category of commutative magmas
    sage: Monoids().Commutative()
    Category of commutative monoids

3.104. 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 an additive magma
```

FinitelyGenerated()

Return the subcategory of the objects of self that are endowed with a distinguished finite set of (multiplicative) magma generators.

EXAMPLES:

This is a shorthand for FinitelyGeneratedAsMagma(), which see:

```
sage: Magmas().FinitelyGenerated()
Category of finitely generated magmas
sage: Semigroups().FinitelyGenerated()
Category of finitely generated semigroups
sage: Groups().FinitelyGenerated()
Category of finitely generated enumerated groups
```

An error is raised if this is ambiguous:

```
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 enumerated rings
```

On the other hand, it does not depend on the multiplicative structure: for example a group is finitely generated if and only if it is finitely generated as a magma. A short hand is provided when there is no ambiguity, and the output tries to reflect that:

```
sage: Semigroups().FinitelyGenerated()
sage: Groups().FinitelyGenerated()
sage: Semigroups().FinitelyGenerated().axioms()
frozenset({'Associative', 'Enumerated', 'FinitelyGeneratedAsMagma'})
```

Note that the set of generators may depend on the actual category; for example, in a group, one can often use less generators since it is allowed to take inverses.

JTrivial()

Return the full subcategory of the $J$-trivial objects of self.

This axiom is in fact only meaningful for semigroups. This stub definition is here as a workaround for trac ticket #20515, in order to define the $J$-trivial axiom as the intersection of the $L$ and $R$-trivial axioms.

See also:

Semigroups.SubcategoryMethods.JTrivial()

Unital()

Return the subcategory of the unital objects of self.
A (multiplicative) magma $\text{Magma} M$ is \textit{unital} if it admits an element 1, called \textit{unit}, such that for all $x \in M$,

$$1 \ast x = x \ast 1 = x$$

This element is necessarily unique, and should be provided as $M.$\texttt{one()}.

\textbf{See also:}

Wikipedia article Unital magma\#unital

\textbf{EXAMPLES:}

```sage
Magmas().Unital()
Category of unital magmas
Semigroups().Unital()
Category of monoids
Monoids().Unital()
Category of monoids
from sage.categories.associative_algebras import AssociativeAlgebras
AssociativeAlgebras(QQ).Unital()
Category of algebras over Rational Field
```

class Subquotients\(\text{(category, *args)}\)

\text{Bases:} sage.categories.subquotients.SubquotientsCategory

The category of subquotient magmas.

See $\text{Sets.SubcategoryMethods.Subquotients()}$ 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 $\text{As}$ 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 $G$.

\textbf{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

\textbf{product}\( (x, y)\)

Return the product of two elements of $\text{self}$.

\textbf{EXAMPLES:}

```sage
S = Semigroups().Subquotients().example()
S
An example of a (sub)quotient semigroup:
a quotient of the left zero semigroup
S.product(S(19), S(3))
19
```

Here is a more elaborate example involving a sub algebra:
sage: Z = SymmetricGroup(5).algebra(QQ).center()
sage: B = Z.basis()

class Unital (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton
class Algebras (category, *args)
Bases: sage.categories.algebra_functor.AlgebrasCategory
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 CartesianProducts (category, *args)
Bases: sage.categories.cartesian_product.CartesianProductsCategory
class ElementMethods
class 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)

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 ElementMethods

class Inverse(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

class CartesianProducts(category, *args)

Bases: sage.categories.cartesian_product.CartesianProductsCategory

extra_super_categories()

Implement the fact that a Cartesian product of magmas with inverses is a magma with inverse.

EXAMPLES:

```sage
C = Magmas().Unital().Inverse().CartesianProducts()
sage: C.extra_super_categories()
[Category of inverse unital magmas]
sage: sorted(C.axioms())
['Inverse', 'Unital']
```

class 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
```

one()

Return the unit of the monoid, that is the unique neutral element for *.

Note: The default implementation is to coerce 1 into self. It is recommended to override this method because the coercion from the integers:
  • is not always meaningful (except for 1);
  • often uses self.one().

EXAMPLES:

```sage
M = Monoids().example(); M
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd ...

sage: M.one()

```

class Realizations(category, *args)

Bases: sage.categories.realizations.RealizationsCategory

class ParentMethods
Return the unit element of \texttt{self}.

```python
sage: from sage.combinat.root_system.extended_affine_weyl_group import ExtendedAffineWeylGroup
sage: PvW0 = ExtendedAffineWeylGroup(['A',2,1]).PvW0()
sage: PvW0 in Magmas().Unital().Realizations() True sage: PvW0.one() 1
```

\textbf{Inverse}()

Return the full subcategory of the inverse objects of \texttt{self}.

An inverse \texttt{ magm }a \texttt{ <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.

\textbf{See also:}

Wikipedia article Inverse_element, Wikipedia article Quasigroup

\textbf{EXAMPLES:}

```python
sage: Magmas().Unital().Inverse()
Category of inverse unital magmas
sage: Monoids().Inverse()
Category of groups
```

\textbf{additional_structure}()

Return \texttt{self}.

Indeed, the category of unital magmas defines an additional structure, namely the unit of the magma which shall be preserved by morphisms.

\textbf{See also:}

\texttt{Category.additional_structure}()

\textbf{EXAMPLES:}

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

\textbf{super_categories}()

\textbf{EXAMPLES:}

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

\section{3.105 Magmas and Additive Magmas}

\textbf{class} \texttt{sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas} \texttt{(s=None)}

Bases: \texttt{sage.categories.category_singleton.Category_singleton}

The category of sets \((S,\,+,*\,)\) with an additive operation `+' and a multiplicative operation `*`.

\textbf{EXAMPLES:}
This is the base category for the categories of rings and their variants:

```
sage: C.Distributive()
Category of distributive magmas and additive magmas
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().AdditiveInverse()
Category of rngs
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().Unit()
Category of semirings
sage: C.Distributive().Associative().AdditiveAssociative().AdditiveCommutative().AdditiveUnital().AdditiveInverse().Unital()
Category of rings
```

This category is really meant to represent the intersection of the categories of `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
```

### CartesianProducts

```python
class CartesianProducts(category, *args):
    Bases: sage.categories.cartesian_product.CartesianProductsCategory
    extra_super_categories()
    Implement the fact that this structure is stable under Cartesian products.

Distributive
    alias of sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas
```

### SubcategoryMethods

```python
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 $x, y, z \in M$,
    
    $$x * (y + z) = x * y + x * z \quad \text{and} \quad (x + y) * z = x * z + y * z$$

    EXAMPLES:
```

Note: Given that Sage does not know that `MagmasAndAdditiveMagmas` is the intersection of `Magmas` and `AdditiveMagmas`, this method is not available for:
Still, the natural syntax works:

\begin{Verbatim}
\texttt{sage: (Magmas() & AdditiveMagmas()).Distributive()}
\end{Verbatim}

Category of distributive magmas and additive magmas

thanks to a workaround implemented in \texttt{Magmas.SubcategoryMethods.Distributive()}:

\begin{Verbatim}
\texttt{sage: (Magmas() & AdditiveMagmas()).Distributive.__module__}
\texttt{sage.categories.magmas}
\end{Verbatim}

\textit{additional_structure}()

Return None.

Indeed, this category is meant to represent the join of \texttt{AdditiveMagmas} and \texttt{Magmas}. As such, it defines no additional structure.

See also:

\texttt{Category.additional_structure()}

EXAMPLES:

\begin{Verbatim}
\texttt{sage: from sage.categories.magmas_and_additive_magmas import *}
\texttt{MagmasAndAdditiveMagmas}
\texttt{sage: MagmasAndAdditiveMagmas().additional_structure()}
\end{Verbatim}

\textit{super_categories}()

EXAMPLES:

\begin{Verbatim}
\texttt{sage: from sage.categories.magmas_and_additive_magmas import *}
\texttt{MagmasAndAdditiveMagmas}
\texttt{sage: MagmasAndAdditiveMagmas().super_categories()}
\texttt{[Category of magmas, Category of additive magmas]}
\end{Verbatim}

### 3.106 Non-unital non-associative algebras

\texttt{class sage.categories.magmatic_algebras.MagmaticAlgebras}(\texttt{base, name=\texttt{None}})

\texttt{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.

\textbf{Warning:} \texttt{MagmaticAlgebras} will eventually replace the current \texttt{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 additive magmas, Category of modules over Integer Ring]

Associative
alias of sage.categories.associative_algebras.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') over Rational Field
sage: F.algebra_generators()
Family (B[word: a], B[word: b], B[word: c])

Unital
alias of sage.categories.unital_algebras.UnitalAlgebras
class WithBasis (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring
class FiniteDimensional (base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring
class ParentMethods
derivations_basis()
Return a basis for the Lie algebra of derivations of self as matrices.

A derivation $D$ of an algebra is an endomorphism of $A$ such that

$$D(ab) = D(a)b + aD(b)$$

for all $a, b \in A$. The set of all derivations form a Lie algebra.

EXAMPLES:

We construct the Heisenberg Lie algebra as a multiplicative algebra:

sage: p_mult = matrix(QQ,3,3,[[0,0,0],[0,0,-1],[0,0,0]])
sage: q_mult = matrix(QQ,3,3,[[0,0,1],[0,0,0],[0,0,0]])
sage: A = algebras.FiniteDimensional(QQ,.....: [p_mult,q_mult,matrix(QQ,3,3)], 'p,q,z')
sage: A.inject_variables()
Defining p, q, z
sage: p*q
z

(continues on next page)
We construct another example using the exterior algebra and verify we obtain a derivation:

```
sage: A = algebras.Exterior(QQ, 1)
sage: A.derivations_basis()
(\[
[0 0] \\
[0 1]
\])
sage: D = A.module_morphism(matrix=A.derivations_basis()[0], \[\]→codomain=A)
sage: one, e = A.basis()
sage: all(D(a*b) == D(a) * b + a * D(b)  \[\]....: for a in A.basis() for b in A.basis())
True
```
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:**

```python
sage: A = AlgebrasWithBasis(QQ).example()
sage: a, b, c = A.algebra_generators()
sage: A.product(a + 2*b, 3*c)
3*B[ac] + 6*B[bc]
```

### product_on_basis(i, j)

The product of the algebra on the basis (optional).

**INPUT:**

- `i, j` – the indices of two elements of the basis of `self`

Return the product of the two corresponding basis elements indexed by `i` and `j`.

If implemented, `product()` is defined from it by bilinearity.

**EXAMPLES:**

```python
sage: A = AlgebrasWithBasis(QQ).example()
sage: Word = A.basis().keys()
sage: A.product_on_basis(Word("abc"),Word("cba"))
B[abccba]
```

### additional_structure()  

Return `None`. Indeed, the category of (magmatic) algebras defines no new structure: a morphism of modules and of magmas between two (magmatic) algebras is a (magmatic) algebra morphism.

**See also:**

`Category.additional_structure()`

**Todo:** This category should be a `CategoryWithAxiom`, the axiom specifying the compatibility between the magma and module structure.

**EXAMPLES:**

```python
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(ZZ).additional_structure()
```

### super_categories()  

**EXAMPLES:**

```python
sage: from sage.categories.magmatic_algebras import MagmaticAlgebras
sage: MagmaticAlgebras(ZZ).super_categories()
```

```python
[Category of additive commutative additive associative additive unital → distributive magmas and additive magmas, Category of modules over Integer → Ring]
```

```python
sage: from sage.categories.additive_semigroups import AdditiveSemigroups
sage: MagmaticAlgebras(ZZ).is_subcategory((AdditiveSemigroups() & Magmas()).Distributive())
True
```
3.107 Manifolds

class sage.categories.manifolds.ComplexManifolds(base, name=None)
Bases: sage.categories.category_types.Category_over_base_ring

The category of complex manifolds.
A \(d\)-dimensional complex manifold is a manifold whose underlying vector space is \(\mathbb{C}^d\) and has a holomorphic atlas.

super_categories()

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

class sage.categories.manifolds.Manifolds(base, name=None)
Bases: sage.categories.category_types.Category_over_base_ring

The category of manifolds over any topological field.
Let \(k\) be a topological field. A \(d\)-dimensional \(k\)-manifold \(M\) is a second countable Hausdorff space such that the neighborhood of any point \(x \in M\) is homeomorphic to \(k^d\).

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR); C
Category of manifolds over Real Field with 53 bits of precision
sage: C.super_categories()
[Category of topological spaces]
```

class AlmostComplex(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of almost complex manifolds.
An almost complex manifold \(M\) is a manifold with a smooth tensor field \(J\) of rank \((1, 1)\) such that \(J^2 = -1\) when regarded as a vector bundle isomorphism \(J : TM \rightarrow TM\) on the tangent bundle. The tensor field \(J\) is called the almost complex structure of \(M\).

extra_super_categories()

Return the extra super categories of \(self\).
An almost complex manifold is smooth.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR); C
Category of manifolds over Real Field with 53 bits of precision
sage: C.AlmostComplex().super_categories() # indirect doctest
[Category of smooth manifolds over Real Field with 53 bits of precision]
```

class Analytic(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of complex manifolds.
An analytic manifold is a manifold with an analytic atlas.
**extra_super_categories()**

Return the extra super categories of `self`.

An analytic manifold is smooth.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Analytic().super_categories()  # indirect doctest
[Category of smooth manifolds
  over Real Field with 53 bits of precision]
```

class `Connected` *(base_category)*

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

The category of connected manifolds.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).Connected()
sage: TestSuite(C).run(skip="_test_category_over_bases")
```

class `Differentiable` *(base_category)*

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

The category of differentiable manifolds.

A differentiable manifold is a manifold with a differentiable atlas.

class `FiniteDimensional` *(base_category)*

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

Category of finite dimensional manifolds.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).FiniteDimensional()
sage: TestSuite(C).run(skip="_test_category_over_bases")
```

class `ParentMethods`

**`dimension()`**

Return the dimension of `self`.

**EXAMPLES:**

```python
sage: from sage.categories.manifolds import Manifolds
sage: M = Manifolds(RR).example()
sage: M.dimension()
3
```

class `Smooth` *(base_category)*

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring`

The category of smooth manifolds.

A smooth manifold is a manifold with a smooth atlas.

**extra_super_categories()**

Return the extra super categories of `self`.
A smooth manifold is differentiable.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Smooth().super_categories() # indirect doctest
[Category of differentiable manifolds
erover Real Field with 53 bits of precision]
```

```python
class SubcategoryMethods

AlmostComplex()
    Return the subcategory of the almost complex objects of self.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).AlmostComplex()
Category of almost complex manifolds
erover Real Field with 53 bits of precision
```

Analytic()
    Return the subcategory of the analytic objects of self.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Analytic()
Category of analytic manifolds
erover Real Field with 53 bits of precision
```

Complex()
    Return the subcategory of manifolds over \( \mathbb{C} \) of self.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(CC).Complex()
Category of complex manifolds over
Complex Field with 53 bits of precision
```

Connected()
    Return the full subcategory of the connected objects of self.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Connected()
Category of connected manifolds
erover Real Field with 53 bits of precision
```

Differentiable()
    Return the subcategory of the differentiable objects of self.

EXAMPLES:

```python
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Differentiable()
```

(continues on next page)
Category of differentiable manifolds
over Real Field with 53 bits of precision

**FiniteDimensional()**
Return the full subcategory of the finite dimensional objects of `self`.

**EXAMPLES:**
```
sage: from sage.categories.manifolds import Manifolds
sage: C = Manifolds(RR).Connected().FiniteDimensional(); C
Category of finite dimensional connected manifolds
over Real Field with 53 bits of precision
```

**Smooth()**
Return the subcategory of the smooth objects of `self`.

**EXAMPLES:**
```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).Smooth()
Category of smooth manifolds
over Real Field with 53 bits of precision
```

**additional_structure()**
Return None.
Indeed, the category of manifolds defines no new structure: a morphism of topological spaces between
manifolds is a manifold morphism.

**See also:**
Category.additional_structure()

**EXAMPLES:**
```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).additional_structure()
```

**super_categories()**

**EXAMPLES:**
```
sage: from sage.categories.manifolds import Manifolds
sage: Manifolds(RR).super_categories()
[Category of topological spaces]
```

### 3.108 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
```
super_categories()

EXAMPLES:

```python
sage: MatrixAlgebras(QQ).super_categories()
[Category of algebras over Rational Field]
```

3.109 Metric Spaces

class sage.categories.metric_spaces.MetricSpaces(category, *args)

Bases: sage.categories.metric_spaces.MetricSpacesCategory

The category of metric spaces.

A metric on a set \( S \) is a function \( d : S \times S \rightarrow \mathbb{R} \) such that:

- \( d(a, b) \geq 0 \),
- \( d(a, b) = 0 \) if and only if \( a = b \).

A metric space is a set \( S \) with a distinguished metric.

Implementation

Objects in this category must implement either a `dist` on the parent or the elements or `metric` on the parent; otherwise this will cause an infinite recursion.

Todo:

- Implement a general geodesics class.
- Implement a category for metric additive groups and move the generic distance \( d(a, b) = |a - b| \) there.
- Incorporate the length of a geodesic as part of the default distance cycle.

EXAMPLES:

```python
sage: from sage.categories.metric_spaces import MetricSpaces
sage: C = MetricSpaces()

sage: C
Category of metric spaces

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

class Complete(base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of complete metric spaces.

class ElementMethods

abs()

Return the absolute value of `self`.

EXAMPLES:

```python
sage: CC(I).abs()
1.00000000000000
```
dist \( (b) \)
Return the distance between \texttt{self} and \texttt{other}.

EXAMPLES:

```
sage: UHP = HyperbolicPlane().UHP()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1 + I)
sage: p1.dist(p2)
arccosh(33/7)
```

class ParentMethods
dist \( (a, b) \)
Return the distance between \(a\) and \(b\) in \texttt{self}.

EXAMPLES:

```
sage: UHP = HyperbolicPlane().UHP()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1.0 + I)
sage: UHP.dist(p1, p2)
2.23230104635820
```

metric()
Return the metric of \texttt{self}.

EXAMPLES:

```
sage: UHP = HyperbolicPlane().UHP()
sage: m = UHP.metric()
sage: p1 = UHP.get_point(5 + 7*I)
sage: p2 = UHP.get_point(1.0 + I)
sage: m(p1, p2)
2.23230104635820
```

class SubcategoryMethods

Complete()
Return the full subcategory of the complete objects of \texttt{self}.

EXAMPLES:

```
sage: Sets().Metric().Complete()
Category of complete metric spaces
```

class WithRealizations\((\texttt{category}, \ast \texttt{args})\)
Bases: \texttt{sage.categories.with_realizations.WithRealizationsCategory}
class ParentMethods
dist \( (a, b) \)
Return the distance between \(a\) and \(b\) by converting them to a realization of \texttt{self} and doing the computation.
class sage.categories.metric_spaces.MetricSpacesCategory(category, *args)
    Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory
    classmethod default_super_categories(category)
        Return the default super categories of category.Metric().

        Mathematical meaning: if A is a metric space in the category C, then A is also a topological space.

        INPUT:
        • cls – the class MetricSpaces
        • category – a category Cat

        OUTPUT:
        A (join) category

        In practice, this returns category.Metric(), joined together with the result of the method
        RegressiveCovariantConstructionCategory.default_super_categories() (that is the join of category and cat.Metric() for each cat in the super categories of category).

        EXAMPLES:
        Consider category=Groups(). Then, a group G with a metric is simultaneously a topological group by itself, and a metric space:

        sage: Groups().Metric().super_categories()
        [Category of topological groups, Category of metric spaces]
        This resulted from the following call:

        sage: sage.categories.metric_spaces.MetricSpacesCategory.default_super_categories(Groups())
        Join of Category of topological groups and Category of metric spaces

3.110 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

3.110. Modular abelian varieties
class Endset(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

extra_super_categories()
Implement the fact that an endset of modular abelian variety is a ring.
EXAMPLES:

```
sage: ModularAbelianVarieties(QQ).Endsets().extra_super_categories()
[Category of rings]
```

base_field()
EXAMPLES:

```
sage: ModularAbelianVarieties(QQ).base_field()
Rational Field
```

super_categories()
EXAMPLES:

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

### 3.111 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:

\[
r \cdot (x \cdot s) = (r \cdot x) \cdot s \quad \forall r, s \in R \text{ and } x \in M
\]

INPUT:

- base_ring – a ring \( R \) or subcategory of Rings()
- dispatch – a boolean (for internal use; default: True)

When the base ring is a field, the category of vector spaces is returned instead (unless dispatch == False).

**Warning:** Outside of the context of symmetric modules over a commutative ring, the specifications of this category are fuzzy and not yet set in stone (see below). The code in this category and its subcategories is therefore prone to bugs or arbitrary limitations in this case.

EXAMPLES:

```
sage: Modules(ZZ)
Category of modules over Integer Ring
sage: Modules(QQ)
Category of vector spaces over Rational Field
sage: Modules(Rings())
Category of modules over rings
```

(continues on next page)
Todo:

- Clarify the distinction, if any, with \texttt{BiModules(R, R)}. In particular, if $R$ is a commutative ring (e.g. a field), some pieces of the code possibly assume that $M$ is a symmetric `$R$-`$R$'-bimodule:

\[
x \cdot r = r \cdot x \quad \forall r \in R \text{ and } x \in M
\]

- Make sure that non symmetric modules are properly supported by all the code, and advertise it.
- Make sure that non commutative rings are properly supported by all the code, and advertise it.
- Add support for base semirings.
- Implement a \texttt{FreeModules(R)} category, when so prompted by a concrete use case: e.g. modeling a free module with several bases (using \texttt{Sets.SubcategoryMethods.Realizations()}) or with an atlas of local maps (see e.g. trac ticket \#15916).

\begin{Verbatim}
\textbf{sage:} Modules(FiniteFields())
Category of vector spaces over finite enumerated fields
\sage:
\textbf{sage:} Modules(Integers(9))
Category of modules over Ring of integers modulo 9
\sage:
\textbf{sage:} Modules(Integers(9)).super_categories()

\texttt{[Category of bimodules over Ring of integers modulo 9 on the left and Ring of\n\quad \#\quad \textbf{\textsc{\rightarrow}}\text{\textsc{-}} integers modulo 9 on the right]}\texttt{]}
\sage:
\textbf{sage:} Modules(ZZ).super_categories()

\texttt{[Category of bimodules over Integer Ring on the left and Integer Ring on the\n\quad \textbf{\textsc{\rightarrow}}\text{\textsc{-}} right]}\texttt{]}
\sage:
\textbf{sage:} Modules == RingModules
True
\sage:
\textbf{sage:} Modules(ZZ['x']).is_abelian()  # see \#6081
True
\end{Verbatim}

\begin{Verbatim}
\textbf{class CartesianProducts (category, *args)}
\quad \textbf{Bases:} \texttt{sage.categories.cartesian_product.CartesianProductsCategory}

\quad The category of modules constructed as Cartesian products of modules

\quad This construction gives the direct product of modules. The implementation is based on the following resources:

\quad - \url{http://groups.google.fr/group/sage-devel/browse_thread/thread/35a72b1d0a2fc77a/348f42ae77a66d16#348f42ae77a66d16}

\quad - Wikipedia article Direct_product

\textbf{class ParentMethods}

\quad \textbf{base_ring ()}

\quad \textbf{Return the base ring of this Cartesian product.}

\textbf{EXAMPLES:}

3.111. Modules 529
sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
sage: C = cartesian_product([E, F]); C
Free module generated by {1, 2, 3} over Integer Ring (+)
Free module generated by {2, 3, 4} over Integer Ring
sage: C.base_ring()
Integer Ring

extra_super_categories()
A Cartesian product of modules is endowed with a natural module structure.

EXAMPLES:

sage: Modules(ZZ).CartesianProducts().extra_super_categories()
[Category of modules over Integer Ring]
sage: Modules(ZZ).CartesianProducts().super_categories()
[Category of Cartesian products of commutative additive groups,
 Category of modules over Integer Ring]

class ElementMethods

Filtered
alias of sage.categories.filtered_modules.FilteredModules

class FiniteDimensional(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

extra_super_categories()
Implement the fact that a finite dimensional module over a finite ring is finite.

EXAMPLES:

sage: Modules(IntegerModRing(4)).FiniteDimensional().extra_super_categories()
[Category of finite sets]
sage: Modules(ZZ).FiniteDimensional().extra_super_categories()
[]
sage: Modules(GF(5)).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: Modules(ZZ).FiniteDimensional().is_subcategory(Sets().Finite())
False
sage: Modules(Rings().Finite()).FiniteDimensional().is_subcategory(Sets().Finite())
True
sage: Modules(Rings()).FiniteDimensional().is_subcategory(Sets().Finite())
False

Graded
alias of sage.categories.graded_modules.GradedModules

class 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:

```python
sage: Modules(ZZ).Endsets().extra_super_categories()
[Category of magmatic algebras over Integer Ring]

sage: End(ZZ^3) in Algebras(ZZ)
True
```

class ParentMethods

base_ring()
Return the base ring of self.

EXAMPLES:

```python
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.CategoryObject.base_ring()`:

```python
sage: H.base_ring.__module__
```

Here we call it directly:

```python
sage: method = H.category().parent_class.base_ring
sage: method.__get__(H)()
Integer Ring
```

zero()

EXAMPLES:

```python
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
```

base_ring()

EXAMPLES:
Todo: Generalize this so that any homset category of a full subcategory of modules over a base ring is a category over this base ring.

extra_super_categories()
EXAMPLES:
```python
sage: Modules(ZZ).Homsets().extra_super_categories()
[Category of modules over Integer Ring]
```

class ParentMethods

tensor_square()
Returns the tensor square of self

EXAMPLES:
```python
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 6
as a permutation group over Rational Field
# An example of Hopf algebra with basis: the group algebra of the Dihedral
group of order 6 as a permutation group over Rational Field
```

class SubcategoryMethods

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
```

**Filtered** *(base_ring=None)*

Return the subcategory of the filtered objects of `self`.

INPUT:

- `base_ring` – this is ignored

EXAMPLES:

```
sage: Modules(ZZ).Filtered()
Category of filtered modules over Integer Ring
```

```
sage: Coalgebras(QQ).Filtered()
Join of Category of filtered modules over Rational Field
    and Category of coalgebras over Rational Field
```

```
sage: AlgebrasWithBasis(QQ).Filtered()
Category of filtered algebras with basis over Rational Field
```

Todo:

- Explain why this does not commute with `WithBasis()`
- Improve the support for covariant functorial constructions categories over a base ring so as to get rid of the `base_ring` argument.

**FiniteDimensional**

Return the full subcategory of the finite dimensional objects of `self`.

EXAMPLES:

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

(continues on next page)
Graded (base_ring=None)

Return the subcategory of the graded objects of self.

INPUT:

• base_ring – this is ignored

EXAMPLES:

```python
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 Rational Field

sage: AlgebrasWithBasis(QQ).Graded()
Category of graded algebras with basis over Rational Field
```

Todo:

• Explain why this does not commute with WithBasis()

Super (base_ring=None)

Return the super-analogue category of self.

INPUT:

• base_ring – this is ignored

EXAMPLES:

```python
sage: Modules(ZZ).Super()
Category of super modules over Integer Ring

sage: Coalgebras(QQ).Super()
Category of super coalgebras over Rational Field

sage: AlgebrasWithBasis(QQ).Super()
Category of super algebras with basis over Rational Field
```

Todo:

• Explain why this does not commute with WithBasis()

TensorProducts()

Return the full subcategory of objects of self constructed as tensor products.

See also:

• `tensor.TensorProductsCategory`
• RegressiveCovariantFunctorialConstruction.

**EXAMPLES:**

```python
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:**

```python
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
```

**base_ring()**

Return the base ring (category) for self.

This implements a base_ring method for all subcategories of Modules(K).

**EXAMPLES:**

```python
sage: C = Modules(QQ) & Semigroups(); C
Join of Category of semigroups and Category of vector spaces over Rational Field
sage: C.base_ring()
Rational Field
sage: C.base_ring.__module__
'sage.categories.modules'
sage: 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'
```

**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](https://en.wikipedia.org/wiki/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:**

```python
sage: VectorSpaces(QQ).DualObjects()
Category of duals of vector spaces over Rational Field
```

The dual of a vector space is a vector space:

```python
sage: VectorSpaces(QQ).DualObjects().super_categories()
[Category of vector spaces over Rational Field]
```

The dual of an algebra is a coalgebra:

```python
sage: sorted(Algebras(QQ).DualObjects().super_categories(), key=str)
[Category of coalgebras over Rational Field, Category of duals of vector spaces over Rational Field]
```

The dual of a coalgebra is an algebra:

```python
sage: sorted(Coalgebras(QQ).DualObjects().super_categories(), key=str)
[Category of algebras over Rational Field, Category of duals of vector spaces over Rational Field]
```

As a shorthand, this category can be accessed with the `dual()` method:

```python
sage: VectorSpaces(QQ).dual()
Category of duals of vector spaces over Rational Field
```

Super
---
alias of `sage.categories.super_modules.SuperModules`

class `TensorProducts`

```
(category, *args)
```

Bases: `sage.categories.tensor.TensorProductsCategory`

The category of modules constructed by tensor product of modules.

extrasuper_categories()

**EXAMPLES:**

---

536 Chapter 3. Individual Categories
Modules With Basis

WithBasis
   alias of sage.categories.modules_with_basis.ModulesWithBasis

additional_structure()
   Return None.

Indeed, the category of modules defines no additional structure: a bimodule morphism between two modules is a module morphism.

See also:
   Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

sage: Modules(ZZ).additional_structure()

super_categories()

EXAMPLES:

sage: Modules(ZZ).super_categories()

Nota bene:

sage: Modules(QQ)
Category of vector spaces over Rational Field

sage: Modules(QQ).super_categories()

3.112 Modules With Basis

AUTHORS:

- 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:
Let $X$ and $Y$ be two modules with basis. We can build $\hom(X, Y)$:

```python
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 Rational Field
```

The simplest morphism is the zero map:

```python
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 $X$:

```python
sage: x = X.monomial(1) + 3 * X.monomial(2)
sage: H.zero()(x)
0
```

**EXAMPLES:**

We now construct a more interesting morphism by extending a function by linearity:

```python
sage: phi = H(on_basis = lambda i: Y.monomial(i+2)); phi
Generic morphism:
  From: X
  To:  Y
```

We can retrieve the function acting on indices of the basis:

```python
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 $\text{FiniteDimensionalModulesWithBasis}$ and $\text{GradedModulesWithBasis}$:

```python
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.

Note: This category currently requires an implementation of an element method \texttt{support}. Once \texttt{trac ticket #18066} is merged, an implementation of an \texttt{items} method will be required.

class \texttt{CartesianProducts}(\texttt{category, *args})

Bases: \texttt{sage.categories.cartesian_product.CartesianProductsCategory}

The category of modules with basis constructed by Cartesian products of modules with basis.

class \texttt{ParentMethods}

\texttt{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 \texttt{DualObjects}(\texttt{category, *args})

Bases: \texttt{sage.categories.dual.DualObjectsCategory}

\texttt{extra_super_categories}()

EXAMPLES:

```
sage: ModulesWithBasis(ZZ).DualObjects().extra_super_categories()
[Category of modules over Integer Ring]
sage: ModulesWithBasis(QQ).DualObjects().super_categories()
[Category of duals of vector spaces over Rational Field, Category of duals of modules with basis over Rational Field]
```

class \texttt{ElementMethods}

\texttt{coefficient}(\texttt{m})

Return the coefficient of \texttt{m} in \texttt{self} and raise an error if \texttt{m} is not in the basis indexing set.

INPUT:
- \texttt{m} -- a basis index of the parent of \texttt{self}

OUTPUT:

The \(B[m]\)-coordinate of \texttt{self} with respect to the basis \(B\). Here, \(B\) denotes the given basis of the parent of \texttt{self}.

EXAMPLES:
Test that `coefficient` also works for those parents that do not have an `element_class`:

```
sage: H = End(ZZ)
sage: F = CombinatorialFreeModule(QQ, H)
sage: hasattr(H, "element_class")
False
sage: h = H.an_element()
sage: (2*F.monomial(h)).coefficient(h)
2
```

**coefficients** *(sort=True)*

Return a list of the (non-zero) coefficients appearing on the basis elements in `self` (in an arbitrary order).

**INPUT:**
- `sort = (default: True)` to sort the coefficients based upon the default ordering of the indexing set

**See also:**

`dense_coefficient_list()`

**EXAMPLES:**

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.coefficients()
[1, -3]
sage: f = B['c'] - 3*B['a']
sage: f.coefficients()
[-3, 1]
```

**is_zero()**

Return `True` if and only if `self == 0`.

**EXAMPLES:**
```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.is_zero()
False
sage: F.zero().is_zero()
True
sage: s = SymmetricFunctions(QQ).schur()
sage: s([2,1]).is_zero()
False
sage: s(0).is_zero()
True
sage: (s([2,1]) - s([2,1])).is_zero()
True
```

**leading_coefficient** (*args, **kwds*)

Return the leading coefficient of `self`.

This is the coefficient of the term whose corresponding basis element is maximal. Note that this may not be the term which actually appears first when `self` is printed.

If the default term ordering is not what is desired, a comparison key, `key(x,y)`, can be provided.

**EXAMPLES:**

```python
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_coefficient()
1
sage: def key(x):
    return -x
sage: x.leading_coefficient(key=key)
3
sage: s = SymmetricFunctions(QQ).schur()
sage: f.leading_coefficient()
-5
```

**leading_item** (*args, **kwds*)

Return the pair `(k, c)` where

\[ c \cdot (\text{the basis element indexed by } k) \]

is the leading term of `self`.

Here ‘leading term’ means that the corresponding basis element is maximal. Note that this may not be the term which actually appears first when `self` is printed.

If the default term ordering is not what is desired, a comparison function, `key(x)`, can be provided.

**EXAMPLES:**

```python
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X")
x = X
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_item()
(3, 4)
sage: def key(x):
    return -x
(continues on next page)
```
sage: x.leading_item(key=key)
(1, 3)
sage: s = SymmetricFunctions(QQ).schur()
sage: f.leading_item()
([3], -5)

leading_monomial (*args, **kwds)
Return the leading monomial of self.
This is the monomial whose corresponding basis element is maximal. Note that this may not be the
term which actually appears first when self is printed.

If the default term ordering is not what is desired, a comparison key, key(x), can be provided.

EXAMPLES:

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 key(x): return -x
sage: x.leading_monomial(key=key)
B[1]

sage: s = SymmetricFunctions(QQ).schur()
sage: f.leading_monomial()
s[3]

leading_support (*args, **kwds)
Return the maximal element of the support of self.
Note that this may not be the term which actually appears first when self is printed.

If the default ordering of the basis elements is not what is desired, a comparison key, key(x), can be
provided.

EXAMPLES:

sage: X = CombinatorialFreeModule(QQ, [1, 2, 3])
sage: 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 key(x): return -x
sage: x.leading_support(key=key)
1

sage: s = SymmetricFunctions(QQ).schur()
sage: f.leading_support()
[3]

leading_term (*args, **kwds)
Return the leading term of self.
This is the term whose corresponding basis element is maximal. Note that this may not be the term which actually appears first when self is printed.

If the default term ordering is not what is desired, a comparison key, key(x), can be provided.

EXAMPLES:

```python
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename("X"); x = X.basis()
sage: x = 3*X.monomial(1) + 2*X.monomial(2) + X.monomial(3)
sage: x.leading_term()
B[3]
sage: def key(x):
    return -x
sage: x.leading_term(key=key)
3*B[1]
sage: s = SymmetricFunctions(QQ).schur()
sage: f.leading_term()
-5*s[3]
```

```
length()
Return the number of basis elements whose coefficients in self are nonzero.

EXAMPLES:

```python
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.length()
2
```

```
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:

```python
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']
sage: f.map_coefficients(lambda x: 0)
0
```

Killed coefficients are handled properly:

```python
sage: f.map_coefficients(lambda x: 0)
0
sage: list(f.map_coefficients(lambda x: 0))
[]
```
```python
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 \((\text{index}, \text{coeff})\) to other such pairs

Return a new element of \(\text{self}.parent()\) obtained by applying the function \(f\) to all items \((\text{index}, \text{coeff})\) of \(\text{self}\).

**EXAMPLES:**
```
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)
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 \(\text{self}.parent()\) 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]
```
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 \( \text{self}.parent() \) obtained by applying the function \( f \) to all of the objects indexing the basis elements.

**EXAMPLES:**

```python
sage: B = CombinatorialFreeModule(ZZ, [-1, 0, 1])
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:

```python
sage: x.map_support_skip_none(lambda i: 1 if i else None)
5*B[1]
```

monomial_coefficients \((\text{copy=}\text{True})\)
Return a dictionary whose keys are indices of basis elements in the support of \( \text{self} \) and whose values are the corresponding coefficients.

**INPUT:**
- \( \text{copy=}(\text{default: True}) \) if \( \text{self} \) is internally represented by a dictionary \( d \), then make a copy of \( d \); if \( \text{False} \), then this can cause undesired behavior by mutating \( d \)

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']
sage: d = f.monomial_coefficients()
sage: d['a']
1
sage: d['c']
3
```

monomials ()
Return a list of the monomials of \( \text{self} \) (in an arbitrary order).

The monomials of an element \( a \) are defined to be the basis elements whose corresponding coefficients of \( a \) are non-zero.

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 2*B['c']
sage: f.monomials()
[B['a'], B['c']]
```

support ()
Return a list of the objects indexing the basis of \( \text{self}.parent() \) whose corresponding coefficients of \( \text{self} \) are non-zero.
This method returns these objects in an arbitrary order.

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: sorted(f.support())
['a', 'c']

sage: s = SymmetricFunctions(QQ).schur()
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: sorted(z.support())
[[1], [1, 1, 1], [2, 1], [4]]
```

**support_of_term()**

Return the support of `self`, where `self` is a monomial (possibly with coefficient).

**EXAMPLES:**

```python
sage: X = CombinatorialFreeModule(QQ, [1,2,3,4]); X.rename("X")
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:

```python
sage: (X.monomial(2) + X.monomial(3)).support_of_term()
Traceback (most recent call last):
```

**tensor(*elements)**

Return the tensor product of its arguments, as an element of the tensor product of the parents of those elements.

**EXAMPLES:**

```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example()
sage: (a,b,c) = A.algebra_generators()
sage: a.tensor(b, c)
```

FIXME: is this a policy that we want to enforce on all parents?

**terms()**

Return a list of the (non-zero) terms of `self` (in an arbitrary order).

**See also:**

*monomials()*

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] + 2*B['c']
```
trailing_coefficient(*args, **kwds)
Return the trailing coefficient of self.

This is the coefficient of the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key key(x), can be provided.

EXAMPLES:

```
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 key(x):
    return -x
sage: x.trailing_coefficient(key=key)
1
sage: s = SymmetricFunctions(QQ).schur()
sage: f.trailing_coefficient()
2
```

trailing_item(*args, **kwds)
Return the pair (c, k) where c*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 key key(x), can be provided.

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 key(x):
    return -x
sage: x.trailing_item(key=key)
(3, 1)
sage: s = SymmetricFunctions(QQ).schur()
sage: f.trailing_item()
([1], 2)
```

trailing_monomial(*args, **kwds)
Return the trailing monomial of self.

This is the monomial whose corresponding basis element is minimal. Note that this may not be the term which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key key(x), can be provided.

EXAMPLES:
trailing_support (*args, **kwds)
Return the minimal element of the support of self. Note that this may not be the term which actually
appears last when self is printed.

If the default ordering of the basis elements is not what is desired, a comparison key, key(x), can be
provided.

EXAMPLES:

```python
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_support()
1
sage: def key(x): return -x
sage: x.trailing_support(key=key)
3
sage: s = SymmetricFunctions(QQ).schur()
sage: f.trailing_support()
s[1]
```

trailing_term (*args, **kwds)
Return the trailing term of self.

This is the term whose corresponding basis element is minimal. Note that this may not be the term
which actually appears last when self is printed.

If the default term ordering is not what is desired, a comparison key key(x), can be provided.

EXAMPLES:

```python
sage: X = 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 key(x): return -x
sage: x.trailing_term(key=key)
B[3]
sage: s = SymmetricFunctions(QQ).schur()
```

(continues on next page)
Filtered alias of \texttt{sage.categories.filtered_modules_with_basis.\_\_init\_\_}\_\_\_\_\_FilteredModulesWithBasis

FiniteDimensional alias of \texttt{sage.categories.finite\_dimensional_modules_with_basis.\_\_init\_\_\_\_\_FiniteDimensionalModulesWithBasis}

Graded alias of \texttt{sage.categories.graded\_modules_with_basis.GradedModulesWithBasis.\_\_init\_\_\_\_\_}

\textbf{class Homsets (category, *\texttt{args})}
Bases: \texttt{sage.categories.homsets.HomsetsCategory}

class ParentMethods

class MorphismMethods

\textbf{on\_basis()}

Return the action of this morphism on basis elements.

\textbf{OUTPUT:}

\begin{itemize}
  \item a function from the indices of the basis of the domain to the codomain
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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)
sage: g == f
True
\end{verbatim}

\textbf{class ParentMethods}

\textbf{basis()}

Return the basis of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F.basis()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
\end{verbatim}
cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 4)
sage: S.cardinality()
+Infinity
sage: S = SymmetricGroupAlgebra(GF(2), 4)  # not tested -- MRO bug trac 
˓→#15475
sage: S.cardinality()  # not tested -- MRO bug trac #15475
16777216
sage: S.cardinality().factor()  # not tested -- MRO bug trac #15475
2^24
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.cardinality()
+Infinity
sage: E.<x,y> = ExteriorAlgebra(GF(3))
sage: E.cardinality()
81
sage: s = SymmetricFunctions(GF(2)).s()
sage: s.cardinality()
+Infinity
```

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: A.<x,y> = algebras.DifferentialWeyl(QQ)
sage: A.dimension()
+Infinity
```

echelon_form(elements, row_reduced=False)

Return a basis in echelon form of the subspace spanned by a finite set of elements.

INPUT:

- elements – a list or finite iterable of elements of self
- row_reduced – (default: False) whether to compute the basis for the row reduced echelon form

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[1], x[0] - x[2], x[1] - x[2]]
sage: matrix(list(map(vector, V)))
```

(continues on next page)
\[ \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \]

```
sage: F = CombinatorialFreeModule(ZZ, [1,2,3,4])
sage: B = F.basis()
sage: F.echelon_form(elements)
```

```
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']]
```

```
sage: R.<x,y> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
sage: x = C.basis()
sage: C.echelon_form([x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]])
[x[0] - x[2], x[1] - x[2]]
```

`is_finite()`

Return whether self is finite.

This is true if and only if `self.basis().keys()` and `self.base_ring()` are both finite.

**EXAMPLES:**

```
sage: GroupAlgebra(SymmetricGroup(2), IntegerModRing(10)).is_finite()
True
sage: GroupAlgebra(SymmetricGroup(2)).is_finite()
False
sage: GroupAlgebra(abelianGroup(1), IntegerModRing(10)).is_finite()
False
```

`linear_combination(iter_of_elements_coeff, factor_on_left=True)`

Return the linear combination \( \lambda_1 v_1 + \cdots + \lambda_k v_k \) (resp. the linear combination \( v_1 \lambda_1 + \cdots + v_k \lambda_k \)) where `iter_of_elements_coeff` iterates through the sequence \((\lambda_1, v_1), \ldots, (\lambda_k, v_k))\).

**INPUT:**

- `iter_of_elements_coeff` – iterator of pairs (element, coeff) with element in `self` and coeff in `self.base_ring()`
- `factor_on_left` – (optional) if True, the coefficients are multiplied from the left; if False, the coefficients are multiplied from the right

**EXAMPLES:**

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: J.linear_combination(((a+b, 1), (-2*b + c, -1)))
1 + (3, -1)
```

`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 \(\text{ModulesWithBasis}(R)\) with basis \(x = (x_i)_{i \in I}\).

Exactly one of the four following options must be specified in order to define the morphism:
• on\_basis – a function \(f\) from \(I\) to \(Y\)
• diagonal – a function \(d\) from \(I\) to \(R\)
• function – a function \(f\) from \(X\) to \(Y\)
• matrix – a matrix of size \(\dim Y \times \dim X\) (if the keyword \(\text{side}\) is set to 'left') or \(\dim Y \times \dim X\) (if this keyword is 'right')

Further options include:
• codomain – the codomain \(Y\) of the morphism (default: \(f.\text{codomain()}\) if it's defined; otherwise it must be specified)
• category – a category or None (default: \(\text{None}\))
• zero – the zero of the codomain (default: \(\text{codomain}.\text{zero()}\)); 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 \(\text{leading\_support()}\) of the image of the basis vector \(x_i\) is \(i\), or
  – "lower" - if the \(\text{trailing\_support()}\) 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 \(\text{unitriangular=}'\text{lower}'\) for triangular=’lower’, \(\text{unitriangular=}'\text{True}'\).
• 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:

\[
g \left( a, \sum_i \lambda_i x_i, c \right) = \sum_i \lambda_i f(a, i, c).
\]

By default, the category is the first of \(\text{Modules}(R).\text{WithBasis()}.\text{FiniteDimensional()}, \text{Modules}(R).\text{WithBasis()}, \text{Modules}(R), \text{and CommutativeAdditiveMonoids()}\) that contains both the domain and the codomain:
In this special case, the default category is `Sets()`:

```python
sage: phi.category_for()
Category of sets
```

One can construct morphisms with the base ring as codomain:

```python
sage: X = CombinatorialFreeModule(ZZ, [1, -1])
```

```python
sage: phi = X.module_morphism( on_basis=lambda i: i, codomain=ZZ )
```

```python
sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))
-1
```

```python
sage: phi.category_for()
Category of commutative additive semigroups
```

```
# todo: not implemented (ZZ is currently not in Modules(ZZ))
```

```
# todo: not implemented (RR is currently not in Modules(ZZ))
```

```python
sage: phi = X.module_morphism(on_basis=lambda i: i, codomain=Zmod(4))
```

```python
sage: phi(2 * X.monomial(1) + 3 * X.monomial(-1))
3
```

```
Traceback (most recent call last):
...
ValueError: codomain(=Ring of integers modulo 4) should be a module over
the base ring of the domain(=Y)
```

On can also define module morphisms between free modules over different base rings; here we implement the natural map from \(X = \mathbb{R}^2\) to \(Y = \mathbb{C}\):

```python
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 )
```

(continues on next page)
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 module over the base ring of the domain(=Free module generated by {'x', 'y'} over Real Field with 53 bits of precision)

sage: Y = CombinatorialFreeModule(RR['q'], ['z'])
sage: phi = Y.module_morphism(on_basis=on_basis, codomain=X)
Traceback (most recent call last):
  ... 
ValueError: codomain(=Free module generated by {'x', 'y'} over Real Field with 53 bits of precision) should be a module over the base ring of the domain(=Free module generated by {'z'} over Univariate Polynomial Ring in q over Real Field with 53 bits of precision)

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 $m$:

```python
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])
```

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:

```python
sage: I = list(range(1, 200))
sage: X = CombinatorialFreeModule(QQ, I); X.rename("X"); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, I); Y.rename("Y"); y = Y.basis()
sage: f = Y.sum_of_monomials * divisors
sage: phi = X.module_morphism(f, triangular="upper", codomain = Y)
sage: phi(x[2])
sage: phi(x[30])
sage: phi.preimage(y[2])
sage: phi.preimage(y[6])
sage: phi.preimage(y[30])
sage: (phi^-1)(y[30])
```

Since trac ticket #8678, one can also define a triangular morphism from a function:

```python
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])
sage: phi.preimage(_)
```

For details and further optional arguments, see `sage.modules.with_basis.morphism. _TriangularModuleMorphism`.

**Warning:** As a temporary measure, until multivariate morphisms are implemented, the constructed morphism is in $\text{Hom}(\text{codomain}, \text{domain}, \text{category})$. This is only correct for unary functions.
Todo:
• Should codomain be `self` by default in the diagonal, triangular, and matrix cases?
• Support for diagonal morphisms between modules not sharing the same index set

```python
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`#
```

```python
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
```

```python
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
```

`monomial(i)`
Return the basis element indexed by i.

**INPUT:**
• i—an element of the index set

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial('a')
B['a']
```

F.monomial is in fact (almost) a map:

```python
sage: F.monomial
Term map from {'a', 'b', 'c'} to Free module generated by {'a', 'b', 'c'} over Rational Field
```

`monomial_or_zero_if_none(i)`

**EXAMPLES:**

```python
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.monomial_or_zero_if_none('a')
B['a']
sage: F.monomial_or_zero_if_none(None)
0
```

`random_element(n=2)`
Return a ‘random’ element of `self`.

**INPUT:**
• n—integer (default: 2); number of summands
ALGORITHM:

Return a sum of \( n \) terms, each of which is formed by multiplying a random element of the base ring by a random element of the group.

