Parents and Elements

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

CHAPTER

ONE

SAGE OBJECTS

1.1 Abstract base class for Sage objects

class sage.structure.sage_object.**SageObject**

Bases: object

Base class for all (user-visible) objects in Sage

Every object that can end up being returned to the user should inherit from *SageObject*.

_ascii_art_()

Return an ASCII art representation.

To implement multi-line ASCII art output in a derived class you must override this method. Unlike _repr_(), which is sometimes used for the hash key, the output of *_ascii_art_()* may depend on settings and is allowed to change during runtime.

OUTPUT:

An AsciiArt object, see sage.typeset.ascii_art for details.

EXAMPLES:

You can use the ascii_art() function to get the ASCII art representation of any object in Sage:

```
sage: result = ascii_art(integral(exp(x+x^2)/(x+1), x))                #␣
↪needs sage.symbolic
...
sage: result                                                            #␣
↪needs sage.symbolic
  /
 |
 |    2
 |   x  + x
 | e
 | ------- dx
 |  x + 1
 |
/
```

Alternatively, you can use the %display ascii_art/simple magic to switch all output to ASCII art and back:

```
sage: # needs sage.combinat
sage: from sage.repl.interpreter import get_test_shell
sage: shell = get_test_shell()
sage: shell.run_cell('tab = StandardTableaux(3)[2]; tab')
[[1, 2], [3]]
sage: shell.run_cell('%display ascii_art')
sage: shell.run_cell('tab')
1  2
3
sage: shell.run_cell('Tableaux.options(ascii_art="table", convention="French")')
sage: shell.run_cell('tab')
+---+
| 3 |
+---+---+
| 1 | 2 |
+---+---+
sage: shell.run_cell('%display plain')
sage: shell.run_cell('Tableaux.options._reset()')
sage: shell.quit()
```

_cache_key()

Return a hashable key which identifies this objects for caching. The output must be hashable itself, or a tuple of objects which are hashable or define a `_cache_key`.

This method will only be called if the object itself is not hashable.

Some immutable objects (such as p-adic numbers) cannot implement a reasonable hash function because their == operator has been modified to return `True` for objects which might behave differently in some computations:

```
sage: # needs sage.rings.padics
sage: K.<a> = Qq(9)
sage: b = a + O(3)
sage: c = a + 3
sage: b
a + O(3)
sage: c
a + 3 + O(3^20)
sage: b == c
True
sage: b == a
True
sage: c == a
False
```

If such objects defined a non-trivial hash function, this would break caching in many places. However, such objects should still be usable in caches. This can be achieved by defining an appropriate `_cache_key`:

```
sage: # needs sage.rings.padics
sage: hash(b)
Traceback (most recent call last):
...
TypeError: unhashable type: 'sage.rings.padics.qadic_flint_CR.
↪qAdicCappedRelativeElement'
```

(continues on next page)

(continued from previous page)

```
sage: @cached_method
....: def f(x): return x==a
sage: f(b)
True
sage: f(c)  # if b and c were hashable, this would return True
False
sage: b._cache_key()
(..., ((0, 1),), 0, 1)
sage: c._cache_key()
(..., ((0, 1), (1,)), 0, 20)
```

An implementation must make sure that for elements a and b, if `a != b`, then also `a._cache_key() != b._cache_key()`. In practice this means that the `_cache_key` should always include the parent as its first argument:

```
sage: S.<a> = Qq(4)                                                         #␣
↪needs sage.rings.padics
sage: d = a + O(2)                                                          #␣
↪needs sage.rings.padics
sage: b._cache_key() == d._cache_key()  # this would be True if the parents␣
↪were not included              # needs sage.rings.padics
False
```

category()

dump(*filename*, *compress=True*)

Same as self.save(filename, compress)

dumps(*compress=True*)

Dump `self` to a string `s`, which can later be reconstituted as `self` using `loads(s)`.

There is an optional boolean argument `compress` which defaults to `True`.

EXAMPLES:

```
sage: from sage.misc.persist import comp
sage: O = SageObject()
sage: p_comp = O.dumps()
sage: p_uncomp = O.dumps(compress=False)
sage: comp.decompress(p_comp) == p_uncomp
True
sage: import pickletools
sage: pickletools.dis(p_uncomp)
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.structure.sage_object SageObject'
   41: q    BINPUT     ...
   43: )    EMPTY_TUPLE
   44: \x81 NEWOBJ
   45: q    BINPUT     ...
   47: .    STOP
highest protocol among opcodes = 2
```

get_custom_name()

Return the custom name of this object, or `None` if it is not renamed.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: P.get_custom_name() is None
True
sage: P.rename('A polynomial ring')
sage: P.get_custom_name()
'A polynomial ring'
sage: P.reset_name()
sage: P.get_custom_name() is None
True
```

parent()

Return the type of `self` to support the coercion framework.

EXAMPLES:

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t                          #␣
↪needs sage.symbolic
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: u = t.maxima_methods()                                              #␣
↪needs sage.symbolic
sage: u.parent()                                                          #␣
↪needs sage.symbolic
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
```

rename(*x=None*)

Change self so it prints as x, where x is a string.

If x is `None`, the existing custom name is removed.

Note: This is *only* supported for Python classes that derive from SageObject.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x', sparse=True).gen()
sage: g = x^3 + x - 5
sage: g
x^3 + x - 5
sage: g.rename('a polynomial')
sage: g
a polynomial
sage: g + x
x^3 + 2*x - 5
sage: h = g^100
sage: str(h)[:20]
'x^300 + 100*x^298 - '
sage: h.rename('x^300 + ...')
sage: h
x^300 + ...
sage: g.rename(None)
sage: g
x^3 + x - 5
```

Real numbers are not Python classes, so rename is not supported:

```
sage: a = 3.14
sage: type(a)                                                          #␣
↪needs sage.rings.real_mpfr
<... 'sage.rings.real_mpfr.RealLiteral'>
sage: a.rename('pi')                                                   #␣
↪needs sage.rings.real_mpfr
Traceback (most recent call last):
...
NotImplementedError: object does not support renaming: 3.14000000000000
```

Note: The reason C-extension types are not supported by default is if they were then every single one would have to carry around an extra attribute, which would be slower and waste a lot of memory.

To support them for a specific class, add a `cdef public _SageObject__custom_name` attribute.

reset_name()

Remove the custom name of an object.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: P
Univariate Polynomial Ring in x over Rational Field
sage: P.rename('A polynomial ring')
sage: P
A polynomial ring
sage: P.reset_name()
sage: P
Univariate Polynomial Ring in x over Rational Field
```

save(*filename=None*, *compress=True*)

Save self to the given filename.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x = SR.var("x")
sage: f = x^3 + 5
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix=".sobj") as t:
....:     f.save(t.name)
....:     load(t.name)
x^3 + 5
```

1.2 Base class for objects of a category

CLASS HIERARCHY:

- *SageObject*
  - **CategoryObject**
    - *Parent*

Many category objects in Sage are equipped with generators, which are usually special elements of the object. For example, the polynomial ring $\mathbf{Z}[x, y, z]$ is generated by x, y, and z. In Sage the `i` th generator of an object `X` is obtained using the notation `X.gen(i)`. From the Sage interactive prompt, the shorthand notation `X.i` is also allowed.

The following examples illustrate these functions in the context of multivariate polynomial rings and free modules.

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 3, 'x')
sage: R.ngens()
3
sage: R.gen(0)
x0
sage: R.gens()
(x0, x1, x2)
sage: R.variable_names()
('x0', 'x1', 'x2')
```

This example illustrates generators for a free module over $\mathbf{Z}$.

```
sage: # needs sage.modules
sage: M = FreeModule(ZZ, 4)
sage: M
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: M.ngens()
4
sage: M.gen(0)
(1, 0, 0, 0)
sage: M.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

class `sage.structure.category_object.`**`CategoryObject`**

Bases: *SageObject*

An object in some category.

`Hom`(*codomain*, *cat=None*)

Return the homspace `Hom(self, codomain, cat)` of all homomorphisms from `self` to `codomain` in the category `cat`.

The default category is determined by `self.category()` and `codomain.category()`.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.Hom(QQ)
Set of Homomorphisms
```

(continues on next page)

(continued from previous page)

```
from Multivariate Polynomial Ring in x, y over Rational Field
  to Rational Field
```

Homspaces are defined for very general Sage objects, even elements of familiar rings.

```
sage: n = 5; Hom(n,7)
Set of Morphisms from 5 to 7 in Category of elements of Integer Ring
sage: z = 2/3; Hom(z, 8/1)
Set of Morphisms from 2/3 to 8 in Category of elements of Rational Field
```

This example illustrates the optional third argument:

```
sage: QQ.Hom(ZZ, Sets())
Set of Morphisms from Rational Field to Integer Ring in Category of sets
```

base()

base_ring()

Return the base ring of `self`.

INPUT:

- `self` – an object over a base ring; typically a module

EXAMPLES:

```
sage: from sage.modules.module import Module
sage: Module(ZZ).base_ring()
Integer Ring

sage: F = FreeModule(ZZ, 3)                                                 #␣
↪needs sage.modules
sage: F.base_ring()                                                         #␣
↪needs sage.modules
Integer Ring
sage: F.__class__.base_ring                                                 #␣
↪needs sage.modules
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>
```

Note that the coordinates of the elements of a module can lie in a bigger ring, the `coordinate_ring`:

```
sage: # needs sage.modules
sage: M = (ZZ^2) * (1/2)
sage: v = M([1/2, 0])
sage: v.base_ring()
Integer Ring
sage: parent(v[0])
Rational Field
sage: v.coordinate_ring()
Rational Field
```

More examples:

```
sage: F = FreeAlgebra(QQ, 'x')                                         #␣
↪needs sage.combinat sage.modules
sage: F.base_ring()                                                    #␣
↪needs sage.combinat sage.modules
Rational Field
sage: F.__class__.base_ring                                            #␣
↪needs sage.combinat sage.modules
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>

sage: # needs sage.modules
sage: E = CombinatorialFreeModule(ZZ, [1,2,3])
sage: F = CombinatorialFreeModule(ZZ, [2,3,4])
sage: H = Hom(E, F)
sage: H.base_ring()
Integer Ring
sage: H.__class__.base_ring
<method 'base_ring' of 'sage.structure.category_object.CategoryObject' objects>
```

Todo: Move this method elsewhere (typically in the Modules category) so as not to pollute the namespace of all category objects.

categories()

Return the categories of `self`.

EXAMPLES:

```
sage: ZZ.categories()
[Join of Category of euclidean domains
     and Category of infinite enumerated sets
     and Category of metric spaces,
 Category of euclidean domains,
 Category of principal ideal domains,
 Category of unique factorization domains,
 Category of gcd domains,
 Category of integral domains,
 Category of domains,
 Category of commutative rings, ...
 Category of monoids, ...,
 Category of commutative additive groups, ...,
 Category of sets, ...,
 Category of objects]
```

category()

gens_dict(*copy=True*)

Return a dictionary whose entries are `{name:variable,...}`, where `name` stands for the variable names of this object (as strings) and `variable` stands for the corresponding defining generators (as elements of this object).

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()                                  #␣
↪needs sage.rings.polynomial.pbori
sage: B.gens_dict()                                                          #␣
↪needs sage.rings.polynomial.pbori
{'a': a, 'b': b, 'c': c, 'd': d}
```

gens_dict_recursive()

Return the dictionary of generators of `self` and its base rings.

OUTPUT:

- a dictionary with string names of generators as keys and generators of `self` and its base rings as values.

EXAMPLES:

```
sage: R = QQ['x,y']['z,w']
sage: sorted(R.gens_dict_recursive().items())
[('w', w), ('x', x), ('y', y), ('z', z)]
```

inject_variables(*scope=None*, *verbose=True*)

Inject the generators of `self` with their names into the namespace of the Python code from which this function is called.

Thus, e.g., if the generators of `self` are labeled 'a', 'b', and 'c', then after calling this method the variables a, b, and c in the current scope will be set equal to the generators of self.

NOTE: If Foo is a constructor for a Sage object with generators, and Foo is defined in Cython, then it would typically call `inject_variables()` on the object it creates. E.g., `PolynomialRing(QQ, 'y')` does this so that the variable y is the generator of the polynomial ring.

latex_name()

latex_variable_names()

Returns the list of variable names suitable for latex output.

All `_SOMETHING` substrings are replaced by `_{SOMETHING}` recursively so that subscripts of subscripts work.

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ, 'x', 12).objgens()
sage: x
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11)
sage: R.latex_variable_names ()
['x_{0}', 'x_{1}', 'x_{2}', 'x_{3}', 'x_{4}', 'x_{5}', 'x_{6}',
 'x_{7}', 'x_{8}', 'x_{9}', 'x_{10}', 'x_{11}']
sage: f = x[0]^3 + 15/3 * x[1]^10
sage: print(latex(f))
5 x_{1}^{10} + x_{0}^{3}
```

objgen()

Return the tuple `(self, self.gen())`.

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ,'x').objgen()
sage: R
```

(continues on next page)

(continued from previous page)

```
Univariate Polynomial Ring in x over Rational Field
sage: x
x
```

objgens()

Return the tuple `(self, self.gens())`.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 3, 'x'); R
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
sage: R.objgens()
(Multivariate Polynomial Ring in x0, x1, x2 over Rational Field, (x0, x1, x2))
```

variable_name()

Return the first variable name.

OUTPUT: a string

EXAMPLES:

```
sage: R.<z,y,a42> = ZZ[]
sage: R.variable_name()
'z'
sage: R.<x> = InfinitePolynomialRing(ZZ)
sage: R.variable_name()
'x'
```

variable_names()

Return the list of variable names corresponding to the generators.

OUTPUT: a tuple of strings

EXAMPLES:

```
sage: R.<z,y,a42> = QQ[]
sage: R.variable_names()
('z', 'y', 'a42')
sage: S = R.quotient_ring(z+y)
sage: S.variable_names()
('zbar', 'ybar', 'a42bar')
```

```
sage: T.<x> = InfinitePolynomialRing(ZZ)
sage: T.variable_names()
('x',)
```

sage.structure.category_object.**certify_names**(*names*)

Check that `names` are valid variable names.

INPUT:

- `names` – an iterable with strings representing variable names

OUTPUT: `True` (for efficiency of the Cython call)

EXAMPLES:

```
sage: from sage.structure.category_object import certify_names as cn
sage: cn(["a", "b", "c"])
1
sage: cn("abc")
1
sage: cn([])
1
sage: cn([""])
Traceback (most recent call last):
...
ValueError: variable name must be nonempty
sage: cn(["_foo"])
Traceback (most recent call last):
...
ValueError: variable name '_foo' does not start with a letter
sage: cn(["x'"])
Traceback (most recent call last):
...
ValueError: variable name "x'" is not alphanumeric
sage: cn(["a", "b", "b"])
Traceback (most recent call last):
...
ValueError: variable name 'b' appears more than once
```

sage.structure.category_object.**check_default_category**(*default_category*, *category*)

sage.structure.category_object.**normalize_names**(*ngens*, *names*)

Return a tuple of strings of variable names of length ngens given the input names.

INPUT:

- `ngens` – integer: number of generators. The value `ngens=-1` means that the number of generators is unknown a priori.
- `names` – any of the following:
  - a tuple or list of strings, such as `('x', 'y')`
  - a comma-separated string, such as `x,y`
  - a string prefix, such as 'alpha'
  - a string of single character names, such as 'xyz'

OUTPUT: a tuple of `ngens` strings to be used as variable names.

EXAMPLES:

```
sage: from sage.structure.category_object import normalize_names as nn
sage: nn(0, "")
()
sage: nn(0, [])
()
sage: nn(0, None)
()
sage: nn(1, 'a')
('a',)
```

(continues on next page)

(continued from previous page)

```
sage: nn(2, 'z_z')
('z_z0', 'z_z1')
sage: nn(3, 'x, y, z')
('x', 'y', 'z')
sage: nn(2, 'ab')
('a', 'b')
sage: nn(2, 'x0')
('x00', 'x01')
sage: nn(3, (' a ', ' bb ', ' ccc '))
('a', 'bb', 'ccc')
sage: nn(4, ['a1', 'a2', 'b1', 'b11'])
('a1', 'a2', 'b1', 'b11')
```

Arguments are converted to strings:

```
sage: nn(1, u'a')
('a',)
sage: var('alpha')                                                        #␣
↪needs sage.symbolic
alpha
sage: nn(2, alpha)                                                        #␣
↪needs sage.symbolic
('alpha0', 'alpha1')
sage: nn(1, [alpha])                                                      #␣
↪needs sage.symbolic
('alpha',)
```

With an unknown number of generators:

```
sage: nn(-1, 'a')
('a',)
sage: nn(-1, 'x, y, z')
('x', 'y', 'z')
```

Test errors:

```
sage: nn(3, ["x", "y"])
Traceback (most recent call last):
...
IndexError: the number of names must equal the number of generators
sage: nn(None, "a")
Traceback (most recent call last):
...
TypeError: 'NoneType' object cannot be interpreted as an integer
sage: nn(1, "")
Traceback (most recent call last):
...
ValueError: variable name must be nonempty
sage: nn(1, "foo@")
Traceback (most recent call last):
...
ValueError: variable name 'foo@' is not alphanumeric
sage: nn(2, "_foo")
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: variable name '_foo0' does not start with a letter
sage: nn(1, 3/2)
Traceback (most recent call last):
...
ValueError: variable name '3/2' is not alphanumeric
```

CHAPTER

TWO

PARENTS

2.1 Parents

2.1.1 Base class for parent objects

CLASS HIERARCHY:

```
SageObject
    CategoryObject
        Parent
```

A simple example of registering coercions:

```
sage: class A_class(Parent):
....:   def __init__(self, name):
....:       Parent.__init__(self)
....:       self._populate_coercion_lists_()
....:       self.rename(name)
....:
....:   def category(self):
....:       return Sets()
....:
....:   def _element_constructor_(self, i):
....:       assert(isinstance(i, (int, Integer)))
....:       return ElementWrapper(self, i)
sage: A = A_class("A")
sage: B = A_class("B")
sage: C = A_class("C")

sage: def f(a):
....:   return B(a.value+1)
sage: class MyMorphism(Morphism):
....:   def __init__(self, domain, codomain):
....:       Morphism.__init__(self, Hom(domain, codomain))
....:
....:   def _call_(self, x):
....:       return self.codomain()(x.value)
sage: f = MyMorphism(A,B)
sage: f
    Generic morphism:
```

(continues on next page)

(continued from previous page)

```
      From: A
      To:   B
sage: B.register_coercion(f)
sage: C.register_coercion(MyMorphism(B,C))
sage: A(A(1)) == A(1)
True
sage: B(A(1)) == B(1)
True
sage: C(A(1)) == C(1)
True

sage: A(B(1))
Traceback (most recent call last):
...
AssertionError
```

When implementing an element of a ring, one would typically provide the element class with `_rmul_` and/or `_lmul_` methods for the action of a base ring, and with `_mul_` for the ring multiplication. However, prior to github issue #14249, it would have been necessary to additionally define a method `_an_element_()` for the parent. But now, the following example works:

```
sage: from sage.structure.element import RingElement
sage: class MyElement(RingElement):
....:     def __init__(self, parent, x, y):
....:         RingElement.__init__(self, parent)
....:     def _mul_(self, other):
....:         return self
....:     def _rmul_(self, other):
....:         return self
....:     def _lmul_(self, other):
....:         return self
sage: class MyParent(Parent):
....:     Element = MyElement
```

Now, we define

```
sage: P = MyParent(base=ZZ, category=Rings())
sage: a = P(1,2)
sage: a*a is a
True
sage: a*2 is a
True
sage: 2*a is a
True
```

class sage.structure.parent.**EltPair**

Bases: `object`

short_repr()

class sage.structure.parent.**Parent**

Bases: *CategoryObject*

Base class for all parents.