EXAMPLES:

```
sage: DihedralGroup(6).algebra(QQ).random_element()
-1/95*(()) - 1/2*(1,4)(2,5)(3,6)
```

Note, this result can depend on the PRNG state in libgap in a way that depends on which packages are loaded, so we must re-seed GAP to ensure a consistent result for this example:

```
sage: libgap.set_seed(0)
sage: SU(2, 13).algebra(QQ).random_element()
1/2*[ 1 9*a + 2]
[2*a + 12 2]
sage: CombinatorialFreeModule(ZZ, Partitions(4)).random_element()  # random
2*B[[2, 1, 1]] + B[[2, 2]]
```

```
submodule(gens, check=True, already_echelonized=False, unitriangular=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
- `unitriangular` – (default: `False`) whether the lift morphism is unitriangular

If `already_echelonized` is `False`, then the generators are put in reduced echelon form using `echelonize()`, and reindexed by 0, 1, ....

**Warning:** At this point, this method only works for finite dimensional submodules and if matrices can be echelonized over the base ring.

If in addition `unitriangular` is `True`, then the generators are made such that the coefficients of the pivots are 1, so that lifting map is unitriangular.

The basis of the submodule uses the same index set as the generators, and the lifting map sends \( y_i \) to \( gens[i] \).

See also:

- `ModulesWithBasis.FiniteDimensional.ParentMethods.quotient_module()`
- `sage.modules.with_basis.subquotient.SubmoduleWithBasis`

EXAMPLES:

We construct a submodule of the free \( \mathbb{Q} \)-module generated by \( x_0, x_1, x_2 \). The submodule is spanned by \( y_0 = x_0 - x_1 \) and \( y_1 = x_1 - x_2 \), and its basis elements are indexed by 0 and 1:

```
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]]
```

(continues on next page)
By using a family to specify a basis of the submodule, we obtain a submodule whose index set coincides with the index set of the family:

```python
sage: X = CombinatorialFreeModule(QQ, range(3), prefix="x")
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=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 $S_3$:

```python
sage: S3 = SymmetricGroup(3)
```
sage: S3A = S3.algebra(QQ)
sage: basis = S3A.annihilator_basis(S3A.algebra_generators(), S3A.bracket)
sage: basis
\(((), (1,2,3) + (1,3,2), (2,3) + (1,2) + (1,3))\)
sage: center = S3A.submodule(basis,
....: category=AlgebrasWithBasis(QQ).Subobjects(),
....: already_echelonized=True)
sage: center
Free module generated by \{0, 1, 2\} over Rational Field
sage: center in Algebras
True
sage: center.print_options(prefix='c')
sage: c = center.basis()
sage: c[1].lift()
\((1,2,3) + (1,3,2)\)
sage: c[0]^2
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()
```

We can also automatically construct a basis such that the lift morphism is (lower) unitriangular:

```
sage: R.<a,b> = QQ[]
sage: C = CombinatorialFreeModule(R, range(3), prefix='x')
sage: x = C.basis()
sage: gens = [x[0] - x[1], 2*x[1] - 2*x[2], x[0] - x[2]]
sage: Y = C.submodule(gens, unitriangular=True)
sage: Y.lift.matrix()
\[
\begin{bmatrix}
1 & 0 \\
0 & 1 \\
-1 & -1
\end{bmatrix}
\]
```

**sum_of_monomials()**
Return the sum of the basis elements with indices in `indices`.

**INPUT:**
- `indices` – an list (or iterable) of indices of basis elements

**EXAMPLES:**

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.sum_of_monomials(['a', 'b', 'c'])
B['a'] + B['b']
sage: F.sum_of_monomials(['a', 'b'])
2*B['a'] + B['b']
```

`F.sum_of_monomials` is in fact (almost) a map:

```
sage: F.sum_of_monomials
A map to Free module generated by \{\text{a}', \text{b}', \text{c}'\} over Rational Field
```

**sum_of_terms**

```
Construct a sum of terms of self.

**INPUT:**
- `terms` – a list (or iterable) of pairs (index, coeff)

**OUTPUT:**
Sum of `coeff * B[index]` over all (index, coeff) in `terms`, where `B` is the basis of self.

**EXAMPLES:**

```python
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: J.sum_of_terms([(0, 2), (2, -3)])
2 + (0, -3)
```

**tensor** (*parents*)
Return the tensor product of the parents.

**EXAMPLES:**

```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A")
sage: A.tensor(A,A)
A # A # A
sage: A.rename(None)
```

**term** *(index, coeff=None)*
Construct a term in self.

**INPUT:**
- `index` – the index of a basis element
- `coeff` – an element of the coefficient ring (default: one)

**OUTPUT:**
`coeff * B[index]`, where `B` is the basis of self.

**EXAMPLES:**

```python
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: J.term(1, -2)
0 + (-2, 0)
```

Design: should this do coercion on the coefficient ring?

**Super**
alias of `sage.categories.super_modules_with_basis.SuperModulesWithBasis`

class **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
• \text{codomain} – the codomain of \( f \) (optional)

By the universal property of the tensor product, \( f \) induces a linear morphism from \( \text{self.parent()} \) to the target module. Returns the result of applying that morphism to \( \text{self} \).

The codomain is used for optimizations purposes only. If it’s not provided, it’s recovered by calling \( f \) on the zero input.

**EXAMPLES:**

We start with simple (admittedly not so interesting) examples, with two modules \( A \) and \( B \):

```python
sage: 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 \( B \otimes A \):

```python
sage: def f(a,b):
    ....:     return tensor([b,a])
```

Now, calling applying \( f \) on \( a \otimes b \) returns the same as \( f(a, b) \):

```python
sage: a = A.monomial(1) + 2 * A.monomial(2); a
sage: b = B.monomial(3) - 2 * B.monomial(4); b
sage: f(a,b)
sage: tensor([a,b]).apply_multilinear_morphism(f)
```

\( f \) may be a bilinear morphism to any module over the base ring of \( A \) and \( B \). Here the codomain is \( \mathbb{Z} \):

```python
sage: def f(a,b):
    ....:     return sum(a.coefficients(), 0) * sum(b.coefficients(), 0)
sage: f(a,b)
-3
sage: type(f(A.zero(), B.zero()))
<... 'int'>
```

Mind the 0 in the sums above; otherwise \( f \) would not return 0 in \( \mathbb{Z} \):

```python
sage: def f(a,b):
    ....:     return sum(a.coefficients()) * sum(b.coefficients())
sage: type(f(A.zero(), B.zero()))
<... 'int'>
```

Which would be wrong and break this method:

```python
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 ParentMethods

    Implements operations on tensor products of modules with basis.

    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]

    is_abelian()

    Return whether this category is abelian.

    This is the case if and only if the base ring is a field.

    EXAMPLES:

    sage: ModulesWithBasis(QQ).is_abelian()
    True
    sage: ModulesWithBasis(ZZ).is_abelian()
    False
3.113 Monoid algebras

sage.categories.monoid_algebras.MonoidAlgebras\( (\text{base\_ring}) \)

The category of monoid algebras over \( \text{base\_ring} \)

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

This is just an alias for:

\begin{verbatim}
sage: C is Monoids().Algebras(QQ)
True
\end{verbatim}

3.114 Monoids

class sage.categories.monoids.Monoids\( (\text{base\_category}) \)

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 \( \ast \) which is associative and admits a unit (see Wikipedia article Monoid).

EXAMPLES:

\begin{verbatim}
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')

sage: S = Monoids().example()
sage: x = S("aa")
sage: x^0, x^1, x^2, x^3, x^4, x^5
('', 'aa', 'aaaa', 'aaaaaa', 'aaaaaaaa', 'aaaaaaaaaa')
\end{verbatim}
class Algebras(category, *args)
    Bases: sage.categories.algebra_functor.AlgebrasCategory

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
    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 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', 'b': 'b', 'c': 'c', 'd': 'd'}
    sage: M.algebra(ZZ).algebra_generators()
    Finite family {'a': B['a'], 'b': B['b'], 'c': B['c'], 'd': B['d']}
    sage: Z12 = Monoids().Finite().example(); Z12
    An example of a finite multiplicative monoid:
    the integers modulo 12
    sage: Z12.monoid_generators()
    Traceback (most recent call last):
    ... AttributeError: 'IntegerModMonoid_with_category' object has no attribute 'monoid_generators'
    sage: Z12.semigroup_generators()
    Family (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
    sage: Z12.algebra(QQ).algebra_generators()

    (continues on next page)
sage: GroupAlgebras(QQ).example(AlternatingGroup(10)).algebra_.generators()
Finite family {0: (8,9,10), 1: (1,2,3,4,5,6,7,8,9)}

sage: A = DihedralGroup(3).algebra(QQ); A
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
sage: A.algebra_generators()
Finite family {0: (1,2,3), 1: (1,3)}

one_basis()
Return the unit of the monoid, which indexes the unit of this algebra, as per
AlgebrasWithBasis.ParentMethods.one_basis().

EXAMPLES:

sage: A = Monoids().example().algebra(ZZ)
sage: A.one_basis()
'

sage: A.one()
B['']
sage: A(3)
3*B['']

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 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 {...}}
```

```python
extra_super_categories()
```

A Cartesian product of monoids is endowed with a natural group structure.

EXAMPLES:

```
sage: C = Monoids().CartesianProducts()
sage: C.extra_super_categories()
[Category of monoids]
sage: sorted(C.super_categories(), key=str)
[Category of Cartesian products of semigroups,
 Category of Cartesian products of unital magmas,
 Category of monoids]
```

class Commutative(base_category)

Bases: `sage.categories.category_with_axiom.CategoryWithAxiom_singleton`

Category of commutative (abelian) monoids.

A monoid $M$ is **commutative** if $xy = yx$ for all $x, y \in M$.

```python
static free(index_set=None, names=None, **kwds)
```

Return a free abelian monoid on $n$ generators or with the generators indexed by a set $I$.

A free monoid is constructed by specifying either:
- the number of generators and/or the names of the generators, or
- the indexing set for the generators.

**INPUT:**
- `index_set` – (optional) an index set for the generators; if an integer, then this represents \{0, 1, ..., 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: F.<x,y,z> = Monoids().Commutative().free(); F
Free abelian monoid indexed by ('x', 'y', 'z')
```

class ElementMethods

```python
is_one()
```

Return whether `self` is the one of the monoid.

The default implementation is to compare with `self.one()`.

```python
powers(n)
```

Return the list $[x^0, x^1, ..., x^{n-1}]$.

**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]]

Finite
alias of sage.categories.finite_monoids.FiniteMonoids

Inverse
alias of sage.categories.groups.Groups

class ParentMethods

prod(args)

\text{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:
```python
sage: S = R.subsemigroup([R(3), R(5)]); S
A subsemigroup of (Ring of integers modulo 15) with 2 generators
sage: S.list()
[3, 5, 9, 0, 10, 12, 6]
```

This method is really a shorthand for subsemigroup:

```python
sage: M2 = R.subsemigroup([R(3), R(5)], one=R.one())
sage: M2 is M
True
```

class `Subquotients` (`category, *args`)

Bases: `sage.categories.subquotients.SubquotientsCategory`

class `ParentMethods`

```python
one()
Returns the multiplicative unit of this monoid, obtained by retracting that of the ambient monoid.

EXAMPLES:
```
```python
sage: S = Monoids().Subquotients().example()  # todo: not implemented
sage: S.one()  # todo: not implemented
```

class `WithRealizations` (`category, *args`)

Bases: `sage.categories.with_realizations.WithRealizationsCategory`

class `ParentMethods`

```python
one()
Return the unit of this monoid.

This default implementation returns the unit of the realization of self given by
`a_realization()`.

EXAMPLES:
```
```python
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[{}]
```

static `free` (`index_set=None, names=None, **kwds`)
Return a free monoid on `n` generators or with the generators indexed by a set `I`.

A free monoid is constructed by specifying either:

- the number of generators and/or the names of the generators
- the indexing set for the generators

INPUT:

- `index_set` – (optional) an index set for the generators; if an integer, then this represents
  \{0, 1, \ldots, n - 1\}
- `names` – a string or list/tuple/iterable of strings (default: 'x'); the generator names or name prefix

568 Chapter 3. Individual Categories
3.115 Number fields

```
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')
```

The category of number fields.

```
EXAMPLES:

We create the category of number fields:

sage: C = NumberFields()
sage: C
Category of number fields

By definition, it is infinite:

sage: NumberFields().Infinite() is NumberFields()
True

Notice that the rational numbers \( \mathbb{Q} \) are considered as an object in this category:

sage: RationalField() in C
True

However, we can define a degree 1 extension of \( \mathbb{Q} \), which is of course also in this category:

sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: K = NumberField(x - 1, 'a'); K
Number Field in a with defining polynomial x - 1
sage: K in C
True

Number fields all lie in this category, regardless of the name of the variable:

sage: K = NumberField(x^2 + 1, 'a')
sage: K in C
True
```

```
class ElementMethods
class ParentMethods
class super_categories()

EXAMPLES:

sage: NumberFields().super_categories()
[Category of infinite fields]
```
3.116 Objects

```python
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()
    []
```

**class ParentMethods**
Methods for all category objects

**class 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
```

**Note:** Background
Information, code, documentation, and tests about the category of homsets of a category Cs should go in the nested class Cs.Homsets. They will then be made available to homsets of any subcategory of Cs.

Assume, for example, that homsets of Cs are Cs themselves. This information can be implemented in the method Cs.Homsets.extra_super_categories to make Cs.Homsets() a subcategory of Cs().

Methods about the homsets themselves should go in the nested class Cs.Homsets. ParentMethods.
Methods about the morphisms can go in the nested class `Cs.Homsets.ElementMethods`. However it’s generally preferable to put them in the nested class `Cs.MorphismMethods`; indeed they will then apply to morphisms of all subcategories of `Cs`, and not only full subcategories.

See also:

FunctorialConstruction

Todo:

• Design a mechanism to specify that an axiom is compatible with taking subsets. Examples: Finite, Associative, Commutative (when meaningful), but not Infinite nor Unital.

• Design a mechanism to specify that, when \(B\) is a subcategory of \(A\), a \(B\)-homset is a subset of the corresponding \(A\) homset. And use it to recover all the relevant axioms from homsets in super categories.

• For instances of redundant code due to this missing feature, see:
  
  – AdditiveMonoids.Homsets.extra_super_categories()
  – HomsetsCategory.extra_super_categories() (slightly different nature)
  – plus plenty of spots where this is not implemented.

**additional_structure()**

Return None

Indeed, by convention, the category of objects defines no additional structure.

See also:

Category.additional_structure()

**EXAMPLES:**

```
sage: Objects().additional_structure()
None
```

**super_categories()**

EXEMPLES:

```
sage: Objects().super_categories()
[]
```

### 3.117 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 \(x \ast z \leq y \ast z\) and \(z \ast x \leq z \ast y\).

See Wikipedia article Ordered_monoid

**EXAMPLES:**

```
sage: PartiallyOrderedMonoids()
Category of partially ordered monoids
sage: PartiallyOrderedMonoids().super_categories()
(Category of posets, Category of monoids)
```
3.118 Permutation groups

class sage.categories.permutation_groups.PermutationGroups(s=\texttt{\textit{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:

\begin{verbatim}
sage: PermutationGroups()
Category of permutation groups
sage: PermutationGroups().super_categories()
[Category of groups]
\end{verbatim}

The category of permutation groups defines additional structure that should be preserved by morphisms, namely the distinguished action:

\begin{verbatim}
sage: PermutationGroups().additional_structure()
Category of permutation groups
\end{verbatim}

**Finite**

alias of sage.categories.finite_permutation_groups.FinitePermutationGroups

super_categories()

Return a list of the immediate super categories of self.

EXAMPLES:

\begin{verbatim}
sage: PermutationGroups().super_categories()
[Category of groups]
\end{verbatim}

3.119 Pointed sets

class sage.categories.pointed_sets.PointedSets(s=\texttt{\textit{None}})
Bases: sage.categories.category_singleton.Category_singleton

...
The category of pointed sets.

EXAMPLES:

```
sage: PointedSets()
Category of pointed sets
```

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

### 3.120 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 \( \mathbb{Q} \):

```
sage: PolyhedralSets(QQ)
Category of polyhedral sets over Rational Field
```

```
sage: PolyhedralSets(QQ).super_categories()
[Category of commutative magmas, Category of additive monoids]
```

### 3.121 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 \texttt{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)
```

(continues on next page)
The other comparison methods are called \texttt{lt()}, \texttt{ge()}, \texttt{gt()}, following Python’s naming convention in \texttt{operator}. Default implementations are provided:

\begin{verbatim}
sage: P.lt(x, x)
False
sage: P.ge(y, x)
True
\end{verbatim}

Unless the poset is a facade (see \texttt{Sets.Facade}), one can compare directly its elements using the usual Python operators:

\begin{verbatim}
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
\end{verbatim}

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:

\begin{verbatim}
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
\end{verbatim}

See also:

\texttt{Poset()}, \texttt{FinitePosets}, \texttt{LatticePosets}

\textbf{class} \texttt{ElementMethods}

\textbf{Finite}

\begin{verbatim}
alias of \texttt{sage.categories.finite_posets.FinitePosets}
\end{verbatim}

\textbf{class} \texttt{ParentMethods}

\textbf{CartesianProduct}

\begin{verbatim}
alias of \texttt{sage.combinat.posets.cartesian_product.CartesianProductPoset}
\end{verbatim}

\textbf{directed_subset} \texttt{(elements, direction)}

Return the order filter or the order ideal generated by a list of elements.
If \texttt{direction} is ‘up’, the order filter (upper set) is being returned. If \texttt{direction} is ‘down’, the order ideal (lower set) is being returned.

**INPUT:**
- elements – a list of elements.
- direction – ‘up’ or ‘down’.

**EXAMPLES:**

```python
sage: B = posets.BooleanLattice(4)
sage: B.directed_subset([3, 8], 'up')
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: B.directed_subset([7, 10], 'down')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

**ge** \((x, y)\)

Return whether \(x \geq y\) in the poset \texttt{self}.

**INPUT:**
- \(x, y\) – elements of \texttt{self}.

This default implementation delegates the work to \texttt{le()}. 

**EXAMPLES:**

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.ge( 6, 3 )
True
sage: D.ge( 3, 3 )
True
sage: D.ge( 3, 5 )
False
```

**gt** \((x, y)\)

Return whether \(x > y\) in the poset \texttt{self}.

**INPUT:**
- \(x, y\) – elements of \texttt{self}.

This default implementation delegates the work to \texttt{lt()}. 

**EXAMPLES:**

```python
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 \texttt{self}.

**INPUT:**
- \(o\) – an iterable (e. g., list, set, or tuple) containing some elements of \texttt{self}

**OUTPUT:**

True if the subset of \texttt{self} consisting of the entries of \(o\) is an antichain of \texttt{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(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])), R(set([1, 4])),
R(set([4, 5]))))
False
**is_chain_of_poset** \((o, \text{ordered}=False)\)

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 \(\text{ordered}\) is set to \(True\).

**INPUT:**
- \(o\) – an iterable (e.g., list, set, or tuple) containing some elements of \(self\)
- \(\text{ordered}\) – a Boolean (default: \(False\)) which decides whether the notion of a chain includes being ordered

**OUTPUT:**

If \(\text{ordered}\) 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 \(\text{ordered}\) 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 \(\text{ordered}\) 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)
```

(continues on next page)
from sage.categories.examples.posets import FiniteSetsOrderedByInclusion

sage: R = FiniteSetsOrderedByInclusion()
sage: R.is_chain_of_poset([R(set([3, 1, 2])), R(set([1, 2])), R(set([1, 5]))], ordered=True)
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

from sage.categories.examples.posets import PositiveIntegersOrderedByDivisibilityFacade

sage: T = PositiveIntegersOrderedByDivisibilityFacade()
sage: T.is_chain_of_poset((T(3), T(4), T(7)))
False
sage: T.is_chain_of_poset((T(3), T(6), T(3)))
True
sage: T.is_chain_of_poset((T(3), T(6), T(3)), ordered=True)
False
sage: T.is_chain_of_poset((T(3), T(6)), ordered=True)
False
sage: T.is_chain_of_poset((T(3), T(6)), ordered=True, linear_extension=True)
True

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([])
False
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:**

```python
sage: P = Poset((divisors(12), attrcall("divides")), facade=True, linear_extension=True)
sage: sorted(P.list())
[1, 2, 3, 4, 6, 12]
sage: P.is_order_ideal([1, 3])
True
sage: P.is_order_ideal([])
True
sage: P.is_order_ideal({1, 3})
True
sage: P.is_order_ideal([1, 3, 4])
False
```

**le** *(x, y)*

Return whether \( x \leq y \) in the poset self.

**INPUT:**

- x, y - elements of self

**EXAMPLES:**

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.le( 3, 6 )
True
sage: D.le( 3, 3 )
True
sage: D.le( 3, 5 )
False
```

**lower_covers** *(x)*

Return the lower covers of \( x \), that is, the elements \( y \) such that \( y < x \) and there exists no \( z \) such that \( y < z < x \).

**EXAMPLES:**

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.lower_covers(15)
[3, 5]
```

**lt** *(x, y)*

Return whether \( x < y \) in the poset self.

**INPUT:**

- x, y - elements of self.

This default implementation delegates the work to **le**().
EXAMPLES:

```
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* ∉ *I* but every element of *P* smaller than *v* is in *I*;
- the set $I \setminus \{v\}$, if *v* ∈ *I* but no element of *P* greater than *v* is in *I*;
- *I* otherwise.

This image always is an order ideal of *P*.

EXAMPLES:

```
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)
```

(continues on next page)
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:

```python
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:

```python
sage: D = Poset((divisors(30), attrcall("divides")))
sage: D.upper_covers(3)
[6, 15]
```

**example** *(choice=None)*

Return examples of objects of *Posets()* as per *Category.example()*.

EXAMPLES:

```python
sage: Posets().example()
An example of a poset: sets ordered by inclusion
sage: Posets().example("facade")
An example of a facade poset: the positive integers ordered by divisibility
```

**super_categories** *

Return a list of the (immediate) super categories of *self*, as per *Category.super_categories()*.

EXAMPLES:

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

### 3.122 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:

```python
sage: PrincipalIdealDomains()
Category of principal ideal domains
sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]
```

See also *Wikipedia article Principal_ideal_domain*

**class** *ElementMethods*

**class** *ParentMethods*
additional_structure()

Return None.

Indeed, the category of principal ideal domains defines no additional structure: a ring morphism between two principal ideal domains is a principal ideal domain morphism.

EXAMPLES:

```
sage: PrincipalIdealDomains().additional_structure()
```

super_categories()

EXAMPLES:

```
sage: PrincipalIdealDomains().super_categories()
[Category of unique factorization domains]
```

### 3.123 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]
```

class ElementMethods

denominator()

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__'
```

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 trac 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)
\textbf{sage:} \quad \text{factor}(\gcd(p,q))
(x - 1)^{-1} \ast (x + 1) \ast (x^2 + 1)^{-1} \ast (x^4 + x^3 + x^2 + x + 1)^{-1}
\textbf{sage:} \quad \text{factor}(\gcd(p,1+x))
(x - 1)^{-1} \ast (x + 1) \ast (x^2 + 1)^{-1}
\textbf{sage:} \quad \text{factor}(\gcd(1+x,q))
(x - 1)^{-1} \ast (x + 1) \ast (x^2 + 1)^{-1}

\textbf{lcm}(\text{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).

\textbf{AUTHOR:}

- Simon King (2011-02): See trac ticket #10771

\textbf{EXAMPLES:}

\textbf{sage:} \quad \text{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.

\textbf{sage:} \quad \text{lcm}(1/3, 1/5) \quad \text{lcm}(1/3, 1/6) \quad 1/3

Some more involved examples:

\textbf{sage:} \quad R.<x> = QQ[]
\textbf{sage:} \quad p = (1+x)^3*(1+2*x^2)/(1-x^5)
\textbf{sage:} \quad q = (1+x)^2*(1+3*x^2)/(1-x^4)
\textbf{sage:} \quad \text{factor}(p)
(-2) \ast (x - 1)^{-1} \ast (x + 1)^{3} \ast (x^2 + 1/2) \ast (x^4 + x^3 + x^2 + x + 1)^{-1}
\textbf{sage:} \quad \text{factor}(q)
(-3) \ast (x - 1)^{-1} \ast (x + 1) \ast (x^2 + 1)^{-1} \ast (x^2 + 1/3)
\textbf{sage:} \quad \text{factor}(\text{lcm}(p,q))
(x - 1)^{-1} \ast (x + 1)^{3} \ast (x^2 + 1/3) \ast (x^2 + 1/2)
\textbf{sage:} \quad \text{factor}(\text{lcm}(p,1+x))
(x + 1)^{3} \ast (x^2 + 1/2)
\textbf{sage:} \quad \text{factor}(\text{lcm}(1+x,q))
(x + 1) \ast (x^2 + 1/3)

\textbf{numerator}()

Constructor for abstract methods

\textbf{EXAMPLES:}

\textbf{sage:} \quad \text{def } f(x):
...: \quad "doc of f"
...: \quad \text{return } 1
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:

```python
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^5 + 2/3)/(t^2 - t + 1), 2/(t^2 + 2)]
sage: sum(parts) == q
True
sage: q = 2*t / (t + 3)^2
sage: q.partial_fraction_decomposition()
(0, [2/(t + 3), -6/(t^2 + 6*t + 9)])
sage: for p in q.partial_fraction_decomposition()[1]: print(p.factor())
(2) * (t + 3)^-1
(-6) * (t + 3)^-2
sage: q.partial_fraction_decomposition(decompose_powers=False)
(0, [2*t/(t^2 + 6*t + 9)])
```

We can decompose over a given algebraic extension:

```python
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)]
```
We can also ask Sage to find the least extension where the denominator factors in linear terms:

```sage
r = 1/(x^4+2)
r = 1/(x^4+2)
N = r.denominator().splitting_field('a')
N
Number Field in a with defining polynomial x^8 - 8*x^6 + 28*x^4 + 16*x^2 + 36

R1.<x1>=N[]
r1 = 1/(x1^4+2)
r1.partial_fraction_decomposition()
```

Or we may work directly over an algebraically closed field:

```sage
R.<x> = QQbar[]
r = 1/(x^4+1)
r.partial_fraction_decomposition()
```

We do the best we can over inexact fields:

```sage
R.<x> = RealField(20)[]
q = 1/(x^2 + x + 2)^2 + 1/(x-1); q
q.partial_fraction_decomposition(); parts
```

**xgcd**

Return a triple \((g, s, t)\) of elements of that field such that \(g\) is the greatest common divisor of \(self\) and \(other\) and \(g = s*\text{self} + t*\text{other}\).
Note: In a field, the greatest common divisor is not very informative, as it is only determined up to a unit. But in the fraction field of an integral domain that provides both \texttt{xgcd} and \texttt{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:

\begin{verbatim}
 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

 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)
\end{verbatim}

An example without a well defined gcd or xgcd on its base ring:

\begin{verbatim}
 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)
\end{verbatim}

class ParentMethods

super_categories()

EXAMPLES:

\begin{verbatim}
 sage: QuotientFields().super_categories()
 [Category of fields]
\end{verbatim}
3.124 Quantum Group Representations

AUTHORS:
• Travis Scrimshaw (2018): initial version

class sage.categories.quantum_group_representations.QuantumGroupRepresentations(base, name=None):
    Bases: sage.categories.category_types.Category_module

    The category of quantum group representations.

class ParentMethods

    cartan_type()
    Return the Cartan type of self.

    EXAMPLES:

    sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
    sage: C = crystals.Tableaux(['C',4], shape=[1])
    sage: R = ZZ['q'].fraction_field()
    sage: V = MinusculeRepresentation(R, C)
    sage: V.cartan_type()
    ['C', 4]

    index_set()
    Return the index set of self.

    EXAMPLES:

    sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
    sage: C = crystals.Tableaux(['C',4], shape=[1])
    sage: R = ZZ['q'].fraction_field()
    sage: V = MinusculeRepresentation(R, C)
    sage: V.index_set()
    (1, 2, 3, 4)

    q()
    Return the quantum parameter $q$ of self.

    EXAMPLES:

    sage: from sage.algebras.quantum_groups.representations import MinusculeRepresentation
    sage: C = crystals.Tableaux(['C',4], shape=[1])
    sage: R = ZZ['q'].fraction_field()
    sage: V = MinusculeRepresentation(R, C)
    sage: V.q()
    q

class TensorProducts(category, *args)
    Bases: sage.categories.tensor.TensorProductsCategory

    The category of quantum group representations constructed by tensor product of quantum group rep-
Warning: We use the reversed coproduct in order to match the tensor product rule on crystals.

class ParentMethods

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.representations import _
˓→MinusculeRepresentation
sage: C = crystals.Tableaux(['C',2], shape=[1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculeRepresentation(R, C)
sage: T = tensor([V,V])
sage: T.cartan_type()
['C', 2]
```

eextra_super_categories()

EXAMPLES:

```
sage: from sage.categories.quantum_group_representations import _
˓→QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.TensorProducts().extra_super_categories()
[Category of quantum group representations over 
  Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

class WithBasis (base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

The category of quantum group representations with a distinguished basis.

class ElementMethods

K (i, power=1)

Return the action of $K_i$ on self to the power power.

INPUT:

• $i$ – an element of the index set
• $power$ – (default: 1) the power of $K_i$

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.representations import _
˓→AdjointRepresentation
sage: K = crystals.KirillovReshetikhin(['D',4,2], 1,1)
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, K)
sage: v = V.an_element(); v
2*B[[]] + 2*B[[[1]]] + 3*B[[[2]]]
sage: v.K(0)
2*B[[]] + 2/q^2*B[[[1]]] + 3*B[[[2]]]
sage: v.K(1)
2*B[[]] + 2*q^2*B[[[1]]] + 3/q^2*B[[[2]]]
sage: v.K(1, 2)
2*B[[]] + 2*q^4*B[[[1]]] + 3/q^4*B[[[2]]]
```

(continues on next page)
\textbf{\texttt{sage: v.K(1, -1)}}
\begin{verbatim}
2*B[[]] + 2/q^2*B[[[1]]] + 3*q^2*B[[[2]]]
\end{verbatim}

\textbf{\texttt{e(i)}}

Return the action of $e_i$ on \texttt{self}.

**INPUT:**

- \texttt{i} – an element of the index set

**EXAMPLES:**

\begin{verbatim}
sage: from sage.algebras.quantum_groups.representations import ....: MinusculeRepresentation, AdjointRepresentation
  sage: C = crystals.Tableaux(['G',2], shape=[1,1])
  sage: R = ZZ['q'].fraction_field()
  sage: V = AdjointRepresentation(R, C)
  sage: v = V.an_element(); v
  2*B[[]] + 2*B[[[1], [2]]] + 3*B[[[1], [3]]] + 3*B[[[2], [3]]]
  sage: v.e(1)
  ((3*q^4+3*q^2+3)/q^2)*B[[[1], [3]]]
  sage: v.e(2)
  2*B[[[1], [2]]]
\end{verbatim}

\textbf{\texttt{f(i)}}

Return the action of $f_i$ on \texttt{self}.

**INPUT:**

- \texttt{i} – an element of the index set

**EXAMPLES:**

\begin{verbatim}
sage: from sage.algebras.quantum_groups.representations import ....: MinusculeRepresentation, AdjointRepresentation
  sage: K = crystals.KirillovReshetikhin(['D',4,1], 2,1)
  sage: R = ZZ['q'].fraction_field()
  sage: V = AdjointRepresentation(R, K)
  sage: v = V.an_element(); v
  2*B[[]] + 2*B[[[1], [2]]] + 3*B[[[1], [3]]] + 3*B[[[2], [3]]]
  sage: v.f(0)
  ((2*q^2+2)/q)*B[[[1], [2]]]
  sage: v.f(1)
  3*B[[[2], [3]]]
  sage: v.f(2)
  2*B[[[1], [3]]]
  sage: v.f(3)
  3*B[[[1], [4]]]
  sage: v.f(4)
  3*B[[[1], [-4]]]
\end{verbatim}

\textbf{\texttt{class ParentMethods}}

\textbf{\texttt{tensor(*factors)}}

Return the tensor product of \texttt{self} with the representations \texttt{factors}.

**EXAMPLES:**

\begin{verbatim}
sage: from sage.algebras.quantum_groups.representations import ....: MinusculeRepresentation, AdjointRepresentation
\end{verbatim}
```python
sage: R = ZZ['q'].fraction_field()
sage: CM = crystals.Tableaux(['D',4], shape=[1])
sage: CA = crystals.Tableaux(['D',4], shape=[1,1])
sage: V = MinusculeRepresentation(R, CM)
sage: V.tensor(V, V)
V((1, 0, 0, 0)) # V((1, 0, 0, 0)) # V((1, 0, 0, 0))
sage: A = MinusculeRepresentation(R, CA)
sage: V.tensor(A)
V((1, 0, 0, 0)) # V((1, 1, 0, 0))
sage: B = crystals.Tableaux(['A',2], shape=[1])
sage: W = MinusculeRepresentation(R, B)
sage: tensor([W,V])
Traceback (most recent call last):
  ... ValueError: all factors must be of the same Cartan type
sage: tensor([V,A,W])
Traceback (most recent call last):
  ... ValueError: all factors must be of the same Cartan type
```

```python
class TensorProducts(category, *args)
    Bases: sage.categories.tensor.TensorProductsCategory

The category of quantum group representations with a distinguished basis constructed by tensor product of quantum group representations with a distinguished basis.

class ParentMethods

K_on_basis (i, b, power=1)
    Return the action of $K_i$ on the basis element indexed by $b$ to the power $power$.

    INPUT:
    • $i$ – an element of the index set
    • $b$ – an element of basis keys
    • $power$ – (default: 1) the power of $K_i$

    EXAMPLES:
```
```
e_on_basis \((i, b)\)
Return the action of \(e_i\) on the basis element indexed by \(b\).

INPUT:
• \(i\) – an element of the index set
• \(b\) – an element of basis keys

EXAMPLES:

```python
sage: from sage.algebras.quantum_groups.representations import \\
....: MinusculeRepresentation, AdjointRepresentation
sage: R = ZZ['q'].fraction_field()
sage: CM = crystals.Tableaux(['D',4], shape=[1])
sage: VM = MinusculeRepresentation(R, CM)
sage: CA = crystals.Tableaux(['D',4], shape=[1,1])
sage: VA = AdjointRepresentation(R, CA)
sage: v = tensor([VM.an_element(), VA.an_element()]); v
4*B[[[1]]] # B[[[1], [2]]] + 4*B[[[1]]] # B[[[1], [3]]]
+ 6*B[[[1]]] # B[[[2], [3]]] + 6*B[[[2]]] # B[[[1], [3]]]
+ 6*B[[[2]]] # B[[[1], [3]]] + 6*B[[[2]]] # B[[[1], [2]]]
+ 9*B[[[1]]] # B[[[2], [3]]]
sage: v.e(1) # indirect doctest
4*B[[[1]]] # B[[[1], [2]]]
+ (4*q+6)/q*B[[[1]]] # B[[[1], [3]]]
+ 6*B[[[1]]] # B[[[2], [3]]]
+ 6*q*B[[[2]]] # B[[[1], [3]]]
+ 9*B[[[1]]] # B[[[2], [3]]]
sage: v.e(2) # indirect doctest
4*B[[[1]]] # B[[[1], [2]]]
+ (6*q+4)/q*B[[[2]]] # B[[[1], [2]]]
+ 6*B[[[2]]] # B[[[1], [3]]]
+ 9*B[[[2]]] # B[[[1], [3]]]
+ 6*q*B[[[3]]] # B[[[1], [2]]]
sage: v.e(3) # indirect doctest
0
sage: v.e(4) # indirect doctest
0
```

f_on_basis \((i, b)\)
Return the action of \(f_i\) on the basis element indexed by \(b\).

INPUT:
• \(i\) – an element of the index set
• \(b\) – an element of basis keys

EXAMPLES:

```python
sage: from sage.algebras.quantum_groups.representations import \\
....: MinusculeRepresentation, AdjointRepresentation
sage: R = ZZ['q'].fraction_field()
sage: KM = crystals.KirillovReshetikhin(['B',3,1], 3,1)
sage: VM = MinusculeRepresentation(R, KM)
sage: KA = crystals.KirillovReshetikhin(['B',3,1], 2,1)
sage: VA = AdjointRepresentation(R, KA)
sage: v = tensor([VM.an_element(), VA.an_element()]); v
4*B[[++, [1]]] # B[[[1]]] + 4*B[[++, [1]]] # B[[[1], [2]]]
+ 6*B[[++, [1]]] # B[[[1], [3]]] + 4*B[[++, [1]]] # B[[[1]]]
+ 4*B[[++, [1]]] # B[[[1], [2]]]
+ 6*B[[++, [1]]] # B[[[1], [3]]] + 6*B[[++, [1]]] # B[[[1]]]
```
(continues on next page)
extra_super_categories()

EXAMPLES:

```python
sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.WithBasis().TensorProducts().extra_super_categories()
[Category of quantum group representations with basis over
 Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

example()

Return an example of a quantum group representation as per `Category.example`.

EXAMPLES:

```python
sage: from sage.categories.quantum_group_representations import QuantumGroupRepresentations
sage: Cat = QuantumGroupRepresentations(ZZ['q'].fraction_field())
sage: Cat.example()
V((2, 1, 0))
```

super_categories()

Return the super categories of `self`.

EXAMPLES:
3.125 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 \( \varphi_i \) or \( \varepsilon_i \)) holds, these crystals are sometimes called seminormal or semiregular.

EXAMPLES:

```python
sage: C = RegularCrystals()
sage: C
Category of regular crystals
sage: C.super_categories()
[Category of crystals]
sage: C.example()
Highest weight crystal of type A_3 of highest weight omega_1
```

class ElementMethods

demazure_operator_simple(i, ring=None)

Return the Demazure operator \( D_i \) applied to \( \text{self} \).

INPUT:

- \( i \) – an element of the index set of the underlying crystal
- \( ring \) – (default: \( \mathbb{Q} \)) a ring

OUTPUT:

An element of the ring-free module indexed by the underlying crystal.

Let \( r = (\text{wt}(b), \alpha_i^\vee) \), then \( D_i(b) \) is defined as follows:

- If \( r \geq 0 \), this returns the sum of the elements obtained from \( \text{self} \) by application of \( f_i^k \) for \( 0 \leq k \leq r \).
- If \( r < 0 \), this returns the opposite of the sum of the elements obtained by application of \( e_i^k \) for \( 0 < k < -r \).

REFERENCES:

- [Li1995]
- [Ka1993]

EXAMPLES:
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t = T(rows=[[1,2],[2]])
sage: t.demazure_operator_simple(2)
B([[1, 2], [2]]) + B([[1, 3], [2]]) + B([[1, 3], [3]])
sage: t.demazure_operator_simple(2).parent()
Algebra of The crystal of tableaux of type ['A', 2] and shape(s) [[2, 1]]
  over Integer Ring
sage: t.demazure_operator_simple(1)
0
sage: K = crystals.KirillovReshetikhin(['A',2,1],2,1)
sage: t = K(rows=[[3],[2]])
sage: t.demazure_operator_simple(0)
B([[1, 2]]) + B([[2, 3]])

dual_equivalence_class (index_set=None)

Return the dual equivalence class indexed by index_set of self.

The dual equivalence class of an element \( b \in B \) is the set of all elements of \( B \) reachable from \( b \) via sequences of \( i \)-elementary dual equivalence relations (i.e., \( i \)-elementary dual equivalence transformations and their inverses) for \( i \) in the index set of \( B \).

For this to be well-defined, the element \( b \) has to be of weight 0 with respect to \( I \); that is, we need to have \( \varepsilon_j(b) = \varphi_j(b) \) for all \( j \in I \).

See [As2008]. See also dual_equivalence_graph() for a definition of \( i \)-elementary dual equivalence transformations.

INPUT:
- index_set – (optional) the index set \( I \) (default: the whole index set of the crystal); this has to be a subset of the index set of the crystal (as a list or tuple)

OUTPUT:
The dual equivalence class of self indexed by the subset index_set. This class is returned as an undirected edge-colored multigraph. The color of an edge is the index \( i \) of the dual equivalence relation it encodes.

See also:
- dual_equivalence_graph()
- sage.combinat.partition.Partition.dual_equivalence_graph()

EXAMPLES:

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),
([1, 2], [3, 4]), ([1, 3], [2, 4]), 4)

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, 2, 4], [3, 4, 5]), 2),
([1, 3, 5], [2, 4]), ([1, 2, 4], [3, 4, 5]), 3),
([1, 2, 4], [3, 4, 5]), ([1, 2, 4], [3, 4, 5]), 4),
([1, 2, 4], [3, 4, 5]), ([1, 2, 4], [3, 4, 5]), 5),
([1, 2, 4], [3, 4, 5]), ([1, 2, 4], [3, 4, 5]), 6)],
epsilon \((i)\)
Return \(\varepsilon_i\) of self.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```

phi \((i)\)
Return \(\varphi_i\) of self.

EXAMPLES:

```python
sage: C = crystals.Letters(['A', 5])
sage: C(1).phi(1)
1
sage: C(2).phi(1)
0
```

stembridgeDel_depth \((i,j)\)
Return the difference in the \(j\)-depth of self and \(f_i\) of self, where \(i\) and \(j\) are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The \(i\)-depth of a crystal node \(x\) is \(\varepsilon_i(x)\).

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: t = T(rows=[[1, 1], [2]])
sage: t.stembridgeDel_depth(1, 2)
0
sage: s = T(rows=[[1, 3], [3]])
sage: s.stembridgeDel_depth(1, 2)
-1
```

stembridgeDel_rise \((i,j)\)
Return the difference in the \(j\)-rise of self and \(f_i\) of self, where \(i\) and \(j\) are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The \(i\)-rise of a crystal node \(x\) is \(\varphi_i(x)\).

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: t = T(rows=[[1, 1], [2]])
sage: t.stembridgeDel_rise(1, 2)
-1
sage: s = T(rows=[[1, 3], [3]])
sage: s.stembridgeDel_rise(1, 2)
0
```

stembridgeDelta_depth \((i,j)\)
Return the difference in the \(j\)-depth of self and \(\varepsilon_i\) of self, where \(i\) and \(j\) are in the index set of the underlying crystal. This function is useful for checking the Stembridge local axioms for crystal bases.

The \(i\)-depth of a crystal node \(x\) is \(-\varepsilon_i(x)\).

EXAMPLES:
```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t=T(rows=[[1,2],[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:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t=T(rows=[[1,1],[2]])
sage: t.stembridgeDelta_rise(1,2)
sage: s=T(rows=[[2,3],[3]])
sage: s.stembridgeDelta_rise(1,2)

-1

stembridgeTriple(i,j)
Let A be the Cartan matrix of the crystal, x a crystal element, and let i and j be in the index set of the crystal. Further, set b=stembridgeDelta_depth(x,i,j), and c=stembridgeDelta_rise(x,i,j). If x.e(i) is non-empty, this function returns the triple (A_{ij}, b, c); otherwise it returns None. By the Stembridge local characterization of crystal bases, one should have A_{ij} = b + c.

EXAMPLES:

```python
sage: T = crystals.Tableaux(['A',2], shape=[2,1])
sage: t=T(rows=[[1,1],[2]])
sage: t.stembridgeTriple(1,2)
sage: s=T(rows=[[1,2],[2]])
sage: s.stembridgeTriple(1,2)

(-1, 0, -1)

weight()
Return the weight of this crystal element.

EXAMPLES:

```python
sage: C = crystals.Letters(['A',5])
sage: C(1).weight()
```
class MorphismMethods

is_isomorphism()
Check if self is a crystal isomorphism, which is true if and only if this is a strict embedding with
the same number of connected components.

EXAMPLES:

```
sage: La = RootSystem(['A', 2, 1]).weight_space(extended=True).fundamental_weights()
sage: B = crystals.LSPaths(La[0])
sage: La = RootSystem(['A', 2, 1]).weight_lattice(extended=True) .fundamental_weights()
sage: C = crystals.GeneralizedYoungWalls(2, La[0])
sage: H = Hom(B, C)
sage: from sage.categories.highest_weight_crystals import HighestWeightCrystalMorphism
sage: class Psi(HighestWeightCrystalMorphism):
    ....: def is_strict(self):
    ....:     return True
sage: psi = Psi(H, C.module_generators)
sage: psi
['A', 2, 1] Crystal morphism:
  From: The crystal of LS paths of type ['A', 2, 1] and weight Lambda[0]
  To: Highest weight crystal of generalized Young walls of Cartan type ['A', 2, 1]
  and highest weight Lambda[0]
  Defn: (Lambda[0],) |--> []
sage: psi.is_isomorphism()
True
```

class 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, 0, -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[[[2, 2, 2, 2]]]
sage: e == T.demazure_operator(e,[1])
True
sage: all(T.demazure_operator(T.demazure_operator(C(t),[1]),[1]) == T.demazure_operator(C(t),[1]) for t in T)
True

demazure_subcrystal (element, reduced_word, only_support=True)

Return the subcrystal corresponding to the application of Demazure operators $D_i$ for $i$ from
reduced_word on element.

INPUT:

• element – an element of a free module indexed by the underlying crystal
• reduced_word – a reduced word of the Weyl group of the same type as the underlying crystal
• only_support – (default: True) only include arrows corresponding to the support of reduced_word

OUTPUT:

• the Demazure subcrystal

EXAMPLES:

```python
sage: T = crystals.Tableaux(["A",2], shape=[2,1])
sage: t = T.highest_weight_vector()
sage: S = T.demazure_subcrystal(t, [1,2])
sage: list(S)
[[[1, 1], [2]], [[1, 1], [3]], [[1, 2], [2]], [[1, 2], [3]], [[2, 2], [3]]]
sage: S = T.demazure_subcrystal(t, [2,1])
sage: list(S)
[[[1, 1], [2]], [[1, 1], [3]], [[1, 2], [2]], [[1, 3], [2]], [[1, 3], [3]]]
```

We construct an example where we don’t only want the arrows indicated by the support of the reduced word:

```python
sage: K = crystals.KirillovReshetikhin(["A",1,1], 1, 2)
sage: mg = K.module_generator()
sage: S = K.demazure_subcrystal(mg, [1])
sage: S.digraph().edges()
[(([1, 1], [1, 2], 1), ([1, 2], [1, 1], 0)),
 ([1, 2], [1, 1], 0),
 ([1, 1], [2, 1], 1),
 ([1, 2], [2, 1], 1),
 ([2, 1], [1, 2], 1)]
```

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 [As2008] Section 4 (where our \( \varphi_j(b) \) and \( \varepsilon_j(b) \) are denoted by \( \epsilon(b, j) \) and \( -\delta(b, j) \), respectively).

The dual equivalence graph of \( B \) is defined to be the colored graph whose vertices are the elements of \( B \) of weight 0, and whose edges of color \( i \) (for \( i \in I \)) connect pairs \( \{b, b'\} \) such that \( b' \) is an \( i \)-elementary dual equivalence transformation of \( b \).

**Note:** This dual equivalence graph is a generalization of \( G(X) \) in [As2008] Section 4 except we do not require \( \varepsilon_i(b) = 0, 1 \) for all \( i \).

This definition can be generalized by choosing a subset \( X \) of the set of all vertices of \( B \) of weight 0, and restricting the dual equivalence graph to the vertex set \( X \).

**INPUT:**
- \( X \) – (optional) the vertex set \( X \) (default: the whole set of vertices of \( \text{self} \) of weight 0)
- \( \text{index-set} \) – (optional) the index set \( I \) (default: the whole index set of \( \text{self} \)); this has to be a subset of the index set of \( \text{self} \) (as a list or tuple)
- \( \text{directed} \) – (default: \( \text{True} \)) 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:**

\begin{verbatim}
sage.combinat.partition.Partition.dual_equivalence_graph()
\end{verbatim}

**EXAMPLES:**

\begin{verbatim}
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: G = T.dual_equivalence_graph()[[[1, 3, 5], [2, 4]], [[1, 3, 4], [2, 5]], 4),
 [[[1, 3, 5], [2, 4]], [[1, 2, 5], [3, 4]], 2),
 [[[1, 2, 5], [3, 4]], [[1, 3, 5], [2, 4]], 3],
 [[[1, 2, 5], [3, 4]], [[1, 3, 5], [2, 4]], 3),
 [[[1, 2, 5], [3, 4]], [[1, 3, 5], [2, 4]], 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]]
\end{verbatim}

**class** TensorProducts (**category, *args**)  
**Bases:** sage.categories.tensor.TensorProductsCategory
The category of regular crystals constructed by tensor product of regular crystals.

**extra_super_categories()**

**EXAMPLES:**

```python
sage: RegularCrystals().TensorProducts().extra_super_categories()
[Category of regular crystals]
```

**additional_structure()**

Return None.

Indeed, the category of regular crystals defines no new structure: it only relates $\varepsilon_a$ and $\varphi_a$ to $e_a$ and $f_a$ respectively.

See also:

*Category.additional_structure()*

Todo: Should this category be a *CategoryWithAxiom*?

**EXAMPLES:**

```python
sage: RegularCrystals().additional_structure()
```

**example (n=3)**

Returns an example of highest weight crystals, as per *Category.example()*.

**EXAMPLES:**

```python
sage: B = RegularCrystals().example(); B
Highest weight crystal of type A_3 of highest weight omega_1
```

**super_categories()**

**EXAMPLES:**

```python
sage: RegularCrystals().super_categories()
[Category of crystals]
```

### 3.126 Regular Supercrystals

**class** `sage.categories.regular_supercrystals.RegularSuperCrystals(s=None)`

**Bases:** `sage.categories.category_singleton.Category_singleton`

The category of crystals for super Lie algebras.

**EXAMPLES:**

```python
sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: C = RegularSuperCrystals()
sage: C
Category of regular super crystals
sage: C.super_categories()
[Category of finite crystals]
```

Parents in this category should implement the following methods:

- either an attribute `_cartan_type` or a method `cartan_type`
• module_generators: a list (or container) of distinct elements that generate the crystal using \( f_i \) and \( e_i \)

Furthermore, their elements \( x \) should implement the following methods:

• \( x.e(i) \) (returning \( e_i(x) \))
• \( x.f(i) \) (returning \( f_i(x) \))
• \( x.weight() \) (returning \( wt(x) \))

**EXAMPLES:**

```
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: abstract_methods_of_class(RegularSuperCrystals().element_class)
{'optional': [], 'required': ['e', 'f', 'weight']}
```

class ElementMethods

**epsilon\( (i) \)**

Return \( \varepsilon_i \) of \( self \).

**EXAMPLES:**

```
sage: C = crystals.Tableaux(['A', [1,2]], shape = [2,1])
sage: c = C.an_element(); c
[-2, -2], [-1]
sage: c.epsilon(2)
0
sage: c.epsilon(0)
0
sage: c.epsilon(-1)
0
```

**is_genuine_highest_weight \( (index_set=None) \)**

Return whether \( self \) is a genuine highest weight element.

**INPUT:**

• index_set – (optional) the index set of the (sub)crystal on which to check

**EXAMPLES:**

```
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: for b in B.highest_weight_vectors():
....:     print("{} {}")

[[[-2, -2, -2], [-1, -1], [1]]] True
[[[-2, -2, -2], [-1, 2], [1]]] False
[[[-2, -2, 2], [-1, -1], [1]]] False
sage: [b for b in B if b.is_genuine_highest_weight([-1,0])]
[[[-2, -2, -2], [-1, -1], [1]],
 [[-2, -2, -2], [-1, -1], [2]],
 [[-2, -2, 2], [-1, 2], [2]],
 [[-2, -2, 2], [-1, -1], [2]],
 [[-2, -2, 2], [-1, 2], [2]],
 [[-2, -2, -2], [-1, 1], [1]],
 [[-2, -2, 2], [-1, 2], [-1]],
 [[-2, -2, -2], [-1, 2], [1]]]
```

**is_genuine_lowest_weight \( (index_set=None) \)**

Return whether \( self \) is a genuine lowest weight element.

**EXAMPLES:**

```
sage: C = crystals.Tableaux(['A', [1,2]], shape = [2,1])
sage: c = C.an_element(); c
[-2, -2], [-1]
sage: c.is_genuine_lowest_weight(3)
True
```

3.126. Regular Supercrystals  603
INPUT:
- \texttt{index\_set} – (optional) the index set of the (sub)crystal on which to check

**EXAMPLES:**

```python
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: for b in sorted(B.lowest_weight_vectors()):
    ....: print("{} {}").format(b, b.is_genuine_lowest_weight())
[[-2, 1, 2], [-1, 2], [1]] False
[[-2, 1, 2], [-1, 2], [2]] False
[[-1, 1, 2], [1, 2], [2]] True
sage: [b for b in B if b.is_genuine_lowest_weight([-1,0])]
[[[-2, -1, 1], [-1, 1], [1]],
 [[-2, -1, 1], [-1, 1], [2]],
 [[-2, 1, 2], [-1, 1], [2]],
 [[-2, 1, 2], [-1, 1], [1]],
 [[-1, -1, 1], [1, 2], [2]],
 [[-1, -1, 1], [1, 2], [1]],
 [[-1, 1, 2], [1, 2], [2]],
 [[-1, 1, 2], [1, 2], [1]]]
```

**\texttt{phi} (i)**

Return \( \varphi_i \) of \texttt{self}.

**EXAMPLES:**

```python
sage: C = crystals.Tableaux(['A',[1,2]], shape = [2,1])
sage: c = C.an_element(); c
[-2, -2, -1]
sage: c.phi(1)
0
sage: c.phi(2)
0
sage: c.phi(0)
1
```

**class ParentMethods**

**\texttt{character} ()**

Return the character of \texttt{self}.

**Todo:** Once the \texttt{WeylCharacterRing} is implemented, make this consistent with the implementation in \texttt{sage.categories.classical\_crystals.ClassicalCrystals.ParentMethods.character()}.  

**EXAMPLES:**

```python
sage: B = crystals.Letters(['A',[1,2]])
sage: B.character()
B[(1, 0, 0, 0, 0)] + B[(0, 1, 0, 0, 0)] + B[(0, 0, 1, 0, 0)] + B[(0, 0, 0, 1, 0)] + B[(0, 0, 0, 0, 1)]
```

**\texttt{connected\_components} ()**

Return the connected components of \texttt{self} as subcrystals.

**EXAMPLES:**
sage: B = crystals.Letters(['A', [1,2]])
sage: B.connected_components()
[Subcrystal of The crystal of letters for type ['A', [1, 2]]]
sage: T = B.tensor(B)
sage: T.connected_components()
[Subcrystal of Full tensor product of the crystals
 [The crystal of letters for type ['A', [1, 2]],
  The crystal of letters for type ['A', [1, 2]]],
Subcrystal of Full tensor product of the crystals
 [The crystal of letters for type ['A', [1, 2]],
  The crystal of letters for type ['A', [1, 2]]]]

connected_components_generators()

Return the tuple of genuine highest weight elements of self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_highest_weight_vectors()
(-2,)
sage: T = B.tensor(B)
sage: T.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])

digraph()

Return the DiGraph associated to self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,3]])
sage: G = B.digraph(); G
Multi-digraph on 6 vertices
sage: Q = crystals.Letters(['Q',3])
sage: G = Q.digraph(); G
Multi-digraph on 3 vertices
sage: G.edges()
[(1, 2, -1), (1, 2, 1), (2, 3, -2), (2, 3, 2)]

The edges of the crystal graph are by default colored using blue for edge 1, red for edge 2, green for edge 3, and dashed with the corresponding color for barred edges. Edge 0 is dotted black:

sage: view(G)  # optional - dot2tex graphviz, not tested (opens external window)

genuine_highest_weight_vectors()

Return the tuple of genuine highest weight elements of self.

EXAMPLES:

sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_highest_weight_vectors()
(-2,)
sage: T = B.tensor(B)
sage: T.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])
sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_highest_weight_vectors()
([-2, -1], [-2, -2])

**genuine_lowest_weight_vectors()**

Return the tuple of genuine lowest weight elements of self.

**EXAMPLES:**

sage: B = crystals.Letters(['A', [1,2]])
sage: B.genuine_lowest_weight_vectors()
(3,)
sage: T = B.tensor(B)
sage: T.genuine_lowest_weight_vectors()
([3, 3], [3, 2])
sage: s1, s2 = T.connected_components()
sage: s = s1 + s2
sage: s.genuine_lowest_weight_vectors()
([3, 3], [3, 2])

**highest_weight_vectors()**

Return the highest weight vectors of self.