Parents are the Sage/mathematical analogues of container objects in computer science.

INPUT:

- `base` – An algebraic structure considered to be the "base" of this parent (e.g. the base field for a vector space).
- `category` – a category or list/tuple of categories. The category in which this parent lies (or list or tuple thereof). Since categories support more general super-categories, this should be the most specific category possible. If category is a list or tuple, a JoinCategory is created out of them. If category is not specified, the category will be guessed (see `CategoryObject`), but will not be used to inherit parent's or element's code from this category.
- `names` – Names of generators.
- `normalize` – Whether to standardize the names (remove punctuation, etc)
- `facade` – a parent, or tuple thereof, or `True`

If `facade` is specified, then `Sets().Facade()` is added to the categories of the parent. Furthermore, if `facade` is not `True`, the internal attribute `_facade_for` is set accordingly for use by `Sets.Facade.ParentMethods.facade_for()`.

Internal invariants:

- `self._element_init_pass_parent == guess_pass_parent(self, self._element_constructor)` Ensures that `__call__()` passes down the parent properly to `_element_constructor()`. See github issue #5979.

Todo: Eventually, category should be `Sets` by default.

`__call__`(*x=0*, **args*, ***kwds*)

This is the generic call method for all parents.

When called, it will find a map based on the Parent (or type) of x. If a coercion exists, it will always be chosen. This map will then be called (with the arguments and keywords if any).

By default this will dispatch as quickly as possible to `_element_constructor_()` though faster pathways are possible if so desired.

`_populate_coercion_lists_`(*coerce_list=[]*, *action_list=[]*, *convert_list=[]*, *embedding=None*, *convert_method_name=None*, *element_constructor=None*, *init_no_parent=None*, *unpickling=False*)

This function allows one to specify coercions, actions, conversions and embeddings involving this parent.

IT SHOULD ONLY BE CALLED DURING THE __INIT__ method, often at the end.

INPUT:

- `coerce_list` – a list of coercion Morphisms to self and parents with canonical coercions to self
- `action_list` – a list of actions on and by self
- **`convert_list` – a list of conversion Maps to self and**
  parents with conversions to self
- `embedding` – a single Morphism from self
- `convert_method_name` – a name to look for that other elements can implement to create elements of self (e.g. _integer_)

- `init_no_parent` – if True omit passing self in as the first argument of element_constructor for conversion. This is useful if parents are unique, or element_constructor is a bound method (this latter case can be detected automatically).

`__mul__`(*x*)

This is a multiplication method that more or less directly calls another attribute `_mul_` (single underscore). This is because `__mul__` cannot be implemented via inheritance from the parent methods of the category, but `_mul_` can be inherited. This is, e.g., used when creating twosided ideals of matrix algebras. See github issue #7797.

EXAMPLES:

```
sage: MS = MatrixSpace(QQ, 2, 2)                                          #␣
↪needs sage.modules
```

This matrix space is in fact an algebra, and in particular it is a ring, from the point of view of categories:

```
sage: MS.category()                                                       #␣
↪needs sage.modules
Category of infinite finite dimensional algebras with basis
 over (number fields and quotient fields and metric spaces)
sage: MS in Rings()                                                       #␣
↪needs sage.modules
True
```

However, its class does not inherit from the base class `Ring`:

```
sage: isinstance(MS, Ring)                                                #␣
↪needs sage.modules
False
```

Its `_mul_` method is inherited from the category, and can be used to create a left or right ideal:

```
sage: # needs sage.modules
sage: MS._mul_.__module__
'sage.categories.rings'
sage: MS * MS.1      # indirect doctest
Left Ideal
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS * [MS.1, 2]
Left Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS.1 * MS
Right Ideal
```

(continues on next page)

(continued from previous page)

```
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: [MS.1, 2] * MS
Right Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

__contains__(*x*)

True if there is an element of self that is equal to x under ==, or if x is already an element of self. Also, True in other cases involving the Symbolic Ring, which is handled specially.

For many structures we test this by using *__call__()* and then testing equality between x and the result.

The Symbolic Ring is treated differently because it is ultra-permissive about letting other rings coerce in, but ultra-strict about doing comparisons.

EXAMPLES:

```
sage: 2 in Integers(7)
True
sage: 2 in ZZ
True
sage: Integers(7)(3) in ZZ
True
sage: 3/1 in ZZ
True
sage: 5 in QQ
True
sage: I in RR                                                          #␣
↪needs sage.rings.real_mpfr sage.symbolic
False
sage: RIF(1, 2) in RIF                                                 #␣
↪needs sage.rings.real_interval_field
True

sage: # needs sage.symbolic
sage: SR(2) in ZZ
True
sage: pi in RIF  # there is no element of RIF equal to pi
False
sage: sqrt(2) in CC
True
sage: pi in RR
True
```

(continues on next page)

(continued from previous page)

```
sage: pi in CC
True
sage: pi in RDF
True
sage: pi in CDF
True
```

Note that we have

```
sage: 3/2 in RIF                                                            #␣
↪needs sage.rings.real_interval_field
True
```

because 3/2 has an exact representation in RIF (i.e. can be represented as an interval that contains exactly one value):

```
sage: RIF(3/2).is_exact()                                                   #␣
↪needs sage.rings.real_interval_field
True
```

On the other hand, we have

```
sage: 2/3 in RIF                                                            #␣
↪needs sage.rings.real_interval_field
False
```

because 2/3 has no exact representation in RIF. Since RIF(2/3) is a nontrivial interval, it cannot be equal to anything (not even itself):

```
sage: RIF(2/3).is_exact()                                                   #␣
↪needs sage.rings.real_interval_field
False
sage: RIF(2/3).endpoints()                                                  #␣
↪needs sage.rings.real_interval_field
(0.666666666666666, 0.666666666666667)
sage: RIF(2/3) == RIF(2/3)                                                  #␣
↪needs sage.rings.real_interval_field
False
```

_coerce_map_from_(*S*)

Override this method to specify coercions beyond those specified in coerce_list.

If no such coercion exists, return None or False. Otherwise, it may return either an actual Map to use for the coercion, a callable (in which case it will be wrapped in a Map), or True (in which case a generic map will be provided).

_convert_map_from_(*S*)

Override this method to provide additional conversions beyond those given in convert_list.

This function is called after coercions are attempted. If there is a coercion morphism in the opposite direction, one should consider adding a section method to that.

This MUST return a Map from S to self, or None. If None is returned then a generic map will be provided.

_get_action_(*S*, *op*, *self_on_left*)

Override this method to provide an action of self on S or S on self beyond what was specified in action_list.

This must return an action which accepts an element of self and an element of S (in the order specified by self_on_left).

_an_element_()

Return an element of `self`.

Want it in sufficient generality that poorly-written functions will not work when they are not supposed to. This is cached so does not have to be super fast.

EXAMPLES:

```
sage: QQ._an_element_()
1/2
sage: ZZ['x,y,z']._an_element_()
x
```

_repr_option(*key*)

Metadata about the `_repr_()` output.

INPUT:

- `key` – string. A key for different metadata informations that can be inquired about.

Valid `key` arguments are:

- `'ascii_art'`: The `_repr_()` output is multi-line ascii art and each line must be printed starting at the same column, or the meaning is lost.
- `'element_ascii_art'`: same but for the output of the elements. Used in `sage.repl.display.formatter`.
- `'element_is_atomic'`: the elements print atomically, that is, parenthesis are not required when *printing* out any of $x - y$, $x + y$, x^y and x/y.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: ZZ._repr_option('ascii_art')
False
sage: MatrixSpace(ZZ, 2)._repr_option('element_ascii_art')                #␣
↪needs sage.modules
True
```

_init_category_(*category*)

Initialize the category framework.

Most parents initialize their category upon construction, and this is the recommended behavior. For example, this happens when the constructor calls `Parent.__init__()` directly or indirectly. However, some parents defer this for performance reasons. For example, `sage.matrix.matrix_space.MatrixSpace` does not.

EXAMPLES:

```
sage: P = Parent()
sage: P.category()
Category of sets
sage: class MyParent(Parent):
....:     def __init__(self):
....:         self._init_category_(Groups())
sage: MyParent().category()
Category of groups
```

_is_coercion_cached(*domain*)

Test whether the coercion from `domain` is already cached.

EXAMPLES:

```
sage: R.<XX> = QQ
sage: R._remove_from_coerce_cache(QQ)
sage: R._is_coercion_cached(QQ)
False
sage: _ = R.coerce_map_from(QQ)
sage: R._is_coercion_cached(QQ)
True
```

_is_conversion_cached(*domain*)

Test whether the conversion from `domain` is already set.

EXAMPLES:

```
sage: P = Parent()
sage: P._is_conversion_cached(P)
False
sage: P.convert_map_from(P)
Identity endomorphism of <sage.structure.parent.Parent object at ...>
sage: P._is_conversion_cached(P)
True
```

Hom(*codomain*, *category=None*)

Return the homspace `Hom(self, codomain, category)`.

INPUT:

- `codomain` – a parent
- `category` – a category or `None` (default: `None`) If `None`, the meet of the category of `self` and `codomain` is used.

OUTPUT:

The homspace of all homomorphisms from `self` to `codomain` in the category `category`.

See also:

`Hom()`

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.Hom(QQ)
```

(continues on next page)

(continued from previous page)

```
Set of Homomorphisms from Multivariate Polynomial Ring in x, y over Rational␣
↪Field to Rational Field
```

Homspaces are defined for very general Sage objects, even elements of familiar rings:

```
sage: n = 5; Hom(n,7)
Set of Morphisms from 5 to 7 in Category of elements of Integer Ring
sage: z=(2/3); Hom(z,8/1)
Set of Morphisms from 2/3 to 8 in Category of elements of Rational Field
```

This example illustrates the optional third argument:

```
sage: QQ.Hom(ZZ, Sets())
Set of Morphisms from Rational Field to Integer Ring in Category of sets
```

A parent may specify how to construct certain homsets by implementing a method `_Hom_`(codomain, category). See :func:`~sage.categories.homset.Hom()` for details.

an_element()

Returns a (preferably typical) element of this parent.

This is used both for illustration and testing purposes. If the set `self` is empty, *an_element()* raises the exception `EmptySetError`.

This calls *_an_element_()* (which see), and caches the result. Parent are thus encouraged to override *_an_element_()*.

EXAMPLES:

```
sage: CDF.an_element()                                                    #␣
↪needs sage.rings.complex_double
1.0*I
sage: ZZ[['t']].an_element()
t
```

In case the set is empty, an `EmptySetError` is raised:

```
sage: Set([]).an_element()
Traceback (most recent call last):
...
EmptySetError
```

category()

EXAMPLES:

```
sage: P = Parent()
sage: P.category()
Category of sets
sage: class MyParent(Parent):
....:     def __init__(self): pass
sage: MyParent().category()
Category of sets
```

coerce(*x*)

Return x as an element of self, if and only if there is a canonical coercion from the parent of x to self.

EXAMPLES:

```
sage: QQ.coerce(ZZ(2))
2
sage: ZZ.coerce(QQ(2))
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Integer Ring
```

We make an exception for zero:

```
sage: V = GF(7)^7                                                         #␣
↪needs sage.modules
sage: V.coerce(0)                                                         #␣
↪needs sage.modules
(0, 0, 0, 0, 0, 0, 0)
```

coerce_embedding()

Return the embedding of `self` into some other parent, if such a parent exists.

This does not mean that there are no coercion maps from `self` into other fields, this is simply a specific morphism specified out of `self` and usually denotes a special relationship (e.g. sub-objects, choice of completion, etc.)

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 + x^2 + 1, embedding=1)
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 + x^2 + 1
        with a = -1.465571231876768?
  To:   Real Lazy Field
  Defn: a -> -1.465571231876768?
sage: K.<a> = NumberField(x^3 + x^2 + 1, embedding=CC.gen())
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 + x^2 + 1
        with a = 0.2327856159383841? + 0.7925519925154479?*I
  To:   Complex Lazy Field
  Defn: a -> 0.2327856159383841? + 0.7925519925154479?*I
```

coerce_map_from(*S*)

Return a `Map` object to coerce from `S` to `self` if one exists, or `None` if no such coercion exists.

EXAMPLES:

By github issue #12313, a special kind of weak key dictionary is used to store coercion and conversion maps, namely `MonoDict`. In that way, a memory leak was fixed that would occur in the following test:

```
sage: import gc
sage: _ = gc.collect()
sage: K = GF(1<<55,'t')                                                   #␣
↪needs sage.rings.finite_rings
sage: for i in range(50):                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings sage.schemes
....:     a = K.random_element()
....:     E = EllipticCurve(j=a)
....:     b = K.has_coerce_map_from(E)
sage: _ = gc.collect()
sage: len([x for x in gc.get_objects() if isinstance(x, type(E))])          #␣
↪needs sage.rings.finite_rings sage.schemes
1
```

convert_map_from(*S*)

This function returns a `Map` from S to $self$, which may or may not succeed on all inputs. If a coercion map from S to self exists, then the it will be returned. If a coercion from $self$ to S exists, then it will attempt to return a section of that map.

Under the new coercion model, this is the fastest way to convert elements of S to elements of $self$ (short of manually constructing the elements) and is used by `__call__()`.

EXAMPLES:

```
sage: m = ZZ.convert_map_from(QQ)
sage: m
Generic map:
  From: Rational Field
  To:   Integer Ring
sage: m(-35/7)
-5
sage: parent(m(-35/7))
Integer Ring
```

element_class()

The (default) class for the elements of this parent

FIXME's and design issues:

- If self.Element is "trivial enough", should we optimize it away with: self.element_class = dynamic_class("%s.element_class"%self.__class__.__name__, (category.element_class,), self.Element)
- This should lookup for Element classes in all super classes

get_action(*S*, *op=None*, *self_on_left=True*, *self_el=None*, *S_el=None*)

Returns an action of self on S or S on self.

To provide additional actions, override `_get_action_()`.

> **Warning:** This is not the method that you typically want to call. Instead, call `coercion_model.get_action(...)` which caches results (this `Parent.get_action` method does not).

has_coerce_map_from(*S*)

Return `True` if there is a natural map from `S` to `self`. Otherwise, return `False`.

EXAMPLES:

```
sage: RDF.has_coerce_map_from(QQ)
True
sage: RDF.has_coerce_map_from(QQ['x'])
False
sage: RDF['x'].has_coerce_map_from(QQ['x'])
True
sage: RDF['x,y'].has_coerce_map_from(QQ['x'])
True
```

hom(*im_gens*, *codomain=None*, *check=None*, *base_map=None*, *category=None*, ***kwds*)

Return the unique homomorphism from self to codomain that sends `self.gens()` to the entries of `im_gens`.

This raises a `TypeError` if there is no such homomorphism.

INPUT:

- `im_gens` – the images in the codomain of the generators of this object under the homomorphism
- `codomain` – the codomain of the homomorphism
- `base_map` – a map from the base ring to the codomain. If not given, coercion is used.
- `check` – whether to verify that the images of generators extend to define a map (using only canonical coercions).

OUTPUT:

A homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of `im_gens`, in which case return the (if it exists) natural map to X.

EXAMPLES:

Polynomial Ring: We first illustrate construction of a few homomorphisms involving a polynomial ring:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
Traceback (most recent call last):
...
ValueError: relations do not all (canonically) map to 0
under map determined by images of generators

sage: # needs sage.rings.finite_rings
sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49,'a'))
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
```

(continues on next page)

(continued from previous page)

```
  Defn: x |--> 3
sage: f(x + 6)
2
sage: f(x^2 + 1)
3
```

Natural morphism:

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a `TypeError` is raised:

```
sage: QQ.hom(ZZ)
Traceback (most recent call last):
...
TypeError: natural coercion morphism from Rational Field to Integer Ring not␣
↪defined
```

is_exact()

Test whether the ring is exact.

Note: This defaults to true, so even if it does return `True` you have no guarantee (unless the ring has properly overloaded this).

OUTPUT:

Return True if elements of this ring are represented exactly, i.e., there is no precision loss when doing arithmetic.

EXAMPLES:

```
sage: QQ.is_exact()
True
sage: ZZ.is_exact()
True
sage: Qp(7).is_exact()                                                  #␣
↪needs sage.rings.padics
False
sage: Zp(7, type='capped-abs').is_exact()                               #␣
↪needs sage.rings.padics
False
```

register_action(*action*)

Update the coercion model to use `action` to act on self.

`action` should be of type `sage.categories.action.Action`.

EXAMPLES:

```
sage: import sage.categories.action
sage: import operator

sage: class SymmetricGroupAction(sage.categories.action.Action):
....:     "Act on a multivariate polynomial ring by permuting the generators."
....:     def __init__(self, G, M, is_left=True):
....:         sage.categories.action.Action.__init__(self, G, M, is_left,␣
↪operator.mul)
....:
....:     def _act_(self, g, a):
....:         D = {}
....:         for k, v in a.dict().items():
....:             nk = [0]*len(k)
....:             for i in range(len(k)):
....:                 nk[g(i+1)-1] = k[i]
....:             D[tuple(nk)] = v
....:         return a.parent()(D)

sage: # needs sage.groups
sage: R.<x, y, z> = QQ['x, y, z']
sage: G = SymmetricGroup(3)
sage: act = SymmetricGroupAction(G, R)
sage: t = x + 2*y + 3*z

sage: # needs sage.groups
sage: act(G((1, 2)), t)
2*x + y + 3*z
sage: act(G((2, 3)), t)
x + 3*y + 2*z
sage: act(G((1, 2, 3)), t)
3*x + y + 2*z
```

This should fail, since we have not registered the left action:

```
sage: G((1,2)) * t                                                        #␣
↪needs sage.groups
Traceback (most recent call last):
...
TypeError: ...
```

Now let's make it work:

```
sage: # needs sage.groups
sage: R._unset_coercions_used()
sage: R.register_action(act)
sage: G((1, 2)) * t
2*x + y + 3*z
```

register_coercion(*mor*)

Update the coercion model to use $mor : P \to$ self to coerce from a parent P into `self`.

For safety, an error is raised if another coercion has already been registered or discovered between P and `self`.

EXAMPLES:

```
sage: K.<a> = ZZ['a']
sage: L.<b> = ZZ['b']
sage: L_into_K = L.hom([-a]) # non-trivial automorphism
sage: K.register_coercion(L_into_K)

sage: K(0) + b
-a
sage: a + b
0
sage: K(b) # check that convert calls coerce first; normally this is just a
-a

sage: L(0) + a in K # this goes through the coercion mechanism of K
True
sage: L(a) in L # this still goes through the convert mechanism of L
True

sage: K.register_coercion(L_into_K)
Traceback (most recent call last):
...
AssertionError: coercion from Univariate Polynomial Ring in b over Integer Ring␣
↪to Univariate Polynomial Ring in a over Integer Ring already registered or␣
↪discovered
```

register_conversion(*mor*)

Update the coercion model to use mor : $P \to$ self to convert from P into `self`.

EXAMPLES:

```
sage: K.<a> = ZZ['a']
sage: M.<c> = ZZ['c']
sage: M_into_K = M.hom([a]) # trivial automorphism
sage: K._unset_coercions_used()
sage: K.register_conversion(M_into_K)

sage: K(c)
a
sage: K(0) + c
Traceback (most recent call last):
...
TypeError: ...
```

register_embedding(*embedding*)

Add embedding to coercion model.

This method updates the coercion model to use embedding : self $\to P$ to embed `self` into the parent P.

There can only be one embedding registered; it can only be registered once; and it must be registered before using this parent in the coercion model.

EXAMPLES:

```
sage: S3 = AlternatingGroup(3)                                          #␣
↪needs sage.groups
sage: G = SL(3, QQ)                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
sage: p = S3[2]; p.matrix()                                          #␣
↪needs sage.groups
[0 0 1]
[1 0 0]
[0 1 0]
```

In general one cannot mix matrices and permutations:

```
sage: # needs sage.groups
sage: G(p)
Traceback (most recent call last):
...
TypeError: unable to convert (1,3,2) to a rational
sage: phi = S3.hom(lambda p: G(p.matrix()), codomain=G)
sage: phi(p)
[0 0 1]
[1 0 0]
[0 1 0]
sage: S3._unset_coercions_used()
sage: S3.register_embedding(phi)
```

By github issue #14711, coerce maps should be copied when using outside of the coercion system:

```
sage: phi = copy(S3.coerce_embedding()); phi                         #␣
↪needs sage.groups
Generic morphism:
  From: Alternating group of order 3!/2 as a permutation group
  To:   Special Linear Group of degree 3 over Rational Field
sage: phi(p)                                                         #␣
↪needs sage.groups
[0 0 1]
[1 0 0]
[0 1 0]
```

This does not work since matrix groups are still old-style parents (see github issue #14014):

```
sage: G(p)                      # not implemented                    #␣
↪needs sage.groups
```

Though one can have a permutation act on the rows of a matrix:

```
sage: G(1) * p                                                       #␣
↪needs sage.groups
[0 0 1]
[1 0 0]
[0 1 0]
```

Some more advanced examples:

```
sage: # needs sage.rings.number_field
sage: x = QQ['x'].0
sage: t = abs(ZZ.random_element(10^6))
```

(continues on next page)

(continued from previous page)

```
sage: K = NumberField(x^2 + 2*3*7*11, "a"+str(t))
sage: a = K.gen()
sage: K_into_MS = K.hom([a.matrix()])
sage: K._unset_coercions_used()
sage: K.register_embedding(K_into_MS)

sage: # needs sage.rings.number_field
sage: L = NumberField(x^2 + 2*3*7*11*19*31,
....:                 "b" + str(abs(ZZ.random_element(10^6))))
sage: b = L.gen()
sage: L_into_MS = L.hom([b.matrix()])
sage: L._unset_coercions_used()
sage: L.register_embedding(L_into_MS)

sage: K.coerce_embedding()(a)                                           #␣
↪needs sage.rings.number_field
[   0    1]
[-462    0]
sage: L.coerce_embedding()(b)                                           #␣
↪needs sage.rings.number_field
[      0       1]
[-272118       0]

sage: a.matrix() * b.matrix()                                           #␣
↪needs sage.rings.number_field
[-272118       0]
[      0    -462]
sage: a.matrix() * b.matrix()                                           #␣
↪needs sage.rings.number_field
[-272118       0]
[      0    -462]
```

class sage.structure.parent.**Set_generic**

Bases: *Parent*

Abstract base class for sets.

object()

Return the underlying object of self.

EXAMPLES:

```
sage: Set(QQ).object()
Rational Field
```

sage.structure.parent.**is_Parent**(*x*)

Return True if x is a parent object, i.e., derives from sage.structure.parent.Parent and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent import is_Parent
sage: is_Parent(2/3)
False
sage: is_Parent(ZZ)
```

(continues on next page)

(continued from previous page)

```
True
sage: is_Parent(Primes())
True
```

2.1.2 Indexed Generators

class sage.structure.indexed_generators.**IndexedGenerators**(*indices*, *prefix='x'*, ***kwds*)

Bases: object

Abstract base class for parents whose elements consist of generators indexed by an arbitrary set.

Options controlling the printing of elements:

- prefix – string, prefix used for printing elements of this module (optional, default 'x'). With the default, a monomial indexed by 'a' would be printed as x['a'].
- latex_prefix – string or None, prefix used in the LaTeX representation of elements (optional, default None). If this is anything except the empty string, it prints the index as a subscript. If this is None, it uses the setting for prefix, so if prefix is set to "B", then a monomial indexed by 'a' would be printed as B_{a}. If this is the empty string, then don't print monomials as subscripts: the monomial indexed by 'a' would be printed as a, or as [a] if latex_bracket is True.
- names – dict with strings as values or list of strings (optional): a mapping from the indices of the generators to strings giving the generators explicit names. This is used instead of the print options prefix and bracket when names is specified.
- latex_names – dict with strings as values or list of strings (optional): same as names except using the LaTeX representation
- bracket – None, bool, string, or list or tuple of strings (optional, default None): if None, use the value of the attribute self._repr_option_bracket, which has default value True. (self._repr_option_bracket is available for backwards compatibility. Users should set bracket instead. If bracket is set to anything except None, it overrides the value of self._repr_option_bracket.) If False, do not include brackets when printing elements: a monomial indexed by 'a' would be printed as B'a', and a monomial indexed by (1,2,3) would be printed as B(1,2,3). If True, use "[" and "]" as brackets. If it is one of "[", "(", or "{", use it and its partner as brackets. If it is any other string, use it as both brackets. If it is a list or tuple of strings, use the first entry as the left bracket and the second entry as the right bracket.
- latex_bracket – bool, string, or list or tuple of strings (optional, default False): if False, do not include brackets in the LaTeX representation of elements. This option is only relevant if latex_prefix is the empty string; otherwise, brackets are not used regardless. If True, use "left[" and "right]" as brackets. If this is one of "[", "(", "\{", "|", or "||", use it and its partner, prepended with "left" and "right", as brackets. If this is any other string, use it as both brackets. If this is a list or tuple of strings, use the first entry as the left bracket and the second entry as the right bracket.
- scalar_mult – string to use for scalar multiplication in the print representation (optional, default "*")
- latex_scalar_mult – string or None (default: None), string to use for scalar multiplication in the latex representation. If None, use the empty string if scalar_mult is set to "*", otherwise use the value of scalar_mult.
- tensor_symbol – string or None (default: None), string to use for tensor product in the print representation. If None, use sage.categories.tensor.symbol and sage.categories.tensor.unicode_symbol.
- sorting_key – a key function (default: lambda x: x), to use for sorting elements in the output of elements

- `sorting_reverse` – bool (default: `False`), if `True` sort elements in reverse order in the output of elements
- `string_quotes` – bool (default: `True`), if `True` then display string indices with quotes
- `iterate_key` – bool (default: `False`) iterate through the elements of the key and print the result as comma separated objects for string output

Note: These print options may also be accessed and modified using the *print_options()* method, after the parent has been defined.

EXAMPLES:

We demonstrate a variety of the input options:

```
sage: from sage.structure.indexed_generators import IndexedGenerators
sage: I = IndexedGenerators(ZZ, prefix='A')
sage: I._repr_generator(2)
'A[2]'
sage: I._latex_generator(2)
'A_{2}'

sage: I = IndexedGenerators(ZZ, bracket='(')
sage: I._repr_generator(2)
'x(2)'
sage: I._latex_generator(2)
'x_{2}'

sage: I = IndexedGenerators(ZZ, prefix="", latex_bracket='(')
sage: I._repr_generator(2)
'[2]'
sage: I._latex_generator(2)
\left( 2 \right)

sage: I = IndexedGenerators(ZZ, bracket=['|', '>'])
sage: I._repr_generator(2)
'x|2>'
```

indices()

Return the indices of `self`.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])                  #␣
↪needs sage.modules
sage: F.indices()                                                       #␣
↪needs sage.modules
{'a', 'b', 'c'}
```

prefix()

Return the prefix used when displaying elements of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])                  #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
sage: F.prefix()                                                          #␣
↪needs sage.modules
'B'
```

```
sage: X = SchubertPolynomialRing(QQ)                                      #␣
↪needs sage.combinat sage.modules
sage: X.prefix()                                                          #␣
↪needs sage.combinat sage.modules
'X'
```

print_options(***kwds*)

Return the current print options, or set an option.

INPUT: all of the input is optional; if present, it should be in the form of keyword pairs, such as `latex_bracket='('`. The allowable keywords are:

- `prefix`
- `latex_prefix`
- `names`
- `latex_names`
- `bracket`
- `latex_bracket`
- `scalar_mult`
- `latex_scalar_mult`
- `tensor_symbol`
- `string_quotes`
- `sorting_key`
- `sorting_reverse`
- `iterate_key`

See the documentation for `IndexedGenerators` for descriptions of the effects of setting each of these options.

OUTPUT: if the user provides any input, set the appropriate option(s) and return nothing. Otherwise, return the dictionary of settings for print and LaTeX representations.

EXAMPLES:

```
sage: # needs sage.modules
sage: F = CombinatorialFreeModule(ZZ, [1,2,3], prefix='x')
sage: F.print_options()
{...'prefix': 'x'...}
sage: F.print_options(bracket='(')
sage: F.print_options()
{...'bracket': '('...}
```

sage.structure.indexed_generators.**parse_indices_names**(*names*, *index_set*, *prefix*, *kwds=None*)

Parse the names, index set, and prefix input, along with setting default values for keyword arguments kwds.

OUTPUT:

The triple (N, I, p):

- N is the tuple of variable names,
- I is the index set, and
- p is the prefix.

This modifies the dictionary kwds.

Note: When the indices, names, or prefix have not been given, it should be passed to this function as None.

Note: For handling default prefixes, if the result will be None if it is not processed in this function.

EXAMPLES:

```
sage: from sage.structure.indexed_generators import parse_indices_names
sage: d = {}
sage: parse_indices_names('x,y,z', ZZ, None, d)
(('x', 'y', 'z'), Integer Ring, None)
sage: d
{}
sage: d = {}
sage: parse_indices_names('x,y,z', None, None, d)
(('x', 'y', 'z'), {'x', 'y', 'z'}, '')
sage: d
{'string_quotes': False}
sage: d = {}
sage: parse_indices_names(None, ZZ, None, d)
(None, Integer Ring, None)
sage: d
{}
```

```
sage: d = {'string_quotes':True, 'bracket':'['}
sage: parse_indices_names(['a','b','c'], ZZ, 'x', d)
(('a', 'b', 'c'), Integer Ring, 'x')
sage: d
{'bracket': '[', 'string_quotes': True}
sage: parse_indices_names('x,y,z', None, 'A', d)
(('x', 'y', 'z'), {'x', 'y', 'z'}, 'A')
sage: d
{'bracket': '[', 'string_quotes': True}
```

sage.structure.indexed_generators.**split_index_keywords**(*kwds*)

Split the dictionary kwds into two dictionaries, one containing keywords for *IndexedGenerators*, and the other is everything else.

OUTPUT:

The dictionary containing only they keywords for *IndexedGenerators*. This modifies the dictionary kwds.

> **Warning:** This modifies the input dictionary kwds.

EXAMPLES:

```
sage: from sage.structure.indexed_generators import split_index_keywords
sage: d = {'string_quotes': False, 'bracket': None, 'base': QQ}
sage: split_index_keywords(d)
{'bracket': None, 'string_quotes': False}
sage: d
{'base': Rational Field}
```

sage.structure.indexed_generators.**standardize_names_index_set**(*names=None*, *index_set=None*, *ngens=None*)

Standardize the `names` and `index_set` inputs.

INPUT:

- `names` – (optional) the variable names
- `index_set` – (optional) the index set
- `ngens` – (optional) the number of generators

If `ngens` is a negative number, then this does not check that the number of variable names matches the size of the index set.

OUTPUT:

A pair `(names_std, index_set_std)`, where `names_std` is either `None` or a tuple of strings, and where `index_set_std` is a finite enumerated set. The purpose of `index_set_std` is to index the generators of some object (e.g., the basis of a module); the strings in `names_std`, when they exist, are used for printing these indices. The `ngens`

If `names` contains exactly one name `X` and `ngens` is greater than 1, then `names_std` are `Xi` for `i` in `range(ngens)`.

2.1.3 Precision management for non-exact objects

Manage the default precision for non-exact objects such as power series rings or Laurent series rings.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(QQ)
sage: R.default_prec()
20
sage: cos(x)
1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 - 1/3628800*x^10 +
 1/479001600*x^12 - 1/87178291200*x^14 + 1/20922789888000*x^16 -
 1/6402373705728000*x^18 + O(x^20)
```

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=10)
sage: R.default_prec()
10
sage: cos(x)
1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 + O(x^10)
```

Note: Subclasses of *Nonexact* which require to change the default precision should implement a method `set_default_prec`.

`class sage.structure.nonexact.`**`Nonexact`**`(prec=20)`

Bases: `object`

A non-exact object with default precision.

INPUT:

- `prec` – a non-negative integer representing the default precision of `self` (default: `20`)

`default_prec()`

Return the default precision for `self`.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: R = QQ[[x]]
sage: R.default_prec()
20
```

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=10)
sage: R.default_prec()
10
```

2.1.4 Global options

The *GlobalOptions* class provides a generic mechanism for setting and accessing **global** options for parents in one or several related classes, typically for customizing the representation of their elements. This class will eventually also support setting options on a parent by parent basis.

These options should be "attached" to one or more classes as an options method.

See also:

For good examples of *GlobalOptions* in action see `sage.combinat.partition.Partitions.options` and `sage.combinat.tableau.Tableaux.options`.

Construction of options classes

The general setup for creating a set of global options is:

```
sage: from sage.structure.global_options import GlobalOptions
sage: class MyOptions(GlobalOptions):
....:     '''
....:     Nice options
....:
....:     @OPTIONS@
....:     '''
....:     NAME = 'option name'
....:     module = 'sage.some_module.some_file'
....:     option_class = 'name_of_class_controlled_by_options'
```

(continues on next page)

(continued from previous page)

```
....:     first_option = dict(default='with_bells',
....:                        description='Changes the functionality of _repr_',
....:                        values=dict(with_bells='causes _repr_ to print with bells',
....:                                    with_whistles='causes _repr_ to print with␣
↪whistles'),
....:                        alias=dict(bells='option1', whistles='option2'))
....:     # second_option = dict(...)
....:     # third_option = dict(...)
```

Note the syntax using the `class` keyword. However, because of some metaclass magic, the resulting `MyOptions` object becomes an instance of `GlobalOptions` instead of a subclass. So, despite the `class` syntax, `MyOptions` is not a class.

The options constructed by *`GlobalOptions`* have to be explicitly associated to the class that they control using the following arguments:

- `NAME` – A descriptive name for the options class. This is optional; the default is the name of the constructed class.
- `module` – The sage module containing the options class (optional)
- `option_class` – The name of the options class. This is optional and defaults to `NAME` if not explicitly set.

It is only possible to pickle a *`GlobalOptions`* class if the corresponding module is specified *and* if the options are explicitly attached to the corresponding class as a *options* method.

Each option is specified as a dictionary which describes the possible values for the option and its documentation. The possible entries in this dictionary are:

- `alias` – Allows for several option values to do the same thing.
- `alt_name` – An alternative name for this option.
- `checker` – A validation function which returns whether a user supplied value is valid or not. This is typically useful for large lists of legal values such as `NN`.
- `default` – Gives the default value for the option.
- `description` – A one line description of the option.
- `link_to` – Links this option to another one in another set of global options. This is used for example to allow `Partitions` and `Tableaux` to share the same `convention` option.
- `setter` – A function which is called **after** the value of the option is changed.
- `values` – A dictionary assigning each valid value for the option to a short description of what it does.
- `case_sensitive` – (Default: `True`) `True` or `False` depending on whether the values of the option are case sensitive.

For each option, either a complete list of possible values, via `values`, or a validation function, via `checker`, must be given. The values can be quite arbitrary, including user-defined functions which customize the default behaviour of the classes such as the output of `_repr_` or `latex()`. See *Dispatchers* below, and `_dispatcher()`, for more information.

The documentation for the options is automatically constructed from the docstring of the class by replacing the magic word `@OPTIONS@` with a description of each option.

The basic structure for defining a *`GlobalOptions`* class is best illustrated by an example:

```
sage: from sage.structure.global_options import GlobalOptions
sage: class Menu():
....:     class options(GlobalOptions):
....:         '''
```

(continues on next page)

(continued from previous page)

```
....:         Fancy documentation
....:         -------------------
....:
....:         @OPTIONS@
....:
....:         The END!
....:         '''
....:         NAME = 'menu'
....:         entree = dict(default='soup',
....:                     description='The first course of a meal',
....:                     values=dict(soup='soup of the day', bread='oven baked'),
....:                     alias=dict(rye='bread'))
....:         appetizer = dict(alt_name='entree')
....:         main = dict(default='pizza', description='Main meal',
....:                   values=dict(pizza='thick crust', pasta='penne arrabiata'),
....:                   case_sensitive=False)
....:         dessert = dict(default='espresso', description='Dessert',
....:                      values=dict(espresso='life begins again',
....:                                  cake='waist begins again',
....:                                  cream='fluffy, white stuff'))
....:         tip = dict(default=10, description='Reward for good service',
....:                    checker = lambda tip: tip in range(0,20))
sage: Menu.options
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
```

In the examples above, the options are constructed when the `options` object is created. However, it is also possible to construct the options dynamically using the `GlobalOptions._add_to_options()` methods.

For more details see *GlobalOptions*.

Accessing and setting option values

All options and their values, when they are strings, are forced to be lower case. The values of an options class can be set and accessed by calling the class or by treating the class as an array.