**EXAMPLES:**

sage: B = crystals.Letters(['A', [1,2]])
sage: B.highest_weight_vectors()
(-2,)
sage: T = B.tensor(B)
sage: T.highest_weight_vectors()
([-2, -2], [-2, -1])

We give an example from [BKK2000] that has fake highest weight vectors:

sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: B.highest_weight_vectors()
([[-2, -2, -2], [-1, -1, [1]],
  [[-2, -2, -2], [-1, 2], [1]],
  [[-2, -2, 2], [-1, -1, [1]]])
sage: B.genuine_highest_weight_vectors()
([[-2, -2, -2], [-1, -1, [1]],]

**lowest_weight_vectors()**

Return the lowest weight vectors of self.

**EXAMPLES:**

sage: B = crystals.Letters(['A', [1,2]])
sage: B.lowest_weight_vectors()
(3,)

(continues on next page)
We give an example from [BKK2000] that has fake lowest weight vectors:

```
sage: B = crystals.Tableaux(['A', [1,1]], shape=[3,2,1])
sage: sorted(B.lowest_weight_vectors())
[[[-2, 1, 2], [-1, 2], [1]],
 [[-2, 1, 2], [-1, 2], [2]],
 [[-1, 1, 2], [1, 2], [2]]]
sage: B.genuine_lowest_weight_vectors()
([[[-1, 1, 2], [1, 2], [2]],])
```

```python
tensor(*crystals, **options)
```

Return the tensor product of self with the crystals B.

**EXAMPLES:**

```
sage: B = crystals.Letters(['A', [1,2]])
sage: C = crystals.Tableaux(['A', [1,2]], shape = [2,1])
sage: T = C.tensor(B); T
Full tensor product of the crystals [Crystal of letters for type ['A', [1, 2]], Crystal of BKK tableaux of shape [2, 1] of gl(2|3)]
sage: S = B.tensor(C); S
Full tensor product of the crystals [The crystal of letters for type ['A', [1, 2]], Crystal of BKK tableaux of shape [2, 1] of gl(2|3)]
sage: G = T.digraph()
sage: H = S.digraph()
sage: G.is_isomorphic(H, edge_labels=True)
True
```

```
class TensorProducts(category, *args)
```

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]
```

**super_categories()**

**EXAMPLES:**

```
sage: from sage.categories.regular_supercrystals import RegularSuperCrystals
sage: C = RegularSuperCrystals()
sage: C.super_categories()
[Category of finite crystals]
```

3.126. Regular Supercrystals
3.127 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 multiplication by elements of the rng

EXAMPLES:

```python
sage: RightModules(QQ)
Category of right modules over Rational Field
sage: RightModules(QQ).super_categories()
[Category of commutative additive groups]
```

class ElementMethods
class ParentMethods

super_categories()

EXAMPLES:

```python
sage: RightModules(QQ).super_categories()
[Category of commutative additive groups]
```

3.128 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:

```python
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:
3.129 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:

```python
sage: Rings()
Category of rings
sage: sorted(Rings().super_categories(), key=str)
[Category of rngs, Category of semirings]
```

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

```python
sage: Rings() is (CommutativeAdditiveGroups() & Monoids()).Distributive()
True
sage: Rings() is Rngs().Unital()
True
sage: Rings() is Semirings().AdditiveInverse()
True
```

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 sage.categories.commutative_rings.CommutativeRings

Division
alias of sage.categories.division_rings.DivisionRings

class ElementMethods

inverse_of_unit()

Return the inverse of this element if it is a unit.

OUTPUT:

An element in the same ring as this element.

EXAMPLES:
sage: R.<x> = ZZ[]
sage: S = R.quo(x^2 + x + 1)
sage: S(1).inverse_of_unit()
1

This method fails when the element is not a unit:

sage: 2.inverse_of_unit()
Traceback (most recent call last):
  ...
ArithmeticError: inverse does not exist

The inverse returned is in the same ring as this element:

sage: a = -1
sage: a.parent()
Integer Ring
sage: a.inverse_of_unit().parent()
Integer Ring

Note that this is often not the case when computing inverses in other ways:

sage: (-a).parent()
Rational Field
sage: (1/a).parent()
Rational Field

\textbf{is\_unit()} \\
Return whether this element is a unit in the ring.

\textbf{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.

\textbf{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()
False

class MorphismMethods

\textbf{is\_injective()} \\
Return whether or not this morphism is injective.

\textbf{EXAMPLES:}

This often raises a \texttt{NotImplementedError} as many homomorphisms do not implement this method:
If the domain is a field, the homomorphism is injective:

```
sage: K.<x> = FunctionField(QQ)
sage: L.<y> = FunctionField(QQ)
sage: f = K.hom([y]); f
Function Field morphism:
  From: Rational function field in x over Rational Field
  To:   Rational function field in y over Rational Field
  Defn: x |--> y
sage: f.is_injective()
True
```

Unless the codomain is the zero ring:

```
sage: codomain = Integers(1)
sage: f = QQ.hom([Zmod(1)(0)], check=False)
sage: f.is_injective()
False
```

Homomorphism from rings of characteristic zero to rings of positive characteristic can not be injective:

```
sage: R.<x> = ZZ[]
sage: f = R.hom([GF(3)(1)]); f
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Finite Field of size 3
  Defn: x |--> 1
sage: f.is_injective()
False
```

A morphism whose domain is an order in a number field is injective if the codomain has characteristic zero:

```
sage: K.<x> = FunctionField(QQ)
sage: f = ZZ.hom(K); f
Composite map:
  From: Integer Ring
  To:   Rational hom field in x over Rational Field
  Defn: Conversion via FractionFieldElement_lpoly_field map:
    From: Integer Ring
    To:   Fraction Field of Univariate Polynomial Ring in x over Rational Field
  Isomorphism:
    From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
    To:   Rational function field in x over Rational Field
```

(continues on next page)
A coercion to the fraction field is injective:

```sage
R = ZpFM(3)
R.fraction_field().coerce_map_from(R).is_injective()
```

```
True
```

---

### 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()
F.bracket(a, b)
```

```
B[word: ab] - B[word: ba]
```

This measures the default of commutation between $x$ and $y$. $F$ endowed with the bracket operation is a Lie algebra; in particular, it satisfies Jacobi’s identity:

```sage
F.bracket(F.bracket(a, b), c) + F.bracket(F.bracket(b, c), a) + F.bracket(F.bracket(c, a), b)
```

```
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.

EXAMPLES:

```python
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.

EXAMPLES:

```python
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:

```python
sage: MS.ideal_monoid() is MS.ideal_monoid()
True
```

`is_ring()`

Return True, since this is in an object of the category of rings.

EXAMPLES:
is_zero()  
Return True if this is the zero ring.

EXAMPLES:

```sage
sage: Integers(1).is_zero()
True  
sage: Integers(2).is_zero()
False  
sage: QQ.is_zero()
False  
sage: R.<x> = ZZ[]
```
```sage
sage: R.quo(1).is_zero()
True
```
```sage
sage: R.<x> = GF(101)[]
```
```sage
sage: R.quo(77).is_zero()
True
```
```sage
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().

EXAMPLES:

```sage
sage: MS = MatrixSpace(QQ,2)
```
```sage
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
sage: isinstance(MS,sage.rings.ring.Ring)
False
```
```sage
sage: isinstance(MS,Rings().parent_class)
True
```
```sage
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
```
```
```sage
[ 1 0]  
[0 0],
```
```sage
[0 1]  
[0 0],
```
```sage
[0 0]  
[1 0],
```
```sage
[0 0]
```
(continues on next page)
quotient \(I, names=None\)

Quotient of a ring by a two-sided ideal.

INPUT:
- \(I\): A two-sided 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 \texttt{sage.rings.ring.Ring.quotient()}. So, we need a bit of effort to make the following example work with the category framework:

```python
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: from sage.rings.noncommutative_ideals import Ideal_nc
sage: from itertools import product
sage: class PowerIdeal(Ideal_nc):
....:     def __init__(self, R, n):
....:         self._power = n
....:         Ideal_nc.__init__(self, R, [R.prod(m) for m in product(R.gens(), repeat=n)])
....:     def reduce(self, x):
....:         R = self.ring()
....:         return add([c*R(m) for m,c in x if len(m) < self._power], R(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, x^2*y, x^2*z, x*y*x, x*y^2, x*z*x, x*z*y, x*z^2, y*x^2, y*x*y, y*x*z, y^2*x, y^3, y^2*z, y*z*x, y*z*y, y*z^2, z*x^2, z*x*y, z*x*z, z*y*x, z*y^2, z*y*z, z^2*x, z^2*y, z^3)
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 \texttt{quotient()}. 

EXAMPLES:

```python
sage: MS = MatrixSpace(QQ,2)
sage: I = MS*MS.gens()*MS
```

\(MS\) is not an instance of \texttt{Ring}, but it is an instance of the parent class of the category of rings. The quotient method is inherited from there:
class 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:

```python
sage: Rings().Division()
Category of division rings
sage: Rings().Commutative().Division()
Category of fields
```

NoZeroDivisors()

Return the full subcategory of the objects of self having no nonzero zero divisors.

A zero divisor in a ring $R$ is an element $x \in R$ such that there exists a nonzero element $y \in R$ such that $x \cdot y = 0$ or $y \cdot x = 0$ (see Wikipedia article Zero_divisor).

EXAMPLES:

```python
sage: Rings().NoZeroDivisors()
Category of domains
```

Note: This could be generalized to MagmasAndAdditiveMagmas.Distributive.AdditiveUnital.
3.130 Rngs

class sage.categories.rngs.Rngs(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of rngs.

An rng \((S, +, \cdot)\) is similar to a ring but not necessarily unital. In other words, it is a combination of a commutative additive group \((S, +)\) and a multiplicative semigroup \((S, \cdot)\), where \(\cdot\) distributes over \(+\).

EXAMPLES:

```python
sage: C = Rngs(); C
Category of rngs
sage: sorted(C.super_categories(), key=str)
[Category of associative additive commutative additive associative additive unital distributive magmas and additive magmas,
 Category of commutative additive groups]
```

```python
sage: C.is(C.CommutativeAdditiveGroups() & Semigroups()).Distributive()
True
sage: C.Unital()
Category of rings
```

Unital

alias of `sage.categories.rings.Rings`

3.131 R-trivial semigroups

class sage.categories.r_trivial_semigroups.RTrivialSemigroups(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

Commutative_extra_super_categories()

Implement the fact that a commutative \(R\)-trivial semigroup is \(J\)-trivial.

EXAMPLES:

```python
sage: Semigroups().RTrivial().Commutative_extra_super_categories()
[Category of j trivial semigroups]
```

extra_super_categories()

Implement the fact that a \(R\)-trivial semigroup is \(H\)-trivial.

EXAMPLES:

```python
sage: Semigroups().RTrivial().extra_super_categories()
[Category of h trivial semigroups]
```
3.132 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
the test_category_over_bases()).

This is currently incompatible with the dispatching below.

```
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
```

```
sage: Schemes(Spec(ZZ)).base_scheme()
Spectrum of Integer Ring
```

```
sage: Schemes(Spec(ZZ)).super_categories()
[Category of schemes]
```
### 3.133 Semigroups

```python
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:
    
    ```python
    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
    ```
```
```

```python
class Algebras(category, *args):
    Bases: sage.categories.algebra_functor.AlgebrasCategory

    class ParentMethods

    algebra_generators()

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

        They correspond to the generators of the semigroup.

        EXAMPLES:
        
        ```python
        sage: M = FiniteSemigroups().example(); M
        An example of a finite semigroup: the left regular band generated by ('a', 'b', 'c', 'd')
        sage: M.semigroup_generators()
        Family ('a', 'b', 'c', 'd')
        sage: M.algebra(ZZ).algebra_generators()
        Finite family {0: B['a'], 1: B['b'], 2: B['c'], 3: B['d']}
        ```
```
```
**gens()**

Return the generators of self.

**EXAMPLES:**

```
sage: a, b = SL2Z.algebra(ZZ).gens(); a, b
([[ 0 -1]
  [ 1  0]],
 [[ 1  1]
  [ 0  1]])
sage: 2*a + b
2*[[ 0 -1]
  [ 1  0]]
+[[ 1  1]
  [ 0  1]]
```

**ngens()**

Return the number of generators of self.

**EXAMPLES:**

```
sage: SL2Z.algebra(ZZ).ngens()
2
sage: DihedralGroup(4).algebra(RR).ngens()
2
```

**product_on_basis**(g1, g2)

Product, on basis elements, as per `MagmaticAlgebras.WithBasis.ParentMethods.product_on_basis()`.

The product of two basis elements is induced by the product of the corresponding elements of the group.

**EXAMPLES:**

```
sage: S = FiniteSemigroups().example(); S
An example of a finite semigroup: the left regular band generated by ( 
\rightarrow 'a', 'b', 'c', 'd')
sage: A = S.algebra(QQ)
sage: a,b,c,d = A.algebra_generators()
sage: a * b + b * d * c * d
B['ab'] + B['bdc']
```

**regular_representation**(side='left')

Return the regular representation of self.

**INPUT:**

- side (default: "left") whether this is the "left" or "right" regular representation

**EXAMPLES:**

```
sage: G = groups.permutation.Dihedral(4)
sage: A = G.algebra(QQ)
sage: V = A.regular_representation()
sage: V == G.regular_representation(QQ)
True
```

**trivial_representation**(side='twosided')

Return the trivial representation of self.
INPUT:
• side — ignored

EXAMPLES:
```
sage: G = groups.permutation.Dihedral(4)
sage: A = G.algebra(QQ)
sage: V = A.trivial_representation()
sage: V == G.trivial_representation(QQ)
True
```

extra_super_categories()
Implement the fact that the algebra of a semigroup is indeed a (not necessarily unital) algebra.

EXAMPLES:
```
sage: Semigroups().Algebras(QQ).extra_super_categories()
[Category of semigroups]
sage: Semigroups().Algebras(QQ).super_categories()
[Category of associative algebras over Rational Field,
 Category of magma algebras over Rational Field]
```

Aperiodic
alias of sage.categories.aperiodic_semigroups.AperiodicSemigroups

class CartesianProducts(category, *args)
Bases: sage.categories.cartesian_product.CartesianProductsCategory

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 ElementMethods

Finite
alias of sage.categories.finite_semigroups.FiniteSemigroups

FinitelyGeneratedAsMagma
alias of sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups

HTrivial
alias of sage.categories.h_trivial_semigroups.HTrivialSemigroups

JTrivial
alias of sage.categories.j_trivial_semigroups.JTrivialSemigroups

LTrivial
alias of sage.categories.l_trivial_semigroups.LTrivialSemigroups

class 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):G.vertices()}, bgcolor=(0,0,0), color_by_label=True, xres=700, yres=700, iterations=200) # long time (less than a minute)
sage: G.num_edges()
120
```

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 in range(4)),[]) ]
sage: G = M.cayley_graph(elements = elements)
sage: G.num_verts(), G.num_edges()
(85, 84)
sage: G.show3d(color_by_label=True, edge_size=0.001, vertex_size=0.01)
```

We now illustrate the side and simple options on a semigroup:
Todo:
- Add more options for constructing subgraphs of the Cayley graph, handling the standard use cases when exploring large/infinite semigroups (a predicate, generators of an ideal, a maximal length in term of the generators)
- Specify good default layout/plot/latex options in the graph
- Generalize to combinatorial modules with module generators / operators

AUTHORS:
- Bobby Moretti (2007-08-10)
- Nicolas M. Thiery (2008-12): extension to semigroups, side, simple, and elements options, ...

magma_generators()
An alias for semigroup_generators().

EXAMPLES:


```python
sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', ...
\rightarrow 'd')
sage: S.magma_generators()
Family ('a', 'b', 'c', 'd')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

\textbf{prod} (\textit{args})

Return the product of the list of elements \textit{args} inside \textit{self}.

\textbf{EXAMPLES:}

```python
sage: S = Semigroups().example("free")
sage: S.prod([S('a'), S('b'), S('c')])
'abc'
sage: S.prod([])
Traceback (most recent call last):
  ...
AssertionError: Cannot compute an empty product in a semigroup
```

\textbf{regular_representation} (\textit{base_ring}=None, \textit{side}='left')

Return the regular representation of \textit{self} over \textit{base_ring}.

- \textit{side} – (default: "left") whether this is the "left" or "right" regular representation

\textbf{EXAMPLES:}

```python
sage: G = groups.permutation.Dihedral(4)
sage: G.regular_representation()
Left Regular Representation of Dihedral group of order 8 as a permutation group over Integer Ring
```

\textbf{semigroup_generators} ()

Return distinguished semigroup generators for \textit{self}.

\textbf{OUTPUT:} a family

This method is optional.

\textbf{EXAMPLES:}

```python
sage: S = Semigroups().example("free"); S
An example of a semigroup: the free semigroup generated by ('a', 'b', 'c', ...
\rightarrow 'd')
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

\textbf{subsemigroup} (\textit{generators}, \textit{one}=None, \textit{category}=None)

Return the multiplicative subsemigroup generated by \textit{generators}.

\textbf{INPUT:}

- \textit{generators} – a finite family of elements of \textit{self}, or a list, iterable, ... that can be converted into one (see Family).
- \textit{one} – a unit for the subsemigroup, or None.
- \textit{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 \texttt{AutomaticSemigroup} for details.

\textbf{EXAMPLES:}

```python
```
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 explicitly:

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

Todo:
- Fix the failure in TESTS by providing a default implementation of __invert__ for finite groups (or even finite monoids).
- Provide a default implementation of one for a finite monoid, so that we would not need to specify it explicitly?

trivial_representation\( (base\_ring=\text{None},\ side='\text{twosided}') \)

Return the trivial representation of self over base\_ring.

INPUT:
- base\_ring – (optional) the base ring; the default is $\mathbb{Z}$
- side – ignored

EXAMPLES:
class Quotients (category, *args)
Bases: sage.categories.quotients.QuotientsCategory

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 implemented

example()
    Return an example of quotient of a semigroup, as per Category.example().

    EXAMPLES:
    
    sage: Semigroups().Quotients().example()
    An example of a (sub)quotient semigroup: a quotient of the left zero semigroup

RTrivial
    alias of sage.categories.r_trivial_semigroups.RTrivialSemigroups

class SubcategoryMethods

    Aperiodic()
    Return the full subcategory of the aperiodic objects of self.

    A (multiplicative) semigroup \( S \) is aperiodic if for any element \( s \in S \), the sequence \( s, s^2, s^3, ... \) eventually stabilizes.

    In terms of variety, this can be described by the equation \( s^\omega s = s \).

    EXAMPLES:
    
    sage: Semigroups().Aperiodic()
    Category of aperiodic semigroups

An aperiodic semigroup is \( H \)-trivial:

    sage: Semigroups().Aperiodic().axioms()
    frozenset({'Aperiodic', 'Associative', 'HTrivial'})

In the finite case, the two notions coincide:

    sage: Semigroups().Aperiodic().Finite() is Semigroups().HTrivial().Finite()
    True
HTrivial()  
Return the full subcategory of the \(H\)-trivial objects of \(self\).

Let \(S\) be (multiplicative) semigroup. Two elements of \(S\) are in the same \(H\)-class if they are in the same \(L\)-class and in the same \(R\)-class. The semigroup \(S\) is \(H\)-trivial if all its \(H\)-classes are trivial (that is of cardinality 1).

EXAMPLES:

```
sage: C = Semigroups().HTrivial(); C  
Category of h trivial semigroups
sage: Semigroups().HTrivial().Finite().example()  
NotImplemented
```

See also:

- Wikipedia article Aperiodic_semigroup
- Semigroups.SubcategoryMethods.RTrivial
- Semigroups.SubcategoryMethods.LTrivial
- Semigroups.SubcategoryMethods.JTrivial
- Semigroups.SubcategoryMethods.Aperiodic

JTrivial()  
Return the full subcategory of the \(J\)-trivial objects of \(self\).

Let \(S\) be (multiplicative) semigroup. The \(J\)-preorder \(\leq J\) on \(S\) is defined by:

\[ x \leq J y \iff x \in SyS \]

The \(J\)-classes are the equivalence classes for the associated equivalence relation. The semigroup \(S\) is \(J\)-trivial if all its \(J\)-classes are trivial (that is of cardinality 1), or equivalently if the \(J\)-preorder is in fact a partial order.

EXAMPLES:

```
sage: C = Semigroups().JTrivial(); C  
Category of j trivial semigroups
```

A semigroup is \(J\)-trivial if and only if it is \(L\)-trivial and \(R\)-trivial:

```
sage: sorted(C.axioms())  
['Associative', 'HTrivial', 'JTrivial', 'LTrivial', 'RTrivial']
sage: Semigroups().LTrivial().RTrivial()  
Category of j trivial semigroups
```

For a commutative semigroup, all three axioms are equivalent:

```
sage: Semigroups().Commutative().LTrivial()  
Category of commutative j trivial semigroups
sage: Semigroups().Commutative().RTrivial()  
Category of commutative j trivial semigroups
```

See also:
• Wikipedia article Green’s_relations
• Semigroups.SubcategoryMethods.LTrivial
• Semigroups.SubcategoryMethods.RTrivial
• Semigroups.SubcategoryMethods.HTrivial

**LTrivial()**

Return the full subcategory of the $L$-trivial objects of `self`.

Let $S$ be (multiplicative) `semigroup`. The $L$-preorder $\preceq_L$ on $S$ is defined by:

$$x \preceq_L y \iff x \in S y$$

The $L$-classes are the equivalence classes for the associated equivalence relation. The semigroup $S$ is $L$-trivial if all its $L$-classes are trivial (that is of cardinality 1), or equivalently if the $L$-preorder is in fact a partial order.

**EXAMPLES:**

```python
sage: C = Semigroups().LTrivial(); C
Category of l trivial semigroups
```

A $L$-trivial semigroup is $H$-trivial:

```python
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'LTrivial']
```

See also:

• Wikipedia article Green’s_relations
• Semigroups.SubcategoryMethods.RTrivial
• Semigroups.SubcategoryMethods.JTrivial
• Semigroups.SubcategoryMethods.HTrivial

**RTrivial()**

Return the full subcategory of the $R$-trivial objects of `self`.

Let $S$ be (multiplicative) `semigroup`. The $R$-preorder $\preceq_R$ on $S$ is defined by:

$$x \preceq_R y \iff x \in y S$$

The $R$-classes are the equivalence classes for the associated equivalence relation. The semigroup $S$ is $R$-trivial if all its $R$-classes are trivial (that is of cardinality 1), or equivalently if the $R$-preorder is in fact a partial order.

**EXAMPLES:**

```python
sage: C = Semigroups().RTrivial(); C
Category of r trivial semigroups
```

An $R$-trivial semigroup is $H$-trivial:

```python
sage: sorted(C.axioms())
['Associative', 'HTrivial', 'RTrivial']
```

See also:

• Wikipedia article Green’s_relations
• Semigroups.SubcategoryMethods.LTrivial
• Semigroups.SubcategoryMethods.JTrivial
• Semigroups.SubcategoryMethods.HTrivial
class Subquotients\(\textit{category}, \ast\textit{args}\)
Bases: sage.categories.subquotients.SubquotientsCategory

The category of subquotient semi-groups.

EXAMPLES:

```
class Subquotients(simag, *args):

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 \textit{Category.example()}.

EXAMPLES:

```

```

```

Unital

alias of sage.categories.monoids.Monoids

example\(\textit{choice}='\textit{leftzero}', \ast\textit{kwds}\)

Returns an example of a semigroup, as per \textit{Category.example()}.

INPUT:

- \textit{choice} – str (default: ‘leftzero’). Can be either ‘leftzero’ for the left zero semigroup, or ‘free’ for the free semigroup.

- \textit{**kwds} – keyword arguments passed onto the constructor for the chosen semigroup.

EXAMPLES:

```

```

```

3.133. Semigroups

629
3.134 Semirings

class sage.categories.semirings.Semirings(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom_singleton

The category of semirings.

A semiring \((S, +, \cdot)\) 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 \((S, \cdot)\), where \(\cdot\) distributes over \(+\).

See also:

Wikipedia article Semiring

EXAMPLES:

```python
sage: Semirings()
Category of semirings
sage: Semirings().super_categories()
[Category of associative additive commutative additive associative additive unital distributive magmas and additive magmas, Category of monoids]
sage: sorted(Semirings().axioms())
['AdditiveAssociative', 'AdditiveCommutative', 'AdditiveUnital', 'Associative', 'Distributive', 'Unital']
sage: Semirings().is (CommutativeAdditiveMonoids() & Monoids()).Distributive()
True
sage: Semirings().AdditiveInverse()
Category of rings
```

3.135 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:

```python
sage: from sage.categories.semisimple_algebras import SemisimpleAlgebras
sage: C = SemisimpleAlgebras(QQ); C
Category of semisimple algebras over Rational Field
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]
```

This category is best constructed as:

```python
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:

Wikipedia article Semisimple_algebra

class FiniteDimensional (base_category)
  Bases: sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

  WithBasis
    alias of sage.categories.finite_dimensional_semisimple_algebras_with_basis.
    FiniteDimensionalSemisimpleAlgebrasWithBasis

class 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()
()
```

  super_categories()
    EXAMPLES:
    ```
sage: Algebras(QQ).Semisimple().super_categories()
[Category of algebras over Rational Field]
```

3.136 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]
```
The category of sets.

The base category for collections of elements with \( = \) (equality).

This is also the category whose objects are all parents.

**EXAMPLES:**

```python
sage: Sets()
Category of sets
sage: Sets().super_categories()
[Category of sets with partial maps]
```

Let us consider an example of set:

```python
sage: P = Sets().example("inherits")
sage: P
Set of prime numbers
```

See `P??` for the code.

P is in the category of sets:

```python
sage: P.category()
Category of sets
```

and therefore gets its methods from the following classes:

```python
sage: for cl in P.__class__.mro(): print(cl)
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits_with_category'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Inherits'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract'>
<class 'sage.structure.unique_representation.UniqueRepresentation'>
<class 'sage.structure.unique_representation.CachedRepresentation'>
<type 'sage.misc.fast_methods.WithEqualityById'>
<class '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'>
<... 'object'>
```

We run some generic checks on P:
sage: TestSuite(P).run(verbos=True)
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
running ._test_elements() ... 
  Running the test suite of self.an_element()
running ._test_category() ... pass
running ._test_eq() ... pass
running ._test_new() ... pass
running ._test_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_eq() ... pass
running ._test_new() ... pass
running ._test_not_implemented_methods() ... pass
running ._test_pickling() ... pass
running ._test_some_elements() ... pass

Now, we manipulate some elements of P:

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.ModuleElement'>
<class 'sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element'>
<class 'sage.categories.sets_cat.Sets.element_class'>
<class 'sage.categories.sets_with_partial_maps.SetsWithPartialMaps.element_class'>
<class 'sage.categories.objects.Objects.element_class'>
<... 'object'>

FIXME: Objects.element_class is not very meaningful ...
class Algebras(*category, *args)
    Bases: sage.categories.algebra_functor.AlgebrasCategory

class ParentMethods

construction()
    Return the functorial construction of self.

    EXAMPLES:
    sage: A = GroupAlgebra(KleinFourGroup(), QQ)
    sage: A.construction()
    (GroupAlgebraFunctor, Rational Field)

elementMethods()
    EXAMPLES:
    sage: Sets().Algebras(ZZ).extra_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 CartesianProducts(*category, *args)
    Bases: sage.categories.cartesian_product.CartesianProductsCategory

elementMethods
    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()
sage: S = cartesian_product([F, G, H])
sage: x = S.monomial((0,4)) + 2 * S.monomial((0,5)) + 3 * S.
    monomial((1,6)) + 4 * S.monomial((2,4)) + 5 * S.monomial((2,7))
sage: x.cartesian_factors()
sage: [s.parent() for s in x.cartesian_factors()]
[F, G, H]
sage: S.zero().cartesian_factors()
(0, 0, 0)
sage: [s.parent() for s in S.zero().cartesian_factors()]
[F, G, H]

cartesian_projection(i)

Return the projection of self onto the i-th factor of the Cartesian product.

INPUT:
• i – the index of a factor of the Cartesian product

EXAMPLES:

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)
sage: x.cartesian_projection(1)
3*B[6]

class 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

element() (continued on next page)

sage: Sets().CartesianProducts().example()
The Cartesian product of (Set of prime numbers (basic implementation),
An example of an infinite enumerated set: the non negative integers,
An example of a finite enumerated set: {1,2,3})

extra_super_categories() (continued on next page)

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 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?

**Enumerated**
alias of *sage.categories.enumerated_sets.EnumeratedSets*

**Facade**
alias of *sage.categories.facade_sets.FacadeSets*

**Finite**
alias of *sage.categories.finite_sets.FiniteSets*

**class Infinite**(base_category)
Bases: *sage.categories.category_with_axiom.CategoryWithAxiom_singleton*

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
```

class IsomorphicObjects**(category, *args)**
Bases: *sage.categories.isomorphic_objects.IsomorphicObjectsCategory*

A category for isomorphic objects of sets.

EXAMPLES:
```python
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]
```

```python
class ParentMethods

Metric
    alias of sage.categories.metric_spaces.MetricSpaces

class MorphismMethods

is_injective()
    Return whether this map is injective.

EXAMPLES:

```python
sage: f = ZZ.hom(GF(3)); f
Natural morphism:
    From: Integer Ring
    To:   Finite Field of size 3
sage: f.is_injective()
False
```

Note that many maps do not implement this method:

```python
sage: R.<x> = ZZ[]
sage: f = R.hom([x])
sage: f.is_injective()
Traceback (most recent call last):
  ...()
NotImplementedError
```

```python
class ParentMethods

CartesianProduct
    alias of sage.sets.cartesian_product.CartesianProduct

algebra(base_ring, category=None, **kwds)
    Return the algebra of self over base_ring.

INPUT:
    • self -- a parent S
    • base_ring -- a ring K
    • category -- a super category of the category of S, or None

This returns the space of formal linear combinations of elements of G with coefficients in R, endowd with whatever structure can be induced from that of S. See the documentation of sage.categories.algebra_functor for details.

EXAMPLES:

If S is a group, the result is its group algebra KS:
```

3.136. Sets
This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication of $G$ to a multiplication on $RG$:

\[
sage: a = A.an_element(); a \\
() + (1,3) + 2*(1,3)(2,4) + 3*(1,4,3,2)
\]

If $S$ is a monoid, the result is its monoid algebra $KS$:

\[
sage: S = Monoids().example(); S \\
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
sage: A = S.algebra(QQ); A \\
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd') over Rational Field
sage: A.category() \\
Category of monoid algebras over Rational Field
\]

Similarly, we can construct algebras for additive magmas, monoids, and groups. One may specify for which category one takes the algebra; here we build the algebra of the additive group $GF_3$:

\[
sage: from sage.categories.additive_groups import AdditiveGroups \\
sage: S = GF(7); S \\
Algebra of Finite Field of size 7 over Rational Field
sage: A = S.algebra(QQ, category=AdditiveGroups()); A \\
Algebra of Finite Field of size 7 over Rational Field
sage: A.category() \\
Category of finite dimensional additive group algebras over Rational Field
sage: a = A(S(1)) \\
sage: a \\
1
sage: 1 + a * a * a \\
0 + 3
\]

Note that the category keyword needs to be fed with the structure on $S$ to be used, not the induced structure on the result.

**an_element()**

Return a (preferably typical) element of this parent.

This is used both for illustration and testing purposes. If the set *self* is empty, *an_element()* should raise the exception *EmptySetError*.

This default implementation calls **an_element()** and caches the result. Any parent should implement either *an_element()* or **an_element()**.
EXAMPLES:

```python
sage: CDF.an_element()
1.0*I
sage: ZZ[['t']].an_element()
t
```

**cartesian_product (**parents, **kwargs)**

Return the Cartesian product of the parents.

**INPUT:**

- `parents` – a list (or other iterable) of parents.
- `category` – (default: None) the category the Cartesian product belongs to. If None is passed, then `category_from_parents()` is used to determine the category.
- `extra_category` – (default: None) a category that is added to the Cartesian product in addition to the categories obtained from the parents.
- other keyword arguments will passed on to the class used for this Cartesian product (see also `CartesianProduct`).

**OUTPUT:**

The Cartesian product.

**EXAMPLES:**

```python
sage: C = AlgebrasWithBasis(QQ)
sage: A = C.example(); A.rename("A")
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)
sage: cartesian_product([ZZ, ZZ], category=Sets()).category()
Category of sets
sage: cartesian_product([ZZ, ZZ]).category()
Join of
Category of Cartesian products of commutative rings and
Category of Cartesian products of enumerated sets
sage: cartesian_product([ZZ, ZZ], extra_category=Posets()).category()
Join of
Category of Cartesian products of commutative rings and
Category of posets and
Category of Cartesian products of enumerated sets
```

**construction()**

Return a pair `(functor, parent)` such that `functor(parent)` returns `self`. If `self` does not have a functorial construction, return `None`.

**EXAMPLES:**

```python
sage: QQ.construction()
(FractionField, Integer Ring)
sage: f, R = QQ['x'].construction()
sage: f
Poly[x]
sage: R
(continues on next page)```
Rational Field

\texttt{sage: f(R)}
Univariate Polynomial Ring in x over Rational Field

\texttt{\texttt{is\_parent\_of}(element)}

Return whether \texttt{self} is the parent of \texttt{element}.

\textbf{INPUT:}

\begin{itemize}
  \item element – any object
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: S = ZZ}
\texttt{sage: S.is\_parent\_of(1)}
\texttt{True}
\texttt{sage: S.is\_parent\_of(2/1)}
\texttt{False}
\end{verbatim}

This method differs from \texttt{\_\_contains\_\_()} because it does not attempt any coercion:

\begin{verbatim}
\texttt{sage: 2/1 in S, S.is\_parent\_of(2/1)}
\texttt{(True, False)}
\texttt{sage: int(1) in S, S.is\_parent\_of(int(1))}
\texttt{(True, False)}
\end{verbatim}

\texttt{\texttt{some\_elements}()}  

Return a list (or iterable) of elements of \texttt{self}.  

This is typically used for running generic tests (see \texttt{TestSuite}).  

This default implementation calls \texttt{an\_element\()\}.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: S = Sets().example(); S}
Set of prime numbers (basic implementation)
\texttt{sage: S.an\_element()}
47
\texttt{sage: S.some\_elements()}
[47]
\texttt{sage: S = Set([])}
\texttt{sage: S.some\_elements()}
[]
\end{verbatim}

This method should return an iterable, \textit{not} an iterator.

\textbf{class Quotients (category, *args)}

\textbf{Bases:} \texttt{sage.categories.quotients.QuotientsCategory}

A category for quotients of sets.

\textbf{See also:}

Sets().Quotients()

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: Sets().Quotients()}
Category of quotients of sets
\texttt{sage: Sets().Quotients().all\_super\_categories()}
\end{verbatim}
[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 Realizations(category, *args)
    Bases: sage.categories.realizations.RealizationsCategory

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 SubcategoryMethods

    Algebras(base_ring)
    Return the category of objects constructed as algebras of objects of self over base_ring.
    INPUT:
    • base_ring -- a ring
    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

    CartesianProducts()
    Return the full subcategory of the objects of self constructed as Cartesian products.
    See also:
• `cartesian_product.CartesianProductFunctor`
  • `RegressiveCovariantFunctorialConstruction`

**EXAMPLES:**

```python
sage: Sets().CartesianProducts()
Category of Cartesian products of sets
sage: Semigroups().CartesianProducts()
Category of Cartesian products of semigroups
sage: EuclideanDomains().CartesianProducts()
Category of Cartesian products of commutative rings
```

**Enumerated()**

Return the full subcategory of the enumerated objects of `self`.

An enumerated object can be iterated to get its elements.

**EXAMPLES:**

```python
sage: Sets().Enumerated()
Category of enumerated sets
sage: Rings().Finite().Enumerated()
Category of finite enumerated rings
sage: Rings().Infinite().Enumerated()
Category of infinite enumerated rings
```

**Facade()**

Return the full subcategory of the facade objects of `self`.

**What is a facade set?**

Recall that, in Sage, sets are modelled by "parents", and their elements know which distinguished set they belong to. For example, the ring of integers \( \mathbb{Z} \) is modelled by the parent \( \mathbb{Z} \), and integers know that they belong to this set:

```python
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:

```python
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 \( P \):

```python
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 |
\longrightarrow
\text{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 $\mathbb{Z}$. The disadvantage is that this introduces an ambiguity when writing $2 < 3$: does this compare $2$ and $3$ w.r.t. the natural order on integers or w.r.t. divisibility?:

```
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:

\begin{verbatim}
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)
\end{verbatim}

Specifications

A parent which is a facade must either:

- call `Parent.__init__()` using the `facade` parameter to specify a parent, or tuple thereof.
- overload the method `facade_for()`.

Note: The concept of facade parents was originally introduced in the computer algebra system MuPAD.

**Finite()**

Return the full subcategory of the finite objects of `self`.

**EXAMPLES:**

\begin{verbatim}
sage: Sets().Finite() Category of finite sets
sage: Rings().Finite() Category of finite rings
\end{verbatim}

**Infinite()**

Return the full subcategory of the infinite objects of `self`.

**EXAMPLES:**

\begin{verbatim}
sage: Sets().Infinite() Category of infinite sets
sage: Rings().Infinite() Category of infinite rings
\end{verbatim}

**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 \( B = \{1, 2, 3\} \):

\begin{verbatim}
sage: A = FiniteEnumeratedSets().IsomorphicObjects().example(); A
The image by some isomorphism of An example of a finite enumerated set:
\end{verbatim}
Since \( B \) is a finite enumerated set, so is \( A \):

```python
sage: A in FiniteEnumeratedSets()
True
sage: A.cardinality()
3
sage: A.list()
[1, 4, 9]
```

The isomorphism from \( B \) to \( A \) is available as:

```python
sage: A.retract(3)
9
```

and its inverse as:

```python
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 \( B = \mathbb{Z} \), endowed with \( \max \) as addition, and \( + \) as multiplication instead of the usual \( + \) and \( \ast \). One can construct \( A \) as isomorphic to \( B \) as an infinite enumerated set. However \( A \) is not isomorphic to \( B \) as a ring; for example, for \( a \in A \) and \( a \in B \), the expressions \( a + A(b) \) and \( B(a) + b \) give completely different results; hence we would not want the expression \( a + b \) to be implicitly resolved to any one of above two, as the coercion mechanism would do.

Coercions also cannot be used with facade parents (see `Sets.Facade`) like in the example above.

We now look at a category of isomorphic objects:

```python
sage: C = Sets().IsomorphicObjects(); C
Category of isomorphic objects of sets
sage: C.super_categories()
[Category of subobjects of sets, Category of quotients of sets]
```

```python
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:

```python
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

\textbf{Metric()}

Return the subcategory of the metric objects of \texttt{self}.

\textbf{Quotients()}

Return the full subcategory of the objects of \texttt{self} constructed as quotients.

Given a concrete category \texttt{As()} (i.e. a subcategory of \texttt{Sets()}), \texttt{As().Quotients()} returns the category of objects of \texttt{As()} endowed with a distinguished description as quotient (in fact homomorphic image) of some other object of \texttt{As()}.

Implementing an object of \texttt{As().Quotients()} is done in the same way as for \texttt{As().Subquotients()}; namely by providing an ambient space and a lift and a retract map. See \texttt{Subquotients()} for detailed instructions.

See also:

• \texttt{Subquotients()} for background
• \texttt{quotients.QuotientsCategory}
• \texttt{RegressiveCovariantFunctorialConstruction}

\textbf{EXAMPLES:}

\begin{verbatim}
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]
\end{verbatim}

The caller is responsible for checking that the given category admits a well defined category of quotients:

\begin{verbatim}
sage: EuclideanDomains().Quotients()
Join of Category of euclidean domains and Category of subquotients of monoids and Category of quotients of semigroups
\end{verbatim}

\textbf{Subobjects()}

Return the full subcategory of the objects of \texttt{self} constructed as subobjects.

Given a concrete category \texttt{As()} (i.e. a subcategory of \texttt{Sets()}), \texttt{As().Subobjects()} returns the category of objects of \texttt{As()} endowed with a distinguished embedding into some other object of \texttt{As()}.

Implementing an object of \texttt{As().Subobjects()} is done in the same way as for \texttt{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 \texttt{Subquotients()} for detailed instructions.

See also:
• *Subquotients()* for background
• *subobjects.SubobjectsCategory*
• RegressiveCovariantFunctorialConstruction

EXAMPLES:

```python
sage: C = Sets().Subobjects(); C
Category of subobjects of sets
sage: C.super_categories()
[Category of subquotients of sets]
sage: C.all_super_categories()
[Category of subobjects of sets,
 Category of subquotients of sets,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

Unless something specific about subobjects is implemented for this category, one actually gets an
optimized super category:

```python
sage: C = Semigroups().Subobjects(); C
Join of Category of subquotients of semigroups
    and Category of subobjects of sets
```

The caller is responsible for checking that the given category admits a well defined category of sub-
objects.

**Subquotients()**

Return the full subcategory of the objects of self constructed as subquotients.

Given a concrete category `self == As()` (i.e. a subcategory of `Sets()`), `As().Subquotients()` returns the category of objects of `As()` endowed with a distinguished description as subquotient of some other object of `As()`.

EXAMPLES:

```python
sage: Monoids().Subquotients()
Category of subquotients of monoids
```

A parent `A` in `As()` is further in `As().Subquotients()` if there is a distinguished parent `B` in `As()`, called the ambient set, a subobject `B'` of `B`, and a pair of maps:

```
\[ l : A \to B' \text{ and } r : B' \to A \]
```

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 \texttt{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 \texttt{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 \( \mathbb{Z}/n\mathbb{Z} \).
See also:

- `Quotients()`, `Subobjects()`, `IsomorphicObjects()`
- `subquotients.SubquotientsCategory`
- `RegressiveCovariantFunctorialConstruction`

**Topological()**

Return the subcategory of the topological objects of `self`.

### class `Subobjects` (category, *args)

**Bases:** `sage.categories.subobjects.SubobjectsCategory`

A category for subobjects of sets.

See also:

`Sets().Subobjects()`

**EXAMPLES:**

```python
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 `Subquotients` (category, *args)

**Bases:** `sage.categories.subquotients.SubquotientsCategory`

A category for subquotients of sets.

See also:

`Sets().Subquotients()`

**EXAMPLES:**

```python
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:**

```python
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)
```

(continues on next page)
class 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:

retract(x)
Retract \( x \) to self.

INPUT:
• \( x \) – an element of the ambient space for self

See also:

EXAMPLES:

sage: S = Semigroups().Subquotients().example()
sage: s = S.ambient().an_element()
sage: s, s.parent()
(42, An example of a semigroup: the left zero semigroup)
Topological
alias of `sage.categories.topological_spaces.TopologicalSpaces`

class WithRealizations(category, *args)
Bases: `sage.categories.with_realizations.WithRealizationsCategory`

class ParentMethods

class Realizations(parent_with_realization)
Bases: `sage.categories.realizations.Category_realization_of_parent`

super_categories()

EXAMPLES:

```sage
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]
```

a_realization()
Return a realization of self.

EXAMPLES:

```sage
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
```

facade_for()
Return the parents self is a facade for, that is the realizations of self

EXAMPLES:

```sage
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: A.facade_for()
[The subset algebra of {1, 2, 3} over Rational Field in the Fundamental basis, The subset algebra of {1, 2, 3} over Rational Field in the In basis, The subset algebra of {1, 2, 3} over Rational Field in the Out basis]
```

3.136. Sets
inject_shorthands (shorthands=None, verbose=True)
Import standard shorthands into the global namespace.

INPUT:
• shorthands – a list (or iterable) of strings (default: self._shorthands) or "all" (for self._shorthands_all)
• verbose – boolean (default True); whether to print the defined shorthands

EXAMPLES:

When computing with a set with multiple realizations, like SymmetricFunctions or SubsetAlgebra, it is convenient to define shorthands for the various realizations, but cumbersome to do it by hand:

```
sage: S = SymmetricFunctions(ZZ); S
Symmetric Functions over Integer Ring
sage: s = S.s(); s
Symmetric Functions over Integer Ring in the Schur basis
sage: e = S.e(); e
Symmetric Functions over Integer Ring in the elementary basis
```

This method automatizes the process:

```
sage: S.inject_shorthands()
Defining e as shorthand for Symmetric Functions over Integer Ring in → the elementary basis
Defining f as shorthand for Symmetric Functions over Integer Ring in → the forgotten basis
Defining h as shorthand for Symmetric Functions over Integer Ring in → the homogeneous basis
Defining m as shorthand for Symmetric Functions over Integer Ring in → the monomial basis
Defining p as shorthand for Symmetric Functions over Integer Ring in → the powersum basis
Defining s as shorthand for Symmetric Functions over Integer Ring in → the Schur basis
```

```
```

```
sage: e
Symmetric Functions over Integer Ring in the elementary basis
sage: p
Symmetric Functions over Integer Ring in the powersum basis
sage: s
Symmetric Functions over Integer Ring in the Schur basis
```

Sometimes, like for symmetric functions, one can request for all shorthands to be defined, including less common ones:

```
sage: S.inject_shorthands("all")
```

(continues on next page)
Defining `m` as shorthand for Symmetric Functions over Integer Ring in the monomial basis
Defining `o` as shorthand for Symmetric Functions over Integer Ring in the orthogonal basis
Defining `p` as shorthand for Symmetric Functions over Integer Ring in the powersum basis
Defining `s` as shorthand for Symmetric Functions over Integer Ring in the Schur basis
Defining `sp` as shorthand for Symmetric Functions over Integer Ring in the symplectic basis
Defining `st` as shorthand for Symmetric Functions over Integer Ring in the irreducible symmetric group character basis
Defining `w` as shorthand for Symmetric Functions over Integer Ring in the Witt basis

The messages can be silenced by setting `verbose=False`:

```
sage: Q = QuasiSymmetricFunctions(ZZ)
sage: Q.inject_shorthands(\text{verbose}=\text{False})
```

```
```

```
sage: F
Quasisymmetric functions over the Integer Ring in the Fundamental basis
sage: M
Quasisymmetric functions over the Integer Ring in the Monomial basis
```

One can also just import a subset of the shorthands:

```
sage: SQ = SymmetricFunctions(QQ)
sage: SQ.inject_shorthands([\text{'p'}, \text{'s'}], \text{verbose}=\text{False})
```

```
sage: p
Symmetric Functions over Rational Field in the powersum basis
sage: s
Symmetric Functions over Rational Field in the Schur basis
```

Note that `e` is left unchanged:

```
sage: e
Symmetric Functions over Integer Ring in the elementary basis
```

**realizations()**

Return all the realizations of `self` that `self` is aware of.

**EXAMPLES:**

```
sage: A = Sets().\text{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 subset algebra of \{1, 2, 3\} over Rational Field in the In basis, The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis]
```
Note: Constructing a parent $P$ in the category $A.Realizations()$ automatically adds $P$ to this list by calling $A._register_realization(A)$

example($base\_ring=None, set=None$)

Return an example of set with multiple realizations, as per $Category.example()$.

EXAMPLES:

```
sage: Sets().WithRealizations().example()
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
```

extra_super_categories()

A set with multiple realizations is a facade parent.

EXAMPLES:

```
sage: Sets().WithRealizations().extra_super_categories()
[Category of facade sets]
sage: Sets().WithRealizations().super_categories()
[Category of facade sets]
```

example($choice=None$)

Return examples of objects of $Sets()$, as per $Category.example()$.

EXAMPLES:

```
sage: Sets().example()
Set of prime numbers (basic implementation)
sage: Sets().example("inherits")
Set of prime numbers
sage: Sets().example("facade")
Set of prime numbers (facade implementation)
sage: Sets().example("wrapper")
Set of prime numbers (wrapper implementation)
```

super_categories()

We include SetsWithPartialMaps between Sets and Objects so that we can define morphisms between sets that are only partially defined. This is also to have the Homset constructor not complain that SetsWithPartialMaps is not a supercategory of Fields, for example.

EXAMPLES:

```
sage: Sets().super_categories()
[Category of sets with partial maps]
```

sage.categories.sets_cat.print_compare(x,y)


INPUT:

- $x$ – an element
• \( y \) – an element

EXAMPLES:

```python
sage: from sage.categories.sets_cat import print_compare
sage: print_compare(1, 2)
1 != 2
sage: print_compare(1, 1)
1 == 1
```

### 3.137 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 \emph{set with a grading} is a set \( S \) equipped with a grading by some other set \( I \) (by default the set \( \mathbb{N} \) of the non-negative integers):

\[
S = \bigcup_{i \in I} S_i
\]

where the \emph{graded components} \( S_i \) are (usually finite) sets. The \emph{grading} function maps each element \( s \) of \( S \) to its \emph{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 \emph{graded_component()} or \emph{subset()}. If only \emph{subset()} is implemented, the first argument must be the grading for compatibility with \emph{graded_component()}. Additionally either the parent must implement \emph{grading()} or its elements must implement a method \emph{grade()}. See the example \emph{sage.categories.examples.sets_with_grading.NonNegativeIntegers}.

Finally, if the graded set is enumerated (see \emph{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:

```python
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 \emph{grading function} is given by \emph{N.grading}:

```python
sage: N.grading(4)
4
```

The graded component \( S_i \) is the set of all integer partitions of \( i \):
Here are some information about this category:

```python
sage: SetsWithGrading()
Category of sets with grading
sage: SetsWithGrading().super_categories()
[Category of sets]
sage: SetsWithGrading().all_super_categories()
[Category of sets with grading,
 Category of sets,
 Category of sets with partial maps,
 Category of objects]
```

Todo:

- This should be moved to `Sets().WithGrading()`.
- Should the grading set be a parameter for this category?
- Does the enumeration need to be compatible with the grading? Be careful that the fact that graded components are allowed to be finite or infinite make the answer complicated.

```python
class ParentMethods

    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:

```python
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:

```python
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:

```python
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]
```

**super_categories()**

EXAMPLES:

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

3.138 SetsWithPartialMaps

```python
class sage.categories.sets_with_partial_maps.SetsWithPartialMaps(r=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:

```python
sage: SetsWithPartialMaps()
Category of sets with partial maps
sage: SetsWithPartialMaps().super_categories()
[Category of objects]
```

```python
class sage.categories.sets_with_partial_maps.SetsWithPartialMaps(r=None)
```
3.139 Shephard Groups

class sage.categories.shephard_groups.ShephardGroups(s=None)
    Bases: sage.categories.category_singleton.Category_singleton

The category of Shephard groups.

EXAMPLES:

```python
sage: from sage.categories.shephard_groups import ShephardGroups
c
sage: C = ShephardGroups(); C
Category of shephard groups
```

```
sage: from sage.categories.shephard_groups import ShephardGroups
c
sage: ShephardGroups().super_categories()
[Category of finite generalized coxeter groups]
```

3.140 Simplicial Complexes

class sage.categories.simplicial_complexes.SimplicialComplexes(s=\texttt{None})
    Bases: sage.categories.category_singleton.Category_singleton

The category of abstract simplicial complexes.

An abstract simplicial complex \( A \) is a collection of sets \( X \) such that:

- \( \emptyset \in A \),
- if \( X \subset Y \in A \), then \( X \in A \).

Todo: Implement the category of simplicial complexes considered as \textit{CW complexes} and rename this to the category of \texttt{AbstractSimplicialComplexes} with appropriate functors.

EXAMPLES:

```python
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
c
sage: C = SimplicialComplexes(); C
Category of simplicial complexes
```

```python
sage: from sage.categories.simplicial_complexes import SimplicialComplexes
c
sage: SimplicialComplexes().super_categories()
[Category of finite generalized coxeter groups]
```

class Finite(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

Category of finite simplicial complexes.

class ParentMethods
class ParentMethods

faces()

Return the faces of self.

EXAMPLES:

    sage: S = SimplicialComplex([[1,3,4], [1,2],[2,5],[4,5]])
    sage: S.faces()
    {-1: {()},
     0: { (1,), (2,), (3,), (4,), (5,) },
     1: { (1, 2), (1, 3), (1, 4), (2, 5), (3, 4), (4, 5) },
     2: { (1, 3, 4) }}

facets()

Return the facets of self.

EXAMPLES:

    sage: S = SimplicialComplex([[1,3,4], [1,2],[2,5],[4,5]])
    sage: sorted(S.facets())
    [(1, 2), (1, 3, 4), (2, 5), (4, 5)]

super_categories()

EXAMPLES:

    sage: from sage.categories.simplicial_complexes import SimplicialComplexes
    sage: SimplicialComplexes().super_categories()
    [Category of sets]

3.141 Simplicial Sets

class sage.categories.simplicial_sets.SimplicialSets(s=None)

Bases: sage.categories.category_singleton.Category_singleton

The category of simplicial sets.

A simplicial set $X$ is a collection of sets $X_i$, indexed by the non-negative integers, together with maps

\[
\begin{align*}
   d_i : X_n &\to X_{n-1}, \quad 0 \leq i \leq n \quad \text{(face maps)} \\
   s_j : X_n &\to X_{n+1}, \quad 0 \leq j \leq n \quad \text{(degeneracy maps)}
\end{align*}
\]
satisfying the simplicial identities:

\[
\begin{align*}
    d_i d_j &= d_{j-1} d_i & \text{if } i < j \\
    d_i s_j &= s_{j-1} d_i & \text{if } i < j \\
    d_j s_j &= 1 = d_{j+1} s_j \\
    d_i s_j &= s_j d_i & \text{if } i > j + 1 \\
    s_i s_j &= s_{j+1} s_i & \text{if } i \leq j
\end{align*}
\]

Morphisms are sequences of maps \( f_i : X_i \to Y_i \) which commute with the face and degeneracy maps.

**EXAMPLES:**

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: C = SimplicialSets(); C
Category of simplicial sets

class Finite(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

    Category of finite simplicial sets.

    The objects are simplicial sets with finitely many non-degenerate simplices.

class Homsets(category, *args):
    Bases: sage.categories.homsets.HomsetsCategory

class Endset(base_category):
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class ParentMethods

    one()
    Return the identity morphism in \( \text{Hom}(S, S) \).

    EXAMPLES:

    ```python
    sage: T = simplicial_sets.Torus()
    sage: Hom(T, T).identity()
    Simplicial set endomorphism of Torus
       Defn: Identity map
    ```

    is_finite()
    Return True if this simplicial set is finite, i.e., has a finite number of nondegenerate simplices.

    EXAMPLES:

    ```python
    sage: simplicial_sets.Torus().is_finite()
    True
    sage: C5 = groups.misc.MultiplicativeAbelian([5])
    sage: simplicial_sets.ClassifyingSpace(C5).is_finite()
    False
    ```

    is_pointed()
    Return True if this simplicial set is pointed, i.e., has a base point.

    EXAMPLES:
sage: from sage.homology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: w = AbstractSimplex(0)
sage: e = AbstractSimplex(1)
sage: X = SimplicialSet({e: (v, w)})
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)
sage: X.is_pointed()
False
sage: Y.is_pointed()
True

set_base_point(point)
Return a copy of this simplicial set in which the base point is set to point.

INPUT:
- point - a 0-simplex in this simplicial set

EXAMPLES:

sage: from sage.homology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v_0')
sage: w = AbstractSimplex(0, name='w_0')
sage: e = AbstractSimplex(1)
sage: X = SimplicialSet({e: (v, w)})
sage: Y = SimplicialSet({e: (v, w)}, base_point=w)
sage: Y.base_point()
w_0
sage: X_star = X.set_base_point(w)
sage: X_star.base_point()
w_0
sage: Y_star = Y.set_base_point(v)
sage: Y_star.base_point()
v_0

class Pointed(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class Finite(base_category)
Bases: sage.categories.category_with_axiom.CategoryWithAxiom

class ParentMethods

fat_wedge(n)
Return the n-th fat wedge of this pointed simplicial set.

This is the subcomplex of the n-fold product \( X^n \) consisting of those points in which at least one factor is the base point. Thus when \( n = 2 \), this is the wedge of the simplicial set with itself, but when \( n \) is larger, the fat wedge is larger than the n-fold wedge.

EXAMPLES:

sage: S1 = simplicial_sets.Sphere(1)
sage: S1.fat_wedge(0)
Point
sage: S1.fat_wedge(1)
S^1
sage: S1.fat_wedge(2).fundamental_group()
(continues on next page)
Finitely presented group \(< e_0, e_1 | >\)
\[
sage: S1.fat_wedge(4).homology()
{0: 0, 1: \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}, 2: \mathbb{Z}^6, 3: \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}}
\]

**smash_product** (*others*)
Return the smash product of this simplicial set with *others*.

**INPUT:**
- *others* — one or several simplicial sets

**EXAMPLES:**
\[
\begin{align*}
sage: S1 & = \text{simplicial_sets.Sphere}(1) \\
sage: RP2 & = \text{simplicial_sets.RealProjectiveSpace}(2) \\
sage: X & = S1.\text{smash_product}(RP2) \\
sage: X.\text{homology}(\text{base_ring}=\text{GF}(2)) \\
& \{0: \text{Vector space of dimension 0 over Finite Field of size 2}, \\
& 1: \text{Vector space of dimension 0 over Finite Field of size 2}, \\
& 2: \text{Vector space of dimension 1 over Finite Field of size 2}, \\
& 3: \text{Vector space of dimension 1 over Finite Field of size 2}\}
\end{align*}
\]

**unset_base_point**()
Return a copy of this simplicial set in which the base point has been forgotten.

**EXAMPLES:**
\[
\begin{align*}
sage: \text{from sage.homology.simplicial_set import } \text{AbstractSimplex, } \text{SimplicialSet} \\
sage: v & = \text{AbstractSimplex}(0, \text{name}='v_0') \\
sage: w & = \text{AbstractSimplex}(0, \text{name}='w_0') \\
sage: e & = \text{AbstractSimplex}(1) \\
sage: Y & = \text{SimplicialSet}({e: (v, w)}, \text{base_point}=w) \\
sage: Y.\text{is_pointed()} \\
& \text{True} \\
sage: Y.\text{base_point()} \\
& w_0 \\
sage: Z & = Y.\text{unset_base_point()} \\
sage: Z.\text{is_pointed()} \\
& \text{False}
\end{align*}
\]

**class** *ParentMethods*

**base_point**()
Return this simplicial set’s base point

**EXAMPLES:**
\[
\begin{align*}
sage: \text{from sage.homology.simplicial_set import } \text{AbstractSimplex, } \text{SimplicialSet} \\
sage: v & = \text{AbstractSimplex}(0, \text{name}='*') \\
sage: e & = \text{AbstractSimplex}(1) \\
sage: S1 & = \text{SimplicialSet}({e: (v, v)}, \text{base_point}=v)
\end{align*}
\]

(continued from previous page)

```python
sage: S1.is_pointed()
True
sage: S1.base_point()
```

**base_point_map** *(domain=None)*

Return a map from a one-point space to this one, with image the base point.

This raises an error if this simplicial set does not have a base point.

**INPUT:**

- **domain** – optional, default None. Use this to specify a particular one-point space as the domain. The default behavior is to use the `sage.homology.simplicial_set.Point()` function to use a standard one-point space.

**EXAMPLES:**

```python
sage: T = simplicial_sets.Torus()
sage: f = T.base_point_map(); f
Simplicial set morphism:
  From: Point
  To:  Torus
  Defn: Constant map at (v_0, v_0)
sage: S3 = simplicial_sets.Sphere(3)
sage: g = S3.base_point_map()
sage: f.domain() == g.domain()
True
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
sage: temp = simplicial_sets.Simplex(0)
sage: pt = temp.set_base_point(temp.n_cells(0)[0])
sage: h = RP3.base_point_map(domain=pt)
sage: f.domain() == h.domain()
False
sage: C5 = groups.misc.MultiplicativeAbelian([5])
sage: BC5 = simplicial_sets.ClassifyingSpace(C5)
sage: BC5.base_point_map()
Simplicial set morphism:
  From: Point
  To:  Classifying space of Multiplicative Abelian group isomorphic to C5
  Defn: Constant map at 1
```

**connectivity** *(max_dim=None)*

Return the connectivity of this pointed simplicial set.

**INPUT:**

- **max_dim** – specify a maximum dimension through which to check. This is required if this simplicial set is simply connected and not finite. The dimension of the first nonzero homotopy group. If simply connected, this is the same as the dimension of the first nonzero homology group.

**Warning:** See the warning for the `is_simply_connected()` method.

The connectivity of a contractible space is `+Infinity`.

**EXAMPLES:**
Return the fundamental group of this pointed simplicial set.

**INPUT:**
- `simplify` (bool, optional `True`) – if `False`, then return a presentation of the group in terms of generators and relations. If `True`, the default, simplify as much as GAP is able to.

**Algorithm:** we compute the edge-path group – see Section 19 of [Kan1958] and Wikipedia article Fundamental_group. Choose a spanning tree for the connected component of the 1-skeleton containing the base point, and then the group’s generators are given by the non-degenerate edges. There are two types of relations: $e = 1$ if $e$ is in the spanning tree, and for every 2-simplex, if its faces are $e_0$, $e_1$, and $e_2$, then we impose the relation $e_0e_1^{-1}e_2 = 1$, where we first set $e_i = 1$ if $e_i$ is degenerate.

**EXAMPLES:**

```python
sage: S1 = simplicial_sets.Sphere(1)
sage: eight = S1.wedge(S1)
sage: eight.fundamental_group()  # free group on 2 generators
Finitely presented group < e0, e1 | >
```

The fundamental group of a disjoint union of course depends on the choice of base point:

```python
sage: T = simplicial_sets.Torus()
sage: K = simplicial_sets.KleinBottle()
sage: X = T.disjoint_union(K)
sage: X_0 = X.set_base_point(X.n_cells(0)[0])
sage: X_0.fundamental_group().is_abelian()  # True
sage: X_1 = X.set_base_point(X.n_cells(0)[1])
sage: X_1.fundamental_group().is_abelian()  # False
```

Compute the fundamental group of some classifying spaces:

```python
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
sage: RP3.fundamental_group()  # Finitely presented group < e | e^2 >
```
```python
sage: C5 = groups.misc.MultiplicativeAbelian([5])
sage: BC5 = C5.nerve()
sage: BC5.fundamental_group()
Finitely presented group < e0 | e0^5 >
sage: Sigma3 = groups.permutation.Symmetric(3)
sage: BSigma3 = Sigma3.nerve()
sage: pi = BSigma3.fundamental_group(); pi
Finitely presented group < e0, e1 | e0^2, e1^3, (e0*e1^-1)^2 >
sage: pi.order()
6
sage: pi.is_abelian()
False
```

**is_simply_connected()**

Return True if this pointed simplicial set is simply connected.

**Warning:** Determining simple connectivity is not always possible, because it requires determining when a group, as given by generators and relations, is trivial. So this conceivably may give a false negative in some cases.

**EXAMPLES:**

```python
sage: T = simplicial_sets.Torus()
sage: T.is_simply_connected()
False
sage: T.suspension().is_simply_connected()
True
sage: simplicial_sets.KleinBottle().is_simply_connected()
False
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: (S2.wedge(S3)).is_simply_connected()
True
sage: X = S2.disjoint_union(S3)
sage: X = X.set_base_point(X.n_cells(0)[0])
sage: X.is_simply_connected()
False
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)
sage: BC3.is_simply_connected()
False
```

**class SubcategoryMethods**

**Pointed()**

A simplicial set is pointed if it has a distinguished base point.

**EXAMPLES:**

```python
sage: from sage.categories.simplicial_sets import SimplicialSets
sage: SimplicialSets().Pointed().Finite()
Category of finite pointed simplicial sets
```

(continues on next page)
3.142 Super Algebras

**class** `sage.categories.super_algebras.SuperAlgebras(base_category)`

**Bases:** `sage.categories.super_modules.SuperModulesCategory`

The category of super algebras.

An \(R\)-super algebra is an \(R\)-super module \(A\) endowed with an \(R\)-algebra structure satisfying

\[
A_0 A_0 \subseteq A_0, \quad A_0 A_1 \subseteq A_1, \quad A_1 A_0 \subseteq A_1, \quad A_1 A_1 \subseteq A_0
\]

and \(1 \in A_0\).

**EXAMPLES:**

```python
sage: Algebras(ZZ).Super()
Category of super algebras over Integer Ring
```

**class ParentMethods**

**graded_algebra()**

Return the associated graded algebra to `self`.

**Warning:** Because a super module \(M\) is naturally \(\mathbb{Z}/2\mathbb{Z}\)-graded, and graded modules have a natural filtration induced by the grading, if \(M\) has a different filtration, then the associated graded module \(\text{gr} M \neq M\). This is most apparent with super algebras, such as the differential Weyl algebra, and the multiplication may not coincide.

**extra_super_categories()**

**EXAMPLES:**

```python
sage: Algebras(ZZ).Super().super_categories()  # indirect doctest
[Category of graded algebras over Integer Ring,
 Category of super modules over Integer Ring]
```

3.143 Super algebras with basis

**class** `sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis(base_category)`

**Bases:** `sage.categories.super_modules.SuperModulesCategory`

...
The category of super algebras with a distinguished basis

EXAMPLES:

```python
sage: C = Algebras(ZZ).WithBasis().Super(); C
Category of super algebras with basis over Integer Ring
```

class ParentMethods

```python
graded_algebra()
```

Return the associated graded module to `self`.

See `AssociatedGradedAlgebra` for the definition and the properties of this.

See also:

graded_algebra()

EXAMPLES:

```python
sage: W.<x,y> = algebras.DifferentialWeyl(QQ)
sage: W.graded_algebra()
Graded Algebra of Differential Weyl algebra of polynomials in x, y over Rational Field
```

```python
extra_super_categories()
```

EXAMPLES:

```python
sage: C = Algebras(ZZ).WithBasis().Super()
sage: sorted(C.super_categories(), key=str) # indirect doctest
[Category of graded algebras with basis over Integer Ring,
 Category of super algebras over Integer Ring,
 Category of super modules with basis over Integer Ring]
```

### 3.144 Super Hopf algebras with basis

class sage.categories.super_hopf_algebras_with_basis.SuperHopfAlgebrasWithBasis(base_category)

Bases: sage.categories.super_modules.SuperModulesCategory

The category of super Hopf algebras with a distinguished basis.

EXAMPLES:

```python
sage: C = HopfAlgebras(ZZ).WithBasis().Super(); C
Category of super hopf algebras with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of super algebras with basis over Integer Ring,
 Category of super coalgebras with basis over Integer Ring,
 Category of super hopf algebras over Integer Ring]
```

### 3.145 Super modules

class sage.categories.super_modules.SuperModules(base_category)

Bases: sage.categories.super_modules.SuperModulesCategory
The category of super modules.

An \( R \)-super module (where \( R \) is a ring) is an \( R \)-module \( M \) equipped with a decomposition \( M = M_0 \oplus M_1 \) into two \( R \)-submodules \( M_0 \) and \( M_1 \) (called the even part and the odd part of \( M \), respectively).

Thus, an \( R \)-super module automatically becomes a \( \mathbb{Z}/2\mathbb{Z} \)-graded \( R \)-module, with \( M_0 \) being the degree-0 component and \( M_1 \) being the degree-1 component.

**EXAMPLES:**

```python
sage: Modules(ZZ).Super()
Category of super modules over Integer Ring
sage: Modules(ZZ).Super().super_categories()
[Category of graded modules over Integer Ring]
```

The category of super modules defines the super structure which shall be preserved by morphisms:

```python
class ElementMethods

    is_even()
    Return if self is an even element.

    EXAMPLES:

    sage: cat = Algebras(QQ).WithBasis().Super()
    sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
    sage: C.degree_on_basis = sum
    sage: C.basis()[2,2,1].is_even()
    False
    sage: C.basis()[2,2].is_even()
    True

    is_even_odd()
    Return 0 if self is an even element or 1 if an odd element.

    Note: The default implementation assumes that the even/odd is determined by the parity of degree().

    Overwrite this method if the even/odd behavior is desired to be independent.

    EXAMPLES:

    sage: cat = Algebras(QQ).WithBasis().Super()
    sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
    sage: C.degree_on_basis = sum
    sage: C.basis()[2,2,1].is_even_odd()
    1
    sage: C.basis()[2,2].is_even_odd()
    0

    is_odd()
    Return if self is an odd element.

    EXAMPLES:
```python
sage: cat = Algebras(QQ).WithBasis().Super()
sage: C = CombinatorialFreeModule(QQ, Partitions(), category=cat)
sage: C.degree_on_basis = sum
sage: C.basis()[2,2,1].is_odd()
True
sage: C.basis()[2,2].is_odd()
False
```

### class ParentMethods

#### extra_super_categories()

Adds `VectorSpaces` to the super categories of `self` if the base ring is a field.

**EXAMPLES:**

```python
sage: Modules(QQ).Super().extra_super_categories()
[Category of vector spaces over Rational Field]
sage: Modules(ZZ).Super().extra_super_categories()
[]
```

This makes sure that `Modules(QQ).Super()` returns an instance of `SuperModules` and not a join category of an instance of this class and of `VectorSpaces(QQ)`:  

```python
sage: type(Modules(QQ).Super())
<class 'sage.categories.super_modules.SuperModules_with_category'>
```

**Todo:** Get rid of this workaround once there is a more systematic approach for the alias `Modules(QQ) -> VectorSpaces(QQ)`. Probably the latter should be a category with axiom, and covariant constructions should play well with axioms.

#### super_categories()

**EXAMPLES:**

```python
sage: Modules(ZZ).Super().super_categories()
[Category of graded modules over Integer Ring]
```

**Nota bene:**

```python
sage: Modules(QQ).Super()
Category of super modules over Rational Field
sage: Modules(QQ).Super().super_categories()
[Category of graded modules over Rational Field]
```

```python
class sage.categories.super_modules.SuperModulesCategory(base_category)

Bases: sage.categories.covariant_functorial_construction.CovariantConstructionCategory, sage.categories.category_types.Category_over_base_ring

**EXAMPLES:**

```python
sage: C = Algebras(QQ).Super()
sage: C
Category of super algebras over Rational Field
sage: C.base_category()
Category of algebras over Rational Field
sage: sorted(C.super_categories(), key=str)
```
[Category of graded algebras over Rational Field,  
Category of super modules over Rational Field]

```python
sage: AlgebrasWithBasis(QQ).Super().base_ring()
Rational Field
sage: HopfAlgebrasWithBasis(QQ).Super().base_ring()
Rational Field
```

```python
classmethod default_super_categories(category, *args)
```

Return the default super categories of \( F_{\mathcal{C}}(A, B, \ldots) \) for \( A, B, \ldots \) parents in \( \mathcal{C} \).

**INPUT:**
- `cls` – the category class for the functor \( F \)
- `category` – a category \( \mathcal{C} \)
- `*args` – further arguments for the functor

**OUTPUT:**
A join category.

This implements the property that subcategories constructed by the set of whitelisted axioms is a subcategory.

**EXAMPLES:**

```python
sage: HopfAlgebras(ZZ).WithBasis().FiniteDimensional().Super()
# indirect doctest
Category of finite dimensional super hopf algebras with basis over Integer Ring
```

### 3.146 Super modules with basis

```python
class sage.categories.super_modules_with_basis.SuperModulesWithBasis(base_category)
```

Bases: `sage.categories.super_modules.SuperModulesCategory`

The category of super modules with a distinguished basis.

An \( R \)-super module with a distinguished basis is an \( R \)-super module equipped with an \( R \)-module basis whose elements are homogeneous.

**EXAMPLES:**

```python
sage: C = GradedModulesWithBasis(ZZ); C
Category of graded modules with basis over Integer Ring
sage: sorted(C.super_categories(), key=str)
[Category of filtered modules with basis over Integer Ring,  
Category of graded modules over Integer Ring]
```

```python
sage: C is ModulesWithBasis(ZZ).Graded()
True
```

```python
class ElementMethods
```

```python
even_component()
```

Return the even component of `self`.  

---

**Chapter 3. Individual Categories**

---

**Sage Reference Manual: Category Framework, Release 8.7**

(continued from previous page)
EXAMPLES:

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x*y + x - 3*y + 4
sage: a.even_component()
x*y + 4
```

### is_even_odd()
Return 0 if `self` is an even element and 1 if `self` is an odd element.

EXAMPLES:

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x + y
sage: a.is_even_odd()
1
sage: a = x*y + 4
sage: a.is_even_odd()
0
sage: a = x + 4
sage: a.is_even_odd()
Traceback (most recent call last):
  ... ValueError: element is not homogeneous
```

### is_super_homogeneous()
Return whether this element is homogeneous, in the sense of a super module (i.e., is even or odd).

EXAMPLES:

```python
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x*y + 4
sage: a.is_super_homogeneous()
True
sage: a.is_homogeneous()
False
```

The exterior algebra has a \(\mathbb{Z}\) grading, which induces the \(\mathbb{Z}/2\mathbb{Z}\) grading. However the definition of homogeneous elements differs because of the different gradings:

```python
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: (x*y).is_even_odd()
0
```

### odd_component()

### is_super_homogeneous()
Return the odd component of self.

EXAMPLES:

```
sage: Q = QuadraticForm(QQ, 2, [1,2,3])
sage: C.<x,y> = CliffordAlgebra(Q)
sage: a = x*y + x - 3*y + 4
sage: a.odd_component()
x - 3*y
```

class ParentMethods

3.147 Topological Spaces

class sage.categories.topological_spaces.TopologicalSpaces(category, *args)

Bases: sage.categories.topological_spaces.TopologicalSpacesCategory

The category of topological spaces.

EXAMPLES:

```
sage: Sets().Topological()
Category of topological spaces
sage: Sets().Topological().super_categories()
[Category of sets]
```

The category of topological spaces defines the topological structure, which shall be preserved by morphisms:

```
sage: Sets().Topological().additional_structure()
Category of topological spaces
```

class Compact (base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of compact topological spaces.

class Connected (base_category)

Bases: sage.categories.category_with_axiom.CategoryWithAxiom

The category of connected topological spaces.

class SubcategoryMethods

Compact ()

Return the subcategory of the compact objects of self.

EXAMPLES:

```
sage: Sets().Topological().Compact()
Category of compact topological spaces
```

Connected ()

Return the full subcategory of the connected objects of self.

EXAMPLES:

```
sage: Sets().Topological().Connected()
Category of connected topological spaces
```
class sage.categories.topological_spaces.TopologicalSpacesCategory(category, *args):
    Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory

3.148 Kac-Moody Algebras With Triangular Decomposition Basis

AUTHORS:

- Travis Scrimshaw (07-15-2017): Initial implementation

class sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras(base, name=None):
    Bases: sage.categories.category_types.Category_over_base_ring

Category of Kac-Moody algebras with a distinguished basis that respects the triangular decomposition.

We require that the grading group is the root lattice of the appropriate Cartan type.

class ElementMethods

    part()

    Return whether the element v is in the lower, zero, or upper part of self.

    OUTPUT:

    -1 if v is in the lower part, 0 if in the zero part, or 1 if in the upper part

    EXAMPLES:

    sage: L = LieAlgebra(QQ, cartan_type="F4")
    sage: L.inject_variables()
    Defining e1, e2, e3, e4, f1, f2, f3, f4, h1, h2, h3, h4
    sage: e1.part()
    1
    sage: f4.part()
    -1
    sage: (h2 + h3).part()
    0
    sage: (f1.bracket(f2) + 4*f4).part()
    -1
    sage: (e1 + f1).part()
    Traceback (most recent call last):
    ...
    ValueError: element is not in one part

class ParentMethods

    e(i=None)

    Return the generators e of self.

    INPUT:

    - i – (optional) if specified, return just the generator e_i

    EXAMPLES:

    sage: L = lie_algebras.so(QQ, 5)
    sage: L.e()
Finite family {1: E[alpha[1]], 2: E[alpha[2]]}
sage: L.e(1)
E[alpha[1]]

\( f \) (i=None)
Return the generators \( f \) of self.

INPUT:
• \( i \) – (optional) if specified, return just the generator \( f_i \)

EXAMPLES:
sage: L = lie_algebras.so(QQ, 5)
sage: L.f()
Finite family {1: E[-alpha[1]], 2: E[-alpha[2]]}
sage: L.f(1)
E[-alpha[1]]

verma_module (la, basis_key=None, **kwds)
Return the Verma module with highest weight \( \lambda \) over self.

INPUT:
 • basis_key – (optional) a key function for the indexing set of the basis elements of self

EXAMPLES:
sage: L = lie_algebras.sl(QQ, 3)
sage: P = L.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: M = L.verma_module(La[1]+La[2])
sage: M

super_categories ()
EXAMPLES:
sage: from sage.categories.triangular_kac_moody_algebras import TriangularKacMoodyAlgebras
sage: TriangularKacMoodyAlgebras(QQ).super_categories()
[Join of Category of graded lie algebras with basis over Rational Field and Category of kac moody algebras over Rational Field]
class ElementMethods

\texttt{radical(*args, **kwds)}

Return the radical of this element, i.e. the product of its irreducible factors.

This default implementation calls \texttt{squarefree_decomposition} if available, and \texttt{factor} otherwise.

\textbf{See also:}

\texttt{squarefree_part()}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Pol.<x> = QQ[]
sage: (x^2*(x-1)^3).radical()
x^2 - x
sage: pol = 37 * (x-1)^3 * (x-2)^2 * (x-1/3)^7 * (x-3/7)
sage: pol.radical()
37*x^4 - 2923/21*x^3 + 1147/7*x^2 - 1517/21*x + 74/7
sage: Integer(10).radical() 10
sage: Integer(-100).radical() 10
sage: Integer(0).radical()
Traceback (most recent call last):
... ArithmeticError: Radical of 0 not defined.
\end{verbatim}

The next example shows how to compute the radical of a number, assuming no prime > 100000 has exponent > 1 in the factorization:

\begin{verbatim}
sage: n = 2^1000-1; n / radical(n, limit=100000) 125
\end{verbatim}

\texttt{squarefree_part()}

Return the square-free part of this element, i.e. the product of its irreducible factors appearing with odd multiplicity.

This default implementation calls \texttt{squarefree_decomposition}.

\textbf{See also:}

\texttt{radical()}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Pol.<x> = QQ[]
sage: (x^2*(x-1)^3).squarefree_part()
x - 1
sage: pol = 37 * (x-1)^3 * (x-2)^2 * (x-1/3)^7 * (x-3/7)
sage: pol.squarefree_part()
37*x^3 - 1369/21*x^2 + 703/21*x - 37/7
\end{verbatim}

class ParentMethods

\texttt{is_unique_factorization_domain(\emph{proof}=True)}

Return True, since this in an object of the category of unique factorization domains.
EXAMPLES:

```python
sage: Parent(QQ,category=UniqueFactorizationDomains()).is_unique_factorization_domain()
True
```

**additional_structure()**

Return whether `self` is a structure category.

See also:

`Category.additional_structure()`

The category of unique factorization domains does not define additional structure: a ring morphism between unique factorization domains is a unique factorization domain morphism.

EXAMPLES:

```python
sage: UniqueFactorizationDomains().additional_structure()
```

**super_categories()**

EXAMPLES:

```python
sage: UniqueFactorizationDomains().super_categories()
[Category of gcd domains]
```

### 3.150 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 is 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:

```python
sage: from sage.categories.unital_algebras import UnitalAlgebras
sage: C = UnitalAlgebras(ZZ); C
Category of unital algebras over Integer Ring
```

**class 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') over Rational Field
sage: A.from_base_ring(1)
B

```python
class WithBasis(base_category):
    Bases:
        sage.categories.category_with_axiom.CategoryWithAxiom_over_base_ring

    class ParentMethods:

        from_base_ring()

        from_base_ring_from_one_basis(r)
        Implement the canonical embedding from the ground ring.
        INPUT:
        • r – an element of the coefficient ring
        EXAMPLES:

        sage: A = AlgebrasWithBasis(QQ).example()
sage: A.from_base_ring_from_one_basis(3)
         3*B
sage: A.from_base_ring(3)
         3*B
sage: A(3)
         3*B

    one()
    Return the multiplicative unit element.
    EXAMPLES:

    sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one()

    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()
sage: A(4)

    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.

3.150. Unital algebras
EXAMPLES:

```python
sage: A = AlgebrasWithBasis(QQ).example()
sage: A.one_basis()
word:
sage: A.one_from_one_basis()
B[word: ]
sage: A.one()
B[word: ]
```

Even if called in the wrong order, they should returns their respective one:

```python
sage: Bone().parent() is B
True
sage: Aone().parent() is A
True
```

### 3.151 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:**

```python
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`

**extra_super_categories()**

The category of vector spaces is closed under Cartesian products:

```python
sage: C = VectorSpaces(QQ)
Category of vector spaces over Rational Field
sage: C.CartesianProducts()
Category of Cartesian products of vector spaces over Rational Field
sage: C in C.CartesianProducts().super_categories()
True
```

*class* `DualObjects(category, *args)`

**Bases:** `sage.categories.dual.DualObjectsCategory`

**extra_super_categories()**

Returns the dual category

**EXAMPLES:**

The category of algebras over the Rational Field is dual to the category of coalgebras over the same field:
class ElementMethods

class ParentMethods

class TensorProducts\(\text{(category, *args)}\)
Bases: sage.categories.tensor.TensorProductsCategory

\textbf{extra\_super\_categories()}

The category of vector spaces is closed under tensor products:

\begin{verbatim}
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]
\end{verbatim}

class WithBasis\(\text{(base\_category)}\)
Bases: sage.categories.category\_with\_axiom.CategoryWithAxiom\_over\_base\_ring

class CartesianProducts\(\text{(category, *args)}\)
Bases: sage.categories.cartesian\_product.CartesianProductsCategory

\textbf{extra\_super\_categories()}

The category of vector spaces with basis is closed under Cartesian products:

\begin{verbatim}
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
\end{verbatim}

class TensorProducts\(\text{(category, *args)}\)
Bases: sage.categories.tensor.TensorProductsCategory

\textbf{extra\_super\_categories()}

The category of vector spaces with basis is closed under tensor products:

\begin{verbatim}
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
\end{verbatim}

\textbf{is\_abelian()}

Return whether this category is abelian.

This is always True since the base ring is a field.

\textbf{EXAMPLES:}

3.151. Vector Spaces 681
additional_structure()
Return None.
Indeed, the category of vector spaces defines no additional structure: a bimodule morphism between two
vector spaces is a vector space morphism.
See also:
Category.additional_structure()

Todo: Should this category be a CategoryWithAxiom?

EXAMPLES:

```
sage: VectorSpaces(QQ).additional_structure()
```

base_field()
Returns the base field over which the vector spaces of this category are all defined.

EXAMPLES:

```
sage: VectorSpaces(QQ).base_field()
Rational Field
```

super_categories()

EXAMPLES:

```
sage: VectorSpaces(QQ).super_categories()
[Category of modules over Rational Field]
```

3.152 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() # todo: not implemented
The symmetric group on (0, ..., 3)
sage: AffineWeylGroups().example() # todo: not implemented
```
ube (continues on next page)
This one will eventually be also in this category:

```python
sage: SymmetricGroup(4)
Symmetric group of order 4! as a permutation group
```

### class ElementMethods

#### bruhat_lower_covers_coroots()

Return all 2-tuples $(v, \alpha)$ where $v$ is covered by self and $\alpha$ is the positive coroot such that $\text{self} = v s_\alpha$ where $s_\alpha$ is the reflection orthogonal to $\alpha$.

**ALGORITHM:**

See `bruhat_lower_covers()` and `bruhat_lower_covers_reflections()` for Coxeter groups.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_lower_covers_coroots()
 (s3*s2*s1, alphacheck[2]), (s3*s1*s2, alphacheck[1])]
```

#### bruhat_upper_covers_coroots()

Returns all 2-tuples $(v, \alpha)$ where $v$ is covers self and $\alpha$ is the positive coroot such that $\text{self} = v s_\alpha$ where $s_\alpha$ is the reflection orthogonal to $\alpha$.

**ALGORITHM:**

See `bruhat_upper_covers()` and `bruhat_upper_covers_reflections()` for Coxeter groups.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',4], prefix="s")
sage: w = W.from_reduced_word([3,1,2,1])
sage: w.bruhat_upper_covers_coroots()
[(s1*s2*s3*s2*s1, alphacheck[3]),
 (s2*s3*s1*s2*s1, alphacheck[2] + alphacheck[3]),
 (s3*s4*s1*s2*s1, alphacheck[4]),
```

#### inversion_arrangement (side=’right’)

Return the inversion hyperplane arrangement of self.

**INPUT:**

- `side = 'right'` (default) or 'left'

**OUTPUT:**

A (central) hyperplane arrangement whose hyperplanes correspond to the inversions of self given as roots.

The side parameter determines on which side to compute the inversions.
EXAMPLES:

```python
sage: W = WeylGroup(['A',3])
sage: w = W.from_reduced_word([1, 2, 3, 1, 2])
sage: A = w.inversion_arrangement(); A
Arrangement of 5 hyperplanes of dimension 3 and rank 3
sage: A.hyperplanes()
(Hyperplane 0*a1 + 0*a2 + a3 + 0,
 Hyperplane 0*a1 + a2 + 0*a3 + 0,
 Hyperplane 0*a1 + a2 + a3 + 0,
 Hyperplane a1 + a2 + 0*a3 + 0,
 Hyperplane a1 + a2 + a3 + 0)
```

The identity element gives the empty arrangement:

```python
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:**

```python
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:**
**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() for i in [-1, 0, 1, 2]]
[0, 1, 2, 2]
```
sage: W = WeylGroup(['C',4,1])
sage: w = W.from_reduced_word([0,3,1,2,0])
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)

Return the successors of self in the quantum Bruhat graph on the parabolic quotient of the Weyl group determined by the subset of Dynkin nodes index_set.

INPUT:
• self – a Weyl group element, which is assumed to be of minimum length in its coset with respect to the parabolic subgroup
• index_set – (default: None) indicates the set of simple reflections used to generate the parabolic subgroup; the default value indicates that the subgroup is the identity
• roots – (default: False) if True, returns the list of 2-tuples (w, α) where w is a successor and α is the positive root associated with the successor relation
• quantum_only – (default: False) if True, returns only the quantum successors

EXAMPLES:

sage: W = WeylGroup(['A',3], prefix="s")
sage: w = W.from_reduced_word([3,1,2])
sage: w.quantum_bruhat_successors([1], roots = True)
[(s3, alpha[2]), (s1*s2*s3*s2, alpha[3]),
 (s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3])]
sage: w.quantum_bruhat_successors([1,3])
[1, s2*s3*s1*s2]
sage: w.quantum_bruhat_successors(roots = True)
[(s3*s1*s2*s1, alpha[1]),
 (s3*s1, alpha[2]),
 (s1*s2*s3*s2, alpha[3]),
 (s2*s3*s1*s2, alpha[1] + alpha[2] + alpha[3])]
sage: w.quantum_bruhat_successors()
[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])
Traceback (most recent call last):
... ValueError: s2*s3 is not of minimum length in its coset of the parabolic subgroup generated by the reflections (1, 3)

reflection_to_coroot ()
Returns the coroot associated with the reflection self.

EXAMPLES:

```sage
sage: W=WeylGroup(['C',2],prefix="s")
sage: W.from_reduced_word([1,2,1]).reflection_to_coroot()
sage: W.from_reduced_word([1,2]).reflection_to_coroot()
Traceback (most recent call last):
  ... ValueError: s1*s2 is not a reflection
sage: W.long_element().reflection_to_coroot()
Traceback (most recent call last):
  ... ValueError: s2*s1*s2*s1 is not a reflection
```

reflection_to_root()

Returns the root associated with the reflection self.

EXAMPLES:

```sage
sage: W=WeylGroup(['C',2],prefix="s")
sage: W.from_reduced_word([1,2,1]).reflection_to_root()
sage: W.from_reduced_word([1,2]).reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s1*s2 is not a reflection
sage: W.long_element().reflection_to_root()
Traceback (most recent call last):
  ... ValueError: s2*s1*s2*s1 is not a reflection
```

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
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] + 2*m[2, 2, 1, 1] + 4*m[2, 2, 1, 1, 1] + 2*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] + 2*m[2, 2, 1, 1] + 4*m[2, 2, 1, 1, 1] + 2*m[2, 2, 2, 1]
```

(continues on next page)
```python
sage: W = WeylGroup(['C',3,1])
sage: W.from_reduced_word([0,2,1,0]).stanley_symmetric_function()
32*m[1, 1, 1, 1] + 16*m[2, 1, 1] + 8*m[2, 2] + 4*m[3, 1]

sage: W = WeylGroup(['B',3,1])
sage: W.from_reduced_word([3,2,1]).stanley_symmetric_function()
2*m[1, 1, 1] + m[2, 1] + 1/2*m[3]

sage: W = WeylGroup(['B',4])
sage: w = W.from_reduced_word([3,2,3,1])
sage: w.stanley_symmetric_function() # long time (6s on sage.math, 2011)
48*m[1, 1, 1, 1] + 24*m[2, 1, 1] + 12*m[2, 2] + 8*m[3, 1] + 2*m[4]

sage: A = AffinePermutationGroup(['A',4,1])
sage: a = A([-2,0,1,4,12])
sage: a.stanley_symmetric_function()
6*m[1, 1, 1, 1, 1, 1, 1, 1, 1] + 5*m[2, 1, 1, 1, 1, 1, 1, 1, 1] + 4*m[2, 2, 1, 1, 1, 1, 1, 1, 1, 1] + 3*m[2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1] + 2*m[3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1] + m[3, 3, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] + m[4, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

One more example (trac ticket #14095):

```python
sage: G = SymmetricGroup(4)
sage: w = G.from_reduced_word([3,2,3,1])
sage: w.stanley_symmetric_function()
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[2, 2] + m[3, 1] + m[3, 1]
```

REFERENCES:

- [BH1994]
- [Lam2008]
- [LSS2009]
- [Pon2010]

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 \( \prod_i x_i^{c(u_i)} \), with coefficient given by \( \prod_i c(u_i) \), where `c` is a type-dependent statistic on Pieri factors, as returned by the method `u[i].stanley_symm_poly_weight()`.

EXAMPLES:
Algorithm: Induction on the left Pieri factors. Note that this induction preserves subsets of $W$ which are stable by taking right factors, and in particular Grassmanian elements.

**Finite**

alias of `sage.categories.finite_weyl_groups.FiniteWeylGroups`

**class ParentMethods**

```python
coxeter_matrix()
```

Return the Coxeter matrix associated to self.

**EXAMPLES:**

```python
sage: G = WeylGroup(['A', 3])
sage: G.coxeter_matrix()
[1 3 2]
[3 1 3]
[2 3 1]
```

```python
pieri_factors(*args, **keywords)
```

Returns the set of Pieri factors in this Weyl group.

For any type, the set of Pieri factors forms a lower ideal in Bruhat order, generated by all the conjugates of some special element of the Weyl group. In type $A_n$, this special element is $s_n \cdots s_1$, and the
conjugates are obtained by rotating around this reduced word.

These are used to compute Stanley symmetric functions.

See also:

- WeylGroups.ElementMethods.stanley_symmetric_function()
- sage.combinat.root_system.pieri_factors

EXAMPLES:

```python
sage: W = WeylGroup(['A',5,1])
sage: PF = W.pieri_factors()
sage: PF.cardinality()
63

sage: W = WeylGroup(['B',3])
sage: PF = W.pieri_factors()
sage: sorted([w.reduced_word() for w in PF])
[[],
 [1],
 [1, 2],
 [1, 2, 1],
 [1, 2, 3],
 [1, 2, 3, 1],
 [1, 2, 3, 2],
 [1, 2, 3, 2, 1],
 [2],
 [2, 1],
 [2, 3],
 [2, 3, 1],
 [2, 3, 2],
 [2, 3, 2, 1],
 [3],
 [3, 1],
 [3, 1, 2],
 [3, 1, 2, 1],
 [3, 2],
 [3, 2, 1]]

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=())

Return the quantum Bruhat graph of the quotient of the Weyl group by a parabolic subgroup \( W_J \).

**INPUT:**

- index_set – (default: ()) a tuple \( J \) of nodes of the Dynkin diagram

By default, the value for index_set indicates that the subgroup is trivial and the quotient is the full Weyl group.

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',3], prefix="s")
sage: g = W.quantum_bruhat_graph((1,3))
sage: g

Parabolic Quantum Bruhat Graph of Weyl Group of type ['A', 3] (as a matrix group acting on the ambient space) for nodes (1, 3):
Digraph on 6 vertices
```
sage: g.vertices()
[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

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()

super_categories()
EXAMPLES:
sage: WeylGroups().super_categories()
[Category of coxeter groups]

3.153 Technical Categories

3.153.1 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
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?

**example**(choice='subset')

Returns an example of facade set, as per `Category.example()`.

**INPUT:**
• `choice` – ‘union’ or ‘subset’ (default: ‘subset’).

EXAMPLES:

```python
sage: Sets().Facade().example()
An example of facade set: the monoid of positive integers
sage: Sets().Facade().example(choice='union')
An example of a facade set: the integers completed by +-infinity
sage: Sets().Facade().example(choice='subset')
An example of facade set: the monoid of positive integers
```
4.1 Covariant Functorial Constructions

A *functorial construction* is a collection of functors \((F_{\text{Cat}})_{\text{Cat}}\) (indexed by a collection of categories) which associate to a sequence of parents \((A, B, \ldots)\) in a category \(\text{Cat}\) a parent \(F_{\text{Cat}}(A, B, \ldots)\). Typical examples of functorial constructions are \texttt{cartesian_product} and \texttt{tensor_product}.

The category of \(F_{\text{Cat}}(A, B, \ldots)\), which only depends on \(\text{Cat}\), is called the (functorial) construction category.

A functorial construction is *(category)-covariant* if for every categories \(\text{Cat}\) and \(\text{SuperCat}\), the category of \(F_{\text{Cat}}(A, B, \ldots)\) is a subcategory of the category of \(F_{\text{SuperCat}}(A, B, \ldots)\) whenever \(\text{Cat}\) is a subcategory of \(\text{SuperCat}\).

A functorial construction is *(category)-regressive* if the category of \(F_{\text{Cat}}(A, B, \ldots)\) is a subcategory of \(\text{Cat}\).

The goal of this module is to provide generic support for covariant functorial constructions. In particular, given some parents \(A, B, \ldots\) in respective categories \(\text{Cat}_A, \text{Cat}_B, \ldots\), it provides tools for calculating the best known category for the parent \(F(A, B, \ldots)\). For examples, knowing that Cartesian products of semigroups (resp. monoids, groups) have a semigroup (resp. monoid, group) structure, and given a group \(B\) and two monoids \(A\) and \(C\) it can calculate that \(A \times B \times C\) is naturally endowed with a monoid structure.

See \texttt{CovariantFunctorialConstruction}, \texttt{CovariantConstructionCategory} and \texttt{RegressiveCovariantConstructionCategory} for more details.

AUTHORS:

- Nicolas M. Thiery (2010): initial revision

\begin{verbatim}
class sage.categories.covariant_functorial_construction.CovariantConstructionCategory (category, *args)
Bases: sage.categories.covariant_functorial_construction.FunctorialConstructionCategory

Abstract class for categories \(F_{\text{Cat}}\) obtained through a covariant functorial construction

additional_structure ()
    Return the additional structure defined by \texttt{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:

- \texttt{Category.additional_structure()}.
- \texttt{is_construction_defined_by_base()}.  

EXAMPLES:
\end{verbatim}

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

**classmethod default_super_categories**(category, *args)

Return the default super categories of \( F_{\text{Cat}}(A, B, ...) \) for \( A, B, ... \) parents in \( \text{Cat} \).

**INPUT:**

- cls – the category class for the functor \( F \)
- category – a category \( \text{Cat} \)
- *args – further arguments for the functor

**OUTPUT:** a (join) category

The default implementation is to return the join of the categories of \( F(A, B, ...) \) for \( A, B, ... \) in turn in each of the super categories of \( \text{category} \).

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 \( \text{category} \).

**EXAMPLES:**

Bialgebras are both algebras and coalgebras:

```python
sage: Bialgebras(QQ).super_categories()
[Category of algebras over Rational Field, Category of coalgebras over Rational Field]
```

Hence tensor products of bialgebras are tensor products of algebras and tensor products of coalgebras:

```python
sage: Bialgebras(QQ).TensorProducts().super_categories()
[Category of tensor products of algebras over Rational Field, Category of tensor products of coalgebras over Rational Field]
```

Here is how `default_super_categories()` was called internally:

```python
sage: sage.categories.tensor.TensorProductsCategory.default_super_categories(Bialgebras(QQ))
Join of Category of tensor products of algebras over Rational Field and Category of tensor products of coalgebras over Rational Field
```

We now show a similar example, with the Algebra functor which takes a parameter \( Q \):

```python
sage: FiniteMonoids().super_categories()
[Category of monoids, Category of finite semigroups]
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(FiniteMonoids(), QQ)
Join of Category of finite dimensional algebras with basis over Rational Field
and Category of monoid algebras over Rational Field
and Category of finite set algebras over Rational Field

is_construction_defined_by_base()

Return whether the construction is defined by the base of self.

EXAMPLES:
The graded functorial construction is defined by the modules category. Hence this method returns True for graded modules and False for other graded xxx categories:

sage: Modules(ZZ).Graded().is_construction_defined_by_base()
True
sage: Algebras(QQ).Graded().is_construction_defined_by_base()
False
sage: Modules(ZZ).WithBasis().Graded().is_construction_defined_by_base()
False

This is implemented as follows: given the base category 𝐴 and the construction 𝐹 of self, that is self=𝐴.𝐹(), check whether no super category of 𝐴 has 𝐹 defined.

Note: Recall that, when 𝐴 does not implement the construction 𝐹, a join category is returned. Therefore, in such cases, this method is not available:

sage: Coalgebras(QQ).Graded().is_construction_defined_by_base()
Traceback (most recent call last):
  ...
AttributeError: 'JoinCategory_with_category' object has no attribute 'is_construction_defined_by_base'

class sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction

An abstract class for construction functors 𝐹 (eg 𝐹 = Cartesian product, tensor product, Q-algebra, …) such that:

- Each category 𝐶𝑎𝑡 (eg 𝐶𝑎𝑡 = Groups()) can provide a category 𝐹𝐶𝑎𝑡 for parents constructed via this functor (e.g. 𝐹𝐶𝑎𝑡 = CartesianProductsOf(Groups())).
- For every category 𝐶𝑎𝑡, 𝐹𝐶𝑎𝑡 is a subcategory of 𝐹𝑆𝑢𝑝𝑒𝑟𝐶𝑎𝑡 for every super category 𝑆𝑢𝑝𝑒𝑟𝐶𝑎𝑡 of 𝐶𝑎𝑡 (the functorial construction is (category)-covariant).
- For parents 𝑀, 𝑁, …, respectively in the categories 𝐶𝑎𝑡, 𝐶𝑎𝑡, …, the category of 𝐹(𝑀, 𝑁, …) is 𝐹𝐶𝑎𝑡 where 𝐶𝑎𝑡 is the meet of the categories 𝐶𝑎𝑡, 𝐶𝑎𝑡, ….

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:

4.1. Covariant Functorial Constructions 697
• 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, \( \text{CartesianProductsOf}(\text{Groups}()) \) is set automatically as a subcategory of \( \text{CartesianProductsOf}(\text{Monoids}()) \).

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):

• \text{tensor(Cat)}

• \text{tensor((Cat, Cat))}

• \text{tensor.of((Cat, Cat))}

• \text{tensor.category_from_categories((Cat, Cat, Cat))}

• \text{Cat.TensorProducts()}

The syntax \text{Cat.TensorProducts()} does not supports well multivariate constructions like \text{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 \text{CartesianProducts()} construction makes only sense for concrete categories, that is subcategories of \text{Sets()}. Maybe we want to model this one way or the other.

\text{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:

\text{sage: Cat1 = Rings()}
\text{sage: Cat2 = Groups()}
\text{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

\text{sage: cartesian_product.category_from_categories((Cat1, Cat2))}
Category of Cartesian products of monoids

\text{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 $F_{\mathcal{C}at}$ obtained through a functorial construction

base_category ()
Return the base category of the category self.

For any category $B = F_{\mathcal{C}at}$ obtained through a functorial construction $F$, the call $B$.base_category () returns the category $\mathcal{C}at$.

EXAMPLES:

```
sage: Semigroups().Quotients().base_category()
Category of semigroups
```

classmethod category_of (category, *args)
Return the image category of the functor $F_{\mathcal{C}at}$.

This is the main entry point for constructing the category $F_{\mathcal{C}at}$ of parents $F(A, B, ...)$ constructed from parents $A, B, ...$ in $\mathcal{C}at$.

INPUT:
• cls – the category class for the functorial construction $F$
• category – a category $\mathcal{C}at$
• *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)
```

(continues on next page)
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.RegressiveCovariantConstructionCategory

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]
```

4.2 Cartesian Product Functorial Construction

AUTHORS:
class sage.categories.cartesian_product.CartesianProductFunctor


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
```

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, ...]
```
Hence, the role of \texttt{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:

\begin{verbatim}
sage: C.one()
('', 1, 1)
\end{verbatim}

Those are implemented in the nested class \texttt{Monoids.CartesianProducts} of \texttt{Monoids(QQ)}. This nested class is itself a subclass of \texttt{CartesianProductsCategory}.

\begin{verbatim}
class sage.categories.cartesian_product.CartesianProductsCategory(category, *args):
    Bases: sage.categories.covariant_functorial_construction.CovariantConstructionCategory

    An abstract base class for all CartesianProducts categories.

    CartesianProducts()
    Return the category of (finite) Cartesian products of objects of self.

    By associativity of Cartesian products, this is self (a Cartesian product of Cartesian products of \(A\)'s is a Cartesian product of \(A\)'s).

    EXAMPLES:

    \begin{verbatim}
sage: ModulesWithBasis(QQ).CartesianProducts().CartesianProducts()
Category of Cartesian products of vector spaces with basis over Rational Field
\end{verbatim}

    base_ring()
    The base ring of a Cartesian product is the base ring of the underlying category.

    EXAMPLES:

    \begin{verbatim}
sage: Algebras(ZZ).CartesianProducts().base_ring()
Integer Ring
\end{verbatim}
\end{verbatim}

4.3 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 \text{Algebras}(\mathbb{Q}), it is automatically endowed with its natural algebra structure, thanks to the category \text{Algebras}(\mathbb{Q}).\text{TensorProducts}() of tensor products of algebras.
    
    The tensor functor is covariant: if A is a subcategory of B, then A\text{.TensorProducts}() is a subcategory of B\text{.TensorProducts}() (see also \text{CovariantFunctorialConstruction}). Hence, the role of \text{Algebras}(\mathbb{Q}).\text{TensorProducts}() is solely to provide mathematical information and algorithms which are relevant to tensor product of algebras.
    
    Those are implemented in the nested class \text{TensorProducts} of \text{Algebras}(\mathbb{Q}). This nested class is itself a subclass of \text{TensorProductsCategory}.

class sage.categories.tensor.TensorProductsCategory(category, *args)
    Bases: sage.categories.covariant_functorial_construction.CovariantConstructionCategory
    
    An abstract base class for all TensorProducts’s categories

    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 \text{Cat}'s is a tensor product of \text{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 functorial construction
    The tensor product functorial construction

    See TensorProductFunctor for more information

    EXAMPLES:

    sage: tensor
    The tensor functorial construction

4.4 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

4.5 Group algebras and beyond: the Algebra functorial construction

4.5.1 Introduction: group algebras

Let $G$ be a group and $R$ be a ring. For example:

```sage
sage: G = DihedralGroup(3)
sage: R = QQ
```

The group algebra $A = RG$ of $G$ over $R$ is the space of formal linear combinations of elements of $group$ with coefficients in $R$:

```sage
sage: A = G.algebra(R); A
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
sage: a = A.an_element(); a
() + (1,2) + 3*(1,2,3) + 2*(1,3,2)
```

This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication of $G$ to a multiplication on $RG$:

```sage
sage: A in Algebras
True
sage: a * a
14*(()) + 5*(2,3) + 2*(1,2) + 10*(1,2,3) + 13*(1,3,2) + 5*(1,3)
```

In particular, the product of two basis elements is induced by the product of the corresponding elements of the group, and the unit of the group algebra is indexed by the unit of the group:

```sage
sage: (s, t) = A.algebra_generators()
sage: s*t
(1,2)
sage: A.one_basis()
()
sage: A.one()
()
```

For the user convenience and backward compatibility, the group algebra can also be constructed with:

```sage
sage: GroupAlgebra(G, R)
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
```

Since trac ticket #18700, both constructions are strictly equivalent:
Group algebras are further endowed with a Hopf algebra structure; see below.

### 4.5.2 Generalizations

The above construction extends to weaker multiplicative structures than groups: magmas, semigroups, monoids. For a monoid $S$, we obtain the monoid algebra $RS$, which is defined exactly as above:

```sage
sage: S = Monoids().example(); S
An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')

sage: A = S.algebra(QQ); A
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd') over Rational Field

sage: A.category()
Category of monoid algebras over Rational Field
```

This construction also extends to additive structures: magmas, semigroups, monoids, or groups:

```sage
sage: S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')

sage: U = S.algebra(QQ); U
Algebra of An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd') over Rational Field

sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: U(a) * U(b)
B[a + b]
```

Despite saying “free module”, this is really an algebra, whose multiplication is induced by the addition of elements of $S$:

```sage
sage: U in Algebras(QQ)
True
sage: (a,b,c,d) = S.additive_semigroup_generators()
sage: U(a) * U(b)
B[a + b]
```

To cater uniformly for the use cases above and some others, for $S$ a set and $K$ a ring, we define in Sage the algebra of $S$ as the $K$-free module with basis indexed by $S$, endowed with whatever algebraic structure can be induced from that of $S$.

**Warning:** In most use cases, the result is actually an algebra, hence the name of this construction. In other cases this name is misleading:

```sage
sage: A = Sets().example().algebra(QQ); A
Algebra of Set of prime numbers (basic implementation) over Rational Field

sage: A.category()
Category of set algebras over Rational Field
sage: A in Algebras(QQ)
False
```

Suggestions for a uniform, meaningful, and non misleading name are welcome!
To achieve this flexibility, the features are implemented as a Covariant Functorial Constructions that is essentially a hierarchy of categories each providing the relevant additional features:

```
sage: A = DihedralGroup(3).algebra(QQ)
sage: A.categories()

[Category of finite group algebras over Rational Field,
  ... Category of group algebras over Rational Field,
  ... Category of monoid algebras over Rational Field,
  ... Category of semigroup algebras over Rational Field,
  ... Category of unital magma algebras over Rational Field,
  ... Category of magma algebras over Rational Field,
  ... Category of set algebras over Rational Field,
  ...]
```

### 4.5.3 Specifying the algebraic structure

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 semigroup.
  Constructing its algebra is ambiguous.
  Please use, e.g., S.algebra(QQ, category=Semigroups())
```

This ambiguity can be resolved using the category argument of the construction:

```
sage: A = Z3.algebra(QQ, category=Monoids()); A
Algebra of Ring of integers modulo 3 over Rational Field

sage: A.category()
Category of finite dimensional monoid algebras over Rational Field

sage: A = Z3.algebra(QQ, category=CommutativeAdditiveGroups()); A
Algebra of Ring of integers modulo 3 over Rational Field

sage: A.category()
Category of finite dimensional commutative additive group algebras
  over Rational Field
```

In general, the category argument can be used to specify which structure of $S$ shall be extended to $KS$.