Continuing the example from *Construction of options classes*:

```
sage: Menu.options
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
sage: Menu.options.dessert
espresso
sage: Menu.options.dessert = 'cake'
sage: Menu.options.dessert
cake
```

Note that, provided there is no ambiguity, options and their values can be abbreviated:

```
sage: Menu.options('d')
'cake'
sage: Menu.options('m','t',des='esp', ent='sou')  # get and set several values at once
['pizza', 10]
sage: Menu.options(t=15)
sage: Menu.options('tip')
15
sage: Menu.options.tip
15
sage: Menu.options(e='s', m='Pi'); Menu.options()
Current options for menu
  - dessert: cake
  - entree:  soup
  - main:    pizza
  - tip:     15
sage: Menu.options(m='P')
Traceback (most recent call last):
...
ValueError: P is not a valid value for main in the options for menu
```

Setter functions

Each option of a *GlobalOptions* can be equipped with an optional setter function which is called **after** the value of the option is changed. In the following example, setting the option 'add' changes the state of the class by setting an attribute in this class using a `classmethod()`. Note that the options object is inserted after the creation of the class in order to access the `classmethod()` as `A.setter`:

```
sage: from sage.structure.global_options import GlobalOptions
sage: class A(SageObject):
....:     state = 0
....:     @classmethod
....:     def setter(cls, option, val):
....:         cls.state += int(val)
sage: class options(GlobalOptions):
....:     NAME = "A"
....:     add = dict(default=1,
....:                checker=lambda v: int(v)>0,
....:                description='An option with a setter',
....:                setter=A.setter)
sage: A.options = options
sage: A.options
Current options for A
- add: 1
sage: a = A(); a.state
1
sage: a.options()
Current options for A
- add: 1
sage: a.options(add=4)
sage: a.state
```

(continues on next page)

(continued from previous page)

```
5
sage: a.options()
Current options for A
- add: 4
```

Documentation for options

The documentation for a *GlobalOptions* is automatically generated from the supplied options. For example, the generated documentation for the options menu defined in *Construction of options classes* is the following:

```
Fancy documentation
-------------------

OPTIONS:

- ``appetizer`` -- alternative name for ``entree``
- ``dessert`` -- (default: ``espresso``)
  Dessert

  - ``cake``     -- waist begins again
  - ``cream``    -- fluffy, white stuff
  - ``espresso`` -- life begins again

- ``entree`` -- (default: ``soup``)
  The first course of a meal

  - ``bread`` -- oven baked
  - ``rye``   -- alias for ``bread``
  - ``soup``  -- soup of the day

- ``main`` -- (default: ``pizza``)
  Main meal

  - ``pasta`` -- penne arrabiata
  - ``pizza`` -- thick crust

- ``tip`` -- (default: ``10``)
  Reward for good service



The END!

See :class:`~sage.structure.global_options.GlobalOptions` for more features of these␣
↪options.
```

In addition, help on each option, and its list of possible values, can be obtained by (trying to) set the option equal to '?':

```
sage: Menu.options.dessert?                # not tested
- ``dessert`` -- (default: ``espresso``)
```

(continues on next page)

(continued from previous page)

```
Dessert

- ``cake``     -- waist begins again
- ``cream``    -- fluffy, white stuff
- ``espresso`` -- life begins again
```

Dispatchers

The whole idea of a *GlobalOptions* class is that the options change the default behaviour of the associated classes. This can be done either by simply checking what the current value of the relevant option is. Another possibility is to use the options class as a dispatcher to associated methods. To use the dispatcher feature of a *GlobalOptions* class it is necessary to implement separate methods for each value of the option where the naming convention for these methods is that they start with a common prefix and finish with the value of the option.

If the value of a dispatchable option is set equal to a (user defined) function then this function is called instead of a class method.

For example, the options `MyOptions` can be used to dispatch the `_repr_` method of the associated class `MyClass` as follows:

```
class MyClass(...):
    def _repr_(self):
        return self.options._dispatch(self,'_repr_','first_option')
    def _repr_with_bells(self):
        print('Bell!')
    def _repr_with_whistles(self):
        print('Whistles!')
class MyOptions(GlobalOptions):
    ...
```

In this example, `first_option` is an option of `MyOptions` which takes values `bells`, `whistles`, and so on. Note that it is necessary to make `self`, which is an instance of `MyClass`, an argument of the dispatcher because `_dispatch()` is a method of *GlobalOptions* and not a method of `MyClass`. Apart from `MyOptions`, as it is a method of this class, the arguments are the attached class (here `MyClass`), the prefix of the method of `MyClass` being dispatched, the option of `MyOptions` which controls the dispatching. All other arguments are passed through to the corresponding methods of `MyClass`. In general, a dispatcher is invoked as:

```
self.options._dispatch(self, dispatch_to, option, *args, **kargs)
```

Usually this will result in the method `dispatch_to + '_' + MyOptions(options)` of `self` being called with arguments `*args` and `**kargs` (if `dispatch_to[-1] == '_'` then the method `dispatch_to + MyOptions(options)` is called).

If `MyOptions(options)` is itself a function then the dispatcher will call this function instead. In this way, it is possible to allow the user to customise the default behaviour of this method. See `_dispatch()` for an example of how this can be achieved.

The dispatching capabilities of *GlobalOptions* allows options to be applied automatically without needing to parse different values of the option (the cost is that there must be a method for each value). The dispatching capabilities can also be used to make one option control several methods:

```
def __le__(self, other):
    return self.options._dispatch(self, '_le_','cmp', other)
```

(continues on next page)

(continued from previous page)

```
def __ge__(self, other):
    return self.options._dispatch(self, '_ge_','cmp', other)
def _le_option_a(self, other):
    return ...
def _ge_option_a(self, other):
    return ...
def _le_option_b(self, other):
    return ...
def _ge_option_b(self, other):
    return ...
```

See `_dispatch()` for more details.

Doc testing

All of the options and their effects should be doc-tested. However, in order not to break other tests, all options should be returned to their default state at the end of each test. To make this easier, every *GlobalOptions* class has a `_reset()` method for doing exactly this.

Pickling

Options classes can only be pickled if they are the options for some standard sage class. In this case the class is specified using the arguments to *GlobalOptions*. For example `options()` is defined as:

```
class Partitions(UniqueRepresentation, Parent):
    ...
    class options(GlobalOptions):
        NAME = 'Partitions'
        module = 'sage.combinat.partition'
        ...
```

Here is an example to test the pickling of a *GlobalOptions* instance:

```
sage: TestSuite(Partitions.options).run()                                    #␣
↪needs sage.combinat
```

AUTHORS:

- Andrew Mathas (2013): initial version
- **Andrew Mathas (2016): overhaul making the options attributes, enabling**
  pickling and attaching the options to a class.
- Jeroen Demeyer (2017): use subclassing to create instances

class sage.structure.global_options.**GlobalOptions**(*NAME=None*, *module=''*, *option_class=''*, *doc=''*, *end_doc=''*, ***options*)

Bases: `object`

The *GlobalOptions* class is a generic class for setting and accessing global options for Sage objects.

While it is possible to create instances of *GlobalOptions* the usual way, the recommended syntax is to subclass from `GlobalOptions`. Thanks to some metaclass magic, this actually creates an instance of `GlobalOptions` instead of a subclass.

INPUT (as "attributes" of the class):

- `NAME` – specifies a name for the options class (optional; default: class name)
- `module` – gives the module that contains the associated options class
- `option_class` – gives the name of the associated module class (default: `NAME`)
- option = `dict(...)` – dictionary specifying an option

The options are specified by keyword arguments with their values being a dictionary which describes the option. The allowed/expected keys in the dictionary are:

- `alias` – defines alias/synonym for option values
- `alt_name` – alternative name for an option
- `checker` – a function for checking whether a particular value for the option is valid
- `default` – the default value of the option
- `description` – documentation string
- `link_to` – links to an option for this set of options to an option in another *GlobalOptions*
- `setter` – a function (class method) which is called whenever this option changes
- `values` – a dictionary of the legal values for this option (this automatically defines the corresponding `checker`); this dictionary gives the possible options, as keys, together with a brief description of them
- `case_sensitive` – (default: `True`) `True` or `False` depending on whether the values of the option are case sensitive

Options and their values can be abbreviated provided that this abbreviation is a prefix of a unique option.

EXAMPLES:

```
sage: from sage.structure.global_options import GlobalOptions
sage: class Menu():
....:     class options(GlobalOptions):
....:         '''
....:         Fancy documentation
....:         -------------------
....:
....:         @OPTIONS@
....:
....:         End of Fancy documentation
....:         '''
....:         NAME = 'menu'
....:         entree = dict(default='soup',
....:                     description='The first course of a meal',
....:                     values=dict(soup='soup of the day', bread='oven baked'),
....:                     alias=dict(rye='bread'))
....:         appetizer = dict(alt_name='entree')
....:         main = dict(default='pizza', description='Main meal',
....:                   values=dict(pizza='thick crust', pasta='penne arrabiata'),
....:                   case_sensitive=False)
....:         dessert = dict(default='espresso', description='Dessert',
....:                      values=dict(espresso='life begins again',
....:                                  cake='waist begins again',
....:                                  cream='fluffy white stuff'))
```

(continues on next page)

(continued from previous page)

```
....:         tip = dict(default=10, description='Reward for good service',
....:                  checker=lambda tip: tip in range(0,20))
sage: Menu.options
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
sage: Menu.options(entree='s')         # unambiguous abbreviations are allowed
sage: Menu.options(t=15)
sage: (Menu.options['tip'], Menu.options('t'))
(15, 15)
sage: Menu.options()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     15
sage: Menu.options._reset(); Menu.options()
Current options for menu
  - dessert: espresso
  - entree:  soup
  - main:    pizza
  - tip:     10
sage: Menu.options.tip=40
Traceback (most recent call last):
...
ValueError: 40 is not a valid value for tip in the options for menu
sage: Menu.options(m='p')           # ambiguous abbreviations are not allowed
Traceback (most recent call last):
...
ValueError: p is not a valid value for main in the options for menu
```

The documentation for the options class is automatically generated from the information which specifies the options:

```
Fancy documentation
-------------------

OPTIONS:

- dessert:  (default: espresso)
  Dessert

  - ``cake``     -- waist begins again
  - ``cream``    -- fluffy white stuff
  - ``espresso`` -- life begins again

- entree:  (default: soup)
  The first course of a meal

  - ``bread`` -- oven baked
```

(continues on next page)

(continued from previous page)

```
  - ``rye``   -- alias for bread
  - ``soup``  -- soup of the day

- main:  (default: pizza)
  Main meal

  - ``pasta`` -- penne arrabiata
  - ``pizza`` -- thick crust

- tip:  (default: 10)
  Reward for good service

End of Fancy documentation

See :class:`~sage.structure.global_options.GlobalOptions` for more features of␣
↪these options.
```

The possible values for an individual option can be obtained by (trying to) set it equal to '?':

```
sage: Menu.options(des='?')
- ``dessert`` -- (default: ``espresso``)
  Dessert

  - ``cake``     -- waist begins again
  - ``cream``    -- fluffy white stuff
  - ``espresso`` -- life begins again

Current value: espresso
```

class sage.structure.global_options.**GlobalOptionsMeta**(*name*, *bases*, *dict*)

Bases: type

Metaclass for *GlobalOptions*

This class is itself an instance of *GlobalOptionsMetaMeta*, which implements the subclass magic.

class sage.structure.global_options.**GlobalOptionsMetaMeta**

Bases: type

class sage.structure.global_options.**Option**(*options*, *name*)

Bases: object

An option.

Each option for an options class is an instance of this class which implements the magic that allows the options to the attributes of the options class that can be looked up, set and called.

By way of example, this class implements the following functionality.

EXAMPLES:

```
sage: # needs sage.combinat
sage: Partitions.options.display
list
sage: Partitions.options.display = 'compact'
```

(continues on next page)

(continued from previous page)

```
sage: Partitions.options.display('list')
sage: Partitions.options._reset()
```

2.2 Old-Style Parents (Deprecated)

2.2.1 Base class for old-style parent objects

CLASS HIERARCHY:

SageObject

Parent

ParentWithBase

ParentWithGens

`class sage.structure.parent_old.Parent`

Bases: *Parent*

Parents are the Sage / mathematical analogues of container objects in computer science.

2.2.2 Base class for old-style parent objects with a base ring

`class sage.structure.parent_base.ParentWithBase`

Bases: *Parent*

This class is being deprecated, see parent.Parent for the new model.

`base_extend`(X)

2.2.3 Base class for old-style parent objects with generators

Note: This class is being deprecated, see `sage.structure.parent.Parent` and `sage.structure.category_object.CategoryObject` for the new model.

Many parent objects in Sage are equipped with generators, which are special elements of the object. For example, the polynomial ring $\mathbf{Z}[x, y, z]$ is generated by x, y, and z. In Sage the i^{th} generator of an object `X` is obtained using the notation `X.gen(i)`. From the Sage interactive prompt, the shorthand notation `X.i` is also allowed.

REQUIRED: A class that derives from ParentWithGens *must* define the ngens() and gen(i) methods.

OPTIONAL: It is also good if they define gens() to return all gens, but this is not necessary.

The `gens` function returns a tuple of all generators, the `ngens` function returns the number of generators.

The `_assign_names` functions is for internal use only, and is called when objects are created to set the generator names. It can only be called once.

The following examples illustrate these functions in the context of multivariate polynomial rings and free modules.

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 3, 'x')
sage: R.ngens()
3
sage: R.gen(0)
x0
sage: R.gens()
(x0, x1, x2)
sage: R.variable_names()
('x0', 'x1', 'x2')
```

This example illustrates generators for a free module over **Z**.

```
sage: # needs sage.modules
sage: M = FreeModule(ZZ, 4)
sage: M
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: M.ngens()
4
sage: M.gen(0)
(1, 0, 0, 0)
sage: M.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

class `sage.structure.parent_gens.`**`ParentWithGens`**

Bases: *`ParentWithBase`*

EXAMPLES:

```
sage: from sage.structure.parent_gens import ParentWithGens
sage: class MyParent(ParentWithGens):
....:     def ngens(self): return 3
sage: P = MyParent(base=QQ, names='a,b,c', normalize=True, category=Groups())
sage: P.category()
Category of groups
sage: P._names
('a', 'b', 'c')
```

gen(*i=0*)

gens()

Return a tuple whose entries are the generators for this object, in order.

hom(*im_gens*, *codomain=None*, *base_map=None*, *category=None*, *check=True*)

Return the unique homomorphism from `self` to codomain that sends `self.gens()` to the entries of `im_gens` and induces the map `base_map` on the base ring.

This raises a `TypeError` if there is no such homomorphism.

INPUT:

- `im_gens` – the images in the codomain of the generators of this object under the homomorphism
- `codomain` – the codomain of the homomorphism
- `base_map` – a map from the base ring of the domain into something that coerces into the codomain
- `category` – the category of the resulting morphism

- check – whether to verify that the images of generators extend to define a map (using only canonical coercions)

OUTPUT:

- a homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of im_gens, in which case return the (if it exists) natural map to X.

EXAMPLES: Polynomial Ring We first illustrate construction of a few homomorphisms involving a polynomial ring.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
Traceback (most recent call last):
...
ValueError: relations do not all (canonically) map to 0
under map determined by images of generators

sage: # needs sage.rings.finite_rings
sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49, 'a'))
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
  Defn: x |--> 3
sage: f(x + 6)
2
sage: f(x^2 + 1)
3
```

EXAMPLES: Natural morphism

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a TypeError exception is raised.

```
sage: QQ.hom(ZZ)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: natural coercion morphism from Rational Field to Integer Ring not␣
↪defined
```

You can specify a map on the base ring:

```
sage: # needs sage.rings.finite_rings
sage: k = GF(2)
sage: R.<a> = k[]
sage: l.<a> = k.extension(a^3 + a^2 + 1)
sage: R.<b> = l[]
sage: m.<b> = l.extension(b^2 + b + a)
sage: n.<z> = GF(2^6)
sage: m.hom([z^4 + z^3 + 1], base_map=l.hom([z^5 + z^4 + z^2]))
Ring morphism:
  From: Univariate Quotient Polynomial Ring in b over
        Finite Field in a of size 2^3 with modulus b^2 + b + a
  To:   Finite Field in z of size 2^6
  Defn: b |--> z^4 + z^3 + 1
        with map of base ring
```

ngens()

class sage.structure.parent_gens.**localvars**

Bases: `object`

Context manager for safely temporarily changing the variables names of an object with generators.

Objects with named generators are globally unique in Sage. Sometimes, though, it is very useful to be able to temporarily display the generators differently. The new Python `with` statement and the localvars context manager make this easy and safe (and fun!)

Suppose X is any object with generators. Write

```
with localvars(X, names[, latex_names] [,normalize=False]):
     some code
     ...
```

and the indented code will be run as if the names in X are changed to the new names. If you give normalize=True, then the names are assumed to be a tuple of the correct number of strings.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: with localvars(R, 'z,w'):
....:     print(x^3 + y^3 - x*y)
z^3 + w^3 - z*w
```

Note: I wrote this because it was needed to print elements of the quotient of a ring R by an ideal I using the print function for elements of R. See the code in `quotient_ring_element.pyx`.

AUTHOR:

- William Stein (2006-10-31)

2.2.4 Pure python code for abstract base class for objects with generators

`sage.structure.gens_py.abelian_iterator(M)`

`sage.structure.gens_py.multiplicative_iterator(M)`

CHAPTER

THREE

ELEMENTS

3.1 Elements

AUTHORS:

- David Harvey (2006-10-16): changed CommutativeAlgebraElement to derive from CommutativeRingElement instead of AlgebraElement
- David Harvey (2006-10-29): implementation and documentation of new arithmetic architecture
- William Stein (2006-11): arithmetic architecture – pushing it through to completion.
- Gonzalo Tornaria (2007-06): recursive base extend for coercion – lots of tests
- Robert Bradshaw (2007-2010): arithmetic operators and coercion
- Maarten Derickx (2010-07): added architecture for is_square and sqrt
- Jeroen Demeyer (2016-08): moved all coercion to the base class *Element*, see github issue #20767

3.1.1 The Abstract Element Class Hierarchy

This is the abstract class hierarchy, i.e., these are all abstract base classes.

```
SageObject
    Element
        ModuleElement
            RingElement
                CommutativeRingElement
                    IntegralDomainElement
                        DedekindDomainElement
                            PrincipalIdealDomainElement
                                EuclideanDomainElement
                    FieldElement
                    CommutativeAlgebraElement
                    Expression
                AlgebraElement
                    Matrix
                InfinityElement
            AdditiveGroupElement
            Vector

        MonoidElement
```

(continues on next page)

(continued from previous page)

```
          MultiplicativeGroupElement
    ElementWithCachedMethod
```

3.1.2 How to Define a New Element Class

Elements typically define a method `_new_c`, e.g.,

```
cdef _new_c(self, defining data):
    cdef FreeModuleElement_generic_dense x
    x = FreeModuleElement_generic_dense.__new__(FreeModuleElement_generic_dense)
    x._parent = self._parent
    x._entries = v
```

that creates a new sibling very quickly from defining data with assumed properties.

Arithmetic for Elements

Sage has a special system for handling arithmetic operations on Sage elements (that is instances of *`Element`*), in particular to manage uniformly mixed arithmetic operations using the `coercion model`. We describe here the rules that must be followed by both arithmetic implementers and callers.

A quick summary for the impatient

To implement addition for any *`Element`* subclass, override the `def _add_(self, other)` method instead of the usual Python `__add__` special method. Within `_add_(self, other)`, you may assume that `self` and `other` have the same parent.

If the implementation is generic across all elements in a given category C, then this method can be put in `C.ElementMethods`.

When writing *Cython* code, `_add_` should be a cpdef method: `cpdef _add_(self, other)`.

When doing arithmetic with two elements having different parents, the `coercion model` is responsible for "coercing" them to a common parent and performing arithmetic on the coerced elements.

Arithmetic in more detail

The aims of this system are to provide (1) an efficient calling protocol from both Python and Cython, (2) uniform coercion semantics across Sage, (3) ease of use, (4) readability of code.

We will take addition as an example; all other operators are similar. There are two relevant functions, with differing names (single vs. double underscores).

- **def Element.__add__(left, right)**

  This function is called by Python or Cython when the binary "+" operator is encountered. It assumes that at least one of its arguments is an *`Element`*.

  It has a fast pathway to deal with the most common case where both arguments have the same parent. Otherwise, it uses the coercion model to work out how to make them have the same parent. The coercion model then adds the coerced elements (technically, it calls `operator.add`). Note that the result of coercion is not required to be a Sage *`Element`*, it could be a plain Python type.

Note that, although this function is declared as `def`, it doesn't have the usual overheads associated with Python functions (either for the caller or for `__add__` itself). This is because Python has optimised calling protocols for such special functions.

- **def Element._add_(self, other)**

  This is the function that you should override to implement addition in a subclass of *Element*.

  The two arguments to this function are guaranteed to have the **same parent**, but not necessarily the same Python type.

  When implementing `_add_` in a Cython extension type, use `cpdef _add_` instead of `def _add_`.

  In Cython code, if you want to add two elements and you know that their parents are identical, you are encouraged to call this function directly, instead of using `x + y`. This only works if Cython knows that the left argument is an `Element`. You can always cast explicitly: `(<Element>x)._add_(y)` to force this. In plain Python, `x + y` is always the fastest way to add two elements because the special method `__add__` is optimized unlike the normal method `_add_`.

The difference in the names of the arguments (`left, right` versus `self, other`) is intentional: `self` is guaranteed to be an instance of the class in which the method is defined. In Cython, we know that at least one of `left` or `right` is an instance of the class but we do not know a priori which one.

Powering is a special case: first of all, the 3-argument version of `pow()` is not supported. Second, the coercion model checks whether the exponent looks like an integer. If so, the function `_pow_int` is called. If the exponent is not an integer, the arguments are coerced to a common parent and `_pow_` is called. So, if your type only supports powering to an integer exponent, you should implement only `_pow_int`. If you want to support arbitrary powering, implement both `_pow_` and `_pow_int`.

For addition, multiplication and powering (not for other operators), there is a fast path for operations with a C `long`. For example, implement `cdef _add_long(self, long n)` with optimized code for `self + n`. The addition and multiplication are assumed to be commutative, so they are also called for `n + self` or `n * self`. From Cython code, you can also call `_add_long` or `_mul_long` directly. This is strictly an optimization: there is a default implementation falling back to the generic arithmetic function.

Examples

We need some *Parent* to work with:

```
sage: from sage.structure.parent import Parent
sage: class ExampleParent(Parent):
....:     def __init__(self, name, **kwds):
....:         Parent.__init__(self, **kwds)
....:         self.rename(name)
```

We start with a very basic example of a Python class implementing `_add_`:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _add_(self, other):
....:         return 42
sage: p = ExampleParent("Some parent")
sage: x = MyElement(p)
sage: x + x
42
```

When two different parents are involved, this no longer works since there is no coercion:

```
sage: q = ExampleParent("Other parent")
sage: y = MyElement(q)
sage: x + y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Some parent' and 'Other parent'
```

If _add_ is not defined, an error message is raised, referring to the parents:

```
sage: x = Element(p)
sage: x._add_(x)
Traceback (most recent call last):
...
AttributeError: 'sage.structure.element.Element' object has no attribute '_add_'...
sage: x + x
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Some parent' and 'Some parent'
sage: y = Element(q)
sage: x + y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Some parent' and 'Other parent'
```

We can also implement arithmetic generically in categories:

```
sage: class MyCategory(Category):
....:     def super_categories(self):
....:         return [Sets()]
....:     class ElementMethods:
....:         def _add_(self, other):
....:             return 42
sage: p = ExampleParent("Parent in my category", category=MyCategory())
sage: x = Element(p)
sage: x + x
42
```

Implementation details

Implementing the above features actually takes a bit of magic. Casual callers and implementers can safely ignore it, but here are the details for the curious.

To achieve fast arithmetic, it is critical to have a fast path in Cython to call the `_add_` method of a Cython object. So we would like to declare `_add_` as a `cpdef` method of class *Element*. Remember however that the abstract classes coming from categories come after *Element* in the method resolution order (or fake method resolution order in case of a Cython class). Hence any generic implementation of `_add_` in such an abstract class would in principle be shadowed by `Element._add_`. This is worked around by defining `Element._add_` as a `cdef` instead of a `cpdef` method. Concrete implementations in subclasses should be `cpdef` or `def` methods.

Let us now see what happens upon evaluating `x + y` when `x` and `y` are instances of a class that does not implement `_add_` but where `_add_` is implemented in the category. First, `x.__add__(y)` is called, where `__add__` is implemented in *Element*. Assuming that `x` and `y` have the same parent, a Cython call to `x._add_(y)` will be done. The latter is implemented to trigger a Python level call to `x._add_(y)` which will succeed as desired.

In case that Python code calls `x._add_(y)` directly, `Element._add_` will be invisible, and the method lookup will continue down the MRO and find the `_add_` method in the category.

class sage.structure.element.**AdditiveGroupElement**

Bases: *ModuleElement*

Generic element of an additive group.

order()

Return additive order of element

class sage.structure.element.**AlgebraElement**

Bases: *RingElement*

class sage.structure.element.**CommutativeAlgebraElement**

Bases: *CommutativeRingElement*

class sage.structure.element.**CommutativeRingElement**

Bases: *RingElement*

Base class for elements of commutative rings.

divides(*x*)

Return `True` if `self` divides x.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(QQ)
sage: x.divides(x^2)
True
sage: x.divides(x^2 + 2)
False
sage: (x^2 + 2).divides(x)
False
sage: P.<x> = PolynomialRing(ZZ)
sage: x.divides(x^2)
True
sage: x.divides(x^2 + 2)
False
sage: (x^2 + 2).divides(x)
False
```

github issue #5347 has been fixed:

```
sage: K = GF(7)
sage: K(3).divides(1)
True
sage: K(3).divides(K(1))
True
```

```
sage: R = Integers(128)
sage: R(0).divides(1)
False
sage: R(0).divides(0)
True
sage: R(0).divides(R(0))
```

(continues on next page)

(continued from previous page)

```
True
sage: R(1).divides(0)
True
sage: R(121).divides(R(120))
True
sage: R(120).divides(R(121))
False
```

If x has different parent than self, they are first coerced to a common parent if possible. If this coercion fails, it returns a TypeError. This fixes github issue #5759.

```
sage: Zmod(2)(0).divides(Zmod(2)(0))
True
sage: Zmod(2)(0).divides(Zmod(2)(1))
False
sage: Zmod(5)(1).divides(Zmod(2)(1))
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Ring of integers modulo 5' and 'Ring of integers modulo 2'
sage: Zmod(35)(4).divides(Zmod(7)(1))
True
sage: Zmod(35)(7).divides(Zmod(7)(1))
False
```

inverse_mod(*I*)

Return an inverse of self modulo the ideal I, if defined, i.e., if I and self together generate the unit ideal.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(25)
sage: x = F.gen()
sage: z = F.zero()
sage: x.inverse_mod(F.ideal(z))
2*z2 + 3
sage: x.inverse_mod(F.ideal(1))
1
sage: z.inverse_mod(F.ideal(1))
1
sage: z.inverse_mod(F.ideal(z))
Traceback (most recent call last):
...
ValueError: an element of a proper ideal does not have an inverse modulo that␣
↪ideal
```

is_square(*root=False*)

Return whether or not the ring element self is a square.

If the optional argument root is True, then also return the square root (or None, if it is not a square).

INPUT:

- root - whether or not to also return a square root (default: False)

OUTPUT:

- `bool` – whether or not a square
- `object` – (optional) an actual square root if found, and `None` otherwise.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = 12*(x+1)^2 * (x+3)^2
sage: f.is_square()
False
sage: f.is_square(root=True)
(False, None)
sage: h = f/3
sage: h.is_square()
True
sage: h.is_square(root=True)
(True, 2*x^2 + 8*x + 6)
```

Note: This is the is_square implementation for general commutative ring elements. It's implementation is to raise a NotImplementedError. The function definition is here to show what functionality is expected and provide a general framework.

mod(*I*)

Return a representative for `self` modulo the ideal I (or the ideal generated by the elements of I if I is not an ideal.)

EXAMPLES: Integers Reduction of 5 modulo an ideal:

```
sage: n = 5
sage: n.mod(3*ZZ)
2
```

Reduction of 5 modulo the ideal generated by 3:

```
sage: n.mod(3)
2
```

Reduction of 5 modulo the ideal generated by 15 and 6, which is (3).

```
sage: n.mod([15,6])
2
```

EXAMPLES: Univariate polynomials

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^3 + x + 1
sage: f.mod(x + 1)
-1
```

Reduction for $\mathbf{Z}[x]$:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = x^3 + x + 1
```

(continues on next page)

(continued from previous page)

```
sage: f.mod(x + 1)
-1
```

When little is implemented about a given ring, then `mod` may simply return f.

EXAMPLES: Multivariate polynomials We reduce a polynomial in two variables modulo a polynomial and an ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: (x^2 + y^2 + z^2).mod(x + y + z)                                  #␣
↪needs sage.libs.singular
2*y^2 + 2*y*z + 2*z^2
```

Notice above that x is eliminated. In the next example, both y and z are eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x - y, y - z) )                           #␣
↪needs sage.libs.singular
3*z^2
sage: f = (x^2 + y^2 + z^2)^2; f
x^4 + 2*x^2*y^2 + y^4 + 2*x^2*z^2 + 2*y^2*z^2 + z^4
sage: f.mod( (x - y, y - z) )                                           #␣
↪needs sage.libs.singular
9*z^4
```

In this example y is eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x^3, y - z) )                             #␣
↪needs sage.libs.singular
x^2 + 2*z^2
```

sqrt(*extend=True*, *all=False*, *name=None*)

Compute the square root.

INPUT:

- **extend – boolean (default: True); whether to make a ring**
  extension containing a square root if `self` is not a square
- **all – boolean (default: False); whether to return a list of**
  all square roots or just a square root
- **name – required when extend=True and self is not a**
  square. This will be the name of the generator of the extension.

OUTPUT:

- if `all=False`, a square root; raises an error if `extend=False` and `self` is not a square
- if `all=True`, a list of all the square roots (empty if `extend=False` and `self` is not a square)

ALGORITHM:

It uses `is_square(root=true)` for the hard part of the work, the rest is just wrapper code.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: R.<x> = ZZ[]
```

(continues on next page)

(continued from previous page)

```
sage: (x^2).sqrt()
x
sage: f = x^2 - 4*x + 4; f.sqrt(all=True)
[x - 2, -x + 2]
sage: sqrtx = x.sqrt(name="y"); sqrtx
y
sage: sqrtx^2
x
sage: x.sqrt(all=true, name="y")
[y, -y]
sage: x.sqrt(extend=False, all=True)
[]
sage: x.sqrt()
Traceback (most recent call last):
...
TypeError: Polynomial is not a square. You must specify the name
of the square root when using the default extend = True
sage: x.sqrt(extend=False)
Traceback (most recent call last):
...
ValueError: trying to take square root of non-square x with extend = False
```

class sage.structure.element.**DedekindDomainElement**

Bases: *IntegralDomainElement*

class sage.structure.element.**Element**

Bases: *SageObject*

Generic element of a structure. All other types of elements (*RingElement*, *ModuleElement*, etc) derive from this type.

Subtypes must either call `__init__()` to set `_parent`, or may set `_parent` themselves if that would be more efficient.

richcmp(*left*, *right*, *op*)

Basic default implementation of rich comparisons for elements with equal parents.

It does a comparison by id for == and !=. Calling this default method with <, <=, > or >= will return `NotImplemented`.

EXAMPLES:

```
sage: from sage.structure.parent import Parent
sage: from sage.structure.element import Element
sage: P = Parent()
sage: e1 = Element(P); e2 = Element(P)
sage: e1 == e1    # indirect doctest
True
sage: e1 == e2    # indirect doctest
False
sage: e1 < e2     # indirect doctest
Traceback (most recent call last):
...
TypeError: '<' not supported between instances of 'sage.structure.element.
↪Element' and 'sage.structure.element.Element'
```

We now create an `Element` class where we define `_richcmp_` and check that comparison works:

```
sage: # needs sage.misc.cython
sage: cython(
....: '''
....: from sage.structure.richcmp cimport rich_to_bool
....: from sage.structure.element cimport Element
....: cdef class FloatCmp(Element):
....:     cdef float x
....:     def __init__(self, float v):
....:         self.x = v
....:     cpdef _richcmp_(self, other, int op):
....:         cdef float x1 = (<FloatCmp>self).x
....:         cdef float x2 = (<FloatCmp>other).x
....:         return rich_to_bool(op, (x1 > x2) - (x1 < x2))
....: ''')
sage: a = FloatCmp(1)
sage: b = FloatCmp(2)
sage: a <= b, b <= a
(True, False)
```

__add__(*left*, *right*)

Top-level addition operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _add_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: e + e
42
```

__sub__(*left*, *right*)

Top-level subtraction operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _sub_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: e - e
42
```

__neg__()

Top-level negation operator for `Element`.

EXAMPLES:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _neg_(self):
....:         return 42
sage: e = MyElement(Parent())
sage: -e
42
```

__mul__(*left*, *right*)

Top-level multiplication operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _mul_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: e * e
42
```

__truediv__(*left*, *right*)

Top-level true division operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: operator.truediv(2, 3)
2/3
sage: operator.truediv(pi, 3)                                              #␣
↪needs sage.symbolic
1/3*pi
sage: x = polygen(QQ, 'x')
sage: K.<i> = NumberField(x^2 + 1)                                         #␣
↪needs sage.rings.number_field
sage: operator.truediv(2, K.ideal(i + 1))                                  #␣
↪needs sage.rings.number_field
Fractional ideal (-i + 1)
```

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _div_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: operator.truediv(e, e)
42
```

__floordiv__(*left*, *right*)

Top-level floor division operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: 7 // 3
2
sage: 7 // int(3)
2
sage: int(7) // 3
2
```

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _floordiv_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: e // e
42
```

__mod__(*left*, *right*)

Top-level modulo operator for `Element` invoking the coercion model.

See *Arithmetic for Elements*.

EXAMPLES:

```
sage: 7 % 3
1
sage: 7 % int(3)
1
sage: int(7) % 3
1
```

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def _mod_(self, other):
....:         return 42
sage: e = MyElement(Parent())
sage: e % e
42
```

base_extend(*R*)

base_ring()

Return the base ring of this element's parent (if that makes sense).

category()

is_zero()

Return `True` if `self` equals `self.parent()(0)`.

The default implementation is to fall back to `not self.__bool__`.

> **Warning:** Do not re-implement this method in your subclass but implement `__bool__` instead.

n(*prec=None*, *digits=None*, *algorithm=None*)

Alias for `numerical_approx()`.

EXAMPLES:

```
sage: (2/3).n()                                                              #␣
↪needs sage.rings.real_mpfr
0.666666666666667
```

numerical_approx(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of `self` with `prec` bits (or decimal `digits`) of precision.

No guarantee is made about the accuracy of the result.

INPUT:

- `prec` – precision in bits
- `digits` – precision in decimal digits (only used if `prec` is not given)
- `algorithm` – which algorithm to use to compute this approximation (the accepted algorithms depend on the object)

If neither `prec` nor `digits` is given, the default precision is 53 bits (roughly 16 digits).

EXAMPLES:

```
sage: (2/3).numerical_approx()                                               #␣
↪needs sage.rings.real_mpfr
0.666666666666667
sage: pi.n(digits=10)                                                        #␣
↪needs sage.symbolic
3.141592654
sage: pi.n(prec=20)                                                          #␣
↪needs sage.symbolic
3.1416
```

parent(*x=None*)

Return the parent of this element; or, if the optional argument x is supplied, the result of coercing x into the parent of this element.

subs(*in_dict=None*, ***kwds*)

Substitutes given generators with given values while not touching other generators. This is a generic wrapper around `__call__`. The syntax is meant to be compatible with the corresponding method for symbolic expressions.

INPUT:

- `in_dict` - (optional) dictionary of inputs
- `**kwds` - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ,2,'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.subs({x:5})
```

(continues on next page)

(continued from previous page)

```
25*y^2 + y + 30
sage: f.subs(x=5)
25*y^2 + y + 30
sage: (1/f).subs(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).subs(x=4)
5
```

substitute(*in_dict=None*, ***kwds*)

This is an alias for self.subs().

INPUT:

- `in_dict` - (optional) dictionary of inputs
- `**kwds` - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ, 2, 'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.substitute({x: 5})
25*y^2 + y + 30
sage: f.substitute(x=5)
25*y^2 + y + 30
sage: (1/f).substitute(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).substitute(x=4)
5
```

class sage.structure.element.**ElementWithCachedMethod**

Bases: *Element*

An element class that fully supports cached methods.

NOTE:

The `cached_method` decorator provides a convenient way to automatically cache the result of a computation. Since github issue #11115, the cached method decorator applied to a method without optional arguments is faster than a hand-written cache in Python, and a cached method without any arguments (except `self`) is actually faster than a Python method that does nothing more but to return `1`. A cached method can also be inherited from the parent or element class of a category.

However, this holds true only if attribute assignment is supported. If you write an extension class in Cython that does not accept attribute assignment then a cached method inherited from the category will be slower (for *Parent*) or the cache would even break (for *Element*).

This class should be used if you write an element class, cannot provide it with attribute assignment, but want that it inherits a cached method from the category. Under these conditions, your class should inherit from this class rather than *Element*. Then, the cache will work, but certainly slower than with attribute assignment. Lazy attributes work as well.

EXAMPLES:

We define three element extension classes. The first inherits from *Element*, the second from this class, and the third simply is a Python class. We also define a parent class and, in Python, a category whose element and parent classes define cached methods.