### 4.5.4 Group algebras, continued

Let us come back to the case of a group algebra $A = RG$. It is endowed with more structure and in particular that of a Hopf algebra:

```
sage: G = DihedralGroup(3)
sage: A = G.algebra(R); A
```

(continues on next page)
Algebra of Dihedral group of order 6 as a permutation group over Rational Field

```
sage: A in HopfAlgebras(R).FiniteDimensional().WithBasis()
True
```

The basis elements are group-like for the coproduct: $\Delta(g) = g \otimes g$:

```
sage: s
(1,2,3)
sage: s.coproduct()
(1,2,3) # (1,2,3)
```

The counit is the constant function $1$ on the basis elements:

```
sage: A = GroupAlgebra(DihedralGroup(6), QQ)
sage: [A.counit(g) for g in A.basis()]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The antipode is given on basis elements by $\chi(g) = g^{-1}$:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ)
sage: s
(1,2,3)
sage: s.antipode()
(1,3,2)
```

By Maschke’s theorem, for a finite group whose cardinality does not divide the characteristic of the base field, the algebra is semisimple:

```
sage: SymmetricGroup(5).algebra(QQ) in Algebras(QQ).Semisimple()
True
```

4.5.5 Coercions

Let $RS$ be the algebra of some structure $S$. Then $RS$ admits the natural coercion from any other algebra $R'S'$ of some structure $S'$, as long as $R'$ coerces into $R$ and $S'$ coerces into $S$.

For example, since there is a natural inclusion from the dihedral group $D_2$ of order 4 into the symmetric group $S_4$ of order 4!, and since there is a natural map from the integers to the rationals, there is a natural map from $\mathbb{Z}[D_2]$ to $\mathbb{Q}[S_4]$:

```
sage: A = DihedralGroup(2).algebra(ZZ)
sage: B = SymmetricGroup(4).algebra(QQ)
sage: a = A.an_element(); a
() + 2*(3,4) + 3*(1,2) + (1,2) (3,4)
sage: b = B.an_element(); b
() + (2,3,4) + 2*(1,3) (2,4) + 3*(1,4) (2,3)
sage: B(a)
() + 2*(3,4) + 3*(1,2) + (1,2) (3,4)
sage: a * b  # a is automatically converted to an element of B
() + 2*(3,4) + 2*(2,3) + (2,3,4) + 3*(1,2) + (1,2) (3,4) + (1,3,2) + 3*(1,3,4,2) + 5*(1,3) (2,4) + 13*(1,3,2,4) + 12*(1,4,2,3) + 5*(1,4) (2,3)
```

(continues on next page)
sage: parent(a * b)
Symmetric group algebra of order 4 over Rational Field

There is no obvious map in the other direction, though:

sage: A(b)
Traceback (most recent call last):
...TypeError: do not know how to make x (= () + (2,3,4) + 2*(1,3)(2,4) + 3*(1,4)(2,3)) an element of self
(=Algebra of Dihedral group of order 4 as a permutation group over Integer Ring)

If $S$ is a unital (additive) magma, then $RS$ is a unital algebra, and thus admits a coercion from its base ring $R$ and any ring that coerces into $R$.

sage: G = DihedralGroup(2)
sage: A = G.algebra(ZZ)
sage: A(2)
2*()

If $S$ is a multiplicative group, then $RS$ admits a coercion from $S$ and from any group which coerce into $S$:

sage: g = DihedralGroup(2).gen(0); g
(3,4)
sage: A(g)
(3,4)
sage: A(2) * g
2*(3,4)

Note that there is an ambiguity if $S'$ is a group which coerce into both $R$ and $S$. For example) if $S$ is the additive group $(\mathbb{Z}, +)$, and $A = RS$ is its group algebra, then the integer 2 can be coerced into $A$ in two ways – via $S$, or via the base ring $R$ – and the answers are different. It that case the coercion to $R$ takes precedence. In particular, if $\mathbb{Z}$ is the ring (or group) of integers, then $\mathbb{Z}$ will coerce to any $RS$, by sending $\mathbb{Z}$ to $R$. In generic code, it is therefore recommended to always explicitly use $A.monomial(g)$ to convert an element of the group into $A$.

AUTHORS:

- David Loeffler (2008-08-24): initial version
- Martin Raum (2009-08): update to use new coercion model – see trac ticket #6670.
- John Palmieri (2011-07): more updates to coercion, categories, etc., group algebras constructed using CombinatorialFreeModule – see trac ticket #6670.
- Nicolas M. Thiéry (2010-2017), Travis Scrimshaw (2017): generalization to a covariant functorial construction for monoid algebras, and beyond – see e.g. trac ticket #18700.

```python
class sage.categories.algebra_functor.AlgebraFunctor(base_ring)
Bases: sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction

For a fixed ring, a functor sending a group/… to the corresponding group/… algebra.

EXAMPLES:

sage: from sage.categories.algebra_functor import AlgebraFunctor
sage: F = AlgebraFunctor(QQ); F
The algebra functorial construction
```

(continues on next page)
The base ring for this functor can be returned using the `base_ring()` method. Here's an example:

```python
sage: from sage.categories.algebra_functor import AlgebraFunctor
sage: AlgebraFunctor(QQ).base_ring()
Rational Field
```

### Definition: `AlgebrasCategory`

```python
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.
```

**INPUT:**

- `base_ring` — a ring

**EXAMPLES:**

```python
sage: C = Groups().Algebras(QQ); C
Category of group algebras over Rational Field
sage: C = Monoids().Algebras(QQ); C
Category of monoid algebras over Rational Field
sage: C._short_name()
'Algebras'
sage: latex(C) # todo: improve that
\textbf{Algebras}(\textbf{Monoids})
```

### Definition: `GroupAlgebraFunctor`

```python
class sage.categories.algebra_functor.GroupAlgebraFunctor(group)

Bases: sage.categories.pushout.ConstructionFunctor

For a fixed group, a functor sending a commutative ring to the corresponding group algebra.
```

**INPUT:**

- `group` — the group associated to each group algebra under consideration

**EXAMPLES:**

```python
sage: from sage.categories.algebra_functor import GroupAlgebraFunctor
sage: F = GroupAlgebraFunctor(KleinFourGroup()); F
GroupAlgebraFunctor
sage: A = F(QQ); A
Algebra of The Klein 4 group of order 4, as a permutation group over Rational Field
```
group()
Return the group which is associated to this functor.

EXAMPLES:

```
sage: from sage.categories.algebra_functor import GroupAlgebraFunctor
sage: GroupAlgebraFunctor(CyclicPermutationGroup(17)).group() == CyclicPermutationGroup(17)
True
```

### 4.6 Subquotient Functorial Construction

**AUTHORS:**
- Nicolas M. Thiery (2010): initial revision

```
class sage.categories.subquotients.SubquotientsCategory(category, *args)
    Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory
```

### 4.7 Quotients Functorial Construction

**AUTHORS:**
- Nicolas M. Thiery (2010): initial revision

```
class sage.categories.quotients.QuotientsCategory(category, *args)
    Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory
```

*classmethod default_super_categories(category)*

Returns the default super categories of category.Quotients()

Mathematical meaning: if A is a quotient of B in the category C, then A is also a subquotient of B in the category C.

INPUT:
- cls – the class QuotientsCategory
- category – a category Cat

OUTPUT: a (join) category

In practice, this returns category.Subquotients(), joined together with the result of the method RegressiveCovariantConstructionCategory.default_super_categories() (that is the join of category and cat.Quotients() for each cat in the super categories of category).

**EXAMPLES:**
Consider category=Groups(), which has cat=Monoids() as super category. Then, a subgroup of a group G is simultaneously a subquotient of G, a group by itself, and a quotient monoid of G:
sage: Groups().Quotients().super_categories()
[Category of groups, Category of subquotients of monoids, Category of quotients of semigroups]

Mind the last item above: there is indeed currently nothing implemented about quotient monoids.

This resulted from the following call:

sage: sage.categories.quotients.QuotientsCategory.default_super_categories(Groups())
Join of Category of groups and Category of subquotients of monoids and Category of quotients of semigroups

### 4.8 Subobjects Functorial Construction

**AUTHORS:**
- Nicolas M. Thiery (2010): initial revision

```python
class sage.categories.subobjects.SubobjectsCategory(category, *args)
    Bases: sage.categories.covariant Functorial Construction

    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 \( C \)

        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 \( \text{category} = \text{Groups()} \), which has \( \text{cat} = \text{Monoids()} \) as super category. Then, a subgroup of a group \( G \) is simultaneously a subquotient of \( G \), a group by itself, and a submonoid of \( G \):

    sage: Groups().Subobjects().super_categories()
    [Category of groups, Category of subquotients of monoids, Category of subobjects of sets]

    Mind the last item above: there is indeed currently nothing implemented about submonoids.

    This resulted from the following call:

    sage: sage.categories.subobjects.SubobjectsCategory.default_super_categories(Groups())
    Join of Category of groups and Category of subquotients of monoids and Category of subobjects of sets
```

---

4.8. Subobjects Functorial Construction 711
4.9 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.RegressiveCovariantConstructionCategory

  @classmethod
default_super_categories(cls, 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 \textit{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 \textit{category=Groups()}, which has \textit{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:

    \[
    \text{sage: Groups().IsomorphicObjects().super_categories()}
    \]

    \[
    \begin{align*}
    \text{[Category of groups,} \\
    \text{Category of subquotients of monoids,} \\
    \text{Category of quotients of semigroups,} \\
    \text{Category of isomorphic objects of sets]}
    \end{align*}
    \]

    Mind the last item above: there is indeed currently nothing implemented about isomorphic objects of monoids.

    This resulted from the following call:

    \[
    \text{sage: sage.categories.isomorphic_objects.IsomorphicObjectsCategory.default_} \rightarrow \text{super_categories(Groups())}
    \]

    \[
    \begin{align*}
    \text{Join of Category of groups and} \\
    \text{Category of subquotients of monoids and} \\
    \text{Category of quotients of semigroups and} \\
    \text{Category of isomorphic objects of sets}
    \end{align*}
    \]

4.10 Homset categories

class sage.categories.homsets.Homsets(s=None):
  Bases: sage.categories.category_singleton.Category_singleton
The category of all homsets.

EXAMPLES:

```python
sage: from sage.categories.homsets import Homsets
sage: Homsets()
Category of homsets
```

This is a subcategory of `Sets()`:

```python
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:

```python
sage: Schemes().Homsets().is_subcategory(Homsets())
True
sage: AdditiveMagmas().Homsets().is_subcategory(Homsets())
True
sage: AdditiveMagmas().AdditiveUnital().Homsets().is_subcategory(Homsets())
True
```

This is tested in `HomsetsCategory._test_homsets_category()`.

```python
class Endset(base_category)
    Bases: sage.categories.category_with_axiom.CategoryWithAxiom

    The category of all endomorphism sets.

    This category serves two purposes: making sure that the `Endset` axiom is implemented in the category where it’s defined, namely `Homsets`, and specifying that `Endsets` are monoids.

    EXAMPLES:

    ```python
    sage: from sage.categories.homsets import Homsets
    sage: Homsets().Endset()
    Category of endsets
    ```

class ParentMethods

    .. method:: is_endomorphism_set()

        Return `True` as self is in the category of `Endsets`.

        EXAMPLES:

        ```python
        sage: P.<t> = ZZ[]
sage: E = End(P)
sage: E.is_endomorphism_set()
        True
        ```

    .. method:: 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 ParentMethods

is_endomorphism_set()

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

EXAMPLES:

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

class SubcategoryMethods

Endset()

Return the subcategory of the homsets of self that are endomorphism sets.

EXAMPLES:

```python
sage: Sets().Homsets().Endset()
Category of endsets of sets
sage: Posets().Homsets().Endset()
Category of endsets of posets
```

super_categories()

Return the super categories of self.

EXAMPLES:

```python
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

base()

If this homsets category is subcategory of a category with a base, return that base.

Todo: Is this really useful?

EXAMPLES:

```python
sage: ModulesWithBasis(ZZ).Homsets().base()
Integer Ring
```
classmethod default_super_categories(category)

Return the default super categories of category.Homsets().

INPUT:

• cls – the category class for the functor F

• category – a category 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():

```
4.10. Homset categories 715
```
Finally, as a last resort, this method returns a stub category modelling the homsets of this category:

```python
sage: hasattr(Posets, "Homsets")
False
sage: H = HomsetsCategory.default_super_categories(Posets()); H
Category of homsets of posets
sage: type(H)
<class 'sage.categories.homsets.Homsetsof_with_category'>
```

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:

```python
sage: C = (Magmas() & Posets()).Homsets(); C
Category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.Homsetsof_with_category'>
```

```python
super_categories()

Return the super categories of self.

A stub homset category admits a single super category, namely the category of all homsets.

EXAMPLES:

```python
sage: C = (Magmas() & Posets()).Homsets(); C
Category of homsets of magmas and posets
sage: type(C)
<class 'sage.categories.homsets.Homsetsof_with_category'>
```

## 4.11 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.
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
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 realizations of hopf algebras over Rational Field
```

```sage
C.super_categories()
[Category of graded hopf algebras with basis over Rational Field, Category of realizations of hopf algebras over Rational Field]
sage: TestSuite(C).run()
```

See also:

• `Sets().WithRealizations`
• ClasscallMetaclass

Todo: Add an optional argument to allow for:

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

class sage.categories.realizations.RealizationsCategory(category, *args):
    Bases: sage.categories.covariant_functorial_construction.
    RegressiveCovariantConstructionCategory

An abstract base class for all categories of realizations category

Realizations 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:

Sage() . WithRealizations

4.12 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 func-
torial constructions.

sage.categories.with_realizations.WithRealizations(self)

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

INPUT:

• self – a category

See also:

• The documentation and code (sage.categories.examples.with_realizations) of
Sets().WithRealizations().example() for more on how to use and implement a parent with
several realizations.
• Various use cases:
  - SymmetricFunctions
  - QuasiSymmetricFunctions
  - NonCommutativeSymmetricFunctions
  - SymmetricFunctionsNonCommutingVariables
  - DescentAlgebra
  - algebras.Moebius
  - IwahoriHeckeAlgebra
  - ExtendedAffineWeylGroup

718 Chapter 4. Functorial constructions
• The Implementing Algebraic Structures thematic tutorial.
• `sage.categories.realizations`

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

EXAMPLES:

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

Parent with realizations

Let us now explain the concept of realizations. A parent with realizations is a facade parent (see `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}] + 2*Out[{2, 3}] + 4*Out[{1, 2, 3}]
```

In our example, there are three realizations:
\begin{sageexample}
\begin{verbatim}
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]
\end{verbatim}
\end{sageexample}

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

\begin{sageexample}
\begin{verbatim}
sage: A.inject_shorthands()
Defining F as shorthand for The subset algebra of \{1, 2, 3\} over Rational Field
\rightarrow in the Fundamental basis
Defining In as shorthand for The subset algebra of \{1, 2, 3\} over Rational Field
\rightarrow in the In basis
Defining Out as shorthand for The subset algebra of \{1, 2, 3\} over Rational Field
\rightarrow in the Out basis
\end{verbatim}
\end{sageexample}

Rationale

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

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

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

\begin{sageexample}
\begin{verbatim}
sage: P = A['x']; P
Univariate Polynomial Ring in x over The subset algebra of \{1, 2, 3\} over Rational Field
sage: x = P.gen()
\end{verbatim}
\end{sageexample}

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

\begin{sageexample}
\begin{verbatim}
sage: P.one()
F[{}]
sage: (P.an_element() + 1)^2
F[{}]*x^2 + 2*F[{}]*x + F[{}]
\end{verbatim}
\end{sageexample}

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

\begin{sageexample}
\begin{verbatim}
sage: p = P([1, In[1], Out[2]]); p
Out[2]*x^2 + In[1]*x + F[{}]
sage: p^2
Out[2]*x^4
+ (-8*In[{}] + 4*In[1]) + 8*In[2] + 4*In[3] - 4*In[1, 2] - 2*In[1, 3]) +
\end{verbatim}
\end{sageexample}

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

One can easily coerce all coefficient to a given basis with:
Alas, the natural notation for constructing such polynomials does not yet work:

```
sage: In[{1}] * x
Traceback (most recent call last):
... TypeError: unsupported operand parent(s) for *: 'The subset algebra of {1, 2, 3} over Rational Field in the In basis' and 'Univariate Polynomial Ring in x over The subset algebra of {1, 2, 3} over Rational Field'
```

### The category of realizations of $A$

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

```
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, `VectorSpaces(QQ).FiniteDimensional().WithRealizations()` will eventually be in charge, whenever a coercion $\phi : A \rightarrow B$ is registered, to register $\phi^{-1}$ as coercion $B \rightarrow A$ if there is none defined yet. To achieve this, `FiniteDimensionalVectorSpaces` would provide a nested class `WithRealizations` implementing the appropriate logic.

`WithRealizations` is a **regressive covariant functorial construction**. On our example, this simply means that $A$ is automatically in the category of rings with realizations (covariance):

```
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.WithRealizations(C)`. The later is responsible for building the hierarchy of the categories with realizations in parallel to that of their base categories, optimizing away...
those categories that do not provide a WithRealizations nested class. See sage.categories.
covariant_functorial_construction for the technical details.

Note: Design question: currently WithRealizations is a regressive construction. That is self.
WithRealizations() is a subcategory of self by default:

```
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.

class sage.categories.with_realizations.WithRealizationsCategory(category, *args)

Bases: sage.categories.covariant_functorial_construction.RegressiveCovariantConstructionCategory

An abstract base class for all categories of parents with multiple realizations.

See also:
Sets().WithRealizations

The role of this base class is to implement some technical goodies, such as the name for that category.
EXAMPLES OF PARENTS USING CATEGORIES

5.1 Examples of algebras with basis

sage.categories.examples.algebras_with_basis.Example
alias of sage.categories.examples.algebras_with_basis.FreeAlgebra
class sage.categories.examples.algebras_with_basis.FreeAlgebra(R, alpha={"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 Rational Field
sage: A.algebra_generators()
Family (B[a], B[b], B[c])

one_basis()
Returns the empty word, which index the one of this algebra, as per AlgebrasWithBasis.ParentMethods.one_basis().

EXAMPLES:

sage: A = AlgebrasWithBasis(QQ).example() sage: A.one_basis() word: sage: A.one() B[ ]

product_on_basis(w1, w2)
Product of basis elements, as per AlgebrasWithBasis.ParentMethods.product_on_basis().

EXAMPLES:

sage: A = AlgebrasWithBasis(QQ).example() sage: Words = A.basis().keys() sage: A.product_on_basis(Words("acb"), Words("cba"))
B[ ]
sage: (a,b,c) = A.algebra_generators()

(continues on next page)
5.2 Examples of commutative additive monoids

sage.categories.examples.commutative_additive_monoids.Example
alias of sage.categories.examples.commutative_additive_monoids.
FreeCommutativeAdditiveMonoid
class sage.categories.examples.commutative_additive_monoids.FreeCommutativeAdditiveMonoid
  (alphabet=('a', 'b', 'c', 'd'))

Bases: sage.categories.examples.commutative_additive_semigroups.
FreeCommutativeAdditiveSemigroup

An example of a commutative additive monoid: the free commutative monoid

This class illustrates a minimal implementation of a commutative monoid.

EXAMPLES:

sage: S = CommutativeAdditiveMonoids().example(); S
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')
sage: S.category()
Category of commutative additive monoids

This is the free semigroup generated by:

sage: S.additive_semigroup_generators()
Family (a, b, c, d)

with product rule given by $a \times b = a$ for all $a, b$:

sage: (a,b,c,d) = S.additive_semigroup_generators()

We conclude by running systematic tests on this commutative monoid:

sage: TestSuite(S).run( verbose = True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_cardinality() . . . pass
running ._test_category() . . . pass
running ._test_elements() . . .
   Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
   pass
running ._test_elements_eq_reflexive() . . . pass

(continues on next page)
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass

class Element (parent, iterable)
Bases: sage.categories.examples.commutative_additive_semigroups.
          FreeCommutativeAdditiveSemigroup.Element

zero()
Returns the zero of this additive monoid, as per CommutativeAdditiveMonoids.
ParentMethods.zero().

EXAMPLES:

sage: M = CommutativeAdditiveMonoids().example(); M
An example of a commutative monoid: the free commutative monoid generated by ('a', 'b', 'c', 'd')
sage: M.zero()
0

5.3 Examples of commutative additive semigroups

sage.categories.examples.commutative_additive_semigroups.Example
alias of sage.categories.examples.commutative_additive_semigroups.
          FreeCommutativeAdditiveSemigroup

class sage.categories.examples.commutative_additive_semigroups.FreeCommutativeAdditiveSemigroup

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_category() . . . pass
running ._test_pickling() . . . pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

**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', 'd')
```

Internally, elements are represented as dense dictionaries which associate to each generator of the monoid its multiplicity. In order to get an element, we wrap the dictionary into an element via **ElementWrapper**:  

```
sage: x.value
{'a': 2, 'b': 0, 'c': 1, 'd': 5}
```

**additive_semigroup_generators()**

Returns the generators of the semigroup.

**EXAMPLES:**

```
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.additive_semigroup_generators()
Family (a, b, c, d)
```
\textbf{an\_element()}

Returns an element of the semigroup.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = CommutativeAdditiveSemigroups().example()
sage: F.an_element()
a + 2*b + 3*c + 4*d
\end{verbatim}

\textbf{summation(x, y)}

Returns the product of $x$ and $y$ in the semigroup, as per \texttt{CommutativeAdditiveSemigroups.ParentMethodssummation()}.  

\textbf{EXAMPLES:}

\begin{verbatim}
sage: F = CommutativeAdditiveSemigroups().example()
sage: (a,b,c,d) = F.additive_semigroup_generators()
sage: Fsummation(a,b)
a + b
sage: (a+b) + (a+c)
2*a + b + c
\end{verbatim}

5.4 Examples of Coxeter groups

5.5 Example of a crystal

\texttt{class sage.categories.examples.crystals.HighestWeightCrystalOfTypeA(n=3)}

\textbf{Bases:} \texttt{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 $\omega_1$.

The purpose of this class is to provide a minimal template for implementing crystals. See \texttt{CrystalOfLetters} for a full featured and optimized implementation.

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

The elements of this crystal are in the set \{1,\ldots,n + 1\}:

\begin{verbatim}
sage: C.list()
[1, 2, 3, 4]
sage: C.module\_generators[0] 1
\end{verbatim}

The crystal operators themselves correspond to the elementary transpositions:

\begin{verbatim}
sage: b = C.module\_generators[0]
sage: b.f(1)
2
\end{verbatim}
sage: b.f(1).e(1) == b
True

Only the following basic operations are implemented:

• cartan_type() or an attribute _cartan_type
• an attribute module_generators
• Element.e()
• Element.f()

All the other usual crystal operations are inherited from the categories; for example:

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
Returns the action of $e_i$ on self.

EXAMPLES:

```python
guide: C = Crystals().example(choice='naive')
guide: [[c, i, c.e(i)] for i in C.index_set() for c in [C(j) for j in [0..5]] if c.e(i) is not None]
guide: [[1, 1, 0], [2, 1, 1], [3, 1, 2], [5, 1, 3], [4, 2, 0], [5, 2, 4]]
```

Returns the action of $f_i$ on self.

EXAMPLES:

```python
guide: C = Crystals().example(choice='naive')
guide: [[c, i, c.f(i)] for i in C.index_set() for c in [C(j) for j in [0..5]] if c.f(i) is not None]
guide: [[0, 1, 1], [1, 1, 2], [2, 1, 3], [3, 1, 5], [0, 2, 4], [4, 2, 5]]
```

### 5.6 Examples of CW complexes

sage.categories.examples.cw_complexes.Example

alias of sage.categories.examples.cw_complexes.Surface

class sage.categories.examples.cw_complexes.Surface(bdy=(1, 2, 1, 2))

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

An example of a CW complex: a (2-dimensional) surface.

This class illustrates a minimal implementation of a CW complex.

EXAMPLES:

```python
guide: from sage.categories.cw_complexes import CWComplexes
guide: C = CWComplexes().example(); C
An example of a CW complex: the surface given by the boundary map (1, 2, 1, 2)
guide: C.category() Category of finite finite dimensional CW complexes
```

We conclude by running systematic tests on this manifold:

```python
guide: TestSuite(C).run()
```

class Element (parent, dim, name)

Bases: sage.structure.element.Element

A cell in a CW complex.

dimension()

Return the dimension of self.

EXAMPLES:
```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
```

```
sage: f = X.an_element()
sage: f.dimension()
2
```

**an_element()**

Return an element of the CW complex, as per `Sets.ParentMethods.an_element()`.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
```

```
sage: X.an_element()
2-cell f
```

**cells()**

Return the cells of self.

**EXAMPLES:**

```python
sage: from sage.categories.cw_complexes import CWComplexes
sage: X = CWComplexes().example()
```

```
sage: C = X.cells()
sage: sorted((d, C[d]) for d in C.keys())
[(0, (0-cell v,)),
 (1, (0-cell e1, 0-cell e2)),
 (2, (2-cell f,))]
```

### 5.7 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:**

```python
sage: S = Sets().Facade().example("union"); S
An example of a facade set: the integers completed by $+\infty$
```

**class** `sage.categories.examples.facade_sets.PositiveIntegerMonoid`

Bases: `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:**

```python
sage: S = Sets().Facade().example(); S
An example of facade set: the monoid of positive integers
```
5.8 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 \texttt{DihedralGroup} for a full featured and optimized implementation.

**EXAMPLES:**

```python
sage: G = FiniteCoxeterGroups().example()
```

This group is generated by two simple reflections \( s_1 \) and \( s_2 \) subject to the relation \( (s_1s_2)^n = 1 \):

```python
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 \texttt{self.indexset()}):

```python
sage: G.an_element()
(1, 2)
sage: list(G)
[(), (1,), (2,), (1, 2), (2, 1), (1, 2, 1), (2, 1, 2), (1, 2, 1, 2), (2, 1, 2, 1), (1, 2, 1, 2, 1)]
```

This reduced word is unique, except for the longest element where the chosen reduced word is \( (1, 2, 1, 2, 1...) \):

```python
sage: G.long_element()
(1, 2, 1, 2, 1)
```

class Element
    Bases: sage.structure.element_wrapper.ElementWrapper

    apply_simple_reflection_right(i)
        Implements \texttt{CoxeterGroups.ElementMethods.apply_simple_reflection()}. 

**EXAMPLES:**

```python
sage: D5 = FiniteCoxeterGroups().example(5)
sage: [i^2 for i in D5]  # indirect doctest
[(), (), (1, 2, 1, 2, 1), (2, 1, 2, 1, 2), (1, 2, 1, 2, 1)]
sage: [i^5 for i in D5]  # indirect doctest
[(), (1,), (2,), (1, 2), (2, 1), (1, 2, 1, 2, 1)]
```
**has_right_descent** *(i, positive=False, side='right')*

Implements `SemiGroups.ElementMethods.has_right_descent()`.

**EXAMPLES:**

```python
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
```

**coxeter_matrix()**

Return the Coxeter matrix of `self`.

**EXAMPLES:**

```python
sage: FiniteCoxeterGroups().example(6).coxeter_matrix()
[1 6]
[6 1]
```

**degrees()**

Return the degrees of `self`.

**EXAMPLES:**

```python
sage: FiniteCoxeterGroups().example(6).degrees()
(2, 6)
```

**index_set()**


**EXAMPLES:**

```python
sage: D4 = FiniteCoxeterGroups().example(4)
sage: D4.index_set()
(1, 2)
```

**one()**

Implements `Monoids.ParentMethods.one()`.

**EXAMPLES:**

```python
sage: D6 = FiniteCoxeterGroups().example(6)
sage: D6.one()
()```

`sage.categories.examples.finite_coxeter_groups.Example` alias of `sage.categories.examples.finite_coxeter_groups.DihedralGroup`
5.9 Example of a finite dimensional algebra with basis

```
sage.categories.examples.finite_dimensional_algebras_with_basis.Example
   alias of   sage.categories.examples.finite_dimensional_algebras_with_basis.
   KroneckerQuiverPathAlgebra

class  sage.categories.examples.finite_dimensional_algebras_with_basis.KroneckerQuiverPathAlgebra
   Bases:  sage.combinat.free_module.CombinatorialFreeModule

An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver.
This class illustrates a minimal implementation of a finite dimensional algebra with basis. See sage.
quivers.algebra.PathAlgebra for a full-featured implementation of path algebras.

algebra_generators()
   Return algebra generators for this algebra.
   See also:
   Algebras.ParentMethods.algebra_generators().
   EXAMPLES:

   sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example(); A
   An example of a finite dimensional algebra with basis: the path algebra of the Kronecker quiver
   (containing the arrows a:x->y and b:x->y) over Rational Field
   sage: A.algebra_generators()
   Finite family {'x': x, 'y': y, 'a': a, 'b': b}

one()
   Return the unit of this algebra.
   See also:
   AlgebrasWithBasis.ParentMethods.one_basis()
   EXAMPLES:

   sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
   sage: A.one()
   x + y

product_on_basis(w1, w2)
   Return the product of the two basis elements indexed by w1 and w2.
   See also:
   AlgebrasWithBasis.ParentMethods.product_on_basis().
   EXAMPLES:

   sage: A = FiniteDimensionalAlgebrasWithBasis(QQ).example()
   sage: A.product_on_basis([w1, w2])
   x
```

Here is the multiplication table for the algebra:

```
sage: matrix([[p*q for q in A.basis()] for p in A.basis()])
[x 0 a b]
[0 y 0 0]
[0 a 0 0]
[0 b 0 0]
```
Here we take some products of linear combinations of basis elements:

```
sage: x, y, a, b = A.basis()
sage: a * (1-b)^2 * x
0
sage: x*a + b*y
a + b
sage: x*x
x
sage: x*y
0
sage: x*a*y
a
```

### 5.10 Examples of finite enumerated sets

```python
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_new() .... pass
    running ._test_nonzero_equal() .... pass
    running ._test_not_implemented_methods() .... pass
    running ._test_pickling() .... pass
    pass
running ._test_elements_eq_reflexive() .... pass
running ._test_elements_eq_symmetric() .... pass
running ._test_elements_eq_transitive() .... pass
running ._test_elements_neq() .... pass
running ._test_enumerated_set_contains() .... pass
running ._test_enumerated_set_iter_cardinality() .... pass
```

(continues on next page)
class sage.categories.examples.finite_enumerated_sets.IsomorphicObjectOfFiniteEnumeratedSet

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

ambient()

Returns the ambient space for self, as per Sets.Subquotients.ParentMethods.ambient().

EXAMPLES:

sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: {1,2,3}
sage: C.ambient()
An example of a finite enumerated set: {1,2,3}

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 \(\text{self}\), as per \(\text{Sets.Subquotients.ParentMethods.retract()}\).

EXAMPLES:

```
sage: C = FiniteEnumeratedSets().IsomorphicObjects().example(); C
The image by some isomorphism of An example of a finite enumerated set: \([1,2,\rightarrow3]\)
sage: C.retract(3)
9
```

## 5.11 Examples of a finite dimensional Lie algebra with basis

```
class sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation

An example of a finite dimensional Lie algebra with basis: the abelian Lie algebra.

Let \(R\) be a commutative ring, and \(M\) an \(R\)-module. The abelian Lie algebra on \(M\) is the \(R\)-Lie algebra obtained by endowing \(M\) with the trivial Lie bracket \(\langle a, b \rangle = 0\) for all \(a, b \in M\).

This class illustrates a minimal implementation of a finite dimensional Lie algebra with basis.

INPUT:

- \(R\) – base ring
- \(n\) – (optional) a nonnegative integer (default: None)
- \(M\) – an \(R\)-module (default: the free \(R\)-module of rank \(n\)) to serve as the ground space for the Lie algebra
- \(ambient\) – (optional) a Lie algebra; if this is set, then the resulting Lie algebra is declared a Lie subalgebra of \(ambient\)

OUTPUT:

The abelian Lie algebra on \(M\).
```

class Element

```
Bases: sage.categories.examples.lie_algebras.LieAlgebraFromAssociative.Element

lift()

Return the lift of \(self\) to the universal enveloping algebra.
```

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.lift()
2*b0 + 2*b1 + 3*b2
```
**monomial_coefficients** *(copy=True)*

Return the monomial coefficients of `self`.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.monomial_coefficients()
{0: 2, 1: 2, 2: 3}
```

**to_vector**

Return `self` as a vector in `self.parent().module()`.

See the docstring of the latter method for the meaning of this.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: elt = 2*a + 2*b + 3*c
sage: elt.to_vector()
(2, 2, 3)
```

**ambient**

Return the ambient Lie algebra of `self`.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b+c])
sage: S.ambient() == L
True
```

**basis**

Return the basis of `self`.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}
```

**basis_matrix**

Return the basis matrix of `self`.

**EXAMPLES:**

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

**from_vector** *(v)*

Return the element of `self` corresponding to the vector `v` in `self.module()`.

Implement this if you implement `module()`; see the documentation of `sage.categories.lie_algebras.LieAlgebras.module()` for how this is to be done.

5.11. Examples of a finite dimensional Lie algebra with basis

737
EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: u = L.from_vector(vector(QQ, (1, 0, 0))); u
(1, 0, 0)
sage: parent(u) is L
True
```

gens()
Return the generators of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

ideal(gens)
Return the Lie subalgebra of self generated by the elements of the iterable gens.
This currently requires the ground ring $R$ to be a field.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([2*a+b, b + c])
An example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

is_ideal(A)
Return if self is an ideal of the ambient space A.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.isIdeal(L)
True
sage: S1 = L.subalgebra([2*a+b, b + c])
sage: S1.is_ideal(L)
True
sage: S2 = L.subalgebra([2*a+b])
sage: S2.is_ideal(S1)
True
sage: S2.is_ideal(S2)
False
```

lie_algebra_generators()
Return the basis of self.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.basis()
Finite family {0: (1, 0, 0), 1: (0, 1, 0), 2: (0, 0, 1)}
```
module()

Return an $R$-module which is isomorphic to the underlying $R$-module of self.

See sage.categories.lie_algebras.LieAlgebras.module() for an explanation.

In this particular example, this returns the module $M$ that was used to construct self.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

subalgebra(gens)

Return the Lie subalgebra of self generated by the elements of the iterable gens.

This currently requires the ground ring $R$ to be a field.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: a, b, c = L.lie_algebra_generators()
sage: L.subalgebra([2*a+b, b + c])
```

```an example of a finite dimensional Lie algebra with basis:
the 2-dimensional abelian Lie algebra over Rational Field with
basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

zero()

Return the zero element.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.zero()
(0, 0, 0)
```

sage.categories.examples.finite_dimensional_lie_algebras_with_basis.Example
alias of sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra

---

5.12 Examples of finite monoids

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

5.12. Examples of finite monoids

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```

```
sage: L = LieAlgebras(QQ).FiniteDimensional().WithBasis().example()
sage: L.module()
Vector space of dimension 3 over Rational Field

sage: a, b, c = L.lie_algebra_generators()
sage: S = L.subalgebra([2*a+b, b + c])
sage: S.module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 -1/2]
[ 0 1 1]
```
An example of a finite monoid: the integers mod $n$

This class illustrates a minimal implementation of a finite monoid.

EXAMPLES:

```sage
sage: S = FiniteMonoids().example(); S
An example of a finite multiplicative monoid: the integers modulo 12
```

```sage
sage: S.category()
Category of finitely generated finite enumerated monoids
```

We conclude by running systematic tests on this monoid:

```sage
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_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_some_elements() . . . pass
```

class Element

    Bases: sage.structure.element_wrapper.ElementWrapper

    wrapped_class
    alias of sage.rings.integer.Integer

an_element()

    Returns an element of the monoid, as per Sets.ParentMethods.an_element().

    EXAMPLES:

```sage
sage: M = FiniteMonoids().example()
sage: M.an_element()
6
```

one()

    Return the one of the monoid, as per Monoids.ParentMethods.one().

    EXAMPLES:
sage: M = FiniteMonoids().example()
sage: M.one()
1

**product** \((x, y)\)

Return the product of two elements \(x\) and \(y\) of the monoid, as per `Semigroups.ParentMethods.product()`.

**EXAMPLES:**

```python
sage: M = FiniteMonoids().example()
sage: M.product(M(3), M(5))
3
```

**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:**

```python
sage: M = FiniteMonoids().example()
sage: M.semigroup_generators()
Family (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

### 5.13 Examples of finite semigroups

sage.categories.examples.finite_semigroups.Example

alias of `sage.categories.examples.finite_semigroups.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:**

```python
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:

```python
sage: S.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

such that \(x^2 = x\) and \(xyz = xy\) for any \(x\) and \(y\) in \(S\):

```python
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
sage: sorted(S.list())
['a', 'ab', 'abc', 'abcd', 'abd', 'abdc', 'ac', 'acb', 'acbd', 'acd',
 'acdb', 'ad', 'adb', 'adbc', 'adc', 'adcb', 'b', 'ba', 'bac',
 'bacd', 'bad', 'badc', 'bc', 'bca', 'bcad', 'bcd', 'bcda', 'bd',
 'bda', 'bdac', 'bdc', 'bdca', 'c', 'ca', 'cab', 'cabd', 'cad',
 'cadb', 'cb', 'cba', 'cbad', 'cbd', 'cbda', 'cd', 'cda', 'cdab',
 'cdb', 'cdba', 'd', 'da', 'db', 'dabc', 'dac', 'dacb', 'db',
 'dba', 'dbac', 'dbc', 'dbca', 'dc', 'dca', 'dcab', 'dcb', 'dcba']
```

It also follows that there are finitely many of them:

```sage
sage: S.cardinality()
64
```

Indeed:

```sage
sage: 4 * ( 1 + 3 * (1 + 2 * (1 + 1)))
64
```

As expected, all the elements of $S$ are idempotents:

```sage
sage: all( x.is_idempotent() for x in S )
True
```

Now, let us look at the structure of the semigroup:

```sage
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
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_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
```

(continues on next page)
class Element
    Bases: sage.structure.element_wrapper.ElementWrapper

an_element()
    Returns an element of the semigroup.

    EXAMPLES:

    sage: S = FiniteSemigroups().example()
    sage: S.an_element()
    'cdab'
    sage: S = FiniteSemigroups().example(('b'))
    sage: S.an_element()
    'b'

product(x, y)
    Returns the product of two elements of the semigroup.

    EXAMPLES:

    sage: S = FiniteSemigroups().example()
    sage: S('a') * S('b')
    'ab'
    sage: S('a') * S('b') * S('a')
    'ab'
    sage: S('a') * S('a')
    'a'

semigroup_generators()
    Returns the generators of the semigroup.

    EXAMPLES:

    sage: S = FiniteSemigroups().example(alphabet=('x','y'))
    sage: S.semigroup_generators()
    Family ('x', 'y')

5.14 Examples of finite Weyl groups

sage.categories.examples.finite_weyl_groups.Example
    alias of sage.categories.examples.finite_weyl_groups.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 irreducible weyl groups

The elements of this group are permutations of the set \(\{0, \ldots, 3\}\):

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)}

Only the following basic operations are implemented:

- one()
- product()
- simple_reflection()
- cartan_type()
- Element.has_right_descent().

All the other usual Weyl group operations are inherited from the categories:

sage: S.cardinality()
24
sage: S.long_element()
(3, 2, 1, 0)
sage: S.cayley_graph(side = "left").plot()
Graphics object consisting of 120 graphics primitives

Alternatively, one could have implemented `sage.categories.coxeter_groups.CoxeterGroups.ElementMethods.apply_simple_reflection()` instead of `simple_reflection()` and `product()`. See `CoxeterGroups().example()`.

class Element
    Bases: 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()

cartan_type()
    Return the Cartan type of self.

    EXAMPLES:
degrees()
Return the degrees of self.

EXAMPLES:

```python
sage: W = FiniteWeylGroups().example()
sage: W.degrees()
(2, 3, 4)
```

index_set()
Implements CoxeterGroups.ParentMethods.index_set().

EXAMPLES:

```python
sage: FiniteWeylGroups().example().index_set()
[0, 1, 2]
```

one()
Implements Monoids.ParentMethods.one().

EXAMPLES:

```python
sage: FiniteWeylGroups().example().one()
(0, 1, 2, 3)
```

product(x, y)
Implements Semigroups.ParentMethods.product().

EXAMPLES:

```python
sage: s = FiniteWeylGroups().example().simple_reflections()
(0, 2, 3, 1)
```

simple_reflection(i)
Implement CoxeterGroups.ParentMethods.simple_reflection() by returning the transposition \((i, i+1)\).

EXAMPLES:

```python
sage: FiniteWeylGroups().example().simple_reflection(2)
(0, 1, 3, 2)
```

## 5.15 Examples of graded connected Hopf algebras with basis

sage.categories.examples.graded_connected_hopf_algebras_with_basis.

This class illustrates an implementation of a graded Hopf algebra with basis that has one primitive generator of degree 1 and basis elements indexed by non-negative integers.
This Hopf algebra example differs from what topologists refer to as a graded Hopf algebra because the twist operation in the tensor rule satisfies

\[(\mu \otimes \mu) \circ (id \otimes \tau \otimes id) \circ (\Delta \otimes \Delta) = \Delta \circ \mu\]

where \(\tau(x \otimes y) = y \otimes x\).

**coproduct_on_basis** (*i*)

The coproduct of a basis element.

\[\Delta(P_i) = \sum_{j=0}^{i} P_{i-j} \otimes P_j\]

**INPUT:**
- \(i\) – a non-negative integer

**OUTPUT:**
- an element of the tensor square of self

**degree_on_basis** (*i*)

The degree of a non-negative integer is itself

**INPUT:**
- \(i\) – a non-negative integer

**OUTPUT:**
- a non-negative integer

**one_basis**

Returns 0, which index the unit of the Hopf algebra.

**OUTPUT:**
- the non-negative integer 0

**EXAMPLES:**

```python
sage: H = GradedHopfAlgebrasWithBasis(QQ).Connected().example()
sage: H.one_basis()
0
sage: H.one()
P0
```

**product_on_basis** (*i*, *j*)

The product of two basis elements.

The product of elements of degree \(i\) and \(j\) is an element of degree \(i+j\).

**INPUT:**
- \(i, j\) – non-negative integers

**OUTPUT:**
- a basis element indexed by \(i+j\)
5.16 Examples of graded modules with basis

The 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)
```

```sage
Lazy family (Term map from Partitions to An example of a graded module with basis: the free module on partitions over Rational Field(i))_{i in Partitions of the integer 2}
```

```sage
A.basis(6)
```

- There is a class for elements, which inherits from IndexedFreeModuleElement. An element is determined by a dictionary whose keys are partitions and whose corresponding values are the coefficients. The class implements two things: an is_homogeneous method and a degree method.

```sage
p = A.monomial(Partition([3,2,1])); p
```

```sage
P[3, 2, 1]
```

```sage
p.is_homogeneous()
```

(continues on next page)
**basis** (*d=None*)

Return the basis for (the *d*-th homogeneous component of) `self`.

**INPUT:**

- `d` – (optional, default `None`) nonnegative integer or `None`

**OUTPUT:**

If `d` is `None`, returns the basis of the module. Otherwise, returns the basis of the homogeneous component of degree `d` (i.e., the subfamily of the basis of the whole module which consists only of the basis vectors lying in $F_d \setminus \bigcup_{i<d} F_i$).

The basis is always returned as a family.

**EXAMPLES:**

```sage
A = ModulesWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Lazy family (Term map from Partitions to An example of a filtered module with basis: the free module on partitions over Integer Ring(i))_{i in Partitions of the integer 4}
```

Without arguments, the full basis is returned:

```sage
A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.basis(4)
Traceback (most recent call last):
  ...  
NotImplementedError: infinite set
```

Checking this method on a filtered algebra. Note that this will typically raise a `NotImplementedError` when this feature is not implemented.

```sage
A = AlgebrasWithBasis(ZZ).Filtered().example()
sage: A.basis()
Lazy family (Term map from Free abelian monoid indexed by {'x', 'y', 'z'} to An example of a filtered algebra with basis: the universal enveloping algebra of Lie algebra of $\mathbb{R}^3$ with cross product over Integer Ring(i))_{i in Free abelian monoid indexed by {'x', 'y', 'z'}}
```

An example with a graded algebra:
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: E.basis()
Lazy family (Term map from Subsets of {0, 1} to
  The exterior algebra of rank 2 over Rational Field(i))_{i in
Subsets of {0, 1}}

definition (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'>

5.17 Examples of graphs

class sage.categories.examples.graphs.Cycle(n=5)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.
structure.parent.Parent

An example of a graph: the cycle of length n.

This class illustrates a minimal implementation of a graph.

EXAMPLES:

sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example(); C
An example of a graph: the 5-cycle
sage: C.category()
Category of graphs

We conclude by running systematic tests on this graph:

sage: TestSuite(C).run()

class Element

Bases: sage.structure.element_wrapper.ElementWrapper

dimension()

Return the dimension of self.

EXAMPLES:
an_element()  
Return an element of the graph, as per \texttt{Sets.ParentMethods.an_element()}.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.an_element()
0
```

edges()  
Return the edges of \texttt{self}.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.edges()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

vertices()  
Return the vertices of \texttt{self}.

EXAMPLES:

```python
sage: from sage.categories.graphs import Graphs
sage: C = Graphs().example()
sage: C.vertices()
[0, 1, 2, 3, 4]
```

5.18 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 \texttt{algebra_generators()}.

They correspond to the generators of the group.

EXAMPLES:
```python
sage: A = HopfAlgebrasWithBasis(QQ).example(); A
An example of Hopf algebra with basis: the group algebra of the Dihedral
→
group of order 6 as a permutation group over Rational Field
sage: A.algebra_generators()
Finite family {(1,2,3): B[(1,2,3)], (1,3): B[(1,3)]}

antipode_on_basis(g)
Antipode, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.
antipode_on_basis().

It is given, on basis elements, by $\nu(g) = g^{-1}$

EXAMPLES:
```}
```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: A.antipode_on_basis(a)
B[(1,3,2)]
```

```python
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:
```}
```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: A.coproduct_on_basis(a)
B[(1,2,3)] # B[(1,2,3)]
```

```python
counit_on_basis(g)
Counit, on basis elements, as per HopfAlgebrasWithBasis.ParentMethods.
counit_on_basis().

The counit on the basis elements is 1.

EXAMPLES:
```}
```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: (a, b) = A._group.gens()
sage: A.counit_on_basis(a)
1
```

```python
one_basis()
Returns the one of the group, which index the one of this algebra, as per AlgebrasWithBasis.
ParentMethods.one_basis().

EXAMPLES:
```}
```python
sage: A = HopfAlgebrasWithBasis(QQ).example()
sage: A.one_basis()
()
sage: A.one()
B[(())]
```

```python
product_on_basis(g1, g2)
Product, on basis elements, as per AlgebrasWithBasis.ParentMethods.
product_on_basis().

```
```

5.18. Examples of algebras with basis 751
The product of two basis elements is induced by the product of the corresponding elements of the group.

**EXAMPLES:**

```sage
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)]
```

## 5.19 Examples of infinite enumerated sets

```sage
categories.examples.infinite EnumeratedSets().Example
alias of categories.examples.infinite EnumeratedSets().NonNegativeIntegers
class categories.examples.infinite EnumeratedSets().NonNegativeIntegers
Bases: categories.structure.unique_representation.UniqueRepresentation, categories.structure parent.Paren
t
An example of infinite enumerated set: the non negative integers
This class provides a minimal implementation of an infinite enumerated set.

**EXAMPLES:**

```sage
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):
... Not ImplementedError: cannot list an infinite set
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
sage: TestSuite(NN).run(\text{\texttt{verbose = True}})
running ._test_an_element() ... pass
running ._test_cardinality() ... pass
running ._test_category() ... pass
```

(continues on next page)
running ._test_elements() . . .
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_enumerated_set_contains() . . . pass
running ._test_enumerated_set_iter_cardinality() . . . pass
running ._test_enumerated_set_iter_list() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass

**Element**
alias of `sage.rings.integer.Integer`

**an_element()**
EXAMPLES:

```
sage: InfiniteEnumeratedSets().example().an_element()
sage: 42
```

**next (o)**
EXAMPLES:

```
sage: NN = InfiniteEnumeratedSets().example()
sage: NN.next(3)
sage: 4
```

## 5.20 Examples of manifolds

`sage.categories.examples.manifolds.Example`
alias of `sage.categories.examples.manifolds.Plane`

**class** `sage.categories.examples.manifolds.Plane(n=3, base_ring=None)`

**Bases:** `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

An example of a manifold: the $n$-dimensional plane.

This class illustrates a minimal implementation of a manifold.

**EXAMPLES:**

```
sage: from sage.categories.manifolds import Manifolds
sage: M = Manifolds(QQ).example(); M
```

(continues on next page)
An example of a Rational Field manifold: the 3-dimensional plane

\texttt{sage}: \texttt{M.category()}
Category of manifolds over Rational Field

We conclude by running systematic tests on this manifold:

\texttt{sage}: \texttt{TestSuite(M).run()}

\textbf{Element}
alias of \texttt{sage.structure.element_wrapper.ElementWrapper}

\textbf{an_element()}
Return an element of the manifold, as per \texttt{Sets.ParentMethods.an_element()}.

\textbf{EXAMPLES:}

\texttt{sage}: \texttt{from sage.categories.manifolds import Manifolds}
\texttt{sage}: \texttt{M = Manifolds(QQ).example()}
\texttt{sage}: \texttt{M.an_element()}
(0, 0, 0)

\textbf{dimension()}
Return the dimension of \texttt{self}.

\textbf{EXAMPLES:}

\texttt{sage}: \texttt{from sage.categories.manifolds import Manifolds}
\texttt{sage}: \texttt{M = Manifolds(QQ).example()}
\texttt{sage}: \texttt{M.dimension()}
3

\section{5.21 Examples of a Lie algebra}

\texttt{sage.categories.examples.lie_algebras.Example}
alias of \texttt{sage.categories.examples.lie_algebras.LieAlgebraFromAssociative}

\textbf{class} \texttt{sage.categories.examples.lie_algebras.LieAlgebraFromAssociative(gens)}
\textbf{Bases:} \texttt{sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation}

An example of a Lie algebra: a Lie algebra generated by a set of elements of an associative algebra.

This class illustrates a minimal implementation of a Lie algebra.

Let \( R \) be a commutative ring, and \( A \) an associative \( R \)-algebra. The Lie algebra \( A \) (sometimes denoted \( A^- \)) is defined to be the \( R \)-module \( A \) with Lie bracket given by the commutator in \( A \): that is, \([a, b] := ab - ba\) for all \( a, b \in A \).

What this class implements is not precisely \( A^- \), however; it is the Lie subalgebra of \( A^- \) generated by the elements of the iterable \texttt{gens}. This specific implementation does not provide a reasonable containment test (i.e., it does not allow you to check if a given element \( a \) of \( A^- \) belongs to this Lie subalgebra); it, however, allows computing inside it.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{gens} – a nonempty iterable consisting of elements of an associative algebra \( A \)
\end{itemize}
The Lie subalgebra of $A^-$ generated by the elements of gens

EXAMPLES:

We create a model of $\mathfrak{sl}_2$ using matrices:

```python
sage: gens = [matrix([[0,1],[0,0]]), matrix([[0,0],[1,0]]), matrix([[1,0],[0,-1]])]
sage: for g in gens:
....:     g.set_immutable()
sage: L = LieAlgebras(QQ).example(gens)
sage: e,f,h = L.lie_algebra_generators()
sage: e.bracket(f) == h
True
sage: h.bracket(e) == 2*e
True
sage: h.bracket(f) == -2*f
True
```

class Element

Bases: sage.structure.element_wrapper.ElementWrapper

Wrap an element as a Lie algebra element.

lie_algebra_generators()

Return the generators of self as a Lie algebra.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).example()
sage: L.lie_algebra_generators()
Family ([2, 1, 3], [2, 3, 1])
```

zero()

Return the element 0.