```
sage: # needs sage.misc.cython
sage: cython_code = ["from sage.structure.element cimport Element,␣
↪ElementWithCachedMethod",
....:     "from sage.structure.richcmp cimport richcmp",
....:     "cdef class MyBrokenElement(Element):",
....:     "    cdef public object x",
....:     "    def __init__(self, P, x):",
....:     "        self.x = x",
....:     "        Element.__init__(self, P)",
....:     "    def __neg__(self):",
....:     "        return MyBrokenElement(self.parent(), -self.x)",
....:     "    def _repr_(self):",
....:     "        return '<%s>' % self.x",
....:     "    def __hash__(self):",
....:     "        return hash(self.x)",
....:     "    cpdef _richcmp_(left, right, int op):",
....:     "        return richcmp(left.x, right.x, op)",
....:     "    def raw_test(self):",
....:     "        return -self",
....:     "cdef class MyElement(ElementWithCachedMethod):",
....:     "    cdef public object x",
....:     "    def __init__(self, P, x):",
....:     "        self.x = x",
....:     "        Element.__init__(self, P)",
....:     "    def __neg__(self):",
....:     "        return MyElement(self.parent(), -self.x)",
....:     "    def _repr_(self):",
....:     "        return '<%s>' % self.x",
....:     "    def __hash__(self):",
....:     "        return hash(self.x)",
....:     "    cpdef _richcmp_(left, right, int op):",
....:     "        return richcmp(left.x, right.x, op)",
....:     "    def raw_test(self):",
....:     "        return -self",
....:     "class MyPythonElement(MyBrokenElement): pass",
....:     "from sage.structure.parent cimport Parent",
....:     "cdef class MyParent(Parent):",
....:     "    Element = MyElement"]
sage: cython('\n'.join(cython_code))
sage: cython_code = ["from sage.misc.cachefunc import cached_method",
....:     "from sage.misc.cachefunc import cached_in_parent_method",
....:     "from sage.categories.category import Category",
....:     "from sage.categories.objects import Objects",
....:     "class MyCategory(Category):",
....:     "    @cached_method",
....:     "    def super_categories(self):",
....:     "        return [Objects()]",
....:     "    class ElementMethods:",
```

(continues on next page)

(continued from previous page)

```
....:     "        @cached_method",
....:     "        def element_cache_test(self):",
....:     "            return -self",
....:     "        @cached_in_parent_method",
....:     "        def element_via_parent_test(self):",
....:     "            return -self",
....:     "    class ParentMethods:",
....:     "        @cached_method",
....:     "        def one(self):",
....:     "            return self.element_class(self,1)",
....:     "        @cached_method",
....:     "        def invert(self, x):",
....:     "            return -x"]
sage: cython('\n'.join(cython_code))
sage: C = MyCategory()
sage: P = MyParent(category=C)
sage: ebroken = MyBrokenElement(P, 5)
sage: e = MyElement(P, 5)
```

The cached methods inherited by `MyElement` works:

```
sage: # needs sage.misc.cython
sage: e.element_cache_test()
<-5>
sage: e.element_cache_test() is e.element_cache_test()
True
sage: e.element_via_parent_test()
<-5>
sage: e.element_via_parent_test() is e.element_via_parent_test()
True
```

The other element class can only inherit a `cached_in_parent_method`, since the cache is stored in the parent. In fact, equal elements share the cache, even if they are of different types:

```
sage: e == ebroken                                                         #␣
↪needs sage.misc.cython
True
sage: type(e) == type(ebroken)                                             #␣
↪needs sage.misc.cython
False
sage: ebroken.element_via_parent_test() is e.element_via_parent_test()     #␣
↪needs sage.misc.cython
True
```

However, the cache of the other inherited method breaks, although the method as such works:

```
sage: ebroken.element_cache_test()                                         #␣
↪needs sage.misc.cython
<-5>
sage: ebroken.element_cache_test() is ebroken.element_cache_test()         #␣
↪needs sage.misc.cython
False
```

Since e and ebroken share the cache, when we empty it for one element it is empty for the other as well:

```
sage: b = ebroken.element_via_parent_test()                               #␣
↪needs sage.misc.cython
sage: e.element_via_parent_test.clear_cache()                             #␣
↪needs sage.misc.cython
sage: b is ebroken.element_via_parent_test()                              #␣
↪needs sage.misc.cython
False
```

Note that the cache only breaks for elements that do no allow attribute assignment. A Python version of MyBrokenElement therefore allows for cached methods:

```
sage: epython = MyPythonElement(P, 5)                                     #␣
↪needs sage.misc.cython
sage: epython.element_cache_test()                                        #␣
↪needs sage.misc.cython
<-5>
sage: epython.element_cache_test() is epython.element_cache_test()        #␣
↪needs sage.misc.cython
True
```

class sage.structure.element.**EuclideanDomainElement**

Bases: *PrincipalIdealDomainElement*

degree()

leading_coefficient()

quo_rem(*other*)

class sage.structure.element.**Expression**

Bases: *CommutativeRingElement*

Abstract base class for Expression.

This class is defined for the purpose of isinstance() tests. It should not be instantiated.

EXAMPLES:

```
sage: isinstance(SR.var('y'), sage.structure.element.Expression)          #␣
↪needs sage.symbolic
True
```

By design, there is a unique direct subclass:

```
sage: len(sage.structure.element.Expression.__subclasses__()) <= 1
True
```

class sage.structure.element.**FieldElement**

Bases: *CommutativeRingElement*

divides(*other*)

Check whether self divides other, for field elements.

Since this is a field, all values divide all other values, except that zero does not divide any non-zero values.

EXAMPLES:

```
sage: # needs sage.rings.number_field sage.symbolic
sage: K.<rt3> = QQ[sqrt(3)]
sage: K(0).divides(rt3)
False
sage: rt3.divides(K(17))
True
sage: K(0).divides(K(0))
True
sage: rt3.divides(K(0))
True
```

is_unit()

Return `True` if `self` is a unit in its parent ring.

EXAMPLES:

```
sage: a = 2/3; a.is_unit()
True
```

On the other hand, 2 is not a unit, since its parent is $\mathbf{Z}$.

```
sage: a = 2; a.is_unit()
False
sage: parent(a)
Integer Ring
```

However, a is a unit when viewed as an element of QQ:

```
sage: a = QQ(2); a.is_unit()
True
```

quo_rem(*right*)

Return the quotient and remainder obtained by dividing `self` by `right`. Since this element lives in a field, the remainder is always zero and the quotient is `self/right`.

class sage.structure.element.**InfinityElement**

Bases: *RingElement*

class sage.structure.element.**IntegralDomainElement**

Bases: *CommutativeRingElement*

is_nilpotent()

class sage.structure.element.**Matrix**

Bases: *ModuleElement*

class sage.structure.element.**ModuleElement**

Bases: *Element*

Generic element of a module.

additive_order()

Return the additive order of self.

order()

Return the additive order of self.

class sage.structure.element.**ModuleElementWithMutability**

Bases: *ModuleElement*

Generic element of a module with mutability.

is_immutable()

Return True if this vector is immutable, i.e., the entries cannot be changed.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.is_immutable()                    # needs sage.modules
False
sage: v.set_immutable()                                                   # needs sage.modules
sage: v.is_immutable()                                                    # needs sage.modules
True
```

is_mutable()

Return True if this vector is mutable, i.e., the entries can be changed.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.is_mutable()                      # needs sage.modules
True
sage: v.set_immutable()                                                   # needs sage.modules
sage: v.is_mutable()                                                      # needs sage.modules
False
```

set_immutable()

Make this vector immutable. This operation can't be undone.

EXAMPLES:

```
sage: # needs sage.modules
sage: v = vector([1..5]); v
(1, 2, 3, 4, 5)
sage: v[1] = 10
sage: v.set_immutable()
sage: v[1] = 10
Traceback (most recent call last):
...
ValueError: vector is immutable; please change a copy instead (use copy())
```

class sage.structure.element.**MonoidElement**

Bases: *Element*

Generic element of a monoid.

multiplicative_order()

Return the multiplicative order of self.

order()

Return the multiplicative order of self.

powers(*n*)

Return the list $[x^0, x^1, \ldots, x^{n-1}]$.

EXAMPLES:

```
sage: G = SymmetricGroup(4)                                              #␣
↪needs sage.groups
sage: g = G([2, 3, 4, 1])                                                #␣
↪needs sage.groups
sage: g.powers(4)                                                        #␣
↪needs sage.groups
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
```

class sage.structure.element.**MultiplicativeGroupElement**

Bases: *MonoidElement*

Generic element of a multiplicative group.

order()

Return the multiplicative order of self.

class sage.structure.element.**PrincipalIdealDomainElement**

Bases: *DedekindDomainElement*

gcd(*right*)

Return the greatest common divisor of `self` and `other`.

lcm(*right*)

Return the least common multiple of `self` and `right`.

class sage.structure.element.**RingElement**

Bases: *ModuleElement*

abs()

Return the absolute value of `self`. (This just calls the `__abs__` method, so it is equivalent to the `abs()` built-in function.)

EXAMPLES:

```
sage: RR(-1).abs()
1.00000000000000
sage: ZZ(-1).abs()
1
sage: CC(I).abs()
1.00000000000000
sage: Mod(-15, 37).abs()
Traceback (most recent call last):
...
ArithmeticError: absolute value not defined on integers modulo n.
```

additive_order()

Return the additive order of `self`.

is_nilpotent()

Return `True` if `self` is nilpotent, i.e., some power of `self` is 0.

is_one()

is_prime()

Is `self` a prime element?

A *prime* element is a non-zero, non-unit element p such that, whenever p divides ab for some a and b, then p divides a or p divides b.

EXAMPLES:

For polynomial rings, prime is the same as irreducible:

```
sage: # needs sage.libs.singular
sage: R.<x,y> = QQ[]
sage: x.is_prime()
True
sage: (x^2 + y^3).is_prime()
True
sage: (x^2 - y^2).is_prime()
False
sage: R(0).is_prime()
False
sage: R(2).is_prime()
False
```

For the Gaussian integers:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: ZI = K.ring_of_integers()
sage: ZI(3).is_prime()
True
sage: ZI(5).is_prime()
False
sage: ZI(2 + i).is_prime()
True
sage: ZI(0).is_prime()
False
sage: ZI(1).is_prime()
False
```

In fields, an element is never prime:

```
sage: RR(0).is_prime()
False
sage: RR(2).is_prime()
False
```

For integers, *is_prime()* redefines prime numbers to be positive:

```
sage: (-2).is_prime()
False
sage: RingElement.is_prime(-2)                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.pari
True
```

Similarly, `NumberField` redefines *is_prime()* to determine primality in the ring of integers:

```
sage: # needs sage.rings.number_field
sage: (1 + i).is_prime()
True
sage: K(5).is_prime()
False
sage: K(7).is_prime()
True
sage: K(7/13).is_prime()
False
```

However, for rationals, *is_prime() does* follow the general definition of prime elements in a ring (i.e., always returns `False`) since the rationals are not a `NumberField` in Sage:

```
sage: QQ(7).is_prime()
False
```

multiplicative_order()

Return the multiplicative order of `self`, if `self` is a unit, or raise `ArithmeticError` otherwise.

powers(*n*)

Return the list $[x^0, x^1, \ldots, x^{n-1}]$.

EXAMPLES:

```
sage: 5.powers(3)
[1, 5, 25]
```

class sage.structure.element.**Vector**

Bases: *ModuleElementWithMutability*

sage.structure.element.**bin_op**(*x, y, op*)

sage.structure.element.**canonical_coercion**(*x, y*)

`canonical_coercion(x,y)` is what is called before doing an arithmetic operation between `x` and `y`. It returns a pair `(z,w)` such that `z` is got from `x` and `w` from `y` via canonical coercion and the parents of `z` and `w` are identical.

EXAMPLES:

```
sage: A = Matrix([[0, 1], [1, 0]])                                      #␣
↪needs sage.modules
sage: canonical_coercion(A, 1)                                          #␣
↪needs sage.modules
(
[0 1]  [1 0]
[1 0], [0 1]
)
```

sage.structure.element.**coerce_binop**(*method*)

Decorator for a binary operator method for applying coercion to the arguments before calling the method.

Consider a parent class in the category framework, S, whose element class expose a method $binop$. If a and b are elements of S, then $a.binop(b)$ behaves as expected. If a and b are not elements of S, but rather have a common parent T whose element class also exposes $binop$, we would rather expect $a.binop(b)$ to compute $aa.binop(bb)$, where $aa = T(a)$ and $bb = T(b)$. This decorator ensures that behaviour without having to otherwise modify the implementation of $binop$ on the element class of A.

Since coercion will be attempted on the arguments of the decorated method, a $TypeError$ will be thrown if there is no common parent between the elements. An $AttributeError$ or $NotImplementedError$ or similar will be thrown if there is a common parent of the arguments, but its element class does not implement a method of the same name as the decorated method.

EXAMPLES:

Sparse polynomial rings uses $@coerce_binop$ on gcd:

```
sage: S.<x> = PolynomialRing(ZZ, sparse=True)
sage: f = x^2
sage: g = x
sage: f.gcd(g)  #indirect doctest
x
sage: T = PolynomialRing(QQ, name='x', sparse=True)
sage: h = 1/2*T(x)
sage: u = f.gcd(h); u  #indirect doctest
x
sage: u.parent() == T
True
```

Another real example:

```
sage: R1 = QQ['x,y']
sage: R2 = QQ['x,y,z']
sage: f = R1(1)
sage: g = R1(2)
sage: h = R2(1)
sage: f.gcd(g)
1
sage: f.gcd(g, algorithm='modular')
1
sage: f.gcd(h)
1
sage: f.gcd(h, algorithm='modular')
1
sage: h.gcd(f)
1
sage: h.gcd(f, 'modular')
1
```

We demonstrate a small class using $@coerce_binop$ on a method:

```
sage: from sage.structure.element import coerce_binop
sage: class MyRational(Rational):
....:     def __init__(self,value):
....:         self.v = value
....:     @coerce_binop
....:     def test_add(self, other, keyword='z'):
....:         return (self.v, other, keyword)
```

Calls func directly if the two arguments have the same parent:

```
sage: x = MyRational(1)
sage: x.test_add(1/2)
(1, 1/2, 'z')
sage: x.test_add(1/2, keyword=3)
(1, 1/2, 3)
```

Passes through coercion and does a method lookup if the left operand is not the same. If the common parent's element class does not have a method of the same name, an exception is raised:

```
sage: x.test_add(2)
(1, 2, 'z')
sage: x.test_add(2, keyword=3)
(1, 2, 3)
sage: x.test_add(CC(2))
Traceback (most recent call last):
...
AttributeError: 'sage.rings.complex_mpfr.ComplexNumber' object has no attribute
↪'test_add'...
```

sage.structure.element.**coercion_traceback**(*dump=True*)

This function is very helpful in debugging coercion errors. It prints the tracebacks of all the errors caught in the coercion detection. Note that failure is cached, so some errors may be omitted the second time around (as it remembers not to retry failed paths for speed reasons.

For performance and caching reasons, exception recording must be explicitly enabled before using this function.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.record_exceptions()
sage: 1 + 1/5
6/5
sage: coercion_traceback()  # Should be empty, as all went well.
sage: 1/5 + GF(5).gen()
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Rational Field' and 'Finite Field of size 5'
sage: coercion_traceback()
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Rational Field' and 'Finite Field of size 5'
```

sage.structure.element.**get_coercion_model**()

Return the global coercion model.

EXAMPLES:

```
sage: import sage.structure.element as e
sage: cm = e.get_coercion_model()
sage: cm
<sage.structure.coerce.CoercionModel object at ...>
```

(continues on next page)

(continued from previous page)

```
sage: cm is coercion_model
True
```

sage.structure.element.**have_same_parent**(*left*, *right*)

Return True if and only if left and right have the same parent.

> **Warning:** This function assumes that at least one of the arguments is a Sage *Element*. When in doubt, use the slower parent(left) is parent(right) instead.

EXAMPLES:

```
sage: from sage.structure.element import have_same_parent
sage: have_same_parent(1, 3)
True
sage: have_same_parent(1, 1/2)
False
sage: have_same_parent(gap(1), gap(1/2))                                    #␣
↪needs sage.libs.gap
True
```

These have different types but the same parent:

```
sage: a = RLF(2)
sage: b = exp(a)
sage: type(a)
<... 'sage.rings.real_lazy.LazyWrapper'>
sage: type(b)
<... 'sage.rings.real_lazy.LazyNamedUnop'>
sage: have_same_parent(a, b)
True
```

sage.structure.element.**is_AdditiveGroupElement**(*x*)

Return True if x is of type AdditiveGroupElement.

sage.structure.element.**is_AlgebraElement**(*x*)

Return True if x is of type AlgebraElement.

sage.structure.element.**is_CommutativeAlgebraElement**(*x*)

Return True if x is of type CommutativeAlgebraElement.

sage.structure.element.**is_CommutativeRingElement**(*x*)

Return True if x is of type CommutativeRingElement.

sage.structure.element.**is_DedekindDomainElement**(*x*)

Return True if x is of type DedekindDomainElement.

sage.structure.element.**is_Element**(*x*)

Return True if x is of type Element.

EXAMPLES:

```
sage: from sage.structure.element import is_Element
sage: is_Element(2/3)
True
sage: is_Element(QQ^3)                                              #␣
↪needs sage.modules
False
```

sage.structure.element.**is_EuclideanDomainElement**(*x*)

Return True if x is of type EuclideanDomainElement.

sage.structure.element.**is_FieldElement**(*x*)

Return True if x is of type FieldElement.

sage.structure.element.**is_InfinityElement**(*x*)

Return True if x is of type InfinityElement.

sage.structure.element.**is_IntegralDomainElement**(*x*)

Return True if x is of type IntegralDomainElement.

sage.structure.element.**is_Matrix**(*x*)

sage.structure.element.**is_ModuleElement**(*x*)

Return True if x is of type ModuleElement.

This is even faster than using isinstance inline.

EXAMPLES:

```
sage: from sage.structure.element import is_ModuleElement
sage: is_ModuleElement(2/3)
True
sage: is_ModuleElement((QQ^3).0)                                    #␣
↪needs sage.modules
True
sage: is_ModuleElement('a')
False
```

sage.structure.element.**is_MonoidElement**(*x*)

Return True if x is of type MonoidElement.

sage.structure.element.**is_MultiplicativeGroupElement**(*x*)

Return True if x is of type MultiplicativeGroupElement.

sage.structure.element.**is_PrincipalIdealDomainElement**(*x*)

Return True if x is of type PrincipalIdealDomainElement.

sage.structure.element.**is_RingElement**(*x*)

Return True if x is of type RingElement.

sage.structure.element.**is_Vector**(*x*)

sage.structure.element.**make_element**(*_class*, *_dict*, *parent*)

This function is only here to support old pickles.

Pickling functionality is moved to Element.{__getstate__,__setstate__} functions.

sage.structure.element.**parent**(*x*)

Return the parent of the element `x`.

Usually, this means the mathematical object of which `x` is an element.

INPUT:

- `x` – an element

OUTPUT:

- If `x` is a Sage *Element*, return `x.parent()`.
- Otherwise, return `type(x)`.

See also:

Parents, Conversion and Coercion Section in the Sage Tutorial

EXAMPLES:

```
sage: a = 42
sage: parent(a)
Integer Ring
sage: b = 42/1
sage: parent(b)
Rational Field
sage: c = 42.0
sage: parent(c)                                                         #␣
↪needs sage.rings.real_mpfr
Real Field with 53 bits of precision
```

Some more complicated examples:

```
sage: x = Partition([3,2,1,1,1])                                        #␣
↪needs sage.combinat
sage: parent(x)                                                         #␣
↪needs sage.combinat
Partitions
sage: v = vector(RDF, [1,2,3])                                          #␣
↪needs sage.modules
sage: parent(v)                                                         #␣
↪needs sage.modules
Vector space of dimension 3 over Real Double Field
```

The following are not considered to be elements, so the type is returned:

```
sage: d = int(42)  # Python int
sage: parent(d)
<... 'int'>
sage: L = list(range(10))
sage: parent(L)
<... 'list'>
```

3.2 Element Wrapper

Wrapping Sage or Python objects as Sage elements.

AUTHORS:

- Nicolas Thiery (2008-2010): Initial version
- Travis Scrimshaw (2013-05-04): Cythonized version

class sage.structure.element_wrapper.**DummyParent**(*name*)

Bases: *UniqueRepresentation*, *Parent*

A class for creating dummy parents for testing *ElementWrapper*

class sage.structure.element_wrapper.**ElementWrapper**

Bases: *Element*

A class for wrapping Sage or Python objects as Sage elements.

EXAMPLES:

```
sage: from sage.structure.element_wrapper import DummyParent
sage: parent = DummyParent("A parent")
sage: o = ElementWrapper(parent, "bla"); o
'bla'
sage: isinstance(o, sage.structure.element.Element)
True
sage: o.parent()
A parent
sage: o.value
'bla'
```

Note that `o` is not *an instance of* `str`, but rather *contains a* `str`. Therefore, `o` does not inherit the string methods. On the other hand, it is provided with reasonable default implementations for equality testing, hashing, etc.

The typical use case of `ElementWrapper` is for trivially constructing new element classes from pre-existing Sage or Python classes, with a containment relation. Here we construct the tropical monoid of integers endowed with `min` as multiplication. There, it is desirable *not* to inherit the `factor` method from `Integer`:

```
sage: class MinMonoid(Parent):
....:     def _repr_(self):
....:         return "The min monoid"
....:
sage: M = MinMonoid()
sage: class MinMonoidElement(ElementWrapper):
....:     wrapped_class = Integer
....:
....:     def __mul__(self, other):
....:         return MinMonoidElement(self.parent(), min(self.value, other.value))
sage: x = MinMonoidElement(M, 5); x
5
sage: x.parent()
The min monoid
sage: x.value
5
sage: y = MinMonoidElement(M, 3)
```

(continues on next page)

(continued from previous page)

```
sage: x * y
3
```

This example was voluntarily kept to a bare minimum. See the examples in the categories (e.g. `Semigroups().example()`) for several full featured applications.

> **Warning:** Versions before github issue #14519 had parent as the second argument and the value as the first.

value

class sage.structure.element_wrapper.**ElementWrapperCheckWrappedClass**

Bases: *ElementWrapper*

An *element wrapper* such that comparison operations are done against subclasses of `wrapped_class`.

wrapped_class

alias of `object`

class sage.structure.element_wrapper.**ElementWrapperTester**

Bases: *ElementWrapper*

Test class for the default `__copy()` method of subclasses of *ElementWrapper*.

append(x)

3.3 Elements, Array and Lists With Clone Protocol

This module defines several classes which are subclasses of *Element* and which roughly implement the "prototype" design pattern (see [Prototype_pattern], [GHJV1994]). Those classes are intended to be used to model *mathematical* objects, which are by essence immutable. However, in many occasions, one wants to construct the data-structure encoding of a new mathematical object by small modifications of the data structure encoding some already built object. For the resulting data-structure to correctly encode the mathematical object, some structural invariants must hold. One problem is that, in many cases, during the modification process, there is no possibility but to break the invariants.

For example, in a list modeling a permutation of $\{1, \ldots, n\}$, the integers must be distinct. A very common operation is to take a permutation to make a copy with some small modifications, like exchanging two consecutive values in the list or cycling some values. Though the result is clearly a permutations there's no way to avoid breaking the permutations invariants at some point during the modifications.

The main purpose of this module is to define the class

- *ClonableElement* as an abstract super class,

and its subclasses:

- *ClonableArray* for arrays (lists of fixed length) of objects;
- *ClonableList* for (resizable) lists of objects;
- *NormalizedClonableList* for lists of objects with a normalization method;
- *ClonableIntArray* for arrays of int.

See also:

The following parents from *sage.structure.list_clone_demo* demonstrate how to use them:

- IncreasingArrays() (see *IncreasingArray* and the parent class *IncreasingArrays*)
- IncreasingLists() (see *IncreasingList* and the parent class *IncreasingLists*)
- SortedLists() (see *SortedList* and the parent class *SortedLists*)
- IncreasingIntArrays() (see *IncreasingIntArray* and the parent class *IncreasingIntArrays*)

EXAMPLES:

We now demonstrate how *IncreasingArray* works, creating an instance el through its parent IncreasingArrays():

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: P = IncreasingArrays()
sage: P([1, 4 ,8])
[1, 4, 8]
```

If one tries to create this way a list which in not increasing, an error is raised:

```
sage: IncreasingArrays()([5, 4 ,8])
Traceback (most recent call last):
...
ValueError: array is not increasing
```

Once created modifying el is forbidden:

```
sage: el = P([1, 4 ,8])
sage: el[1] = 3
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

However, you can modify a temporarily mutable clone:

```
sage: with el.clone() as elc:
....:      elc[1] = 3
sage: [el, elc]
[[1, 4, 8], [1, 3, 8]]
```

We check that the original and the modified copy now are in a proper immutable state:

```
sage: el.is_immutable(), elc.is_immutable()
(True, True)
sage: elc[1] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

You can break the property that the list is increasing during the modification:

```
sage: with el.clone() as elc2:
....:     elc2[1] = 12
....:     print(elc2)
....:     elc2[2] = 25
[1, 12, 8]
```

(continues on next page)

(continued from previous page)

```
sage: elc2
[1, 12, 25]
```

But this property must be restored at the end of the `with` block; otherwise an error is raised:

```
sage: with elc2.clone() as el3:
....:     el3[1] = 100
Traceback (most recent call last):
...
ValueError: array is not increasing
```

Finally, as an alternative to the `with` syntax one can use:

```
sage: el4 = copy(elc2)
sage: el4[1] = 10
sage: el4.set_immutable()
sage: el4.check()
```

REFERENCES:

- [Prototype_pattern]
- [GHJV1994]

AUTHORS:

- Florent Hivert (2010-03): initial revision

class `sage.structure.list_clone.ClonableArray`

Bases: *ClonableElement*

Array with clone protocol

The class of objects which are *Element* behave as arrays (i.e. lists of fixed length) and implement the clone protocol. See *ClonableElement* for details about clone protocol.

INPUT:

- `parent` – a *Parent*
- `lst` – a list
- `check` – a boolean specifying if the invariant must be checked using method *check()*.
- `immutable` – a boolean telling whether the created element is immutable (defaults to `True`)

See also:

IncreasingArray for an example of usage.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: IA = IncreasingArrays()
sage: ia1 = IA([1, 4, 6]); ia1
[1, 4, 6]
sage: with ia1.clone() as ia2:
....:     ia2[1] = 5
sage: ia2
[1, 5, 6]
```

(continues on next page)

(continued from previous page)

```
sage: with ia1.clone() as ia2:
....:     ia2[1] = 7
Traceback (most recent call last):
...
ValueError: array is not increasing
```

check()

Check that `self` fulfill the invariants

This is an abstract method. Subclasses are supposed to overload `check`.

EXAMPLES:

```
sage: from sage.structure.list_clone import ClonableArray
sage: ClonableArray(Parent(), [1,2,3]) # indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this should never be called, please overload the check␣
↪method
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,4]) # indirect doctest
```

count(*key*)

Return number of `i`'s for which `s[i] == key`

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: c = IncreasingArrays()([1,2,2,4])
sage: c.count(1)
1
sage: c.count(2)
2
sage: c.count(3)
0
```

index(*x*, *start=None*, *stop=None*)

Return the smallest `k` such that `s[k] == x` and `i <= k < j`

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: c = IncreasingArrays()([1,2,4])
sage: c.index(1)
0
sage: c.index(4)
2
sage: c.index(5)
Traceback (most recent call last):
...
ValueError: 5 is not in list
```

class `sage.structure.list_clone.`**ClonableElement**

Bases: *Element*

Abstract class for elements with clone protocol

This class is a subclass of *Element* and implements the "prototype" design pattern (see [Prototype_pattern], [GHJV1994]). The role of this class is:

- to manage copy and mutability and hashing of elements
- to ensure that at the end of a piece of code an object is restored in a meaningful mathematical state.

A class `C` inheriting from *ClonableElement* must implement the following two methods

- `obj.__copy__()` – returns a fresh copy of obj
- `obj.check()` – returns nothing, raise an exception if `obj` doesn't satisfy the data structure invariants

and ensure to call `obj._require_mutable()` at the beginning of any modifying method.

Additionally, one can also implement

- `obj._hash_()` – return the hash value of `obj`.

Then, given an instance `obj` of `C`, the following sequences of instructions ensures that the invariants of `new_obj` are properly restored at the end:

```
with obj.clone() as new_obj:
    ...
    # lot of invariant breaking modifications on new_obj
    ...
# invariants are ensured to be fulfilled
```

The following equivalent sequence of instructions can be used if speed is needed, in particular in Cython code:

```
new_obj = obj.__copy__()
...
# lot of invariant breaking modifications on new_obj
...
new_obj.set_immutable()
new_obj.check()
# invariants are ensured to be fulfilled
```

Finally, if the class implements the `_hash_` method, then *ClonableElement* ensures that the hash value can only be computed on an immutable object. It furthermore caches it so that it is only computed once.

Warning: for the hash caching mechanism to work correctly, the hash value cannot be 0.

EXAMPLES:

The following code shows a minimal example of usage of *ClonableElement*. We implement a class or pairs (x, y) such that $x < y$:

```
sage: from sage.structure.list_clone import ClonableElement
sage: class IntPair(ClonableElement):
....:     def __init__(self, parent, x, y):
....:         ClonableElement.__init__(self, parent=parent)
....:         self._x = x
....:         self._y = y
....:         self.set_immutable()
....:         self.check()
```

(continues on next page)

(continued from previous page)

```
....:       def _repr_(self):
....:           return "(x=%s, y=%s)"%(self._x, self._y)
....:       def check(self):
....:           if self._x >= self._y:
....:               raise ValueError("Incorrectly ordered pair")
....:       def get_x(self): return self._x
....:       def get_y(self): return self._y
....:       def set_x(self, v): self._require_mutable(); self._x = v
....:       def set_y(self, v): self._require_mutable(); self._y = v
```

Note: we don't need to define `__copy__` since it is properly inherited from *Element*.

We now demonstrate the behavior. Let's create an `IntPair`:

```
sage: myParent = Parent()
sage: el = IntPair(myParent, 1, 3); el
(x=1, y=3)
sage: el.get_x()
1
```

Modifying it is forbidden:

```
sage: el.set_x(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

However, you can modify a mutable copy:

```
sage: with el.clone() as el1:
....:       el1.set_x(2)
sage: [el, el1]
[(x=1, y=3), (x=2, y=3)]
```

We check that the original and the modified copy are in a proper immutable state:

```
sage: el.is_immutable(), el1.is_immutable()
(True, True)
sage: el1.set_x(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

A modification which doesn't restore the invariant $x < y$ at the end is illegal and raise an exception:

```
sage: with el.clone() as elc2:
....:       elc2.set_x(5)
Traceback (most recent call last):
...
ValueError: Incorrectly ordered pair
```

clone(*check=True*)

Return a clone that is mutable copy of `self`.

INPUT:

- `check` – a boolean indicating if `self.check()` must be called after modifications.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: with el.clone() as el1:
....:     el1[2] = 5
sage: el1
[1, 2, 5]
```

is_immutable()

Return `True` if `self` is immutable (cannot be changed) and `False` if it is not.

To make `self` immutable use `self.set_immutable()`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: el.is_immutable()
True
sage: copy(el).is_immutable()
False
sage: with el.clone() as el1:
....:     print([el.is_immutable(), el1.is_immutable()])
[True, False]
```

is_mutable()

Return `True` if `self` is mutable (can be changed) and `False` if it is not.

To make this object immutable use `self.set_immutable()`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
sage: el.is_mutable()
False
sage: copy(el).is_mutable()
True
sage: with el.clone() as el1:
....:     print([el.is_mutable(), el1.is_mutable()])
[False, True]
```

set_immutable()

Makes `self` immutable, so it can never again be changed.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: el = IncreasingArrays()([1,2,3])
```

(continues on next page)

(continued from previous page)

```
sage: el1 = copy(el); el1.is_mutable()
True
sage: el1.set_immutable();  el1.is_mutable()
False
sage: el1[2] = 4
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

class sage.structure.list_clone.**ClonableIntArray**

Bases: *ClonableElement*

Array of int with clone protocol

The class of objects which are *Element* behave as list of int and implement the clone protocol. See *ClonableElement* for details about clone protocol.

INPUT:

- parent – a *Parent*
- lst – a list
- check – a boolean specifying if the invariant must be checked using method *check()*
- immutable – a boolean telling whether the created element is immutable (defaults to True)

See also:

IncreasingIntArray for an example of usage.

check()

Check that self fulfill the invariants

This is an abstract method. Subclasses are supposed to overload check.

EXAMPLES:

```
sage: from sage.structure.list_clone import ClonableArray
sage: ClonableArray(Parent(), [1,2,3]) # indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this should never be called, please overload the check␣
↪method
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: el = IncreasingIntArrays()([1,2,4]) # indirect doctest
```

index(*item*)

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: c = IncreasingIntArrays()([1,2,4])
sage: c.index(1)
0
sage: c.index(4)
2
sage: c.index(5)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: list.index(x): x not in list
```

list()

Convert self into a Python list.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: I = IncreasingIntArrays()(range(5))
sage: I == list(range(5))
False
sage: I.list() == list(range(5))
True
sage: I = IncreasingIntArrays()(range(1000))
sage: I.list() == list(range(1000))
True
```

class sage.structure.list_clone.**ClonableList**

Bases: *ClonableArray*

List with clone protocol

The class of objects which are *Element* behave as lists and implement the clone protocol. See *ClonableElement* for details about clone protocol.

See also:

IncreasingList for an example of usage.

append(*el*)

Appends `el` to `self`

INPUT: `el` – any object

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1])
sage: el.append(3)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     elc.append(4)
....:     elc.append(6)
sage: elc
[1, 4, 6]
sage: with el.clone() as elc:
....:     elc.append(4)
....:     elc.append(2)
Traceback (most recent call last):
...
ValueError: array is not increasing
```

extend(*it*)

Extends `self` by the content of the iterable `it`

INPUT: `it` – any iterable

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.extend((10,11))
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.

sage: with el.clone() as elc:
....:     elc.extend((10,11))
sage: elc
[1, 4, 5, 8, 9, 10, 11]

sage: el2 = IncreasingLists()([15, 16])
sage: with el.clone() as elc:
....:     elc.extend(el2)
sage: elc
[1, 4, 5, 8, 9, 15, 16]

sage: with el.clone() as elc:
....:     elc.extend((6,7))
Traceback (most recent call last):
...
ValueError: array is not increasing
```

insert(*index*, *el*)

Inserts `el` in `self` at position `index`

INPUT:

- `el` – any object
- `index` – any int

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.insert(3, 6)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     elc.insert(3, 6)
sage: elc
[1, 4, 5, 6, 8, 9]
sage: with el.clone() as elc:
....:     elc.insert(2, 6)
Traceback (most recent call last):
...
ValueError: array is not increasing
```

pop(*index=-1*)

Remove `self[index]` from `self` and returns it

INPUT: `index` - any int, default to -1

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.pop()
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     print(elc.pop())
9
sage: elc
[1, 4, 5, 8]
sage: with el.clone() as elc:
....:     print(elc.pop(2))
5
sage: elc
[1, 4, 8, 9]
```

remove(*el*)

Remove the first occurrence of `el` from `self`

INPUT: `el` - any object

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: el = IncreasingLists()([1, 4, 5, 8, 9])
sage: el.remove(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: with el.clone() as elc:
....:     elc.remove(4)
sage: elc
[1, 5, 8, 9]
sage: with el.clone() as elc:
....:     elc.remove(10)
Traceback (most recent call last):
...
ValueError: list.remove(x): x not in list
```

class `sage.structure.list_clone.`**NormalizedClonableList**

Bases: `ClonableList`

List with clone protocol and normal form

This is a subclass of `ClonableList` which call a method `normalize()` at creation and after any modification of its instance.

See also:

`SortedList` for an example of usage.

EXAMPLES:

We construct a sorted list through its parent:

```
sage: from sage.structure.list_clone_demo import SortedLists
sage: SL = SortedLists()
sage: sl1 = SL([4,2,6,1]); sl1
[1, 2, 4, 6]
```

Normalization is also performed atfer modification:

```
sage: with sl1.clone() as sl2:
....:     sl2[1] = 12
sage: sl2
[1, 4, 6, 12]
```

normalize()

Normalize `self`

This is an abstract method. Subclasses are supposed to overload *normalize()*. The call `self.normalize()` is supposed to

- call `self._require_mutable()` to check that `self` is in a proper mutable state
- modify `self` to put it in a normal form

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: l = SortedList(SortedLists(), [2,3,2], False, False)
sage: l
[2, 2, 3]
sage: l.check()
Traceback (most recent call last):
...
ValueError: list is not strictly increasing
```

3.4 Elements, Array and Lists With Clone Protocol, demonstration classes

This module demonstrate the usage of the various classes defined in *list_clone*

class sage.structure.list_clone_demo.**IncreasingArray**

Bases: *ClonableArray*

A small extension class for testing *ClonableArray*.

check()

Check that `self` is increasing.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingArrays
sage: IncreasingArrays()([1,2,3]) # indirect doctest
[1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
sage: IncreasingArrays()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class sage.structure.list_clone_demo.**IncreasingArrays**

Bases: *UniqueRepresentation*, *Parent*

A small (incomplete) parent for testing *ClonableArray*

Element

alias of *IncreasingArray*

class sage.structure.list_clone_demo.**IncreasingIntArray**

Bases: *ClonableIntArray*

A small extension class for testing *ClonableIntArray*.

check()

Check that `self` is increasing.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingIntArrays
sage: IncreasingIntArrays()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingIntArrays()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class sage.structure.list_clone_demo.**IncreasingIntArrays**

Bases: *IncreasingArrays*

A small (incomplete) parent for testing *ClonableIntArray*

Element

alias of *IncreasingIntArray*

class sage.structure.list_clone_demo.**IncreasingList**

Bases: *ClonableList*

A small extension class for testing *ClonableList*

check()

Check that `self` is increasing

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import IncreasingLists
sage: IncreasingLists()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingLists()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: array is not increasing
```

class sage.structure.list_clone_demo.**IncreasingLists**

Bases: *IncreasingArrays*

A small (incomplete) parent for testing *ClonableList*

Element

alias of *IncreasingList*

class sage.structure.list_clone_demo.**SortedList**

Bases: *NormalizedClonableList*

A small extension class for testing *NormalizedClonableList*.

check()

Check that `self` is strictly increasing

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: SortedLists()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: SortedLists()([3,2,2]) # indirect doctest
Traceback (most recent call last):
...
ValueError: list is not strictly increasing
```

normalize()

Normalize `self`

Sort the list stored in `self`.