EXAMPLES:

```python
sage: L = LieAlgebras(QQ).example()
sage: L.zero()
0
```

5.22 Examples of a Lie algebra with basis

class sage.categories.examples.lie_algebras_with_basis.AbelianLieAlgebra($R$, $gens$)

Bases: sage.combinat.free_module.CombinatorialFreeModule

An example of a Lie algebra: the abelian Lie algebra.

This class illustrates a minimal implementation of a Lie algebra with a distinguished basis.

class Element

Bases: sage.modules.with_basis.indexed_element.IndexedFreeModuleElement

lift()

Return the lift of self to the universal enveloping algebra.
EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: elt = L.an_element()
sage: elt.lift()
3*P[F[2]] + 2*P[F[1]] + 2*P[F[]]
```

**bracket_on_basis**(x, y)

Return the Lie bracket on basis elements indexed by x and y.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.bracket_on_basis(Partition([4,1]), Partition([2,2,1]))
0
```

**lie_algebra_generators**()

Return the generators of self as a Lie algebra.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: L.lie_algebra_generators()
Lazy family (Term map from Partitions to
An example of a Lie algebra: the abelian Lie algebra on the
generators indexed by Partitions over Rational
Field(i))_{i in Partitions}
```

```
sage.categories.examples.lie_algebras_with_basis.Example
    alias of sage.categories.examples.lie_algebras_with_basis.AbelianLieAlgebra

class sage.categories.examples.lie_algebras_with_basis.IndexedPolynomialRing(R, indices, **kwds)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Polynomial ring whose generators are indexed by an arbitrary set.

Todo: Currently this is just used as the universal enveloping algebra for the example of the abelian Lie algebra. This should be factored out into a more complete class.

**algebra_generators**()

Return the algebra generators of self.

EXAMPLES:

```
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: UEA.algebra_generators()
Lazy family (algebra generator map(i))_{i in Partitions}
```

**one_basis**()

Return the index of element 1.

EXAMPLES:
sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: UEA.one_basis()
1
sage: UEA.one_basis().parent()
Free abelian monoid indexed by Partitions

product_on_basis(x, y)
Return the product of the monomials indexed by x and y.

EXAMPLES:

sage: L = LieAlgebras(QQ).WithBasis().example()
sage: UEA = L.universal_enveloping_algebra()
sage: I = UEA._indices
sage: UEA.product_on_basis(I.an_element(), I.an_element())

5.23 Examples of monoids

sage.categories.examples.monoids.Example
alias of sage.categories.examples.monoids.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_new() . . . pass
    running ._test_not_implemented_methods() . . . pass
    running ._test_pickling() . . . pass
    pass
    running ._test_elements_eq_reflexive() . . . pass
    running ._test_elements_eq_symmetric() . . . pass
    running ._test_elements_eq_transitive() . . . pass
    running ._test_elements_neq() . . . pass
    running ._test_eq() . . . pass
    running ._test_new() . . . pass
    running ._test_pickling() . . . pass
class Element
    Bases: sage.structure.element_wrapper.ElementWrapper

    monoid_generators()
    Return the generators of this monoid.

    EXAMPLES:

    sage: M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    sage: M.monoid_generators()
    Finite family {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd'}
    sage: a,b,c,d = M.monoid_generators()
    sage: a*d*c*b
    'adcb'

    one()
    Returns the one of the monoid, as per Monoids.ParentMethods.one().

    EXAMPLES:

    sage: M = Monoids().example(); M
    An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd')
    sage: M.one()
    '•'

5.24 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:

```python
sage: P = Posets().example(); P
An example of a poset: sets ordered by inclusion

We conclude by running systematic tests on this poset:

```python
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_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass
```

class Element

Bases: sage.structure.element_wrapper.ElementWrapper

```
wrapped_class
    alias of sage.sets.set.Set_objectEnumerated

an_element()  
Returns an element of this poset

EXAMPLES:

```python
sage: B = Posets().example()
sage: B.an_element()
{1, 4, 6}

le(x, y)  
Returns whether \( x \) is a subset of \( y \)

EXAMPLES:

```python
sage: P = Posets().example()
sage: P.le( P(Set([1,3])), P(Set([1,2,3])) )
True
sage: P.le( P(Set([1,3])), P(Set([1,3])) )
True
sage: P.le( P(Set([1,2])), P(Set([1,3])) )
False
```
class sage.categories.examples.posets.PositiveIntegersOrderedByDivisibilityFacade
    
    Bases: 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 ordered by divisibility` is a facade for

sage: 3 in P
True
sage: 0 in P
False
```

class element_class(X)
    
    Bases: sage.sets.set.Set_objectEnumerated, sage.categories.finite_sets.FiniteSets.parent_class

    A finite enumerated set.

    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
```

5.25 Examples of semigroups in cython

class sage.categories.examples.semigroups_cython.IdempotentSemigroups(s=None)
    
    Bases: sage.categories.category.Category

    class ElementMethods

        is_idempotent()

        EXAMPLES:
sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup()
sage: S(2).is_idempotent()
True

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.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. The category itself must be a Python class though.

This is purely a proof of concept. The code obviously needs refactorisation!

Comments:

• one cannot play ugly class surgery tricks (as with _mul_parent). available operations should really be declared to the coercion model!

EXAMPLES:

sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
sage: S = LeftZeroSemigroup(); S
An example of a semigroup: the left zero semigroup

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
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

(continues on next page)
That’s really the only method which is obtained from the category ...

```python
sage: S(42).is_idempotent
<bound method IdempotentSemigroups.element_class.is_idempotent of 42>
sage: S(42).is_idempotent()
True

sage: S(42)._pow_int
<bound method IdempotentSemigroups.element_class._pow_int of 42>
sage: S(42)^10
42

sage: S(42).is_idempotent
<bound method IdempotentSemigroups.element_class.is_idempotent of 42>
sage: S(42).is_idempotent()
True
```

**Element**

alias of `LeftZeroSemigroupElement`

```python
class sage.categories.examples.semigroups_cython.LeftZeroSemigroupElement
    Bases: sage.structure.element.Element

    EXAMPLES:

    ```python
    sage: from sage.categories.examples.semigroups_cython import LeftZeroSemigroup
    sage: S = LeftZeroSemigroup()
    sage: x = S(3)
    sage: TestSuite(x).run()
    ```
```

## 5.26 Examples of semigroups

```python
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 concatenation:

```
sage: S('dab') * S('acb')
dabacb'
```

**class** `Element`

**Bases:** `sage.structure.element_wrapper.ElementWrapper`

The class for elements of the free semigroup.

**an_element()**

Returns an element of the semigroup.

**EXAMPLES:**

```
sage: F = Semigroups().example('free')
sage: F.an_element()
'abcd'
```

**product** *(x, y)*

Returns the product of `x` and `y` in the semigroup, as per `Semigroups.ParentMethods.product()`.

**EXAMPLES:**

```
sage: F = Semigroups().example('free')
sage: F.an_element() * F('a')^5
'abcdaaaaa'
```

**semigroup_generators()**

Returns the generators of the semigroup.

**EXAMPLES:**

```
sage: F = Semigroups().example('free')
sage: F.semigroup_generators()
Family ('a', 'b', 'c', 'd')
```

**class** `sage.categories.examples.semigroups.IncompleteSubquotientSemigroup`(category=None)

**Bases:** `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
```
class Element
    Bases: sage.structure.element_wrapper.ElementWrapper

ambient()
    Returns the ambient semigroup.

    EXAMPLES:
    
    sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
    An example of a semigroup: the left zero semigroup

class sage.categories.examples.semigroups.LeftZeroSemigroup
    Bases: 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

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

    an_element()
    Returns an element of the semigroup.

    EXAMPLES:
    
    sage: Semigroups().example().an_element()
**product** $(x, y)$
Returns the product of $x$ and $y$ in the semigroup, as per `Semigroups.ParentMethods.product()`.

**EXAMPLES:**
```
sage: S = Semigroups().example()
sage: S('hello') * S('world')
'hello'
sage: S(3) * S(1) * S(2)
3
```

**some_elements()**
Returns a list of some elements of the semigroup.

**EXAMPLES:**
```
sage: Semigroups().example().some_elements()
[3, 42, 'a', 3.4, 'raton laveur']
```

**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
```

**class** `Element`

Bases: `sage.structure.element_wrapper.ElementWrapper`

**ambient()**
Returns the ambient semigroup.

**EXAMPLES:**
```
sage: S = Semigroups().Subquotients().example()
sage: S.ambient()
An example of a semigroup: the left zero semigroup
```

5.26. Examples of semigroups 765
an_element()  
Returns an element of the semigroup.

EXAMPLES:

```sage
S = Semigroups().Subquotients().example()
sage: S.an_element()
42
```

lift(x)  
Lift the element \( x \) into the ambient semigroup.

INPUT:

- \( x \) – an element of \( \text{self} \).

OUTPUT:

- an element of \( \text{self.ambient}() \).

EXAMPLES:

```sage
S = Semigroups().Subquotients().example()
x = S.an_element(); x
42
sage: S.lift(x)
42
sage: S.lift(x) in S.ambient()
True
y = S.ambient()(100); y
100
sage: S.lift(S(y))
42
```

retract(x)  
Returns the retract \( x \) onto an element of this semigroup.

INPUT:

- \( x \) – an element of the ambient semigroup \( \text{self.ambient}() \).

OUTPUT:

- an element of \( \text{self} \).

EXAMPLES:

```sage
S = Semigroups().Subquotients().example()
L = S.ambient()
sage: S.retract(L(17))
17
sage: S.retract(L(42))
42
sage: S.retract(L(171))
42
```

some_elements()  
Returns a list of some elements of the semigroup.

EXAMPLES:
sage: S = Semigroups().Subquotients().example()
sage: S.some_elements()
[1, 2, 3, 8, 42, 42]

the_answer()
Returns the Answer to Life, the Universe, and Everything as an element of this semigroup.

EXAMPLES:

sage: S = Semigroups().Subquotients().example()
sage: S.the_answer()
42

5.27 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 \( \mathbb{Z} \) (facade implementation).

This is a minimal implementations. For more advanced examples of implementations, see also:

sage: P = Sets().example("facade")
sage: P = Sets().example("inherits")
sage: P = Sets().example("wrapper")

EXAMPLES:

sage: P = Sets().example()
sage: P(12)
Traceback (most recent call last):
  ...  
AssertionError: 12 is not a prime number
sage: a = P.an_element()
sage: a.parent()
Integer Ring
sage: x = P(13); x
13
sage: type(x)
<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(verbosetrue)
running ._test_an_element() ... pass

(continues on next page)
Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
running ._test_category() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_nonzero_equal() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_new() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_pickling() . . . pass
running ._test_some_elements() . . . pass

\texttt{an\_element()}

Implements \textit{Sets.ParentMethods.an\_element()}.

\texttt{element\_class}

alias of \texttt{sage.rings.integer.Integer}

class \texttt{sage.categories.examples.sets_cat.PrimeNumbers\_Abstract}

\texttt{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 \texttt{PrimeNumbers\_Abstract}.

This is used by:

sage: P = Sets().example(“facade”) sage: P = Sets().example(“inherits”) sage: P = Sets().example(“wrapper”)

class \texttt{Element}

\texttt{Bases: sage.structure.element.Element}

\texttt{is\_prime()}

Return whether \texttt{self} is a prime number.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: P = Sets().example(“inherits”)
sage: x = P.an_element()
sage: P.an_element().is_prime()
True
\end{verbatim}

\texttt{next()}

Return the next prime number.

\texttt{EXAMPLES:}

\begin{verbatim}
sage: P = Sets().example(“inherits”)
sage: p = P.an_element(); p
\end{verbatim}
47
\begin{verbatim}sage: p.next()
53
\end{verbatim}

Note: This method is not meant to implement the protocol iterator, and thus not subject to Python 2 vs Python 3 incompatibilities.

\begin{verbatim}
def an_element()
    Implements \texttt{Sets.ParentMethods.an_element().}

def next(i)
    Return the next prime number.

    EXAMPLES:

    \begin{verbatim}sage: P = Sets().example("inherits")
sage: x = P.next(P.an_element()); x
    \end{verbatim}

\begin{verbatim}sage: x.parent()
Set of prime numbers
\end{verbatim}

\begin{verbatim}
def some_elements()
    Return some prime numbers.

    EXAMPLES:

    \begin{verbatim}sage: P = Sets().example("inherits")
sage: P.some_elements()
[47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
\end{verbatim}
\end{verbatim}

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 \( \mathbb{Z} \) (facade implementation).

EXAMPLES:

\begin{verbatim}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
\end{verbatim}
\end{verbatim}

(continues on next page)
The disadvantage of this implementation is that the elements do not know that they are prime, so that prime
testing is slow:

\begin{verbatim}
sage: pf = Sets().example("facade").an_element()
sage: timeit("pf.is_prime()") # random
625 loops, best of 3: 4.1 us per loop
\end{verbatim}

compared to the other implementations where prime testing is only done if needed during the construction of
the element, and later on the elements “know” that they are prime:

\begin{verbatim}
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
\end{verbatim}

Note also that the `next` method for the elements does not exist:

\begin{verbatim}
sage: pf.next()
Traceback (most recent call last):
  ... AttributeError: 'sage.rings.integer.Integer' object has no attribute 'next'
\end{verbatim}

unlike in the other implementations:

\begin{verbatim}
sage: pw.next()
53
sage: pi.next()
53
\end{verbatim}

\texttt{element_class}

alias of \texttt{sage.rings.integer.Integer}

class \texttt{sage.categories.examples.sets_cat.PrimeNumbers_Inherits}

\texttt{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 \texttt{Integer} (technically \texttt{IntegerWrapper}).

\textbf{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: 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_new() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
(continues on next page)
running ._test_new() . . . 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.PrimeNumbers_Abstract.Element

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)
class Element
Bases: sage.structure.element_wrapper.ElementWrapper, sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element

ElementWrapper
alias of sage.structure.element_wrapper.ElementWrapper

5.28 Example of a set with grading

sage.categories.examples.sets_with_grading.Example
alias of sage.categories.examples.sets_with_grading.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^15 + z^16 + z^17 + z^18 + z^19 + O(z^20)
```

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:

```python
sage: N = SetsWithGrading().example()
sage: N.grading(10)
10
```

## 5.29 Examples of parents endowed with multiple realizations

```python
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:

\[
In_s = \sum_{t \subset s} F_t \quad \text{and} \quad F_s = \sum_{t \supset s} Out_t
\]

Each such basis gives a realization of \( A \), where the elements are represented by their expansion in this basis.

This parent, and its code, demonstrate how to implement this algebra and its three realizations, with coercions and mixed arithmetic between them.

See also:

- \texttt{Sets().WithRealizations}
- the Implementing Algebraic Structures thematic tutorial.

EXAMPLES:

```python
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 \( A \):

```python
sage: F = A.F(); F
The subset algebra of \{1, 2, 3\} over Rational Field in the Fundamental basis
sage: In = A.In(); In
The subset algebra of \{1, 2, 3\} over Rational Field in the In basis
sage: Out = A.Out(); Out
The subset algebra of \{1, 2, 3\} over Rational Field in the Out basis
```

One can quickly define all the bases using the following shortcut:
Accessing the basis elements is done with `basis()` method:

```sage
sage: F.basis().list()
[0, 1, 2, 3, 4, 5, 6, 7]
```

To access a particular basis element, you can use the `from_set()` method:

```sage
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
sage: F[2,3]
F[2, 3]
sage: In[1,3]
In[1, 3]
```

Some conversions:

```sage
sage: F(In[2,3])
F[2, 3]
sage: In(F[2,3])
In[1, 3]
sage: Out(F[3])
Out[1, 3]
sage: F(Out[3])
F[3]
sage: Out(In[2,3])
Out[2, 3]
```

We can now mix expressions:

```sage
sage: (1 + Out[1]) * In[2,3]
```

```

```sage
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.

```5.29. Examples of parents endowed with multiple realizations

775
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}]
```

```python
one()
```

Returns the unit of this algebra.

This default implementation takes the unit in the fundamental basis, and coerces it in `self`.

EXAMPLES:

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

```
sage: Out.one()
Out[{}] + Out[{1}] + Out[{2}] + Out[{3}] + Out[{1, 2}] + Out[{1, 3}] + Out[{2, 3}] + Out[{1, 2, 3}]
```

```
super_categories()
```

EXAMPLES:

```
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: C = A.Bases(); C
Category of bases of The subset algebra of {1, 2, 3} over Rational Field
sage: C.super_categories()
[Category of realizations of The subset algebra of {1, 2, 3} over Rational Field
Join of Category of algebras with basis over Rational Field and
Category of commutative algebras over Rational Field and
Category of realizations of unital magmas]
```

```python
class Fundamental(A)
```

```
Bases:
sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_classBindableClass
```

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 In(A)  
Bases:  
sage.combinat.free_module.CombinatorialFreeModule,  
sage.misc.bindable_classBindableClass

The Subset Algebra, in the In basis

INPUT:
• A – a parent with realization in SubsetAlgebra

EXAMPLES:

```
sage: A = Sets().WithRealizations().example()  
sage: A.In()  
The subset algebra of {1, 2, 3} over Rational Field in the In basis
```
class Out(A)

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_classBindableClass

The Subset Algebra, in the Out basis

INPUT:
• A – a parent with realization in SubsetAlgebra

EXAMPLES:

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

a_realization()
Returns the default realization of self

EXAMPLES:

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

base_set()

EXAMPLES:

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

indices()
The objects that index the basis elements of this algebra.

EXAMPLES:

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

indices_key(x)
A key function on a set which gives a linear extension of the inclusion order.

INPUT:
• x – set

EXAMPLES:

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

supsets(set)
Returns all the subsets of S containing set

INPUT:
- set – a subset of the base set $S$ of self

**EXAMPLES:**

```python
sage: A = Sets().WithRealizations().example(); A
The subset algebra of {1, 2, 3} over Rational Field
sage: A.supsets(Set((2,)))
[[(1, 2, 3), (2, 3), (1, 2), (2)]]
```
6.1 Specific category classes

This is placed in a separate file from categories.py to avoid circular imports (as morphisms must be very low in the hierarchy with the new coercion model).

```python
class sage.categories.category_types.AbelianCategory(s=None):
    Bases: sage.categories.category.Category

    is_abelian() -> bool
    Return True as self is an abelian category.

    EXAMPLES:
    sage: CommutativeAdditiveGroups().is_abelian()  # doctest: +NORMALIZE_WHITESPACE
    True

class sage.categories.category_types.Category_ideal(ambient, name=None):
    Bases: sage.categories.category_types.Category_in_ambient

    classmethod an_instance() -> self
    Return an instance of this class.

    EXAMPLES:
    sage: AlgebraIdeals.an_instance()  # doctest: +NORMALIZE_WHITESPACE
    Category of algebra ideals in Univariate Polynomial Ring in x over Rational Field

    ring() -> self
    Return the ambient ring used to describe objects self.

    EXAMPLES:
    sage: C = Ideals(IntegerRing())
    sage: C.ring()  # doctest: +NORMALIZE_WHITESPACE
    Integer Ring

class sage.categories.category_types.Category_in_ambient(ambient, name=None):
    Bases: sage.categories.category.Category

    Initialize self.

    EXAMPLES:
```
ambient()  
Return the ambient object in which objects of this category are embedded.

EXAMPLES:

```
sage: C = Ideals(IntegerRing())
sage: C.ambient()
Integer Ring
```

class sage.categories.category_types.Category_module(base, name=None)
Bases: sage.categories.category_types.AbelianCategory, sage.categories.category_types.Category_over_base_ring

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

EXAMPLES:

```
sage: C = Algebras(QQ)
sage: C.base()
Rational Field
```

Initialize self.

EXAMPLES:

```python
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:

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

```python
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:

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

**super_categories()**

EXAMPLES:

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

```python
class sage.categories.category_types.Elements(object)
Bases: sage.categories.category.Category

The category of all elements of a given parent.

EXAMPLES:

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

**classmethod an_instance()**

Returns an instance of this class

EXAMPLES:

```python
sage: Elements.an_instance()
Category of elements of Rational Field
```
object ()
EXAMPLES:

```
sage: Elements(ZZ).object()
Integer Ring
```

super_categories ()
EXAMPLES:

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

Todo: Check that this is what we want.

### 6.2 Singleton categories

**class** `sage.categories.category_singleton.Category_contains_method_by_parent_class`

Bases: `object`

Returns whether \( x \) is an object in this category.

More specifically, returns `True` if and only if \( x \) has a category which is a subcategory of this one.

EXAMPLES:

```
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”:

```python
sage: R = QQ['x,y']
sage: R1 = Parent(category = MyRings())
sage: R2 = Parent(category = MyRingsSingleton())
```

```python
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:

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

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

```python
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:

```python
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 \mu s per loop
sage: timeit("R3 in Rings()", number=10000) # not tested
10000 loops, best of 3: 3.01 \mu s 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:

```python
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 \texttt{Category\_singleton} are supported:

\begin{verbatim}
sage: class MyRingsSingleton(Category\_singleton):
    ....:    def super\_categories(self): return Rings().super\_categories()
sage: class Disaster(MyRingsSingleton): pass
sage: Disaster()
Traceback (most recent call last):
  ...
AssertionError: <class '__main__.Disaster'> is not a direct subclass of <class ...
˓→'sage.categories.category\_singleton.Category\_singleton'>
\end{verbatim}

However, it is acceptable for a direct subclass \( R \) of \texttt{Category\_singleton} to create its unique instance as an instance of a subclass of itself (in which case, its the subclass of \( R \) which is concrete, not \( R \) itself). This is used for example to plug in extra category code via a dynamic subclass:

\begin{verbatim}
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.categories.sets\_with\_partial\_maps.SetsWithPartialMaps.˓→subcategory\_class'>,
 <class 'sage.categories.objects.Objects.subcategory\_class'>,
 <... 'object'>]
sage: R() \is\ R()
True
sage: R() \is\ R().__class__()
True
\end{verbatim}

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:

\texttt{Category\_\_classcall\_\_()}, \texttt{Category\_\_init\_\_()}

Note: The \_\_test\_category test is failing because \texttt{MyRingsSingleton()} is not a subcategory of the join of its super categories:

\begin{verbatim}
sage: C = MyRingsSingleton()
\end{verbatim}
Oh well; it's not really relevant for those tests.

6.3 Fast functions for the category framework

AUTHOR:
• Simon King (initial version)

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

6.3. Fast functions for the category framework

AUTHOR:

1. Simon King (initial version)

```python
class sage.categories.category_cy_helper.AxiomContainer
Bases: dict

A fast container for axioms.

This is derived from dict. A key is the name of an axiom. The corresponding value is the “rank” of this axiom, that is used to order the axioms in canonicalize_axioms().

EXAMPLES:
```
```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: isinstance(all_axioms, sage.categories.category_with_axiom.AxiomContainer)
True
```

```python
add(axiom)
Add a new axiom name, of the next rank.

EXAMPLES:
```
```python
sage: all_axioms = sage.categories.category_with_axiom.all_axioms
sage: m = max(all_axioms.values())
sage: all_axioms.add('Awesome')
sage: all_axioms['Awesome'] == m + 1
True
```

To avoid side effects, we remove the added axiom:
```
```python
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'])
('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:
6.4 Coercion methods for categories

The purpose of this Cython module is to hold special coercion methods, which are inserted by their respective categories.

6.5 Poor Man’s map

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

INPUT:

• function – a callable or an iterable of callables. This represents the underlying function used to implement this map. If it is an iterable, then the callables will be composed to implement this map.

• domain – the domain of this map or None if the domain is not known or should remain unspecified

• codomain – the codomain of this map or None if the codomain is not known or should remain unspecified

• name – a name for this map or None if this map has no particular name

EXAMPLES:

sage: from sage.categories.poor_man_map import PoorManMap
sage: f = PoorManMap(factorial, domain = (1, 2, 3), codomain = (1, 2, 6))
sage: f
A map from (1, 2, 3) to (1, 2, 6)
sage: f(3)
6
The composition of several functions can be created by passing in a tuple of functions:

```python
sage: i = PoorManMap((factorial, sqrt), domain= (1, 4, 9), codomain = (1, 2, 6))
```

However, the same effect can also be achieved by just composing maps:

```python
sage: g = PoorManMap(factorial, domain = (1, 2, 3), codomain = (1, 2, 6))
sage: h = PoorManMap(sqrt, domain = (1, 4, 9), codomain = (1, 2, 3))
sage: i == g*h
True
```

**codomain()**

Returns the codomain of `self`

**EXAMPLES:**

```python
sage: from sage.categories.poor_man_map import PoorManMap
dsage: PoorManMap(lambda x: x+1, domain = (1,2,3), codomain = (2,3,4)).codomain()
(2, 3, 4)
```

**domain()**

Returns the domain of `self`

**EXAMPLES:**

```python
sage: from sage.categories.poor_man_map import PoorManMap
dsage: 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
BIBLIOGRAPHY


[StThWi] Christian Stump, Hugh Thomas and Nathan Williams, *Cataland: why the fuss?*, Arxiv 1503.00710
C
sage.categories.action, 147
sage.categories.additive_groups, 150
sage.categories.additive_magmas, 151
sage.categories.additive_monoids, 162
sage.categories.additive_semigroups, 163
sage.categories.affine_weyl_groups, 165
sage.categories.algebra_functor, 704
sage.categories.algebra_ideals, 168
sage.categories.algebra_modules, 169
sage.categories.algebras, 170
sage.categories.algebras_with_basis, 172
sage.categories.aperiodic_semigroups, 177
sage.categories.associative_algebras, 177
sage.categories.bialgebras, 177
sage.categories.bialgebras_with_basis, 178
sage.categories.bimodules, 183
sage.categories.cartesian_product, 700
sage.categories.category, 26
sage.categories.category_cy_helper, 787
sage.categories.category_singleton, 784
sage.categories.category_types, 781
sage.categories.category_with_axiom, 62
sage.categories.classical_crystals, 184
sage.categories.coalgebras, 188
sage.categories.coalgebras_with_basis, 192
sage.categories.coercion_methods, 789
sage.categories.commutative_additive_groups, 194
sage.categories.commutative_additive_monoids, 195
sage.categories.commutative_additive_semigroups, 196
sage.categories.commutative_algebra_ideals, 196
sage.categories.commutative_algebras, 197
sage.categories.commutative_ring_ideals, 197
sage.categories.commutative_rings, 197
sage.categories.complete_discrete_valuation, 200
sage.categories.complex_reflection_groups, 203
sage.categories.complex_reflection_or_generalized_coxeter_groups, 204
sage.categories.integral_domains, 473
sage.categories.isomorphic_objects, 712
sage.categories.j_trivial_semigroups, 473
sage.categories.kac_moody_algebras, 474
sage.categories.l_trivial_semigroups, 501
sage.categories.lattice_posets, 474
sage.categories.left_modules, 475
sage.categories.lie_algebras, 476
sage.categories.lie_algebras_with_basis, 486
sage.categories.lie_groups, 488
sage.categories.loop_crystals, 489
sage.categories.magnmas, 502
sage.categories.magnmas_and_additive_magnmas, 515
sage.categories.magmatic_algebras, 517
sage.categories.manifolds, 521
sage.categories.map, 101
sage.categories.matrix_algebras, 524
sage.categories.metric_spaces, 525
sage.categories.modular_abelian_varieties, 527
sage.categories.modules, 528
sage.categories.modules_with_basis, 537
sage.categories.monoid_algebras, 563
sage.categories.monoids, 563
sage.categories.morphism, 116
sage.categories.number_fields, 569
sage.categories.objects, 570
sage.categories.partially_ordered_monoids, 571
sage.categories.permutation_groups, 572
sage.categories.pointed_sets, 572
sage.categories.polyhedra, 573
sage.categories.poor_man_map, 789
sage.categories.posets, 573
sage.categories.primer, 1
sage.categories.principal_ideal_domains, 582
sage.categories.pushout, 120
sage.categories.quantum_group_representations, 589
sage.categories.quotient_fields, 583
sage.categories.quotients, 710
sage.categories.r_trivial_semigroups, 617
sage.categories.realizations, 716
sage.categories.regular_crystals, 595
sage.categories.regular_supercrystals, 602
sage.categories.right_modules, 608
sage.categories.ring_ideals, 608
sage.categories.rings, 609
sage.categories.rngs, 617
sage.categories.schemes, 618
sage.categories.semirings, 630
sage.categories.semigroups, 619
sage.categories.semirings, 630
sage.categories.semisimple_algebras, 630
sage.categories.sets_cat, 631
sage.categories.sets_with_grading, 657
sage.categories.sets_with_partial_maps, 659
sage.categories.shephard_groups, 660
sage.categories.simplicial_complexes, 660
sage.categories.simplicial_sets, 661
sage.categories.subobjects, 711
sage.categories.subquotients, 710
sage.categories.super_algebras, 668
sage.categories.super_algebras_with_basis, 668
sage.categories.super_hopf_algebras_with_basis, 669
sage.categories.super_modules, 669
sage.categories.super_modules_with_basis, 672
sage.categories.tensor, 702
sage.categories.topological_spaces, 674
sage.categories.triangular_kac_moody_algebras, 675
sage.categories.tutorial, 99
sage.categories.unique_factorization_domains, 676
sage.categories.unital_algebras, 678
sage.categories.vector_spaces, 680
sage.categories.weyl_groups, 682
sage.categories.with_realizations, 718
Symbols
__classcall__() (sage.categories.category.Category static method), 38
__classcall__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 87
__classget__() (sage.categories.category_with_axiom.CategoryWithAxiom static method), 87
__init__() (sage.categories.category.Category method), 39
__init__() (sage.categories.category_with_axiom.CategoryWithAxiom method), 88
_all_super_categories() (sage.categories.category.Category method), 32
_all_super_categories_proper() (sage.categories.category.Category method), 33
_make_named_class() (sage.categories.category.Category method), 34
_make_named_class() (sage.categories.category.CategoryWithParameters.Category method), 58
_repr__() (sage.categories.category.Category method), 35
_repr__() (sage.categories.category.JoinCategory.Category method), 59
_repr_object_names() (sage.categories.category.Category method), 35
_repr_object_names() (sage.categories.category.JoinCategory.Category method), 59
_repr_object_names() (sage.categories.category_with_axiom.CategoryWithAxiom method), 89
_sort() (sage.categories.category.Category static method), 37
_sort_uniq() (sage.categories.category.Category static method), 38
_super_categories() (sage.categories.category.Category method), 32
_super_categories_for_classes() (sage.categories.category.Category method), 32
_test_category() (sage.categories.category.Category method), 35
_test_category_with_axiom() (sage.categories.category_with_axiom.CategoryWithAxiom method), 89
_with_axiom() (sage.categories.category.Category method), 36
_with_axiom_as_tuple() (sage.categories.category.Category method), 36
_without_axioms() (sage.categories.category.Category method), 37
_without_axioms() (sage.categories.category.JoinCategory.Category method), 60
_without_axioms() (sage.categories.category_with_axiom.CategoryWithAxiom method), 89

A
a_realization() (sage.categories.examples.with_realizations.SubsetAlgebra method), 778
a_realization() (sage.categories.sets_cat.Sets.WithRealizations.ParentMethods method), 653
AbelianCategory (class in sage.categories.category_types), 781
AbelianLieAlgebra (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 736
AbelianLieAlgebra (class in sage.categories.examples.lie_algebras_with_basis), 755
AbelianLieAlgebra.Element (class in sage.categories.examples.finite_dimensional_lie_algebras_with_basis), 736
AbelianLieAlgebra.Element (class in sage.categories.examples.lie_algebras_with_basis), 755
abs() (sage.categories.metric_spaces.MetricSpaces.ElementMethods method), 525
absolute_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 225
absolute_le() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 225
absolute_length() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 226
absolute_poset() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 320
act() (sage.categories.action.Action method), 148
Action (class in sage.categories.action), 147
ActionEndomorphism (class in sage.categories.action), 148
actor() (sage.categories.action.Action method), 148
adams_operator() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ElementMethods method), 178
add() (sage.categories.category_cy_helper.AxiomContainer method), 787
addition_table() (sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital method), 156
addition_table() (sage.categories.additive_groups.AdditiveGroups method), 150
addition_table() (sage.categories.additive_monoids.AdditiveMonoids method), 162
additional_structure() (sage.categories.additive_magmas.AdditiveMagmas method), 158
additional_structure() (sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital method), 156
additional_structure() (sage.categories.additive_groups.AdditiveGroups method), 150
additional_structure() (sage.categories.additive_monoids.AdditiveMonoids method), 162
additional_structure() (sage.categories.category.Category method), 44
additional_structure() (sage.categories.category.JoinCategory method), 60
additional_structure() (sage.categories.category_with_axiom.CategoryWithAxiom method), 90
additional_structure() (sage.categories.classical_crystals.ClassicalCrystals method), 187
additional_structure() (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 204
additional_structure() (sage.categories.covariant FunctorialConstruction.CovariantConstructionCategory method), 695
additional_structure() (sage.categories.coxeter_groups.CoxeterGroups method), 252
additional_structure() (sage.categories.enumerated_sets.EnumeratedSets method), 288
additional_structure() (sage.categories.gcd_domains.GcdDomains method), 435
additional_structure() (sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups method), 436
additional_structure() (sage.categories.higher_weight_crystals.HigherWeightCrystals method), 465
additional_structure() (sage.categories.lie_groups.LieGroups method), 488
additional_structure() (sage.categories.magmas.Magmas.Unital method), 515
additional_structure() (sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas method), 517
additional_structure() (sage.categories.magmaic_algebras.MagmaicAlgebras method), 520
additional_structure() (sage.categories.manifolds.Manifolds method), 524
additional_structure() (sage.categories.modules.Modules method), 537
additional_structure() (sage.categories.objects.Objects method), 571
additional_structure() (sage.categories.principal_ideal_domains.PrincipalIdealDomains method), 582
additional_structure() (sage.categories.regular_crystals.RegularCrystals method), 602
additional_structure() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains method), 678
additional_structure() (sage.categories.vector_spaces.VectorSpaces method), 682
additional_structure() (sage.categories.weyl_groups.WeylGroups method), 691
additive_order() (sage.categories.examples.commutative_additive_groups.FiniteCommutativeAdditiveGroups.CartesianProducts.ElementMethods method), 195
additive_semigroup_generators() (sage.categories.examples.commutative_additive_semigroups.FiniteCommutativeAdditiveSemigroups method), 726
AdditiveAssociative (sage.categories.additive_magmas.AdditiveMagmas attribute), 152
AdditiveAssociative() (sage.categories.additive_monoids.AdditiveMonoids.SubcategoryMethods method), 161
AdditiveCommutative (sage.categories.additive_groups.AdditiveGroups attribute), 150
AdditiveCommutative (sage.categories.additive_monoids.AdditiveMonoids attribute), 162
AdditiveCommutative (sage.categories.additive_semigroups.AdditiveSemigroups attribute), 164
AdditiveCommutative() (sage.categories.additive_magmas.AdditiveMagmas.SubcategoryMethods method), 161
AdditiveGroups (class in sage.categories.additive_groups), 150
AdditiveGroups.Algebras (class in sage.categories.additive_groups), 150
AdditiveGroups.Algebras.ParentMethods (class in sage.categories.additive_groups), 150
AdditiveGroups.Finite (class in sage.categories.additive_groups), 151
AdditiveGroups.Finite.Algebras (class in sage.categories.additive_groups), 151
AdditiveGroups.Finite.Algebras.ParentMethods (class in sage.categories.additive_groups), 151
AdditiveInverse (sage.categories.additive_monoids.AdditiveMonoids.attribute), 163
AdditiveInverse() (sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital.SubcategoryMethods method), 155
AdditiveMagmas (class in sage.categories.additive_magmas), 151
AdditiveMagmas.AdditiveCommutative (class in sage.categories.additive_magmas), 152
AdditiveMagmas.AdditiveCommutative.Algebras (class in sage.categories.additive_magmas), 152
AdditiveMagmas.AdditiveCommutative.CartesianProducts (class in sage.categories.additive_magmas), 152
AdditiveMagmas.AdditiveUnital (class in sage.categories.additive_magmas), 153
AdditiveMagmas.AdditiveUnital.AdditiveInverse (class in sage.categories.additive_magmas), 153
AdditiveMagmas.AdditiveUnital.AdditiveInverse.CartesianProducts (class in sage.categories.additive_magmas), 153
AdditiveMagmas.AdditiveUnital.Algebras (class in sage.categories.additive_magmas), 153
AdditiveMagmas.AdditiveUnital.CartesianProducts (class in sage.categories.additive_magmas), 154
AdditiveMagmas.AdditiveUnital.CartesianProducts.ParentMethods (class in sage.categories.additive_magmas), 154
AdditiveMagmas.AdditiveUnital.ElementMethods (class in sage.categories.additive_magmas), 154
AdditiveMagmas.AdditiveUnital.Homsets (class in sage.categories.additive_magmas), 154
AdditiveMagmas.AdditiveUnital.Homsets.ParentMethods (class in sage.categories.additive_magmas), 154
AdditiveMagmas.AdditiveUnital.ParentMethods (class in sage.categories.additive_magmas), 155
AdditiveMagmas.AdditiveUnital.SubcategoryMethods (class in sage.categories.additive_magmas), 155
AdditiveMagmas.AdditiveUnital.WithRealizations (class in sage.categories.additive_magmas), 156
AdditiveMagmas.AdditiveUnital.WithRealizations.ParentMethods (class in sage.categories.additive_magmas), 156
AdditiveMagmas.Algebras (class in sage.categories.additive_magmas), 156
AdditiveMagmas.Algebras.ParentMethods (class in sage.categories.additive_magmas), 156
AdditiveMagmas.CartesianProducts (class in sage.categories.additive_magmas), 157
AdditiveMagmas.CartesianProducts.ElementMethods (class in sage.categories.additive_magmas), 157
AdditiveMagmas.Homsets (class in sage.categories.additive_magmas), 157
AdditiveMagmas.Homsets.ParentMethods (class in sage.categories.additive_magmas), 157
AdditiveMonoids (class in sage.categories.additive_monoids), 162
AdditiveMonoids.Homsets (class in sage.categories.additive_monoids), 163
AdditiveMonoids.ParentMethods (class in sage.categories.additive_monoids), 163
AdditiveSemigroups (class in sage.categories.additive_semigroups), 163
AdditiveSemigroups.Algebras (class in sage.categories.additive_semigroups), 164
AdditiveSemigroups.Algebras.ParentMethods (class in sage.categories.additive_semigroups), 164
AdditiveSemigroups.CartesianProducts (class in sage.categories.additive_semigroups), 165
AdditiveSemigroups.Homsets (class in sage.categories.additive_semigroups), 165
AdditiveSemigroups.ParentMethods (class in sage.categories.additive_semigroups), 165
AdditiveUnital (sage.categories.additive_semigroups.AdditiveSemigroups.attribute), 164
AdditiveUnital() (sage.categories.additive_magmas.AdditiveMagmas.SubcategoryMethods method), 162
adjoint_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 366
affine_grassmannian_elements_of_given_length() (sage.categories.affine_weyl_groups.AffineWeylGroups.PARENT METHODS method), 167
affine_grassmannian_to_core() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 166
affine_grassmannian_to_partition() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 166
AffineWeylGroups (class in sage.categories.affine_weyl_groups), 165
AffineWeylGroups.ElementMethods (class in sage.categories.affine_weyl_groups), 166
affinization() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 491
algebra() (sage.categories.algebra_ideals.AlgebraIdeals method), 168
algebra() (sage.categories.algebra_modules.AlgebraModules method), 169
algebra() (sage.categories.commutative_algebra_ideals.CommutativeAlgebraIdeals method), 196
algebra() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 166
affine_grassmannian_elements_of_given_length() (sage.categories.affine_weyl_groups.AffineWeylGroups.ParentMethods method), 167
affinization() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 494
AlgebraFunctor (class in sage.categories.algebra_functor), 708
AlgebraicClosureFunctor (class in sage.categories.pushout), 120
AlgebraicExtensionFunctor (class in sage.categories.pushout), 120
algebra() (sage.categories.algebra_ideals.ElementMethods method), 168
algebra() (sage.categories.algebra_modules.ElementMethods method), 169
algebra() (sage.categories.commutative_algebra_ideals.ElementMethods method), 196
algebra() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 166
algebra() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 166
AlgebraFunctor (class in sage.categories.algebra_functor), 708
AlgebraicClosureFunctor (class in sage.categories.pushout), 120
AlgebraicExtensionFunctor (class in sage.categories.pushout), 120
Algebras (sage.categories.coxeter_groups.CoxeterGroups attribute), 225
Algebras (sage.categories.groups.Groups attribute), 451
Algebras (sage.categories.sets_cat.Sets.SubcategoryMethods method), 643
Algebras.CartesianProducts (class in sage.categories.algebras), 170
Algebras.DualObjects (class in sage.categories.algebras), 171
Algebras.ElementMethods (class in sage.categories.algebras), 171
Algebras.Quotients (class in sage.categories.algebras), 171
Algebras.Quotients.ParentMethods (class in sage.categories.algebras), 171
Algebras.SubcategoryMethods (class in sage.categories.algebras), 171
Algebras.TensorProducts (class in sage.categories.algebras), 172
Algebras.TensorProducts.ElementMethods (class in sage.categories.algebras), 172
Algebras.TensorProducts.ParentMethods (class in sage.categories.algebras), 172
antichains() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 383
antipode() (sage.categories.hopf_algebras.HopfAlgebras.ElementMethods method), 466
antipode() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.ParentMethods method), 470
antipode_on_basis() (sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra method), 439
antipode_on_basis() (sage.categories.group_algebras.GroupAlgebras.ParentMethods method), 447
antipode_on_basis() (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis.ParentMethods method), 470
aprioric() (sage.categories.semigroups.Semigroups.attribute), 621
AperiodicSemigroups (class in sage.categories.aperiodic_semigroups), 177
apply_conjugation_by_simple_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 206
apply_demazure_product() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 226
apply_multilinear_morphism() (sage.categories.modules_with_basis.ModulesWithBasis.TensorProducts.ElementMethods method), 560
apply_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 206
apply_simple_projection() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 227
apply_simple_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 207
apply_simple_reflection_left() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 208
apply_simple_reflection_right() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 208
apply_simple_reflection_right() (sage.categories.examples.finite_coxeter_groups.DihedralGroup.Element method), 731
apply_simple_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 209
as_finite_dimensional_algebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 367
Associative (sage.categories.magma.Magma.attribute), 503
Associative (sage.categories.magmatic_algebras_MAGmaticAlgebras.attribute), 518
Associative() (sage.categories.magma.Magma.SubcategoryMethods method), 509
AssociativeAlgebras (class in sage.categories.associative_algebras), 177
axiom() (in module sage.categories.category_with_axiom), 93
axiom_of_nested_class() (in module sage.categories.category_with_axiom), 94
AxiomContainer (class in sage.categories.category_cy_helper), 787
axioms() (sage.categories.category.Category méthode), 47
axioms() (sage.categories.category_with_axiom.CategoryWithAxiom method), 90
b_sharp() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 491
baker_campbell_hausdorff() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 479
Bars (class in sage.categories.category_with_axiom), 85
base() (sage.categories.category_types.Category_over_base method), 782
base() (sage.categories.homsets.HomsetsCategory method), 714
base() (sage.categories.tensor.TensorProductsCategory method), 703
Blahs.Unital.Blue (class in sage.categories.category_with_axiom), 87
Blue() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 87
Blue_extra_super_categories() (sage.categories.category_with_axiom.Blahs method), 86
bracket() (sage.categories.lie_algebras.LieAlgebras.ElementMethods method), 477
bracket() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 481
bracket_on_basis() (sage.categories.examples.lie_algebras_with_basis.AbelianLieAlgebra method), 756
bracket_on_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 487
braid_group_as_finitely_presented_group() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 243
braid_orbit() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 243
braid_relations() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_graph() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_graph() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_interval() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 245
bruhat_interval_poset() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 245
bruhat_le() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_lower_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_lower_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 229
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_lower_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_lower_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 229
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_lower_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_lower_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 229
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_lower_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 228
bruhat_lower_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 229
bruhat_lower_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_upper_covers() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 229
bruhat_upper_covers() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 244
bruhat_upper_covers_coroots() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
bruhat_upper_covers_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 230
C
CallMorphism (class in sage.categories.morphism), 116
cambrian_lattice() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 337
canonical_matrix() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 230
canonical_representation() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 246
canonicalize_axioms() (in module sage.categories.category_cy_helper), 787
cardinality() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods method), 185
cardinality() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 324
cardinality() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.CartesianProducts.ParentMethods method), 390
cardinality() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.IsomorphicObjects.ParentMethods method), 392
cardinality() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ParentMethods method), 393
cardinality() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), 396
cardinality() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 491
cardinality() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 550
cardinality() (sage.categories.sets_cat.Sets.CartesianProducts.ParentMethods method), 635
cardinality() (sage.categories.sets_cat.Sets.Infinite.ParentMethods method), 638
cartan_invariants_matrix() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 351
cartan_type() (sage.categories.crystals.CrystalMorphism method), 255
cartan_type() (sage.categories.crystals.Crystals.ElementMethods method), 259
cartan_type() (sage.categories.crystals.Crystals.ParentMethods method), 264
cartan_type() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 744
cartan_type() (sage.categories.kac_moody_algebras.KacMoodyAlgebras.ParentMethods method), 474
cartan_type() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 589
cartan_type() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.TensorProducts.ParentMethods method), 590
cartesian_product() (sage.categories.sets_cat.Sets.ElementMethods method), 637
cartesian_product() (sage.categories.sets_cat.Sets.ParentMethods method), 641
CartesianProduct (sage.categories.posets.Posets.ParentMethods attribute), 574
CartesianProduct (sage.categories.sets_cat.Sets.ParentMethods attribute), 639
CartesianProductFunctor (class in sage.categories.cartesian_product), 701
CartesianProducts() (sage.categories.cartesian_product.CartesianProductsCategory method), 702
CartesianProducts() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 643
CartesianProductsCategory (class in sage.categories.cartesian_product), 702
catalan_number() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated.Irreducible.ParentMethods method), 328
Category (class in sage.categories.category), 27
category() (sage.categories.category.Category method), 47
category() (sage.categories.morphism.Morphism method), 117
Category_contains_method_by_parent_class (class in sage.categories.categorySingleton), 784
category_for() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 104
category_from_categories() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 698
category_from_category() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 698
category_from_parents() (sage.categories.covariant_functorial_construction.CovariantFunctorialConstruction method), 699
category_graph() (in module sage.categories.category), 60
category_graph() (sage.categories.category.Category method), 47
Category_ideal (class in sage.categories.category_types), 781
Category_in_ambient (class in sage.categories.category_types), 781
Category_module (class in sage.categories.category_types), 782
category_of() (sage.categories.covariant_functorial_construction.FunctorialConstructionCategory class method), 699
Category_over_base (class in sage.categories.category_types), 782
Category_over_base_ring (class in sage.categories.category_types), 782
Category_realization_of_parent (class in sage.categories.realizations), 716
category_sample() (in module sage.categories.category), 61
Category_singleton (class in sage.categories.category_singleton), 784
category_sort_key() (in module sage.categories.category_cy_helper), 788
CategoryWithAxiom (class in sage.categories.category_with_axiom), 87
CategoryWithAxiom_over_base_ring (class in sage.categories.category_with_axiom), 92
CategoryWithAxiom_singleton (class in sage.categories.category_with_axiom), 92
CategoryWithParameters (class in sage.categories.category), 57
cayley_graph() (sage.categories.semisimple.SemisimpleGroup.ParenMethods method), 621
cayley_graph_disabled() (sage.categories.finite_groups.FiniteGroups.ParenMethods method), 397
cayley_table() (sage.categories.groups.Groups.ParenMethods method), 453
cell_module() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 347

cell_module_indices() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 347


cell_poset() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 347


cells() (sage.categories.cw_complexes.CWComplexes.ParentMethods method), 276

cells() (sage.categories.examples.cw_complexes.Surface method), 730

cells() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 347

Cellular() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.SubcategoryMethods method), 363

cellular_basis() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods method), 347


center() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 352

center() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 352

center_basis() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 353

center_basis() (sage.categories.group_algebras.GroupAlgebras.ParentMethods method), 447

central_form() (sage.categories.group_algebras.GroupAlgebras.ElementMethods method), 446

central_orthogonal_idempotents() (sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis.Commutative.ParentMethods method), 388

central_orthogonal_idempotents() (sage.categories.finite_dimensional_semisimple_algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods method), 388

centralizer() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 367

centralizer_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 367

ChainComplexes (class in sage.categories.category_types), 783
character() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods method), 185
character() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 604
character_value() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ElementMethods method), 319
characteristic() (sage.categories.rings.Rings.ParentMethods method), 612
chevalley_eilenberg_complex() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 368
classical_decomposition() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 492
classical_weight() (sage.categories.loop_crystals.RegularLoopCrystals.ElementMethods method), 501

ClassicalCrystals (class in sage.categories.classical_crystals), 184
ClassicalCrystals.ElementMethods (class in sage.categories.classical_crystals), 184
ClassicalCrystals.ParentMethods (class in sage.categories.classical_crystals), 185
ClassicalCrystals.TensorProducts (class in sage.categories.classical_crystals), 187
classically_highest_weight_vectors() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 492
Coalgebras (class in sage.categories.coalgebras), 188
Coalgebras.DualObjects (class in sage.categories.coalgebras), 188
Coalgebras.ElementMethods (class in sage.categories.coalgebras), 188
Coalgebras.ParentMethods (class in sage.categories.coalgebras), 189
Coalgebras.Realizations (class in sage.categories.coalgebras), 189
Coalgebras.Realizations.ParentMethods (class in sage.categories.coalgebras), 189
Coalgebras.Super (class in sage.categories.coalgebras), 190
Coalgebras.TensorProducts (class in sage.categories.coalgebras), 191
Coalgebras.TensorProducts.ElementMethods (class in sage.categories.coalgebras), 191
Coalgebras.TensorProducts.ParentMethods (class in sage.categories.coalgebras), 191
Coalgebras.WithRealizations (class in sage.categories.coalgebras), 191
Coalgebras.WithRealizations.ParentMethods (class in sage.categories.coalgebras), 191
CoalgebrasWithBasis (class in sage.categories.coalgebras_with_basis), 192
CoalgebrasWithBasis.ElementMethods (class in sage.categories.coalgebras_with_basis), 192
CoalgebrasWithBasis.ParentMethods (class in sage.categories.coalgebras_with_basis), 193
codegrees() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 324
codegrees() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 338
codomain (sage.categories.map.Map attribute), 105
codomain (sage.categories.action.Action method), 148
codomain (sage.categories.action.InverseAction method), 149
codomain (sage.categories.action.PrecomposedAction method), 150
codomain (sage.categories.functor.Functor method), 98
codomain (sage.categories.homset.Homset method), 113
codomain (sage.categories.poor_man_map.PoorManMap method), 790
coefficient() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 539
coefficients() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 540
cohomology() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 369
common_base() (sage.categories.pushout.ConstructionFunctor method), 126
common_base() (sage.categories.pushout.MultivariateConstructionFunctor method), 133
Commutative (sage.categories.algebras.Algebras attribute), 170
Commutative (sage.categories.division_rings.DivisionRings attribute), 281
Commutative (sage.categories.domains.Domains attribute), 282
Commutative (sage.categories.rings.Rings attribute), 609
Commutative() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 87
Commutative() (sage.categories.magma.MagmaSubcategoryMethods method), 509
Commutative_extra_super_categories() (sage.categories.l_trivial_semigroups.LTrivialSemigroups method), 501
Commutative_extra_super_categories() (sage.categories.r_trivial_semigroups.RTrivialSemigroups method), 617
CommutativeAdditiveGroups (class in sage.categories.commutative_additive_groups), 194
CommutativeAdditiveGroups.Algebras (class in sage.categories.commutative_additive_groups), 194
CommutativeAdditiveGroups.CartesianProducts (class in sage.categories.commutative_additive_groups), 195
CommutativeAdditiveGroups.CartesianProducts.ElementMethods (class in sage.categories.commutative_additive_groups), 195
CommutativeAdditiveMonoids (class in sage.categories.commutative_additive_monoids), 195
CommutativeAdditiveSemigroups (class in sage.categories.commutative_additive_semigroups), 196
CommutativeAlgebraIdeals (class in sage.categories.commutative_algebra_ideals), 196
CommutativeAlgebras (class in sage.categories.commutative_algebras), 197
CommutativeRingIdeals (class in sage.categories.commutative_ring_ideals), 197
CommutativeRings (class in sage.categories.commutative_rings), 197
CommutativeRings.CartesianProducts (class in sage.categories.commutative_rings), 197
CommutativeRings.ElementMethods (class in sage.categories.commutative_rings), 198
CommutativeRings.Finite (class in sage.categories.commutative_rings), 198
CommutativeRings.Finite.ParentMethods (class in sage.categories.commutative_rings), 198
CommutativeRings.ParentMethods (class in sage.categories.commutative_rings), 200
commutes() (sage.categories.pushout.CompletionFunctor method), 123
commutes() (sage.categories.pushout.ConstructionFunctor method), 127
Compact() (sage.categories.topological_spaces.TopologicalSpaces.SubcategoryMethods method), 674
Compact_extra_super_categories() (sage.categories.cw_complexes.CWComplexes method), 275
Complete() (sage.categories.metric_spaces.MetricSpaces.SubcategoryMethods method), 526
CompleteDiscreteValuationFields (class in sage.categories.complete_discrete_valuation), 200
CompleteDiscreteValuationFields.ElementMethods (class in sage.categories.complete_discrete_valuation), 200
CompleteDiscreteValuationRings (class in sage.categories.complete_discrete_valuation), 201
CompleteDiscreteValuationRings.ElementMethods (class in sage.categories.complete_discrete_valuation), 201
CompletionFunctor (class in sage.categories.pushout), 123
Complex() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 523
ComplexManifolds (class in sage.categories.manifolds), 521
ComplexReflectionGroups (class in sage.categories.complex_reflection_groups), 203
ComplexReflectionGroups.ParentMethods (class in sage.categories.complex_reflection_groups), 203
ComplexReflectionOrGeneralizedCoxeterGroups (class in sage.categories.complex_reflection_or_generalized_coxeter_groups), 204
ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods (class in sage.categories.complex_reflection_or_generalized_coxeter_groups), 206
ComplexReflectionOrGeneralizedCoxeterGroups.Irreducible (class in sage.categories.complex_reflection_or_generalized_coxeter_groups), 210
ComplexReflectionOrGeneralizedCoxeterGroups.Irreducible.ParentMethods (class in sage.categories.complex_reflection_or_generalized_coxeter_groups), 210
ComplexReflectionOrGeneralizedCoxeterGroups.SubcategoryMethods (class in sage.categories.complex_reflection_or_generalized_coxeter_groups), 211
CompositeConstructionFunctor (class in sage.categories.pushout), 124
conjugacy_class() (sage.categories.groups.Groups.ElementMethods method), 452
conjugacy_class() (sage.categories.groups.Groups.ParentMethods method), 456
conjugacy_classes() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), 397
conjugacy_classes_representatives() (sage.categories.finite_groups.FiniteGroups.ParentMethods method), 397
Connected() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 87
Connected() (sage.categories.cw_complexes.CWComplexes.SubcategoryMethods method), 276
Connected() (sage.categories.filtered_modules.FilteredModules.SubcategoryMethods method), 303
Connected() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 523
connected_components() (sage.categories.crystals.Crystals.ParentMethods method), 264
connected_components() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 604
connected_components_generators() (sage.categories.crystals.Crystals.ParentMethods method), 265
connected_components_generators() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 462
connected_components_generators() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 605
construction() (sage.categories.sets_cat.Sets.ParentMethods method), 641
construction_tower() (in module sage.categories.pushout), 138
ConstructionFunctor (class in sage.categories.pushout), 125
convolution_product() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ElementMethods method), 179
convolution_product() (sage.categories.bialgebras_with_basis.BialgebrasWithBasis.ParentMethods method), 181
coproduct() (sage.categories.coalgebras.Coalgebras.ElementMethods method), 188
coproduct() (sage.categories.coalgebras.Coalgebras.ElementMethods method), 189
coproduct() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ElementMethods method), 192
coproduct() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ParentMethods method), 193
coproduct_by_coercion() (sage.categories.coalgebras.coalgebras.Realizations.ParentMethods method), 189
coproduct_iterated() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ElementMethods method), 192
coproduct_on_basis() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ElementMethods method), 193
coproduct_on_basis() (sage.categories.coalgebras_with_basis.CoalgebrasWithBasis.ParentMethods method), 193
coproduct_on_basis() (sage.categories.group_with_basis.MyGroupAlgebra method), 751
coproduct_on_basis() (sage.categories.group_algebras.GroupAlgebras.ElementMethods method), 448
counit() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 230
counit() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 448
counit() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 188
counit() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 189
counit() (sage.categories.coalgebras.ElementMethods method), 189
counit() (sage.categories.coalgebras.ElementMethods method), 191
counit() (sage.categories.coalgebras_with_basis.ElementMethods method), 191
counit() (sage.categories.coalgebras_with_basis.ElementMethods method), 193
counit() (sage.categories.coalgebras.ElementMethods method), 190
counit() (sage.categories.coalgebras.ElementMethods method), 194
counit() (sage.categories.coalgebras.ElementMethods method), 439
counit() (sage.categories.coalgebras.ElementMethods method), 448
CovariantConstructionCategory (class in sage.categories.covariant_functorial_construction), 695
CovariantFunctorialConstruction (class in sage.categories.covariant_functorial_construction), 697
cover_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 231
covered_reflections_subgroup() (sage.categories.finite_coxeter_groups.ElementMethods method), 334
coxeter_diagram() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 246
coxeter_element() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 246
coxeter_element() (sage.categories.finite_complex_reflection_groups.ElementMethods method), 331
coxeter_element() (sage.categories.finite_complex_reflection_groups.ElementMethods method), 332
coxeter_knuth_graph() (sage.categories.finite_coxeter_groups.ElementMethods method), 334
coxeter_knuth_neighbor() (sage.categories.finite_coxeter_groups.ElementMethods method), 335
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>coxeter_matrix()</td>
<td>sage.categories.coxeter_groups.CoxeterGroups.ParentMethods</td>
<td>247</td>
</tr>
<tr>
<td>coxeter_matrix()</td>
<td>sage.categories.examples.finite_coxeter_groups.DihedralGroup</td>
<td>732</td>
</tr>
<tr>
<td>coxeter_matrix()</td>
<td>sage.categories.weyl_groups.WeylGroups.ParentMethods</td>
<td>689</td>
</tr>
<tr>
<td>coxeter_number()</td>
<td>sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods</td>
<td>321</td>
</tr>
<tr>
<td>coxeter_sorting_word()</td>
<td>sage.categories.coxeter_groups.CoxeterGroups.ElementMethods</td>
<td>231</td>
</tr>
<tr>
<td>coxeter_type()</td>
<td>sage.categories.coxeter_groups.CoxeterGroups.ParentMethods</td>
<td>247</td>
</tr>
<tr>
<td>CoxeterGroupAlgebras</td>
<td>class in sage.categories.coxeter_group_algebras</td>
<td>221</td>
</tr>
<tr>
<td>CoxeterGroupAlgebras.ParentMethods</td>
<td>class in sage.categories.coxeter_group_algebras</td>
<td>221</td>
</tr>
<tr>
<td>CoxeterGroups</td>
<td>class in sage.categories.coxeter_groups</td>
<td>224</td>
</tr>
<tr>
<td>CoxeterGroups.ElementMethods</td>
<td>class in sage.categories.coxeter_groups</td>
<td>225</td>
</tr>
<tr>
<td>CoxeterGroups.ParentMethods</td>
<td>class in sage.categories.coxeter_groups</td>
<td>243</td>
</tr>
<tr>
<td>crystal_morphism()</td>
<td>sage.categories.crystals.Crystals.ParentMethods</td>
<td>265</td>
</tr>
<tr>
<td>CrystalHomset</td>
<td>class in sage.categories.crystals</td>
<td>252</td>
</tr>
<tr>
<td>CrystalMorphism</td>
<td>class in sage.categories.crystals</td>
<td>254</td>
</tr>
<tr>
<td>CrystalMorphismByGenerators</td>
<td>class in sage.categories.crystals</td>
<td>256</td>
</tr>
<tr>
<td>Crystals</td>
<td>class in sage.categories.crystals</td>
<td>257</td>
</tr>
<tr>
<td>Crystals.ElementMethods</td>
<td>class in sage.categories.crystals</td>
<td>258</td>
</tr>
<tr>
<td>Crystals.MorphismMethods</td>
<td>class in sage.categories.crystals</td>
<td>263</td>
</tr>
<tr>
<td>Crystals.ParentMethods</td>
<td>class in sage.categories.crystals</td>
<td>264</td>
</tr>
<tr>
<td>Crystals.SubcategoryMethods</td>
<td>class in sage.categories.crystals</td>
<td>274</td>
</tr>
<tr>
<td>Crystals.TensorProducts</td>
<td>class in sage.categories.crystals</td>
<td>274</td>
</tr>
<tr>
<td>CWComplexes</td>
<td>class in sage.categories.cw_complexes</td>
<td>275</td>
</tr>
<tr>
<td>CWComplexes.Connected</td>
<td>class in sage.categories.cw_complexes</td>
<td>275</td>
</tr>
<tr>
<td>CWComplexes.ElementMethods</td>
<td>class in sage.categories.cw_complexes</td>
<td>275</td>
</tr>
<tr>
<td>CWComplexes.Finite</td>
<td>class in sage.categories.cw_complexes</td>
<td>275</td>
</tr>
<tr>
<td>CWComplexes.Finite.ParentMethods</td>
<td>class in sage.categories.cw_complexes</td>
<td>276</td>
</tr>
<tr>
<td>CWComplexes.FiniteDimensioned</td>
<td>class in sage.categories.cw_complexes</td>
<td>276</td>
</tr>
<tr>
<td>CWComplexes.ParentMethods</td>
<td>class in sage.categories.cw_complexes</td>
<td>276</td>
</tr>
<tr>
<td>CWComplexes.SubcategoryMethods</td>
<td>class in sage.categories.cw_complexes</td>
<td>276</td>
</tr>
<tr>
<td>Cycle</td>
<td>class in sage.categories.examples.graphs</td>
<td>749</td>
</tr>
<tr>
<td>Cycle.Element</td>
<td>class in sage.categories.examples.graphs</td>
<td>749</td>
</tr>
<tr>
<td>cycle_index()</td>
<td>sage.categories.finite_permutation_groups.FinitePermutationGroups.ParentMethods</td>
<td>404</td>
</tr>
</tbody>
</table>

D

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.covariant Functorial Construction.CovariantConstructionCategory</td>
<td>696</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.covariant Functorial Construction.RegressiveCovariantConstructionCategory</td>
<td>700</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.graded_modules.GradedModules.Category</td>
<td>442</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.homsets.Homsets.Category</td>
<td>714</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.isomorphic_objects.IsomorphicObjects.Category</td>
<td>712</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.metric_spaces.MetricSpaces.Category</td>
<td>527</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.quotients.Quotients.Category</td>
<td>710</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.subobjects.Subobjects.Category</td>
<td>711</td>
</tr>
<tr>
<td>default_super_categories()</td>
<td>sage.categories.super_modules.Supers.Category</td>
<td>672</td>
</tr>
<tr>
<td>degree()</td>
<td>sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods</td>
<td>305</td>
</tr>
</tbody>
</table>

814 Index
degree_negation() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ElementMethods method), 443
degree_negation() (sage.categories.graded_modules_with_basis.GradedModulesWithBasis.ParentMethods method), 444
degree_on_basis() (sage.categories.examples.graded_connected_hopf_algebras_with_basis.GradedConnectedCombinatorialHopfAlgebra method), 746
degree_on_basis() (sage.categories.examples.graded_modules_with_basis.GradedPartitionModule method), 749
degree_on_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 306
degree_on_basis() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis method), 365
degrees() (sage.categories.examples.finite_coxeter_groups.DihedralGroup method), 732
degrees() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 745
degrees() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 325
degrees() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 338
demazure_character() (sage.categories.classical_crystals.ClassicalCrystals.ParentMethods method), 186
demazure_lusztig_eigenvectors() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.ParentMethods method), 221
demazure_lusztig_operator_on_basis() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.ParentMethods method), 222
demazure_lusztig_operators() (sage.categories.coxeter_group_algebras.CoxeterGroupAlgebras.ParentMethods method), 223
demazure_operator() (sage.categories.regular_crystals.RegularCrystals.ParentMethods method), 599
demazure_operator_simple() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 595
demazure_product() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 247
demazure_subcrystal() (sage.categories.regular_crystals.RegularCrystals.ParentMethods method), 600
denominator() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 583
dense_coefficient_list() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ElementMethods method), 380
dehdhar_factor_element() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 231
dehdhar_lift_down() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 232
dehdhar_lift_up() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 232
derivations_basis() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis method), 370
derivative() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 583
derived_series() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ElementMethods method), 370
derived_subalgebra() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ElementMethods method), 371
descents() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 233
differentiable() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 523
digraph() (sage.categories.crystals.Crystals.ParentMethods method), 268
digraph() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 462
digraph() (sage.categories.loop_crystals.LoopCrystals.ParentMethods method), 500
digraph() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 605
DihedralGroup (class in sage.categories.examples.finite_coxeter_groups), 731
DihedralGroup.Element (class in sage.categories.examples.finite_coxeter_groups), 731
dimension() (sage.categories.cw_complexes.CWComplexes.ElementMethods method), 275
dimension() (sage.categories.cw_complexes.CWComplexes.Finite.ParentMethods method), 276
dimension() (sage.categories.cw_complexes.CWComplexes.ParentMethods method), 276
dimension() (sage.categories.examples.cw_complexes.Surface.Element method), 729
dimension() (sage.categories.examples.graphs.Cycle.Element method), 749
dimension() (sage.categories.examples.manifolds.Plane method), 754
dimension() (sage.categories.graphs.Graphs.ParentMethods method), 444
dimension() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 487
dimension() (sage.categories.manifolds.Manifolds.ParentMethods method), 522
dimension() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 550
direct_sum() (sage.categories.crystals.Crystals.ParentMethods method), 269
directed_subset() (sage.categories.posets.Posets.ParentMethods method), 574
DiscreteValuationFields (class in sage.categories.discrete_valuation), 277
DiscreteValuationFields.ElementMethods (class in sage.categories.discrete_valuation), 277
DiscreteValuationFields.ParentMethods (class in sage.categories.discrete_valuation), 277
DiscreteValuationRings (class in sage.categories.discrete_valuation), 278
DiscreteValuationRings.ElementMethods (class in sage.categories.discrete_valuation), 278
dist() (sage.categories.metric_spaces.MetricSpaces.ElementMethods method), 525
dist() (sage.categories.metric_spaces.MetricSpaces.ParentMethods method), 526
distinguished_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 211
distinguished_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 211
Distributive (sage.categories.magasms_and_additive_magmas.MagasmsAndAdditiveMagmas attribute), 516
Distributive() (sage.categories.magasms_and_additive_magmas.MagasmsAndAdditiveMagmas.SubcategoryMethods method), 509
DistributiveMagasmsAndAdditiveMagmas (class in sage.categories.distributive_magmas_and_additive_magmas), 279
DistributiveMagasmsAndAdditiveMagmas.DistributiveAssociative (class in sage.categories.distributive_magmas_and_additive_magmas), 280
DistributiveMagasmsAndAdditiveMagmas.DistributiveAssociative.DistributiveCommutative (class in sage.categories.distributive_magmas_and_additive_magmas), 280
DistributiveMagasmsAndAdditiveMagmas.DistributiveAssociative.DistributiveCommutative.DistributiveUnital (class in sage.categories.distributive_magmas_and_additive_magmas), 280
DistributiveMagasmsAndAdditiveMagmas.CartesianProducts (class in sage.categories.distributive_magmas_and_additive_magmas), 280
DistributiveMagasmsAndAdditiveMagmas.Plus (class in sage.categories.distributive_magmas_and_additive_magmas), 280
Division (sage.categories.rings.Rings.attribute), 609
Division() (sage.categories.rings.Rings.SubcategoryMethods method), 616
DivisionRings (class in sage.categories.division_rings), 280
extra_super_categories() (sage.categories.semigroups.Semigroups.CartesianProducts method), 621
extra_super_categories() (sage.categories.sets_cat.Sets.Algebras method), 634
extra_super_categories() (sage.categories.sets_cat.Sets.CartesianProducts method), 637
extra_super_categories() (sage.categories.sets_cat.Sets.WithRealizations method), 656
extra_super_categories() (sage.categories.super_algebras.SuperAlgebras method), 668
extra_super_categories() (sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis method), 669
extra_super_categories() (sage.categories.vector_spaces.VectorSpaces.CartesianProducts method), 680
extra_super_categories() (sage.categories.vector_spaces.VectorSpaces.DualObjects method), 680
extra_super_categories() (sage.categories.vector_spaces.VectorSpaces.TensorProducts method), 681
extra_super_categories() (sage.categories.vector_spaces.VectorSpaces.WithBasis.TensorProducts method), 681

F

f() (sage.categories.crystals.Crystals.ElementMethods method), 259
f() (sage.categories.examples.crystals.HighestWeightCrystalOfTypeA.Element method), 728
f() (sage.categories.examples.crystals.NaiveCrystal.Element method), 729
f() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.WithBasis.ElementMethods method), 591
f_on_basis() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.WithBasis.TensorProducts.ParentMethods method), 593
f_string() (sage.categories.crystals.Crystals.ElementMethods method), 260
Facade (sage.categories.sets_cat.Sets attribute), 638
Facade() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 644
facade_for() (sage.categories.facade_sets.FacadeSets.ParentMethods method), 691
FacadeSets (class in sage.categories.facade_sets), 691
FacadeSets.ParentMethods (class in sage.categories.facade_sets), 691
faces() (sage.categories.simplicial_complexes.SimplicialComplexes.ParentMethods method), 661
facets() (sage.categories.simplicial_complexes.SimplicialComplexes.ParentMethods method), 661
factor() (sage.categories.fields.Fields.ElementMethods method), 291
factor() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 584
Fields (class in sage.categories.fields), 290
Fields.ElementMethods (class in sage.categories.fields), 290
Fields.ParentMethods (class in sage.categories.fields), 294
Filtered (sage.categories.algebras.Algebras attribute), 171
Filtered (sage.categories.algebras_with_basis.AlgebrasWithBasis attribute), 175
Filtered (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis attribute), 469
Filtered (sage.categories.modules.Modules attribute), 530
Filtered (sage.categories.modules_with_basis.ModulesWithBasis attribute), 549
Filtered() (sage.categories.modules.Modules.SubcategoryMethods method), 533
FilteredAlgebras (class in sage.categories.filtered_algebras), 295
FilteredAlgebras.ParentMethods (class in sage.categories.filtered_algebras), 295
FilteredAlgebrasWithBasis (class in sage.categories.filtered_algebras_with_basis), 295
FilteredAlgebrasWithBasis.ElementMethods (class in sage.categories.filtered_algebras_with_basis), 296
FilteredAlgebrasWithBasis.ParentMethods (class in sage.categories.filtered_algebras_with_basis), 296
FilteredModules (class in sage.categories.filtered_modules), 303
FilteredModules.Connected (class in sage.categories.filtered_modules), 303
FilteredModules.SubcategoryMethods (class in sage.categories.filtered_modules), 303
FilteredModulesCategory (class in sage.categories.filtered_modules), 304
FilteredModulesWithBasis (class in sage.categories.filtered_modules), 304
FilteredModulesWithBasis.ElementMethods (class in sage.categories.filtered_modules_with_basis), 305
FilteredModulesWithBasis.ParentMethods (class in sage.categories.filtered_modules_with_basis), 311
Finite (sage.categories.complex_reflection_groups.ComplexReflectionGroups attribute), 203
Finite (sage.categories.coxeter_groups.CoxeterGroups attribute), 243
Finite (sage.categories.crystals.Crystals attribute), 263
Finite (sage.categories.enumerated_sets.EnumeratedSets attribute), 283
Finite (sage.categories.fields.Fields attribute), 294
Finite (sage.categories.groups.Groups attribute), 453
Finite (sage.categories.lattice_posets.LatticePosets attribute), 475
Finite (sage.categories.monoids.Monoids attribute), 567
Finite (sage.categories.permutation_groups.PermutationGroups attribute), 572
Finite (sage.categories.posets.Posets attribute), 574
Finite (sage.categories.semigroups.Semigroups attribute), 621
Finite (sage.categories.sets_cat.Sets attribute), 638
Finite (sage.categories.weyl_groups.WeylGroups attribute), 689
Finite() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 646
Finite_extra_super_categories() (sage.categories.division_rings.DivisionRings method), 281
Finite_extra_super_categories() (sage.categories.h_trivial_semigroups.HTrivialSemigroups method), 471
FiniteComplexReflectionGroups (class in sage.categories.finite_complex_reflection_groups), 318
FiniteComplexReflectionGroups.ElementMethods (class in sage.categories.finite_complex_reflection_groups), 319
FiniteComplexReflectionGroups.Irreducible (class in sage.categories.finite_complex_reflection_groups), 320
FiniteComplexReflectionGroups.Irreducible.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 320
FiniteComplexReflectionGroups.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 323
FiniteComplexReflectionGroups.SubcategoryMethods (class in sage.categories.finite_complex_reflection_groups), 327
FiniteComplexReflectionGroups.WellGenerated (class in sage.categories.finite_complex_reflection_groups), 328
FiniteComplexReflectionGroups.WellGenerated.Irreducible (class in sage.categories.finite_complex_reflection_groups), 328
FiniteComplexReflectionGroups.WellGenerated.Irreducible.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 328
FiniteComplexReflectionGroups.WellGenerated.ParentMethods (class in sage.categories.finite_complex_reflection_groups), 331
FiniteCoxeterGroups (class in sage.categories.finite_coxeter_groups), 333
FiniteCoxeterGroups.ElementMethods (class in sage.categories.finite_coxeter_groups), 334
FiniteCoxeterGroups.ParentMethods (class in sage.categories.finite_coxeter_groups), 336
FiniteCrystals (class in sage.categories.finite_crystals), 345
FiniteCrystals.TensorProducts (class in sage.categories.finite_crystals), 345
FiniteDimensional (sage.categories.algebras_with_basis.AlgebrasWithBasis attribute), 175
FiniteDimensional (sage.categories.graded_lie_algebras_with_basis.GradedLieAlgebrasWithBasis attribute), 441
FiniteDimensional (sage.categories.hopf_algebras_with_basis.HopfAlgebrasWithBasis attribute), 470
FiniteDimensional (sage.categories.modules_with_basis.ModulesWithBasis attribute), 549
FiniteDimensional() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 87
FiniteDimensional() (sage.categories.cw_complexes.CWComplexes.SubcategoryMethods method), 277
FiniteDimensional() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 524
<table>
<thead>
<tr>
<th>Class / Attribute</th>
<th>Module / Class</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FiniteDimensional()</td>
<td>sage.categories.modules.Modules.SubcategoryMethods</td>
<td>533</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>346</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.Cellular</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>346</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.Cellular.ElementMethods</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>346</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.Cellular.ParentMethods</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>347</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.Cellular.TensorProducts</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>348</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.ElementMethods</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>349</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.ParentMethods</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>351</td>
</tr>
<tr>
<td>FiniteDimensionalAlgebrasWithBasis.SubcategoryMethods</td>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>363</td>
</tr>
<tr>
<td>FiniteDimensionalBialgebrasWithBasis()</td>
<td>sage.categories.finite_dimensional_bialgebras_with_basis</td>
<td>363</td>
</tr>
<tr>
<td>FiniteDimensionalCoalgebrasWithBasis()</td>
<td>sage.categories.finite_dimensional_coalgebras_with_basis</td>
<td>363</td>
</tr>
<tr>
<td>FiniteDimensionalGradedLieAlgebrasWithBasis</td>
<td>sage.categories.finite_dimensional_graded_lie_algebras_with_basis</td>
<td>364</td>
</tr>
<tr>
<td>FiniteDimensionalGradedLieAlgebrasWithBasis.ParentMethods</td>
<td>sage.categories.finite_dimensional_graded_lie_algebras_with_basis</td>
<td>364</td>
</tr>
<tr>
<td>FiniteDimensionalGradedLieAlgebrasWithBasis.Stratified</td>
<td>sage.categories.finite_dimensional_graded_lie_algebras_with_basis</td>
<td>365</td>
</tr>
<tr>
<td>FiniteDimensionalHopfAlgebrasWithBasis</td>
<td>sage.categories.finite_dimensional_hopf_algebras_with_basis</td>
<td>365</td>
</tr>
<tr>
<td>FiniteDimensionalHopfAlgebrasWithBasis.ElementMethods</td>
<td>sage.categories.finite_dimensional_hopf_algebras_with_basis</td>
<td>365</td>
</tr>
<tr>
<td>FiniteDimensionalHopfAlgebrasWithBasis.ParentMethods</td>
<td>sage.categories.finite_dimensional_hopf_algebras_with_basis</td>
<td>366</td>
</tr>
<tr>
<td>FiniteDimensionalLieAlgebrasWithBasis</td>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis</td>
<td>366</td>
</tr>
<tr>
<td>FiniteDimensionalLieAlgebrasWithBasis.ElementMethods</td>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis</td>
<td>366</td>
</tr>
<tr>
<td>FiniteDimensionalLieAlgebrasWithBasis.ParentMethods</td>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis</td>
<td>367</td>
</tr>
<tr>
<td>FiniteDimensionalLieAlgebrasWithBasis.Subobjects</td>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis</td>
<td>379</td>
</tr>
<tr>
<td>FiniteDimensionalModulesWithBasis</td>
<td>sage.categories.finite_dimensional_modules_with_basis</td>
<td>380</td>
</tr>
<tr>
<td>FiniteDimensionalModulesWithBasis.ElementMethods</td>
<td>sage.categories.finite_dimensional_modules_with_basis</td>
<td>380</td>
</tr>
<tr>
<td>FiniteDimensionalModulesWithBasis.MorphismMethods</td>
<td>sage.categories.finite_dimensional_modules_with_basis</td>
<td>382</td>
</tr>
<tr>
<td>FiniteDimensionalNilpotentLieAlgebrasWithBasis</td>
<td>sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis</td>
<td>386</td>
</tr>
</tbody>
</table>
graded_algebra() (sage.categories.graded_algebras.GradedAlgebras.ParentMethods method), 437
graded_algebra() (sage.categories.graded_algebras_with_basis.GradedAlgebrasWithBasis.ParentMethods method), 437
graded_algebra() (sage.categories.super_algebras.SuperAlgebras.ParentMethods method), 668
graded_algebra() (sage.categories.super_algebras_with_basis.SuperAlgebrasWithBasis.ParentMethods method), 669
graded_component() (sage.categories.examples.sets_with_grading.NonNegativeIntegers method), 773
graded_component() (sage.categories.sets_with_grading.SetsWithGrading.ParentMethods method), 658
GradedAlgebras (class in sage.categories.graded_algebras), 436
GradedAlgebras.ElementMethods (class in sage.categories.graded_algebras), 436
GradedAlgebras.ParentMethods (class in sage.categories.graded_algebras), 436
GradedAlgebrasWithBasis (class in sage.categories.graded_algebras_with_basis), 437
GradedAlgebrasWithBasis.ElementMethods (class in sage.categories.graded_algebras_with_basis), 437
GradedAlgebrasWithBasis.ParentMethods (class in sage.categories.graded_algebras_with_basis), 437
GradedBialgebras() (in module sage.categories.graded_bialgebras), 437
GradedBialgebrasWithBasis() (in module sage.categories.graded_bialgebras_with_basis), 438
GradedCoalgebras() (in module sage.categories.graded_coalgebras), 438
GradedCoalgebrasWithBasis() (in module sage.categories.graded_coalgebras_with_basis), 438
GradedConnectedCombinatorialHopfAlgebraWithPrimitiveGenerator (class in sage.categories.examples.graded_connected_hopf_algebras_with_basis), 745
GradedHopfAlgebras() (in module sage.categories.graded_hopf_algebras), 438
GradedHopfAlgebrasWithBasis (class in sage.categories.graded_hopf_algebras_with_basis), 439
GradedHopfAlgebrasWithBasis.Connected (class in sage.categories.graded_hopf_algebras_with_basis), 439
GradedHopfAlgebrasWithBasis.Connected.ElementMethods (class in sage.categories.graded_hopf_algebras_with_basis), 439
GradedHopfAlgebrasWithBasis.Connected.ParentMethods (class in sage.categories.graded_hopf_algebras_with_basis), 439
GradedHopfAlgebrasWithBasis.ElementMethods (class in sage.categories.graded_hopf_algebras_with_basis), 440
GradedHopfAlgebrasWithBasis.ParentMethods (class in sage.categories.graded_hopf_algebras_with_basis), 440
GradedLieAlgebras (class in sage.categories.graded_lie_algebras), 440
GradedLieAlgebras.Stratified (class in sage.categories.graded_lie_algebras), 440
GradedLieAlgebras.Stratified.FiniteDimensional (class in sage.categories.graded_lie_algebras), 440
GradedLieAlgebras.SubcategoryMethods (class in sage.categories.graded_lie_algebras), 441
GradedLieAlgebrasWithBasis (class in sage.categories.graded_lie_algebras_with_basis), 441
GradedModules (class in sage.categories.graded_modules), 441
GradedModules.ElementMethods (class in sage.categories.graded_modules), 442
GradedModules.ParentMethods (class in sage.categories.graded_modules), 442
GradedModules.Category (class in sage.categories.graded_modules), 442
GradedModulesWithBasis (class in sage.categories.graded_modules_with_basis), 443
GradedModulesWithBasis.ElementMethods (class in sage.categories.graded_modules_with_basis), 443
GradedModulesWithBasis.ParentMethods (class in sage.categories.graded_modules_with_basis), 443
GradedPartitionModule (class in sage.categories.graded_modules_with_basis), 444
grading() (sage.categories.examples.sets_with_grading.NonNegativeIntegers method), 773
grading() (sage.categories.sets_with_grading.SetsWithGrading.ParentMethods method), 658
grading_set() (sage.categories.sets_with_grading.SetsWithGrading.ParentMethods method), 659
Graphs (class in sage.categories.graphs), 444
Graphs.ParentMethods (class in sage.categories.graphs), 444
grossmannian_elements() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 248
group() (sage.categories.additive_groups.AdditiveGroups.Algebras.ParentMethods method), 150
group() (sage.categories.algebra_functor.GroupAlgebraFunctor method), 710
group() (sage.categories.group_algebras.GroupAlgebras.ParentMethods method), 449

GroupAlgebraFunctor (class in sage.categories.algebra_functor), 709

GroupAlgebras (class in sage.categories.group_algebras), 446

GroupAlgebras.ElementMethods (class in sage.categories.group_algebras), 446

GroupAlgebras.ParentMethods (class in sage.categories.group_algebras), 447

Groupoid (class in sage.categories.groupoid), 450

Groups (class in sage.categories.groups), 450

Groups.CartesianProducts (class in sage.categories.groups), 451

Groups.CartesianProducts.ElementMethods (class in sage.categories.groups), 451

Groups.CartesianProducts.ParentMethods (class in sage.categories.groups), 451

Groups.Commutative (class in sage.categories.groups), 452

Groups.ElementMethods (class in sage.categories.groups), 452

Groups.ParentMethods (class in sage.categories.groups), 453

Groups.Topological (class in sage.categories.groups), 458

GSets (class in sage.categories.g_sets), 434

gt() (sage.categories.posets.Posets.ParentMethods method), 575

H

has_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 234

has_full_support() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 234

has_left_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 234

has_right_descent() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 235

has_right_descent() (sage.categories.examples.finite_coxeter_groups.DihedralGroup.Element method), 731

has_right_descent() (sage.categories.examples.finite_weyl_groups.SymmetricGroup.Element method), 744

HeckeModules (class in sage.categories.hecke_modules), 458

HeckeModules.Homsets (class in sage.categories.hecke_modules), 459

HeckeModules.Homsets.ParentMethods (class in sage.categories.hecke_modules), 459

HeckeModules.ParentMethods (class in sage.categories.hecke_modules), 459

Heighest_weight_vector() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 462

highest_weight_vectors() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 463


highest_weight_vectors() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 606

HighestWeightCrystalHomset (class in sage.categories.highest_weight_crystals), 459

HighestWeightCrystalMorphism (class in sage.categories.highest_weight_crystals), 460

HighestWeightCrystalOfTypeA (class in sage.categories.examples.crystals), 727

HighestWeightCrystalOfTypeA.Element (class in sage.categories.examples.crystals), 728

HighestWeightCrystals (class in sage.categories.highest_weight_crystals), 460

HighestWeightCrystals.ElementMethods (class in sage.categories.highest_weight_crystals), 460

HighestWeightCrystals.ParentMethods (class in sage.categories.highest_weight_crystals), 462

HighestWeightCrystals.TensorProducts (class in sage.categories.highest_weight_crystals), 465

HighestWeightCrystals.TensorProducts.ParentMethods (class in sage.categories.highest_weight_crystals), 465

hochschild_complex() (sage.categories.algebras_with_basis.AlgebrasWithBasis.ParentMethods method), 175

holomorph() (sage.categories.groups.Groups.ParentMethods method), 457

Index 829
Hom() (in module sage.categories.homset), 111
hom() (in module sage.categories.homset), 116
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 307
homogeneous_component() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 313
homogeneous_component_as_submodule() (sage.categories.finite_dimensional_graded_lie_algebras_with_basis.FiniteDimensionalGradedLieAlgebrasWithBasis.ElementMethods method), 364
homogeneous_component_basis() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 313
homogeneous_degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 308
homology() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ElementMethods method), 371
Homset (class in sage.categories.homset), 113
homset_category() (sage.categories.homset.Homset method), 114
Homsets (class in sage.categories.homsets), 712
Homsets() (sage.categories.objects.Objects.SubcategoryMethods method), 570
Homsets.Endset (class in sage.categories.homsets), 713
Homsets.Endset.ParentMethods (class in sage.categories.homsets), 713
Homsets.ParentMethods (class in sage.categories.homsets), 714
Homsets.SubcategoryMethods (class in sage.categories.homsets), 714
HomsetsCategory (class in sage.categories.homsets), 714
HomsetsOf (class in sage.categories.homsets), 716
HomsetWithBase (class in sage.categories.homset), 115
HopfAlgebras (class in sage.categories.hopf_algebras), 466
HopfAlgebras.DualCategory (class in sage.categories.hopf_algebras), 466
HopfAlgebras.DualCategory.ParentMethods (class in sage.categories.hopf_algebras), 466
HopfAlgebras.ElementMethods (class in sage.categories.hopf_algebras), 466
HopfAlgebras.Morphism (class in sage.categories.hopf_algebras), 467
HopfAlgebras.ParentMethods (class in sage.categories.hopf_algebras), 467
HopfAlgebras.Realizations (class in sage.categories.hopf_algebras), 467
HopfAlgebras.Realizations.ParentMethods (class in sage.categories.hopf_algebras), 467
HopfAlgebras.Super (class in sage.categories.hopf_algebras), 467
HopfAlgebras.TensorProducts (class in sage.categories.hopf_algebras), 467
HopfAlgebras.TensorProducts.ElementMethods (class in sage.categories.hopf_algebras), 467
HopfAlgebras.TensorProducts.ParentMethods (class in sage.categories.hopf_algebras), 467
HopfAlgebrasWithBasis (class in sage.categories.hopf_algebras_with_basis), 468
HopfAlgebrasWithBasis.ElementMethods (class in sage.categories.hopf_algebras_with_basis), 469
HopfAlgebrasWithBasis.ParentMethods (class in sage.categories.hopf_algebras_with_basis), 470
HopfAlgebrasWithBasis.TensorProducts (class in sage.categories.hopf_algebras_with_basis), 470
HopfAlgebrasWithBasis.TensorProducts.ElementMethods (class in sage.categories.hopf_algebras_with_basis), 470
HopfAlgebrasWithBasis.TensorProducts.ParentMethods (class in sage.categories.hopf_algebras_with_basis), 470
HTrivial (sage.categories.semisupideps.Semigroups attribute), 621
HTrivial() (sage.categories.semisupideps.Semigroups.SubcategoryMethods method), 627
HTrivialSemigroups (class in sage.categories.h_trivial_semigroups), 471
hyperplane_index_set() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 214
| ideal() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 738
| ideal() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 372
| ideal() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups.ParentMethods method), 432
| ideal() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 482
| ideal_monoid() (sage.categories.rings.Rings.ParentMethods method), 612
| idempotent_lift() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 353
| idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.ParentMethods method), 428
| IdempotentSemigroups (class in sage.categories.examples.semigroups_cython), 760
| IdempotentSemigroups.ElementMethods (class in sage.categories.examples.semigroups_cython), 760
| identity() (sage.categories.homset.Homset method), 114
| IdentityConstructionFunctor (class in sage.categories.pushout), 128
| IdentityFunctor() (in module sage.categories.functor), 98
| IdentityFunctor_generic (class in sage.categories.functor), 98
| IdentityMorphism (class in sage.categories.morphism), 117
| im_gens() (sage.categories.crystals.CrystalMorphismByGenerators method), 256
| image() (sage.categories.crystals.CrystalMorphismByGenerators method), 256
| image() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 380
| image_basis() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 381
| IncompleteSubquotientSemigroup (class in sage.categories.examples.semigroups), 763
| IncompleteSubquotientSemigroup.Element (class in sage.categories.examples.semigroups), 763
| index_set() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 215
| index_set() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 248
| index_set() (sage.categories.crystals.Crystals.ElementMethods method), 260
| index_set() (sage.categories.examples.finite_coxeter_groups.DihedralGroup), 732
| index_set() (sage.categories.examples.finite_weyl_groups.SymmetricGroup), 745
| index_set() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 589
| IndexedPolynomialRing (class in sage.categories.examples.lie_algebras_with_basis), 756
| indices() (sage.categories.examples.with_realizations.SubsetAlgebra method), 778
| indices_key() (sage.categories.examples.with_realizations.SubsetAlgebra method), 778
| induced_graded_map() (sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis.ParentMethods method), 297
| induced_graded_map() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 314
| Infinite (sage.categories.enumerated_sets.EnumeratedSets attribute), 283
| Infinite() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 646
| InfiniteEnumeratedSets (class in sage.categories.infinite_enum, 472
| InfiniteEnumeratedSets.ParentMethods (class in sage.categories.infinite_enum), 472
| InfinitePolynomialFunctor (class in sage.categories.pushout), 128
| inner_derivations_basis() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 372
IntegerModMonoid (class in sage.categories.examples.finite_monoids), 739
IntegerModMonoid.Element (class in sage.categories.examples.finite_monoids), 740
IntegersCompletion (class in sage.categories.examples.facade_sets), 730
IntegralDomains (class in sage.categories.integral_domains), 473
IntegralDomains.ElementMethods (class in sage.categories.integral_domains), 473
IntegralDomains.ParentMethods (class in sage.categories.integral_domains), 473
Inverse (sage.categories.monoids.Monoids attribute), 567
inverse() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 209
Inverse() (sage.categories.mmagas.Magmas.Unital.SubcategoryMethods method), 515
inverse() (sage.categories.map.Section method), 109
Inverse_extra_super_categories() (sage.categories.h_trivial_semigroups.HTrivialSemigroups method), 471
inverse_of_unit() (sage.categories.fields.Fields.ElementMethods method), 292
inverse_of_unit() (sage.categories.rings.Rings.ElementMethods method), 609
InverseAction (class in sage.categories.action), 148
inversion_arrangement() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 683
inversion_sequence() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 338
inversions() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 684
inversions_as_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 235
Irreducible() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.SubcategoryMethods method), 221
irreducible_component_index_sets() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 215
irreducible_components() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 211
irreducible_components() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 215
irreducibles_poset() (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 398
is_abelian() (sage.categories.category.Category method), 49
is_abelian() (sage.categories.category_types.AbelianCategory method), 781
is_abelian() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 373
is_abelian() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 482
is_abelian() (sage.categories.modules_with_basis.ModulesWithBasis method), 562
is_abelian() (sage.categories.vector_spaces.VectorSpaces.WithBasis method), 681
is_affine_grassmannian() (sage.categories.affine_weyl_groups.AffineWeylGroups.ElementMethods method), 167
is_antichain_of_poset() (sage.categories.posets.Posets.ParentMethods method), 575
is_Category() (in module sage.categories.category), 61
is_central() (sage.categories.monoids.Monoids.Algebras.ElementMethods method), 564
is_chain_of_poset() (sage.categories.posets.Posets.ParentMethods method), 576
is_commutative() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 354
is_commutative() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 482
is_commutative() (sage.categories.mmagas.Magmas.Commutative.ParentMethods method), 504
is_connected() (sage.categories.crystals.Crystals.ParentMethods method), 270
is_construction_defined_by_base() (sage.categories.covariant Functorial_construction.CovariantConstructionCategory method), 697
is_coxeter_element() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ElementMethods method), 335
is_coxeter_sortable() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 235
is_embedding() (sage.categories.crystals.Crystals.MorphismMethods method), 263
is_empty() (sage.categories.additive_magmas.AdditiveMagmas.AdditiveUnital.ParentMethods method), 155
is_empty() (sage.categories.enumerated_setsEnumeratedSets.ParentMethods method), 284
is_empty() (sage.categories.magas.Magas.Unital.ParentMethods method), 514
is_empty() (sage.categories.sets_cat.Sets.CartesianProducts.ParentMethods method), 636
is_empty() (sage.categories.sets_cat.Sets.Infinite.ParentMethods method), 638
is_endomorphism() (sage.categories.morphism.Morphism method), 118
is_endomorphism_set() (sage.categories.homsets.Homsets.Endset.ParentMethods method), 713
is_endomorphism_set() (sage.categories.homsets.Homsets.ParentMethods method), 714
is_Endset() (in module sage.categories.homset), 116
is_euclidean_domain() (sage.categories.euclidean_domains.EuclideanDomains.ParentMethods method), 290
is_even() (sage.categories.super_modules.SuperModules.ElementMethods method), 670
is_even_odd() (sage.categories.super_modules.SuperModules.ElementMethods method), 670
is_even_odd() (sage.categories.super_modules.SuperModules.ElementMethods method), 670
is_field() (sage.categories.fields.Fields.ParentMethods method), 294
is_field() (sage.categories.magas.Magas.Algebras.ParentMethods method), 502
is_finite() (sage.categories.finite_sets.FiniteSets.ParentMethods method), 430
is_finite() (sage.categories-modules_with_basis.ModulesWithBasis.ParentMethods method), 551
is_finite() (sage.categories.sets_cat.Sets.CartesianProducts.ParentMethods method), 636
is_finite() (sage.categories.sets_cat.Sets.Infinite.ParentMethods method), 638
is_finite() (sage.categories.supercrystals.SuperCrystals.ElementMethods method), 603
is_functor() (in module sage.categories.functor), 99
is_genuine_highest_weight() (sage.categories.regular_supercrystals.RegularSuperCrystals.ElementMethods method), 603
is_genuine_lowest_weight() (sage.categories.regular_supercrystals.RegularSuperCrystals.ElementMethods method), 603
is_grassmannian() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 236
is_highest_weight() (sage.categories.crystals.Crystals.ElementMethods method), 260
is_homogeneous() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 308
is_Homset() (in module sage.categories.homset), 116
is_ideal() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 738
is_ideal() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 373
is_idempotent() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 483
is_idempotent() (sage.categories.examples.semirings.LeftZeroSemiring.Element method), 764
is_idempotent() (sage.categories.examples.semirings.cyton.IdempotentSemirings.ElementMethods method), 760
is_idempotent() (sage.categories.magas.Magas.ElementMethods method), 505
is_identity() (sage.categories.morphism.IdentityMorphism method), 117
is_identity() (sage.categories.morphism.Morphism method), 118
is_identity_decomposition_into_orthogonal_idempotents() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ElementMethods method), 354
is_injective() (sage.categories.crystals.CrystalMorphism method), 255
is_injective() (sage.categories.map.FormalCompositeMap method), 102
is_injective() (sage.categories.morphism.IdentityMorphism method), 117
is_injective() (sage.categories.rings.Rings.MorphismMethods method), 610
is_injective() (sage.categories.sets_cat.Sets.MorphismMethods method), 639
is_integral_domain() (sage.categories.group_algebras.GroupAlgebras.ParentMethods method), 449
is_integral_domain() (sage.categories.integral_domains.IntegralDomains.ParentMethods method), 473
is_integrally_closed() (sage.categories.fields.Fields.ParentMethods method), 294
is_irreducible() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 216
is_isomorphism() (sage.categories.crystals.Crystals.MorphismMethods method), 263
is_isomorphism() (sage.categories.regular_crystals.RegularCrystals.MorphismMethods method), 599
is_lattice() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 416
is_lattice_morphism() (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 399
is_left() (sage.categories.action.Action method), 148
is_lowest_weight() (sage.categories.crystals.Crystals.ElementMethods method), 260
is_Map() (in module sage.categories.map), 109
is_Morphism() (in module sage.categories.morphism), 120
is_nilpotent() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 373
is_nilpotent() (sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods method), 386
is_nilpotent() (sage.categories.lie_algebras.LieAlgebras.Nilpotent.ParentMethods method), 479
is_odd() (sage.categories.super_modules.SuperModules.ElementMethods method), 670
is_one() (sage.categories.monoids.Monoids.ElementMethods method), 566
is_order_filter() (sage.categories.posets.Posets.ParentMethods method), 578
is_order_ideal() (sage.categories.posets.Posets.ParentMethods method), 579
is_parent_of() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 417
is_parent_of() (sage.categories.sets_cat.Sets.ParentMethods method), 642
is_parent_of() (sage.categories.simplicial_sets.SimplicialSets.ParentMethods method), 662
is_poset_isomorphism() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 417
is_poset_morphism() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 417
is_prime() (sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element method), 768
is_real() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 325
is_real() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 339
is_reducible() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 216
is_reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 210
is_ring() (sage.categories.rings.Rings.ParentMethods method), 613
is_self_dual() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 418
is_semisimple() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 373
is_simply_connected() (sage.categories.simplicial_sets.SimplicialSets.Pointed.ParentMethods method), 667
is_solvable() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 374
is_solvable() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 483
is_strict() (sage.categories.crystals.Crystals.MorphismMethods method), 264
is_subcategory() (sage.categories.category.Category method), 51
is_subcategory() (sage.categories.category.JoinCategory method), 60
is_super_homogeneous() (sage.categories.super_modules_with_basis.SuperModulesWithBasis.ElementMethods method), 673
is_surjective() (sage.categories.crystals.CrystalMorphism method), 255
is_surjective() (sage.categories.map.FormalCompositeMap method), 103
is_surjective() (sage.categories.map.Map method), 106
is_surjective() (sage.categories.morphism.IdentityMorphism method), 117
is_unique_factorization_domain() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains.PARENT_METHODS
method), 677
is_unit() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 278
is_unit() (sage.categories.fields.Fields.ElementMethods method), 292
is_unit() (sage.categories.rings.Rings.ElementMethods method), 610
is_well_generated() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.PARENT_METHODS
method), 325
is_well_generated() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WELL_GENERATED_PARENT_METHODS
method), 332
is_zero() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 540
is_zero() (sage.categories.rings.Rings.PARENT_METHODS method), 614
IsomorphicObjectOfFiniteEnumeratedSet (class in sage.categories.examples.finite_enumerated_sets), 735
IsomorphicObjects() (sage.categories.sets_cat.Sets.PARENT_METHODS method), 646
IsomorphicObjectsCategory (class in sage.categories.isomorphic_objects), 712
isotypic_projective_modules() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.PARENT_METHODS
method), 356
iterator_range() (sage.categories.enumerated_sets.EnumeratedSets.PARENT_METHODS method), 284
iterator_range() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.PARENT_METHODS method), 393
J
j_classes() (sage.categories.finite_semigroups.FiniteSemigroups.PARENT_METHODS method), 428
j_classes_of_idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.PARENT_METHODS method), 429
j_transversal_of_idempotents() (sage.categories.finite_semigroups.FiniteSemigroups.PARENT_METHODS method), 429
join() (sage.categories.category.Category static method), 52
join() (sage.categories.lattice_posets.LatticePosets.PARENT_METHODS method), 475
join_as_tuple() (in module sage.categories.category_cy_helper), 788
join_irreducibles() (sage.categories.finite_lattice_posets.FiniteLatticePosets.PARENT_METHODS method), 399
join_irreducibles_poset() (sage.categories.finite_lattice_posets.FiniteLatticePosets.PARENT_METHODS method), 400
JoinCategory (class in sage.categories.category), 59
JTrivial (sage.categories.semigroups.Semigroups attribute), 621
JTrivial() (sage.categories.magma.Magmas.SubcategoryMethods method), 511
JTrivial() (sage.categories.semigroups.Semigroups.SubcategoryMethods method), 627
JTrivialSemigroups (class in sage.categories.j_trivial_semigroups), 473
K
K() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.WithBasis.ElementMethods method), 590
K_on_basis() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.WithBasis.TensorProducts.PARENT_METHODS
method), 592
KacMoodyAlgebras (class in sage.categories.kac_moody_algebras), 474
KacMoodyAlgebras.PARENT_METHODS (class in sage.categories.kac_moody_algebras), 474
kernel() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MORPHISM_METHODS
method), 381
kernel_basis() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MORPHISM_METHODS
method), 381
killing_form() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.PARENT_METHODS
method), 374
Index

killing_form() (sage.categories.lie_algebras.LieAlgebras.ElementMethods method), 477
killing_form() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 483
killing_form_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 374
killing_matrix() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 374
KirillovReshetikhinCrystals (class in sage.categories.loop_crystals), 489
KirillovReshetikhinCrystals.ElementMethods (class in sage.categories.loop_crystals), 489
KirillovReshetikhinCrystals.ParentMethods (class in sage.categories.loop_crystals), 490
KirillovReshetikhinCrystals.TensorProducts (class in sage.categories.loop_crystals), 494
KirillovReshetikhinCrystals.TensorProducts.ElementMethods (class in sage.categories.loop_crystals), 494
KirillovReshetikhinCrystals.TensorProducts.ParentMethods (class in sage.categories.loop_crystals), 497
KroneckerQuiverPathAlgebra (class in sage.categories.examples.finite_dimensional_algebras_with_basis), 733

L
Lambda() (sage.categories.crystals.Crystals.ParentMethods method), 264
last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.CartesianProducts.ParentMethods method), 390
last() (sage.categories.finite_enumerated_sets.FiniteEnumeratedSets.ParentMethods method), 393
latex() (sage.categories.crystals.Crystals.ParentMethods method), 270
latex_file() (sage.categories.crystals.Crystals.ParentMethods method), 270
LatticePosets (class in sage.categories.lattice_posets), 474
LatticePosets.ParentMethods (class in sage.categories.lattice_posets), 475
LaurentPolynomialFunctor (class in sage.categories.pushout), 131
lcm() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 278
lcm() (sage.categories.fields.Fields.ElementMethods method), 292
lcm() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 585
le() (sage.categories.examples.posets.FiniteSetsOrderedByInclusion method), 759
le() (sage.categories.examples.posets.PositiveIntegersOrderedByDivisibilityFacade method), 760
le() (sage.categories.posets.Posets.ParentMethods method), 579
leading_coefficient() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 541
leading_item() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 541
leading_monomial() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 542
leading_support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 542
leading_term() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 542
left_base_ring() (sage.categories.bimodules.Bimodules method), 184
left_domain() (sage.categories.action.Action method), 148
left_inversions_as_reflections() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 236
left_pieri_factorizations() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 685
left_precomposition (sage.categories.action.PrecomposedAction attribute), 150
LeftModules (class in sage.categories.left_modules), 475
LeftModules.ElementMethods (class in sage.categories.left_modules), 476
LeftModules.ParentMethods (class in sage.categories.left_modules), 476
LeftRegularBand (class in sage.categories.examples.finite_semigroups), 741
LeftRegularBand.Element (class in sage.categories.examples.finite_semigroups), 743
LeftZeroSemigroup (class in sage.categories.examples.semigroups), 764
LeftZeroSemigroup (class in sage.categories.examples.semigroups_cython), 761
LeftZeroSemigroup.Element (class in sage.categories.examples.semigroups), 764
LeftZeroSemigroup.Element (class in sage.categories.examples.semigroups_cython), 762
length() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 236
length() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 543
lowest_weight_vectors() (sage.categories.regular_supercrystals.RegularSuperCrystals.ParentMethods method), 606
lt() (sage.categories.posets.Posets.ParentMethods method), 579
LTrivial (sage.categories.semigroups.Semigroups.Semigroups attribute), 621
LTrivial() (sage.categories.semigroups.Semigroups.SubcategoryMethods method), 628
LTrivialSemigroups (class in sage.categories.l_trivial_semigroups), 501
lusztig_involution() (sage.categories.classical_crystals.ClassicalCrystals.ElementMethods method), 184
lusztig_involution() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ElementMethods method), 489

M
m_cambrian_lattice() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 339
magma_generators() (sage.categories.finitely_generated_magmas.FinitelyGeneratedMagmas.ParentMethods method), 431
magma_generators() (sage.categories.semigroups.Semigroups.ParentMethods method), 623
Magmas (class in sage.categories.magmas), 502
Magmas.Algebras (class in sage.categories.magmas), 502
Magmas.Algebras.ParentMethods (class in sage.categories.magmas), 502
Magmas.CartesianProducts (class in sage.categories.magmas), 503
Magmas.CartesianProducts.ParentMethods (class in sage.categories.magmas), 503
Magmas.Commutative (class in sage.categories.magmas), 504
Magmas.Commutative.Algebras (class in sage.categories.magmas), 504
Magmas.Commutative.CartesianProducts (class in sage.categories.magmas), 504
Magmas.Commutative.ParentMethods (class in sage.categories.magmas), 504
Magmas.ElementMethods (class in sage.categories.magmas), 504
Magmas.JTrivial (class in sage.categories.magmas), 505
Magmas.ParentMethods (class in sage.categories.magmas), 505
Magmas.Realizations (class in sage.categories.magmas), 508
Magmas.Realizations.ParentMethods (class in sage.categories.magmas), 509
Magmas.SubcategoryMethods (class in sage.categories.magmas), 509
Magmas.Subquotients (class in sage.categories.magmas), 512
Magmas.Subquotients.ParentMethods (class in sage.categories.magmas), 512
Magmas.Unital (class in sage.categories.magmas), 513
Magmas.Unital.Algebras (class in sage.categories.magmas), 513
Magmas.Unital.CartesianProducts (class in sage.categories.magmas), 513
Magmas.Unital.CartesianProducts.ElementMethods (class in sage.categories.magmas), 513
Magmas.Unital.CartesianProducts.ParentMethods (class in sage.categories.magmas), 513
Magmas.Unital.ElementMethods (class in sage.categories.magmas), 513
Magmas.Unital.Inverse (class in sage.categories.magmas), 514
Magmas.Unital.Inverse.CartesianProducts (class in sage.categories.magmas), 514
Magmas.Unital.ParentMethods (class in sage.categories.magmas), 514
Magmas.Unital.Realizations (class in sage.categories.magmas), 514
Magmas.Unital.Realizations.ParentMethods (class in sage.categories.magmas), 514
Magmas.Unital.SubcategoryMethods (class in sage.categories.magmas), 515
MagmasAndAdditiveMagmas (class in sage.categories.magmas_and_additive_magmas), 515
MagmasAndAdditiveMagmas.CartesianProducts (class in sage.categories.magmas_and_additive_magmas), 516
MagmasAndAdditiveMagmas.SubcategoryMethods (class in sage.categories.magmas_and_additive_magmas), 516
MagmaticAlgebras (class in sage.categories.magmatic_algebras), 517
MagmaticAlgebras.ParentMethods (class in sage.categories.magmatic_algebras), 518
MagmaticAlgebras.WithBasis (class in sage.categories.magmatic_algebras), 518
MagmaticAlgebras.WithBasis.FiniteDimensional (class in sage.categories.magmatic_algebras), 518

838
MagmaticAlgebras.WithBasis.FiniteDimensional.ParentMethods (class in sage.categories.magmatic_algebras), 518
MagmaticAlgebras.WithBasis.ParentMethods (class in sage.categories.magmatic_algebras), 519
Manifolds (class in sage.categories.manifolds), 521
Manifolds.AlmostComplex (class in sage.categories.manifolds), 521
Manifolds.Analytic (class in sage.categories.manifolds), 521
Manifolds.Connected (class in sage.categories.manifolds), 522
Manifolds.Differentiable (class in sage.categories.manifolds), 522
Manifolds.FiniteDimensional (class in sage.categories.manifolds), 522
Manifolds.ParentMethods (class in sage.categories.manifolds), 522
Manifolds.Smooth (class in sage.categories.manifolds), 522
Manifolds.SubcategoryMethods (class in sage.categories.manifolds), 523
Map (class in sage.categories.map), 104
map() (sage.categories.enumerated_setsEnumeratedSets.ParentMethods method), 285
map_coefficients() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 543
map_item() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 544
map_support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 544
map_support_skip_none() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 544
matrix() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.MorphismMethods method), 381
MatrixAlgebras (class in sage.categories.matrix_algebras), 524
MatrixFunctor (class in sage.categories.pushout), 131
maximal_degree() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 309
maximal_vector() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 493
meet() (sage.categories.category.Category static method), 53
meet() (sage.categories.lattice_posets.LatticePosets.ParentMethods method), 475
meet_irreducibles() (sage.categories.finite_lattice_posets.FiniteLatticePosets.ParentMethods method), 400
merge() (sage.categories.pushout.AlgebraicClosureFunctor method), 120
merge() (sage.categories.pushout.AlgebraicExtensionFunctor method), 121
merge() (sage.categories.pushout.CompletionFunctor method), 124
merge() (sage.categories.pushout.ConstructionFunctor method), 127
merge() (sage.categories.pushout.InfinitePolynomialFunctor method), 130
merge() (sage.categories.pushout.LaurentPolynomialFunctor method), 131
merge() (sage.categories.pushout.MatrixFunctor method), 132
merge() (sage.categories.pushout.MultiPolynomialFunctor method), 133
merge() (sage.categories.pushout.PermutationGroupFunctor method), 134
merge() (sage.categories.pushout.PolynomialFunctor method), 135
merge() (sage.categories.pushout.QuotientFunctor method), 136
merge() (sage.categories.pushout.SubspaceFunctor method), 136
merge() (sage.categories.pushout.VectorFunctor method), 137
metapost() (sage.categories.crystals.Crystals.ParentMethods method), 270
Metric (sage.categories.sets_cat.Sets attribute), 639
metric() (sage.categories.metric_spaces.MetricSpaces.ParentMethods method), 526
Metric (sage.categories.sets_cat.Sets.SubcategoryMethods method), 648
MetricSpaces (class in sage.categories.metric_spaces), 525
MetricSpaces.Complete (class in sage.categories.metric_spaces), 525
MetricSpaces.ElementMethods (class in sage.categories.metric_spaces), 525
Monoids.Commutative (class in sage.categories.monoids), 566
Monoids.ElementMethods (class in sage.categories.monoids), 566
Monoids.ParentMethods (class in sage.categories.monoids), 567
Monoids.Subquotients (class in sage.categories.monoids), 568
Monoids.Subquotients.ParentMethods (class in sage.categories.monoids), 568
Monoids.WithRealizations (class in sage.categories.monoids), 568
Monoids.WithRealizations.ParentMethods (class in sage.categories.monoids), 568
monomial() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 556
monomial_coefficients() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra.Element method), 736
monomial_coefficients() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 545
monomial_or_zero_if_none() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 556
monomials() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 545
Morphism (class in sage.categories.morphism), 117
morphism() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 376
morphism_class() (sage.categories.category.Category method), 54
multiplication_table() (sage.categories.magmas.Magmas.ParentMethods method), 505
multiplicative_order() (sage.categories.groups.Groups.CartesianProducts.ElementMethods method), 451
MultiPolynomialFunctor (class in sage.categories.pushout), 132
MultivariateConstructionFunctor (class in sage.categories.pushout), 133
MyGroupAlgebra (class in sage.categories.examples.hopf_algebras_with_basis), 750

N
NaiveCrystal (class in sage.categories.examples.crystals), 728
NaiveCrystal.Element (class in sage.categories.examples.crystals), 728
natural_map() (sage.categories.homset.Homset method), 114
nerve() (sage.categories.finite_monoids.FiniteMonoids.ParentMethods method), 402
next() (sage.categories.enumerated_sets.EnumeratedSets.ParentMethods method), 286
next() (sage.categories.examples.infinite_enumerated_sets.NonNegativeIntegers method), 753
next() (sage.categories.examples.sets_cat.PrimeNumbers_Abstract method), 769
next() (sage.categories.examples.sets_cat.PrimeNumbers_Abstract.Element method), 768
ngens() (sage.categories.semi_groups.Semigroups.SubcategoryMethods method), 620
Nilpotent (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.Attribute), 366
Nilpotent() (sage.categories.lie_algebras.LieAlgebras.SubcategoryMethods method), 485
noncrossing_partition_lattice() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.Irreducible.ParentMethods method), 322
NonNegativeIntegers (class in sage.categories.examples.infinite_enumerated_sets), 752
NonNegativeIntegers (class in sage.categories.examples.sets_with_grading), 773
NoZeroDivisors (sage.categories.rings.Rings.Attribute), 612
NoZeroDivisors() (sage.categories.rings.Rings.SubcategoryMethods method), 616
number_of_irreducible_components() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ElementMethods method), 216
number_of_irreducible_hyperplanes() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 326
number_of_reflections() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 326
number_of_simple_reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.Periments method), 216

NumberFields (class in sage.categories.number_fields), 569
NumberFields.ElementMethods (class in sage.categories.number_fields), 569
NumberFields.ParentMethods (class in sage.categories.number_fields), 569
numerator() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 585

O

object() (sage.categories.category_types.Elements method), 783

Objects (class in sage.categories.objects), 570
Objects.ParentMethods (class in sage.categories.objects), 570

odd_component() (sage.categories.super_modules_with_basis.SuperModulesWithBasis.ElementMethods method), 673

on_basis() (sage.categories.modules_with_basis.ModulesWithBasis.MorphismMethods method), 549
on_left_matrix() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ElementMethods method), 350

one() (sage.categories.algebras_with_basis.AlgebrasWithBasis.CartesianProducts.ParentMethods method), 174
one() (sage.categories.algebras_with_basis.AlgebrasWithBasis.ParentMethods method), 175
one() (sage.categories.examples.finite_coxeter_groups.DihedralGroup method), 732
one() (sage.categories.examples.finite_dimensional_algebras_with_basis.KroneckerQuiverPathAlgebra method), 733
one() (sage.categories.examples.finite_monoids.IntegerModMonoid method), 740
one() (sage.categories.examples.finite_weyl_groups.SymmetricGroup method), 745
one() (sage.categories.examples.monoids.FreeMonoid method), 758
one() (sage.categories.examples.with_realizations.SubsetAlgebra.Bases.ParentMethods method), 776
one() (sage.categories.examples.with_realizations.SubsetAlgebra.Fundamental method), 776
one() (sage.categories.homset.Homset method), 115
one() (sage.categories.magmas.Magmas.Unital.ParentMethods method), 514
one() (sage.categories.monoids.Monoids.Subquotients.ParentMethods method), 568
one() (sage.categories.monoids.Monoids.WithRealizations.ParentMethods method), 568
one() (sage.categories.unital_algebras.UnitalAlgebras.Basis破门门门Methods method), 679
one_basis() (sage.categories.algebras_with_basis.AlgebrasWithBasis.TensorProducts.ParentMethods method), 175
one_basis() (sage.categories.examples.algebras_with_basis.FreeAlgebra method), 723
one_basis() (sage.categories.examples.algebras_with_basis.GradedConnectedCombinatorialHopfAlgebraWithBasis method), 746
one_basis() (sage.categories.examples.algebras_with_basis.MyGroupAlgebra method), 751
one_basis() (sage.categories.examples.lie_algebras_with_base.IndexedPolynomialRing method), 766
one_basis() (sage.categories.examples.with_realizations.SubsetAlgebra.Fundamental method), 777
one() (sage.categories.unital_algebras.UnitalAlgebras.Basis破门门门Methods method), 679
one_dimensional_configuration_sum() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.TensorProducts.ParentMethods method), 498
one_from_cartesian_product_of_one_basis() (sage.categories.algebras_with_basis.AlgebrasWithBasis.CartesianProducts.ParentMethods method), 174
one_from_one_basis() (sage.categories.unital_algebras.UnitalAlgebras.Basis破门门门Methods method), 679
op (sage.categories.action.Action attribute), 148
operation() (sage.categories.action.Action method), 148
or_subcategory() (sage.categories.category.Category method), 54
order_filter() (sage.categories.posets.Posets.ParentMethods method), 580
order_filter_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 418
order_ideal() (sage.categories.posets.Posets.ParentMethods method), 580
order_ideal_complement_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 419
order_ideal_generators() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 420
order_ideal_toggle() (sage.categories.posets.Posets.ParentMethods method), 580
order_ideal_toggles() (sage.categories.posets.Posets.ParentMethods method), 581
order_ideals_lattice() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 420
orthogonal_idempotents_central_mod_radical() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis method), 356
panyushev_complement() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 421
panyushev_orbit_iter() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 422
panyushev_orbits() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 423
parent() (sage.categories.map.Map method), 106
parent_class() (sage.categories.category.Category method), 55
part() (sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras.ElementMethods method), 675
partial_fraction_decomposition() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 586
PartiallyOrderedMonoids (class in sage.categories.partially_ordered_monoids), 571
PartiallyOrderedMonoids.ElementMethods (class in sage.categories.partially_ordered_monoids), 571
PartiallyOrderedMonoids.ParentMethods (class in sage.categories.partially_ordered_monoids), 572
pbw_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ParentMethods method), 487
peirce_decomposition() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis method), 358
peirce_summand() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis method), 359
permutahedron() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 340
PermutationGroupFunctor (class in sage.categories.pushout), 134
PermutationGroups (class in sage.categories.permutation_groups), 572
Phi() (sage.categories.crystals.Crystals.ElementMethods method), 258
phi() (sage.categories.crystals.Crystals.ElementMethods method), 260
phi() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 597
phi() (sage.categories.regular_supercrystals.RegularSuperCrystals.ElementMethods method), 604
phi_minus_epsilon() (sage.categories.crystals.Crystals.ElementMethods method), 261
pieri_factors() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 689
Plane (class in sage.categories.examples.manifolds), 753
plot() (sage.categories.crystals.Crystals.ParentMethods method), 271
poincare_birkhoff_witt_basis() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis method), 488
Pointed() (sage.categories.simplicial_sets.SimplicialSets.SubcategoryMethods method), 667
PointedSets (class in sage.categories.pointed_sets), 572
PolyhedralSets (class in sage.categories.polyhedra), 573
PolynomialFunctor (class in sage.categories.pushout), 134
method), 746

product_on_basis() (sage.categories.examples.hopf_algebras_with_basis.MyGroupAlgebra method), 751
product_on_basis() (sage.categories.examples.lie_algebras_with_basis.IndexedPolynomialRing method), 757
product_on_basis() (sage.categories.examples.with_realizations.SubsetAlgebra.Fundamental method), 777
product_space() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 376

projection() (sage.categories.filtered_algebras_with_basis.FilteredAlgebrasWithBasis.ParentMethods method), 302
projection() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ParentMethods method), 317
pseudo_order() (sage.categories.finite_monoids.FiniteMonoids.ElementMethods method), 401
pushforward() (sage.categories.morphism.Morphism method), 118
pushout() (in module sage.categories.pushout), 139
pushout() (sage.categories.pushout.ConstructionFunctor method), 128
pushout_lattice() (in module sage.categories.pushout), 145

Q

q() (sage.categories.quantum_group_representations.QuantumGroupRepresentations.ParentMethods method), 589
q_dimension() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ParentMethods method), 463
q_dimension() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 494
quantum_bruhat_graph() (sage.categories.weyl_groups.WeylGroups.ParentMethods method), 690
quantum_bruhat_successors() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 686
QuantumGroupRepresentations (class in sage.categories.quantum_group_representations), 589
QuantumGroupRepresentations.ParentMethods (class in sage.categories.quantum_group_representations), 589
QuantumGroupRepresentations.TensorProducts (class in sage.categories.quantum_group_representations), 589
QuantumGroupRepresentations.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 590
QuantumGroupRepresentations.WithBasis (class in sage.categories.quantum_group_representations), 590
QuantumGroupRepresentations.WithBasis.ElementMethods (class in sage.categories.quantum_group_representations), 590
QuantumGroupRepresentations.WithBasis.TensorProducts (class in sage.categories.quantum_group_representations), 591
QuantumGroupRepresentations.WithBasis.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 592
QuantumGroupRepresentations.WithBasis.TensorProducts.ParentMethods (class in sage.categories.quantum_group_representations), 592
quo() (sage.categories.rings.Rings.ParentMethods method), 614
quo_rem() (sage.categories.discrete_valuation.DiscreteValuationRings.ElementMethods method), 278
quo_rem() (sage.categories.euclidean_domains.EuclideanDomains.ElementMethods method), 289
quo_rem() (sage.categories.fields.Fields.ElementMethods method), 293
quotient() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 377
quotient() (sage.categories.rings.Rings.ParentMethods method), 615
quotient_module() (sage.categories.finite_dimensional_modules_with_basis.FiniteDimensionalModulesWithBasis.ParentMethods method), 385
quotient_ring() (sage.categories.rings.Rings.ParentMethods method), 615
QuotientFields (class in sage.categories.quotient_fields), 583
QuotientFields.ElementMethods (class in sage.categories.quotient_fields), 583
QuotientFields.ParentMethods (class in sage.categories.quotient_fields), 588
QuotientFunctor (class in sage.categories.pushout), 135

Index 845
QuotientOfLeftZeroSemigroup (class in sage.categories.examples.semigroups), 765
QuotientOfLeftZeroSemigroup.Element (class in sage.categories.examples.semigroups), 765
Quotients() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 648
QuotientsCategory (class in sage.categories.quotients), 710

R
rt() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 494
R_matrix() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 490
radical() (sage.categories.finite_dimensional.algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 360
radical() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains.ElementMethods method), 677
radical_basis() (sage.categories.finite_dimensional.algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 361
radical_basis() (sage.categories.finite_dimensional.semisimple.algebras_with_basis.FiniteDimensionalSemisimpleAlgebrasWithBasis.ParentMethods method), 389
random_element() (sage.categories.enumerated_sets.SemisimpleAlgebras.ParentMethods method), 631
random_element() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ParentMethods method), 327
realization_of() (sage.categories.realizations.Realizations.ParentMethods method), 643
Realizations() (in module sage.categories.realizations), 717
Realizations() (sage.categories.category.Category method), 39
RealizationsCategory (class in sage.categories.realizations), 718
reduced_word() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 238
reduced_word_graph() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 238
reduced_word_reverse_iterator() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 239
reduced_words() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 239
reflection() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 217
reflection_index_set() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 217
reflection_length() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 210
reflection_length() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 240
reflection_length() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.ElementMethods method), 319
reflection_to_coroot() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 686
reflection_to_root() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 687
reflections() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 217
reflections_from_w0() (sage.categories.finite_coxeter_groups.FiniteCoxeterGroups.ParentMethods method), 341
register_as_coercion() (sage.categories.morphism.Morphism method), 118
register_as_conversion() (sage.categories.morphism.Morphism method), 119
RegressiveCovariantConstructionCategory (class in sage.categories.covariant_functorial_construction), 700
RegularCrystals (class in sage.categories.regular_crystals), 595
RegularCrystals.ElementMethods (class in sage.categories.regular_crystals), 595
RegularCrystals.MorphismMethods (class in sage.categories.regular_crystals), 599
RegularCrystals.ParentMethods (class in sage.categories.regular_crystals), 599
RegularCrystals.TensorProducts (class in sage.categories.regular_crystals), 601
RegularLoopCrystals (class in sage.categories.loop_crystals), 500
RegularLoopCrystals.ElementMethods (class in sage.categories.loop_crystals), 501
RegularSuperCrystals (class in sage.categories.regular_supercrystals), 602
RegularSuperCrystals.ElementMethods (class in sage.categories.regular_supercrystals), 603
RegularSuperCrystals.ParentMethods (class in sage.categories.regular_supercrystals), 604
RegularSuperCrystals.TensorProducts (class in sage.categories.regular_supercrystals), 607
required_methods() (sage.categories.category.Category method), 55
residue_field() (sage.categories.discrete_valuation.DiscreteValuationFields.ParentMethods method), 277
retract() (sage.categories.examples.finite_enumenrated_sets.IsomorphicObjectOfFiniteEnumeratedSet method), 735
retract() (sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup method), 766
reversed() (sage.categories.homset.Homset method), 115
rhopolar_congruence() (sage.categories.finite_monoids.FiniteMonoids.ParentMethods method), 403
right_base_ring() (sage.categories.bimodules.Bimodules method), 184
right_domain() (sage.categories.action.Action method), 148
right_precomposition (sage.categories.action.PrecomposedAction attribute), 150
RightModules (class in sage.categories.right_modules), 608
RightModules.ElementMethods (class in sage.categories.right_modules), 608
RightModules.ParentMethods (class in sage.categories.right_modules), 608
ring() (sage.categories.category_types.Category_ideal method), 781
RingIdeals (class in sage.categories.ring_ideals), 608
Rings (class in sage.categories.rings), 609
Rings.ElementMethods (class in sage.categories.rings), 609
Rings.MorphismMethods (class in sage.categories.rings), 610
Rings.ParentMethods (class in sage.categories.rings), 612
Rings.SubcategoryMethods (class in sage.categories.rings), 616
Rngs (class in sage.categories.rings), 617
rowmotion() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 423
rowmotion_orbit_iter() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 424
rowmotion_orbits() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 425
rowmotion_orbits_plots() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 425
RTrivial (sage.categories.semigroups.Semigroups.SubcategoryMethods attribute), 626
RTrivial() (sage.categories.semigroups.Semigroups.SubcategoryMethods method), 628
RTrivial_extra_super_categories() (sage.categories.l_trivial_semigroups.LTrivialSemigroups method), 501
RTrivialSemigroups (class in sage.categories.r_trivial_semigroups), 617

S

s() (sage.categories.crystals.Crystals.ElementMethods method), 261
s() (sage.categories.loop_crystals.KirillovReshetikhinCrystals.ParentMethods method), 494
sage.categories.action (module), 147
sage.categories.additive_groups (module), 150
sage.categories.additive_magmas (module), 151
sage.categories.additive_monoids (module), 162
sage.categories.additive_semigroups (module), 163
sage.categories.affine_weyl_groups (module), 165
sage.categories.algebra_functor (module), 704
sage.categories.algebra_ideals (module), 168
sage.categories.algebra_modules (module), 169
sage.categories.algebras (module), 170
sage.categories.algebras_with_basis (module), 172
sage.categories.aperiodic_semigroups (module), 177
sage.categories.associative_algebras (module), 177
sage.categories.bialgebras (module), 177
sage.categories.bialgebras_with_basis (module), 178
sage.categories.bimodules (module), 183
sage.categories.cartesian_product (module), 700
sage.categories.category (module), 26
sage.categories.category_cy_helper (module), 787
sage.categories.category_singleton (module), 784
sage.categories.category_types (module), 781
sage.categories.category_with_axiom (module), 62
sage.categories.classical_crystals (module), 184
sage.categories.coalgebras (module), 188
sage.categories.coalgebras_with_basis (module), 192
sage.categories.coercion_methods (module), 789
sage.categories.commutative_additive_groups (module), 194
sage.categories.commutative_additive_monoids (module), 195
sage.categories.commutative_additive_semigroups (module), 196
sage.categories.commutative_algebra_ideals (module), 196
sage.categories.commutative_algebras (module), 197
sage.categories.commutative_ring_ideals (module), 197
sage.categories.commutative_rings (module), 197
sage.categories.complex_discrete_valuation (module), 200
sage.categories.complex_reflection_groups (module), 203
sage.categories.complex_reflection_or_generalized_coxeter_groups (module), 204
sage.categories.covariant_functorial_construction (module), 695
sage.categories.coxeter_group_algebras (module), 221
sage.categories.coxeter_groups (module), 224
sage.categories.crystals (module), 252
sage.categories.cw_complexes (module), 275
sage.categories.discrete_valuation (module), 277
sage.categories.distributive_magmas_and_additive_magmas (module), 279
sage.categories.division_rings (module), 280
sage.categories.domains (module), 281
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.categories.dual</td>
<td>703</td>
</tr>
<tr>
<td>sage.categories.enumerated_sets</td>
<td>282</td>
</tr>
<tr>
<td>sage.categories.euclidean_domains</td>
<td>288</td>
</tr>
<tr>
<td>sage.categories.enumerated_sets</td>
<td>282</td>
</tr>
<tr>
<td>sage.categories.euclidean_domains</td>
<td>288</td>
</tr>
<tr>
<td>sage.categories.examples.algebras_with_basis</td>
<td>723</td>
</tr>
<tr>
<td>sage.categories.examples.commutative_additive_monoids</td>
<td>724</td>
</tr>
<tr>
<td>sage.categories.examples.commutative_additive_semigroups</td>
<td>725</td>
</tr>
<tr>
<td>sage.categories.examples.coxeter_groups</td>
<td>727</td>
</tr>
<tr>
<td>sage.categories.examples.crystals</td>
<td>727</td>
</tr>
<tr>
<td>sage.categories.examples.cw_complexes</td>
<td>729</td>
</tr>
<tr>
<td>sage.categories.examples.facade_sets</td>
<td>730</td>
</tr>
<tr>
<td>sage.categories.examples.finite_coxeter_groups</td>
<td>731</td>
</tr>
<tr>
<td>sage.categories.examples.finite_dimensional_algebras_with_basis</td>
<td>733</td>
</tr>
<tr>
<td>sage.categories.examples.finite_dimensional_lie_algebras_with_basis</td>
<td>736</td>
</tr>
<tr>
<td>sage.categories.examples.finite_enumerated_sets</td>
<td>734</td>
</tr>
<tr>
<td>sage.categories.examples.finite_monoids</td>
<td>739</td>
</tr>
<tr>
<td>sage.categories.examples.finite_semigroups</td>
<td>741</td>
</tr>
<tr>
<td>sage.categories.examples.finite_weyl_groups</td>
<td>743</td>
</tr>
<tr>
<td>sage.categories.examples.graded_connected_hopf_algebras_with_basis</td>
<td>745</td>
</tr>
<tr>
<td>sage.categories.examples.graded_modules_with_basis</td>
<td>747</td>
</tr>
<tr>
<td>sage.categories.examples.graphs</td>
<td>749</td>
</tr>
<tr>
<td>sage.categories.examples.hopf_algebras_with_basis</td>
<td>750</td>
</tr>
<tr>
<td>sage.categories.examples.infinite_enumerated_sets</td>
<td>752</td>
</tr>
<tr>
<td>sage.categories.examples.lie_algebras</td>
<td>754</td>
</tr>
<tr>
<td>sage.categories.examples.lie_algebras_with_basis</td>
<td>755</td>
</tr>
<tr>
<td>sage.categories.examples.manifolds</td>
<td>753</td>
</tr>
<tr>
<td>sage.categories.examples.monoids</td>
<td>757</td>
</tr>
<tr>
<td>sage.categories.examples.posets</td>
<td>758</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups</td>
<td>762</td>
</tr>
<tr>
<td>sage.categories.examples.semigroups_cython</td>
<td>760</td>
</tr>
<tr>
<td>sage.categories.examples.sets_cat</td>
<td>767</td>
</tr>
<tr>
<td>sage.categories.examples.sets_with_grading</td>
<td>773</td>
</tr>
<tr>
<td>sage.categories.examples.with_realizations</td>
<td>774</td>
</tr>
<tr>
<td>sage.categories.facade_sets</td>
<td>691</td>
</tr>
<tr>
<td>sage.categories.fields</td>
<td>290</td>
</tr>
<tr>
<td>sage.categories.filtered_algebras</td>
<td>295</td>
</tr>
<tr>
<td>sage.categories.filtered_algebras_with_basis</td>
<td>295</td>
</tr>
<tr>
<td>sage.categories.filtered_modules</td>
<td>303</td>
</tr>
<tr>
<td>sage.categories.filtered_modules_with_basis</td>
<td>304</td>
</tr>
<tr>
<td>sage.categories.finite_complex_reflection_groups</td>
<td>318</td>
</tr>
<tr>
<td>sage.categories.finite_coxeter_groups</td>
<td>333</td>
</tr>
<tr>
<td>sage.categories.finite_cryystals</td>
<td>343</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_algebras_with_basis</td>
<td>345</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_bialgebras_with_basis</td>
<td>363</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_coalgebras_with_basis</td>
<td>363</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_graded_lie_algebras_with_basis</td>
<td>364</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_hopf_algebras_with_basis</td>
<td>365</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_lie_algebras_with_basis</td>
<td>366</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_modules_with_basis</td>
<td>380</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis</td>
<td>386</td>
</tr>
<tr>
<td>sage.categories.finite_dimensional_semisimple_algebras_with_basis</td>
<td>387</td>
</tr>
</tbody>
</table>
sage.categories.finite_enumerated_sets (module), 389
sage.categories.finite_fields (module), 395
sage.categories.finite_groups (module), 395
sage.categories.finite_lattice_posets (module), 398
sage.categories.finite_monoids (module), 401
sage.categories.finite_permutation_groups (module), 404
sage.categories.finite_posets (module), 406
sage.categories.finite_semigroups (module), 428
sage.categories.finite_sets (module), 429
sage.categories.finite_weyl_groups (module), 430
sage.categories.finitely_generated_magmas (module), 431
sage.categories.finitely_generated_semigroups (module), 431
sage.categories.function_fields (module), 433
sage.categories.functor (module), 95
sage.categories.g_sets (module), 434
sage.categories.gcdDomains (module), 434
sage.categories.generalized_coxeter_groups (module), 435
sage.categories.graded_algebras (module), 436
sage.categories.graded_algebras_with_basis (module), 437
sage.categories.graded_bialgebras (module), 437
sage.categories.graded_bialgebras_with_basis (module), 438
sage.categories.graded_coalgebras (module), 438
sage.categories.graded_coalgebras_with_basis (module), 438
sage.categories.graded_hopf_algebras (module), 438
sage.categories.graded_hopf_algebras_with_basis (module), 439
sage.categories.graded_lie_algebras (module), 440
sage.categories.graded_lie_algebras_with_basis (module), 441
sage.categories.graded_modules (module), 441
sage.categories.graded_modules_with_basis (module), 443
sage.categories.graphs (module), 444
sage.categories.group_algebras (module), 445
sage.categories.groupoid (module), 450
sage.categories.groups (module), 450
sage.categories.h_trivial_semigroups (module), 471
sage.categories.hecke_modules (module), 458
sage.categories.highest_weight_crystals (module), 459
sage.categories.homset (module), 109
sage.categories.homsets (module), 712
sage.categories.hopf_algebras (module), 466
sage.categories.hopf_algebras_with_basis (module), 468
sage.categories.infinite_enumerated_sets (module), 472
sage.categories.integral_domains (module), 473
sage.categories.isomorphic_objects (module), 712
sage.categories.j_trivial_semigroups (module), 473
sage.categories.kac_moody_algebras (module), 474
sage.categories.l_trivial_semigroups (module), 501
sage.categories.lattice_posets (module), 474
sage.categories.left_modules (module), 475
sage.categories.lie_algebras (module), 476
sage.categories.lie_algebras_with_basis (module), 486
sage.categories.super_algebras_with_basis (module), 668
sage.categories.super_hopf_algebras_with_basis (module), 669
sage.categories.super_modules (module), 669
sage.categories.super_modules_with_basis (module), 672
sage.categories.tensor (module), 702
sage.categories.topological_spaces (module), 674
sage.categories.triangular_kac_moody_algebras (module), 675
sage.categories.tutorial (module), 99
sage.categories.unique_factorization_domains (module), 676
sage.categories.unital_algebras (module), 678
sage.categories.vector_spaces (module), 680
sage.categories.weyl_groups (module), 682
sage.categories.with_realizations (module), 718
scaling_factors() (sage.categories.crystals.CrystalMorphism method), 255
Schemes (class in sage.categories.schemes), 618
Schemes_over_base (class in sage.categories.schemes), 618
Section (class in sage.categories.map), 108
section() (sage.categories.map.Map method), 108
section() (sage.categories.morphism.IdentityMorphism method), 117
semidirect_product() (sage.categories.groups.Groups.ParentMethods method), 457
semigroup_generators() (sage.categories.examples.finite_monoids.IntegerModMonoid method), 741
semigroup_generators() (sage.categories.examples.finite_semigroups.LeftRegularBand method), 743
semigroup_generators() (sage.categoriesEXAMPLE_free_semigroup method), 763
semigroup_generators() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups.ParentMethods method), 397
semigroup_generators() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups.ParentMethods method), 432
semigroup_generators() (sage.categories.morphisms.IdentityMorphism method), 117
Semigroups (class in sage.categories.semigroups), 619
Semigroups.Algebras (class in sage.categories.semigroups), 619
Semigroups.Algebras.ParentMethods (class in sage.categories.semigroups), 619
Semigroups.CartesianProducts (class in sage.categories.semigroups), 621
Semigroups.ElementMethods (class in sage.categories.semigroups), 621
Semigroups.ParentMethods (class in sage.categories.semigroups), 621
Semigroups.Quotients (class in sage.categories.semigroups), 626
Semigroups.Quotients.ParentMethods (class in sage.categories.semigroups), 626
Semirings (class in sage.categories.semirings), 630
Semisimple (sage.categories.algebras.Algebras attribute), 171
Semisimple() (sage.categories.algebras.Algebras.SubcategoryMethods method), 171
semisimple_quotient() (sage.categories.finite_dimensional_algebras_with_basis.FiniteDimensionalAlgebrasWithBasis.ParentMethods method), 362
SemisimpleAlgebras (class in sage.categories.semisimple_algebras), 630
SemisimpleAlgebras.FiniteDimensional (class in sage.categories.semisimple_algebras), 631
SemisimpleAlgebras.ParentMethods (class in sage.categories.semisimple_algebras), 631
set_base_point() (sage.categories.simplicial_sets.SimplicialSets.ParentMethods method), 663
SimplicialSets.Homsets.Endset (class in sage.categories.simplicial_sets), 662
SimplicialSets.ParentMethods (class in sage.categories.simplicial_sets), 662
SimplicialSets.Pointed (class in sage.categories.simplicial_sets), 663
SimplicialSets.Pointed.Finite (class in sage.categories.simplicial_sets), 663
SimplicialSets.Pointed.Finite.ParentMethods (class in sage.categories.simplicial_sets), 663
SimplicialSets.SubcategoryMethods (class in sage.categories.simplicial_sets), 667
Smooth() (sage.categories.manifolds.Manifolds.SubcategoryMethods method), 524
some_elements() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups.ParentMethods method), 220
some_elements() (sage.categories.enumerated_setsEnumeratedSets.ParentMethods method), 287
some_elements() (sage.categories.examples.semigroups.LeftZeroSemigroup method), 765
some_elements() (sage.categories.examples.semigroups.QuotientOfLeftZeroSemigroup method), 766
some_elements() (sage.categories.examples.sets_cat.PrimeNumbersAbstract method), 769
some_elements() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups.Finite.ParentMethods method), 398
some_elements() (sage.categories.finitely_generated_semigroups.FinitelyGeneratedSemigroups.Finite.ParentMethods method), 432
some_elements() (sage.categories.sets_cat.Sets.ParentMethods method), 642
special_node() (sage.categories.affine_weyl_groups.AffineWeylGroups.AffineWeylGroups.ParentMethods method), 167
squarefree_part() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains.ElementMethods method), 677
standard_coxeter_elements() (sage.categories.coxeter_groups.CoxeterGroups.ParentMethods method), 250
standard_coxeter_elements() (sage.categories.finite_complex_reflection_groups.FiniteComplexReflectionGroups.WellGenerated.ParentMethods method), 332
stanley_symmetric_function() (sage.categories.weyl_groups.WeylGroups.WeylGroups.ElementMethods method), 687
stanley_symmetric_function_as_polynomial() (sage.categories.weyl_groups.WeylGroups.ElementMethods method), 688
stembridgeDel_depth() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 597
stembridgeDelta_depth() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 597
stembridgeDelta_rise() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 598
stembridgeTriple() (sage.categories.regular_crystals.RegularCrystals.ElementMethods method), 598
step() (sage.categories.finite_dimensional_nilpotent_lie_algebras_with_basis.FiniteDimensionalNilpotentLieAlgebrasWithBasis.ParentMethods method), 387
stratified() (sage.categories.graded_lie_algebras.GradedLieAlgebras.SubcategoryMethods method), 441
string_parameters() (sage.categories.highest_weight_crystals.HighestWeightCrystals.ElementMethods method), 461
structure() (sage.categories.category.Category method), 55
structure_coefficients() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 378
subalgebra() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 739
subalgebra() (sage.categories.finite_dimensional_lie_algebras_with_basis.FiniteDimensionalLieAlgebrasWithBasis.ParentMethods method), 378
subalgebra() (sage.categories.lie_algebras.LieAlgebras.ParentMethods method), 484
subcategory_class() (sage.categories.category.Category method), 56
subcrystal() (sage.categories.crystals.Crystals.ElementMethods method), 261
subcrystal() (sage.categories.crystals.Crystals.ParentMethods method), 272
submodule() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 557
submonoid() (sage.categories.monoids.Monoids.ParentMethods method), 567
Subobjects() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 648
SubobjectsCategory (class in sage.categories.subobjects), 711
Subquotients() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 649
SubquotientsCategory (class in sage.categories.subquotients), 710
subsemigroup() (sage.categories.semigroups.Semigroups.ParentMethods method), 624
subset() (sage.categories.sets_with_grading.SetsWithGrading.ParentMethods method), 659
SubsetAlgebra (class in sage.categories.examples.with_realizations), 774
SubsetAlgebra.Bases (class in sage.categories.examples.with_realizations), 775
Super (sage.categories.algebras.Algebras attribute), 172
Super (sage.categories.algebras_with_basis.AlgebrasWithBasis attribute), 175
Super (sage.categories.bialgebras.Bialgebras attribute), 178
Super (sage.categories.coalgebras.Coalgebras attribute), 192
Super_categories() (sage.categories.category.Category method), 57
Super_categories() (sage.categories.category.JoinCategory method), 60
Super_categories() (sage.categories.category_with_axiom.CategoryWithAxiom method), 91
Super_categories() (sage.categories.commutative_algebra_ideals.CommutativeAlgebraIdeals method), 196
Super_categories() (sage.categories.commutative_ring_ideals.CommutativeRingIdeals method), 197
Super_categories() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationFields method), 201

super_categories() (sage.categories.complete_discrete_valuation.CompleteDiscreteValuationRings method), 202
super_categories() (sage.categories.complex_reflection_groups.ComplexReflectionGroups method), 204
super_categories() (sage.categories.complex_reflection_or_generalized_coxeter_groups.ComplexReflectionOrGeneralizedCoxeterGroups method), 221
super_categories() (sage.categories.covariant_functorial_construction.FunctorialConstructionCategory method), 700
super_categories() (sage.categories.coxeter_groups.CoxeterGroups method), 252
super_categories() (sage.categories.crystals.Crystals method), 274
super_categories() (sage.categories.cw_complexes.CWComplexes method), 277
super_categories() (sage.categories.discrete_valuation.DiscreteValuationFields method), 278
super_categories() (sage.categories.discrete_valuation.DiscreteValuationRings method), 279
super_categories() (sage.categories.domains.Domains method), 282
super_categories() (sage.categories.enumerated_sets.EnumeratedSets method), 288
super_categories() (sage.categories.euclidean_domains.EuclideanDomains method), 290
super_categories() (sage.categories.examples.semigroups_cython.IdempotentSemigroups method), 761
super_categories() (sage.categories.function_fields.FunctionFields method), 434
super_categories() (sage.categories.g_sets.GSets method), 434
super_categories() (sage.categories.gcd_domains.GcdDomains method), 435
super_categories() (sage.categories.generalized_coxeter_groups.GeneralizedCoxeterGroups method), 436
super_categories() (sage.categories.highest_weight_crystals.HighestWeightCrystals method), 466
super_categories() (sage.categories.homsets.Homsets method), 714
super_categories() (sage.categories.homsets.HomsetsOf method), 716
super_categories() (sage.categories.loop_crystals.KirillovReshetikhinCrystals method), 498
super_categories() (sage.categories.loop_crystals.LoopCrystals method), 500
super_categories() (sage.categories.loop_crystals.RegularLoopCrystals method), 501
super_categories() (sage.categories.magmas.Magmas method), 515
super_categories() (sage.categories.magmas_and_additive_magmas.MagmasAndAdditiveMagmas method), 517
super_categories() (sage.categories.magnificient_algebras.MagmaticAlgebras method), 520
super_categories() (sage.categories.manifolds.ComplexManifolds method), 521
super_categories() (sage.categories.manifolds.Manifolds method), 524
super_categories() (sage.categories.magmatic_algebras.MatrixAlgebras method), 524
super_categories() (sage.categories.module_ring_modules.ModuleRingModules method), 537
super_categories() (sage.categories.objects.Objects method), 571
super_categories() (sage.categories.permutation_groups.PermutationGroups method), 572
super_categories() (sage.categories.permutative_algebras.PermutativeAlgebras method), 572
super_categories() (sage.categories.pi_objects.Objects method), 573
super_categories() (sage.categories.polyhedra.Polyhedra method), 573
super_categories() (sage.categories.polyhedra.Polyhedra method), 573
super_categories() (sage.categories.posets.Posets method), 582
super_categories() (sage.categories.principal_ideal_domains.PrincipalIdealDomains method), 583
super_categories() (sage.categories.quantum_group_representations.QuantumGroupRepresentations method), 594
super_categories() (sage.categories.quotient_fields.QuotientFields method), 588
super_categories() (sage.categories.regular_crystals.RegularCrystals method), 602
super_categories() (sage.categories.regular_supercrystals.RegularSuperCrystals method), 607
super_categories() (sage.categories.right_modules.RightModules method), 608
super_categories() (sage.categories.ring_ideals.RingIdeals method), 608
super_categories() (sage.categories.schemes.Schemes method), 618
super_categories() (sage.categories.schemes.Schemes_over_base method), 618
super_categories() (sage.categories.semisimple_algebras.SemisimpleAlgebras method), 631
super_categories() (sage.categories.sets_cat.Sets method), 656
super_categories() (sage.categories.sets_with_grading.SetsWithGrading method), 659
super_categories() (sage.categories.sets_with_partial_maps.SetsWithPartialMaps method), 659
super_categories() (sage.categories.shephard_groups.ShephardGroups method), 660
super_categories() (sage.categories.simplicial_complexes.SimplicialComplexes method), 661
super_categories() (sage.categories.simplicial_sets.SimplicialSets method), 668
super_categories() (sage.categories.super_modules.Superalgebras method), 671
super_categories() (sage.categories.triangular_kac_moody_algebras.TriangularKacMoodyAlgebras method), 676
super_categories() (sage.categories.unique_factorization_domains.UniqueFactorizationDomains method), 678
super_categories() (sage.categories.vector_spaces.VectorSpaces method), 682
SuperAlgebras (class in sage.categories.super_algebras), 668
SuperAlgebras.ParentMethods (class in sage.categories.super_algebras), 668
SuperAlgebrasWithBasis (class in sage.categories.super_modules), 668
SuperAlgebrasWithBasis.ParentMethods (class in sage.categories.super_algebras_with_basis), 668
SuperHopfAlgebrasWithBasis (class in sage.categories.super_hopf_algebras_with_basis), 669
SuperModules (class in sage.categories.superalgebras), 669
SuperModules.ElementMethods (class in sage.categories.super_modules), 670
SuperModules.ParentMethods (class in sage.categories.superalgebras), 671
SuperModulesCategory (class in sage.categories.superalgebras), 671
SuperModulesWithBasis (class in sage.categories.superalgebras_with_basis), 672
SuperModulesWithBasis.ElementMethods (class in sage.categories.super_modules_with_basis), 672
SuperModulesWithBasis.ParentMethods (class in sage.categories.super_modules_with_basis), 674
support() (sage.categories.coxeter_groups.CoxeterGroups.ElementMethods method), 241
support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 545
support_of_term() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 546
supsets() (sage.categories.examples.with_realizations.SubsetAlgebra method), 778
Surface (class in sage.categories.examples.cw_complexes), 729
SymmetricGroup (class in sage.categories.examples.finite_weyl_groups), 743
SymmetricGroup.Element (class in sage.categories.examples.finite_weyl_groups), 744

tensor (in module sage.categories.tensor), 703
tensor() (sage.categories.crystals.Crystals.ElementMethods method), 262
tensor() (sage.categories.crystals.Crystals.ParentMethods method), 273
tensor() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 546
tensor() (sage.categories.modules_with_basis.ModulesWithBasis.ParentMethods method), 560
to_vector() (sage.categories.lie_algebras_with_basis.LieAlgebrasWithBasis.ElementMethods method), 486
toggling_orbit_iter() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 425
toggling_orbits() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 427
toggling_orbits_plots() (sage.categories.finite_posets.FinitePosets.ParentMethods method), 427
Topological (sage.categories.sets_cat.Sets.attribute), 653
Topological() (sage.categories.sets_cat.Sets.SubcategoryMethods method), 651
TopologicalSpaces (class in sage.categories.topological_spaces), 674
TopologicalSpaces.Complex (class in sage.categories.topological_spaces), 674
TopologicalSpaces.Connected (class in sage.categories.topological_spaces), 674
TopologicalSpaces.SubcategoryMethods (class in sage.categories.topological_spaces), 674
TopologicalSpacesCategory (class in sage.categories.topological_spaces), 674
trailing_coefficient() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 547
trailing_item() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 547
trailing_monomial() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 547
trailing_support() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 547
trailing_term() (sage.categories.modules_with_basis.ModulesWithBasis.ElementMethods method), 548
TriangularKacMoodyAlgebras (class in sage.categories.triangular_kac_moody_algebras), 675
TriangularKacMoodyAlgebras.ElementMethods (class in sage.categories.triangular_kac_moody_algebras), 675
TriangularKacMoodyAlgebras.ParentMethods (class in sage.categories.triangular_kac_moody_algebras), 675
trivial_representation() (sage.categories.semigroups.Semigroups.ParentMethods method), 625
truncate() (sage.categories.filtered_modules_with_basis.FilteredModulesWithBasis.ElementMethods method), 310
type_to_parent() (in module sage.categories.pushout), 145

U
uncamelcase() (in module sage.categories.category_with_axiom), 95
uniformizer() (sage.categories.discrete_valuation.DiscreteValuationFields.ParentMethods method), 278
UniqueFactorizationDomains (class in sage.categories.unique_factorization_domains), 676
UniqueFactorizationDomains.ElementMethods (class in sage.categories.unique_factorization_domains), 676
UniqueFactorizationDomains.ParentMethods (class in sage.categories.unique_factorization_domains), 677
Unital (sage.categories_associative_algebras.AssociativeAlgebras.attribute), 177
Unital (sage.categories.distributive_magmas_and_additive_magmas.DistributiveMagmasAndAdditiveMagmas.AdditiveAssociative. AdditiveUnital attribute), 280
Unital (sage.categories.magmatic_algebras.MagmaticAlgebras.attribute), 518
Unital (sage.categories.rings.Rings.attribute), 617
Unital (sage.categories.semigroups.Semigroups.attribute), 629
Unital() (sage.categories.category_with_axiom.Blahs.SubcategoryMethods method), 87
Unital() (sage.categories.magmas.Magmas.SubcategoryMethods method), 511
Unital_extra_super_categories() (sage.categories.category_with_axiom.Bars method), 85
UnitalAlgebras (class in sage.categories.unital_algebras), 678
UnitalAlgebras.ParentMethods (class in sage.categories.unital_algebras), 678
UnitalAlgebras_with_basis (class in sage.categories.unital_algebras), 679
universal_enveloping_algebra() (sage.categories.lie_algebras.LieAlgebras.ParameterMethods method), 485
unpickle_map() (in module sage.categories.map), 109
unrank() (sage.categories.enumerated_sets.EnumeratedSets.ParameterMethods method), 287
unrank_range() (sage.categories.enumerated_sets.EnumeratedSets.ParameterMethods method), 287
unrank_range() (sage.categories.finiteEnumeratedSets.FiniteEnumeratedSets.ParentMethods method), 394
upper_covers() (sage.categories.coxeterGroups.CoxeterGroups.ElementMethods method), 241
upper_covers() (sage.categories.posets.Posets.ParentMethods method), 582
\n\nvaluation() (sage.categories.completeDiscreteValuation.CompleteDiscreteValuationFields.ElementMethods method), 200
VectorFunctor (class in sage.categories.pushout), 137
VectorSpaces (class in sage.categories.vector_spaces), 680
VectorSpaces.CartesianProducts (class in sage.categories.vector_spaces), 680
VectorSpaces.DualObjects (class in sage.categories.vector_spaces), 680
VectorSpaces.ElementMethods (class in sage.categories.vector_spaces), 681
VectorSpaces.ParentMethods (class in sage.categories.vector_spaces), 681
VectorSpaces.TensorProducts (class in sage.categories.vector_spaces), 681
VectorSpaces.WithBasis (class in sage.categories.vector_spaces), 681
VectorSpaces.WithBasis.CartesianProducts (class in sage.categories.vector_spaces), 681
VectorSpaces.WithBasis.TensorProducts (class in sage.categories.vector_spaces), 681
verma_module() (sage.categories.triangularKacMoodyAlgebras.TriangularKacMoodyAlgebras.ParentMethods method), 676
vertices() (sage.categories.examples.graphs.Cycle method), 750
virtualization() (sage.categories.crystals.CrystalMorphism method), 256
\nW
w0() (sage.categories.finiteCoxeterGroups.FiniteCoxeterGroups.ParentMethods method), 342
weak_covers() (sage.categories.coxeterGroups.CoxeterGroups.ElementMethods method), 241
weak_lattice() (sage.categories.finiteCoxeterGroups.FiniteCoxeterGroups.ParentMethods method), 342
weak_le() (sage.categories.coxeterGroups.CoxeterGroups.ElementMethods method), 242
weak_order_ideal() (sage.categories.coxeterGroups.CoxeterGroups.ParentMethods method), 251
weak_poset() (sage.categories.finiteCoxeterGroups.FiniteCoxeterGroups.ParentMethods method), 343
weight() (sage.categories.crystals.ElementMethods method), 263
weight() (sage.categories.regularCrystals.RegularCrystals.ElementMethods method), 598
weight_lattice_realization() (sage.categories.crystals.ElementMethods method), 273
weight_lattice_realization() (sage.categories.loopCrystals.Crystals.ElementMethods method), 500
WellGenerated() (sage.categories.finiteComplexReflectionGroups.FiniteComplexReflectionGroups.SubcategoryMethods method), 327
WeylGroups (class in sage.categories.weylGroups), 682
WeylGroups.ElementMethods (class in sage.categories.weylGroups), 683
WeylGroups.ParentMethods (class in sage.categories.weylGroups), 689
WithBasis (sage.categories.algebras.Algebras.Attribute), 172
WithBasis (sage.categories.bialgebras.Bialgebras.Attribute), 178
WithBasis (sage.categories.coalgebras.Coalgebras.Attribute), 191
WithBasis (sage.categories.hopfAlgebras.HopfAlgebras.Attribute), 467
WithBasis (sage.categories.lieAlgebras.LieAlgebras.Attribute), 485
WithBasis (sage.categories.lie_algebras.LieAlgebras.FiniteDimensional attribute), 478
WithBasis (sage.categories.modules.Modules.Module attribute), 537
WithBasis (sage.categories.semisimple_algebras.SemisimpleAlgebras.FiniteDimensional attribute), 631
WithBasis() (sage.categories.modules.Modules.SubcategoryMethods method), 535
WithRealizations() (in module sage.categories.with_realizations), 718
WithRealizations() (sage.categories.category.Category method), 40
WithRealizationsCategory (class in sage.categories.with_realizations), 722
wrapped_class (sage.categories.examples.finite_monoids.IntegerModMonoid.Element attribute), 740
wrapped_class (sage.categories.examples.posets.FiniteSetsOrderedByInclusion.Element attribute), 759

x

xgcd() (sage.categories.fields.Fields.ElementMethods method), 293
xgcd() (sage.categories.quotient_fields.QuotientFields.ElementMethods method), 587

Z

zero() (sage.categories.examples.commutative_additive_monoids.FreeCommutativeAdditiveMonoid method), 725
zero() (sage.categories.examples.finite_dimensional_lie_algebras_with_basis.AbelianLieAlgebra method), 739
zero() (sage.categories.examples.lie_algebras.LieAlgebraFromAssociative method), 755