EXAMPLES:

```
sage: from sage.structure.list_clone_demo import SortedList, SortedLists
sage: l = SortedList(SortedLists(), [3,1,2], False, False)
sage: l         # indirect doctest
[1, 2, 3]
sage: l[1] = 5; l
[1, 5, 3]
sage: l.normalize(); l
[1, 3, 5]
```

class sage.structure.list_clone_demo.**SortedLists**

Bases: *IncreasingLists*

A small (incomplete) parent for testing *NormalizedClonableList*

Element

alias of *SortedList*

CHAPTER

FOUR

MATHEMATICAL DATA STRUCTURES

4.1 Formal sums

AUTHORS:

- David Harvey (2006-09-20): changed FormalSum not to derive from "list" anymore, because that breaks new Element interface
- Nick Alexander (2006-12-06): added test cases.
- William Stein (2006, 2009): wrote the first version in 2006, documented it in 2009.
- Volker Braun (2010-07-19): new-style coercions, documentation added. FormalSums now derives from UniqueRepresentation.

FUNCTIONS:

- `FormalSums(ring)` – create the module of formal finite sums with coefficients in the given ring.
- `FormalSum(list of pairs (coeff, number))` – create a formal sum.

EXAMPLES:

```
sage: A = FormalSum([(1, 2/3)]); A
2/3
sage: B = FormalSum([(3, 1/5)]); B
3*1/5
sage: -B
-3*1/5
sage: A + B
2/3 + 3*1/5
sage: A - B
2/3 - 3*1/5
sage: B*3
9*1/5
sage: 2*A
2*2/3
sage: list(2*A + A)
[(3, 2/3)]
```

class sage.structure.formal_sum.**FormalSum**(*x*, *parent=None*, *check=True*, *reduce=True*)

Bases: *ModuleElement*

A formal sum over a ring.

reduce()

EXAMPLES:

```
sage: a = FormalSum([(-2,3), (2,3)], reduce=False); a
-2*3 + 2*3
sage: a.reduce()
sage: a
0
```

class sage.structure.formal_sum.**FormalSums**

Bases: *UniqueRepresentation*, Module

The R-module of finite formal sums with coefficients in some ring R.

EXAMPLES:

```
sage: FormalSums()
Abelian Group of all Formal Finite Sums over Integer Ring
sage: FormalSums(ZZ)
Abelian Group of all Formal Finite Sums over Integer Ring
sage: FormalSums(GF(7))
Abelian Group of all Formal Finite Sums over Finite Field of size 7
sage: FormalSums(ZZ[sqrt(2)])                                              #␣
↪needs sage.rings.number_field sage.symbolic
Abelian Group of all Formal Finite Sums over Order in Number Field in sqrt2
 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
sage: FormalSums(GF(9,'a'))                                                #␣
↪needs sage.rings.finite_rings
Abelian Group of all Formal Finite Sums over Finite Field in a of size 3^2
```

Element

alias of *FormalSum*

base_extend(*R*)

EXAMPLES:

```
sage: F7 = FormalSums(ZZ).base_extend(GF(7)); F7
Abelian Group of all Formal Finite Sums over Finite Field of size 7
```

The following tests against a bug that was fixed at github issue #18795:

```
sage: isinstance(F7, F7.category().parent_class)
True
```

4.2 Factorizations

The *Factorization* class provides a structure for holding quite general lists of objects with integer multiplicities. These may hold the results of an arithmetic or algebraic factorization, where the objects may be primes or irreducible polynomials and the multiplicities are the (non-zero) exponents in the factorization. For other types of examples, see below.

Factorization class objects contain a `list`, so can be printed nicely and be manipulated like a list of prime-exponent pairs, or easily turned into a plain list. For example, we factor the integer -45:

```
sage: F = factor(-45)
```

This returns an object of type *Factorization*:

```
sage: type(F)
<class 'sage.structure.factorization_integer.IntegerFactorization'>
```

It prints in a nice factored form:

```
sage: F
-1 * 3^2 * 5
```

There is an underlying list representation, which ignores the unit part:

```
sage: list(F)
[(3, 2), (5, 1)]
```

A *Factorization* is not actually a list:

```
sage: isinstance(F, list)
False
```

However, we can access the *Factorization* F itself as if it were a list:

```
sage: F[0]
(3, 2)
sage: F[1]
(5, 1)
```

To get at the unit part, use the *Factorization.unit()* function:

```
sage: F.unit()
-1
```

All factorizations are immutable, up to ordering with `sort()` and simplifying with `simplify()`. Thus if you write a function that returns a cached version of a factorization, you do not have to return a copy.

```
sage: F = factor(-12); F
-1 * 2^2 * 3
sage: F[0] = (5,4)
Traceback (most recent call last):
...
TypeError: 'Factorization' object does not support item assignment
```

EXAMPLES:

This more complicated example involving polynomials also illustrates that the unit part is not discarded from factorizations:

```
sage: # needs sage.libs.pari
sage: x = QQ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
```

(continues on next page)

(continued from previous page)

```
(-5) * (x - 3) * (x - 2)
sage: F.unit()
-5
sage: F.value()
-5*x^2 + 25*x - 30
```

The underlying list is the list of pairs (p_i, e_i), where each p_i is a 'prime' and each e_i is an integer. The unit part is discarded by the list:

```
sage: # needs sage.libs.pari
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
sage: len(F)
2
sage: F[1]
(x - 2, 1)
```

In the ring $\mathbf{Z}[x]$, the integer -5 is not a unit, so the factorization has three factors:

```
sage: # needs sage.libs.pari
sage: x = ZZ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
(-1) * 5 * (x - 3) * (x - 2)
sage: F.universe()
Univariate Polynomial Ring in x over Integer Ring
sage: F.unit()
-1
sage: list(F)
[(5, 1), (x - 3, 1), (x - 2, 1)]
sage: F.value()
-5*x^2 + 25*x - 30
sage: len(F)
3
```

On the other hand, -1 is a unit in $\mathbf{Z}$, so it is included in the unit:

```
sage: # needs sage.libs.pari
sage: x = ZZ['x'].0
sage: f = -1 * (x-2) * (x-3)
sage: F = f.factor(); F
(-1) * (x - 3) * (x - 2)
sage: F.unit()
-1
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
```

Factorizations can involve fairly abstract mathematical objects:

```
sage: # needs sage.modular
sage: F = ModularSymbols(11,4).factorization(); F
```

(continues on next page)

(continued from previous page)

```
(Modular Symbols subspace of dimension 2 of Modular Symbols space
  of dimension 6 for Gamma_0(11) of weight 4 with sign 0 over Rational Field) *
(Modular Symbols subspace of dimension 2 of Modular Symbols space
  of dimension 6 for Gamma_0(11) of weight 4 with sign 0 over Rational Field) *
(Modular Symbols subspace of dimension 2 of Modular Symbols space
  of dimension 6 for Gamma_0(11) of weight 4 with sign 0 over Rational Field)
sage: type(F)
<class 'sage.structure.factorization.Factorization'>


sage: # needs sage.rings.number_field
sage: x = ZZ['x'].0
sage: K.<a> = NumberField(x^2 + 3); K
Number Field in a with defining polynomial x^2 + 3
sage: f = K.factor(15); f
(Fractional ideal (1/2*a + 3/2))^2 * (Fractional ideal (5))
sage: f.universe()
Monoid of ideals of Number Field in a with defining polynomial x^2 + 3
sage: f.unit()
Fractional ideal (1)
sage: g = K.factor(9); g
(Fractional ideal (1/2*a + 3/2))^4
sage: f.lcm(g)
(Fractional ideal (1/2*a + 3/2))^4 * (Fractional ideal (5))
sage: f.gcd(g)
(Fractional ideal (1/2*a + 3/2))^2
sage: f.is_integral()
True
```

AUTHORS:

- William Stein (2006-01-22): added unit part as suggested by David Kohel.
- William Stein (2008-01-17): wrote much of the documentation and fixed a couple of bugs.
- Nick Alexander (2008-01-19): added support for non-commuting factors.
- John Cremona (2008-08-22): added division, lcm, gcd, is_integral and universe functions

class sage.structure.factorization.**Factorization**(*x*, *unit=None*, *cr=False*, *sort=True*, *simplify=True*)

Bases: *SageObject*

A formal factorization of an object.

EXAMPLES:

```
sage: N = 2006
sage: F = N.factor(); F
2 * 17 * 59
sage: F.unit()
1
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
```

(continues on next page)

(continued from previous page)

```
sage: loads(F.dumps()) == F
True
sage: F = Factorization([(x, 1/3)])                                          #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

base_change(U)

Return the factorization `self`, with its factors (including the unit part) coerced into the universe U.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
Integer Ring
sage: P.<x> = ZZ[]
sage: F.base_change(P).universe()
Univariate Polynomial Ring in x over Integer Ring
```

This method will return a `TypeError` if the coercion is not possible:

```
sage: g = x^2 - 1
sage: F = factor(g); F                                                       #␣
↪needs sage.libs.pari
(x - 1) * (x + 1)
sage: F.universe()                                                           #␣
↪needs sage.libs.pari
Univariate Polynomial Ring in x over Integer Ring
sage: F.base_change(ZZ)                                                      #␣
↪needs sage.libs.pari
Traceback (most recent call last):
...
TypeError: Impossible to coerce the factors of (x - 1) * (x + 1) into Integer␣
↪Ring
```

expand()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)                                           #␣
↪needs sage.combinat sage.modules
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F                           #␣
↪needs sage.combinat sage.modules
x^3 * y^2 * x
sage: F.value()                                                              #␣
↪needs sage.combinat sage.modules
x^3*y^2*x
```

gcd(*other*)

Return the gcd of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the gcd. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-30).gcd(factor(-160))
2 * 5
sage: factor(gcd(-30,160))
2 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).gcd(factor(5*x+10))).universe()                    #␣
↪needs sage.libs.pari
Univariate Polynomial Ring in x over Integer Ring
```

is_commutative()

Return whether the factors commute.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.is_commutative()
True

sage: # needs sage.rings.number_field
sage: K = QuadraticField(23, 'a')
sage: F = K.factor(13)
sage: F.is_commutative()
True

sage: # needs sage.combinat sage.modules
sage: R.<x,y,z> = FreeAlgebra(QQ, 3)
sage: F = Factorization([(z, 2)], 3)
sage: F.is_commutative()
False
sage: (F*F^-1).is_commutative()
False
```

is_integral()

Return whether all exponents of this Factorization are non-negative.

EXAMPLES:

```
sage: F = factor(-10); F
-1 * 2 * 5
sage: F.is_integral()
True

sage: F = factor(-10) / factor(16); F
-1 * 2^-3 * 5
sage: F.is_integral()
False
```

lcm(*other*)

Return the lcm of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the lcm. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-10).lcm(factor(-16))
2^4 * 5
sage: factor(lcm(-10,16))
2^4 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).lcm(factor(5*x + 10))).universe()                    #␣
↪needs sage.libs.pari
Univariate Polynomial Ring in x over Integer Ring
```

prod()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)                                      #␣
↪needs sage.combinat sage.modules
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F                      #␣
↪needs sage.combinat sage.modules
x^3 * y^2 * x
sage: F.value()                                                         #␣
↪needs sage.combinat sage.modules
x^3*y^2*x
```

radical()

Return the factorization of the radical of the value of `self`.

First, check that all exponents in the factorization are positive, raise ValueError otherwise. If all exponents are positive, return `self` with all exponents set to 1 and with the unit set to 1.

EXAMPLES:

```
sage: F = factor(-100); F
-1 * 2^2 * 5^2
sage: F.radical()
2 * 5
sage: factor(1/2).radical()
Traceback (most recent call last):
...
ValueError: all exponents in the factorization must be positive
```

radical_value()

Return the product of the prime factors in `self`.

First, check that all exponents in the factorization are positive, raise ValueError otherwise. If all exponents are positive, return the product of the prime factors in self. This should be functionally equivalent to self.radical().value().

EXAMPLES:

```
sage: F = factor(-100); F
-1 * 2^2 * 5^2
sage: F.radical_value()
10
sage: factor(1/2).radical_value()
Traceback (most recent call last):
...
ValueError: all exponents in the factorization must be positive
```

simplify()

Combine adjacent products as much as possible.

sort(*key=None*)

Sort the factors in this factorization.

INPUT:

- key – (default: None); comparison key

OUTPUT:

- changes this factorization to be sorted (inplace)

If key is None, we determine the comparison key as follows:

If the prime in the first factor has a dimension method, then we sort based first on *dimension* then on the exponent.

If there is no dimension method, we next attempt to sort based on a degree method, in which case, we sort based first on *degree*, then exponent to break ties when two factors have the same degree, and if those match break ties based on the actual prime itself.

Otherwise, we sort according to the prime itself.

EXAMPLES:

We create a factored polynomial:

```
sage: x = polygen(QQ, 'x')
sage: F = factor(x^3 + 1); F                                        # needs sage.libs.pari
(x + 1) * (x^2 - x + 1)
```

We sort it by decreasing degree:

```
sage: F.sort(key=lambda x: (-x[0].degree(), x))                     # needs sage.libs.pari
sage: F                                                             # needs sage.libs.pari
(x^2 - x + 1) * (x + 1)
```

subs(**args*, ***kwds*)

Implement the substitution.

This is assuming that each term can be substituted.

There is another mechanism for substitution in symbolic products.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
sage: R.<x,y> = FreeAlgebra(QQ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)])
sage: F(x=4)
(1) * 4^3 * y^2 * 4
sage: F.subs({y:2})
x^3 * 2^2 * x

sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)])
sage: F(x=4)
4 * 4^3 * y^2
sage: F.subs({y:x})
x * x^2 * x^3
sage: F(x=y+x)
(x + y) * y^2 * (x + y)^3
```

unit()

Return the unit part of this factorization.

EXAMPLES:

We create a polynomial over the real double field and factor it:

```
sage: x = polygen(RDF, 'x')
sage: F = factor(-2*x^2 - 1); F                                    # needs numpy
(-2.0) * (x^2 + 0.5000000000000001)
```

Note that the unit part of the factorization is -2.0:

```
sage: F.unit()                                                     # needs numpy
-2.0

sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
```

universe()

Return the parent structure of my factors.

Note: This used to be called `base_ring`, but the universe of a factorization need not be a ring.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
```

(continues on next page)

(continued from previous page)

```
Integer Ring

sage: R.<x,y,z> = FreeAlgebra(QQ, 3)                                    #␣
↪needs sage.combinat sage.modules
sage: F = Factorization([(z, 2)], 3)                                    #␣
↪needs sage.combinat sage.modules
sage: (F*F^-1).universe()                                               #␣
↪needs sage.combinat sage.modules
Free Algebra on 3 generators (x, y, z) over Rational Field

sage: F = ModularSymbols(11,4).factorization()                          #␣
↪needs sage.modular
sage: F.universe()                                                      #␣
↪needs sage.modular
```

value()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.value()
-2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)                                      #␣
↪needs sage.combinat sage.modules
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F                      #␣
↪needs sage.combinat sage.modules
x^3 * y^2 * x
sage: F.value()                                                         #␣
↪needs sage.combinat sage.modules
x^3*y^2*x
```

4.3 IntegerFactorization objects

class sage.structure.factorization_integer.**IntegerFactorization**(*x*, *unit=None*, *cr=False*, *sort=True*, *simplify=True*, *unsafe=False*)

Bases: *Factorization*

A lightweight class for an `IntegerFactorization` object, inheriting from the more general `Factorization` class.

In the `Factorization` class the user has to create a list containing the factorization data, which is then passed to the actual `Factorization` object upon initialization.

However, for the typical use of integer factorization via the `Integer.factor()` method in `sage.rings.integer` this is noticeably too much overhead, slowing down the factorization of integers of up to about 40 bits by a factor of around 10. Moreover, the initialization done in the `Factorization` class is typically unnecessary: the caller can guarantee that the list contains pairs of an `Integer` and an `int`, as well as that the list is sorted.

AUTHOR:

- Sebastian Pancratz (2010-01-10)

4.4 Finite Homogeneous Sequences

A mutable sequence of elements with a common guaranteed category, which can be set immutable.

Sequence derives from list, so has all the functionality of lists and can be used wherever lists are used. When a sequence is created without explicitly given the common universe of the elements, the constructor coerces the first and second element to some *canonical* common parent, if possible, then the second and third, etc. If this is possible, it then coerces everything into the canonical parent at the end. (Note that canonical coercion is very restrictive.) The sequence then has a function `universe()` which returns either the common canonical parent (if the coercion succeeded), or the category of all objects (Objects()). So if you have a list v and type:

```
sage: v = [1, 2/3, 5]
sage: w = Sequence(v)
sage: w.universe()
Rational Field
```

then since `w.universe()` is $\mathbf{Q}$, you're guaranteed that all elements of w are rationals:

```
sage: v[0].parent()
Integer Ring
sage: w[0].parent()
Rational Field
```

If you do assignment to w this property of being rationals is guaranteed to be preserved:

```
sage: w[0] = 2
sage: w[0].parent()
Rational Field
sage: w[0] = 'hi'
Traceback (most recent call last):
...
TypeError: unable to convert 'hi' to a rational
```

However, if you do `w = Sequence(v)` and the resulting universe is `Objects()`, the elements are not guaranteed to have any special parent. This is what should happen, e.g., with finite field elements of different characteristics:

```
sage: v = Sequence([GF(3)(1), GF(7)(1)])
sage: v.universe()
Category of objects
```

You can make a list immutable with `v.freeze()`. Assignment is never again allowed on an immutable list.

Creation of a sequence involves making a copy of the input list, and substantial coercions. It can be greatly sped up by explicitly specifying the universe of the sequence:

```
sage: v = Sequence(range(10000), universe=ZZ)
```

`sage.structure.sequence.Sequence`(*x*, *universe=None*, *check=True*, *immutable=False*, *cr=False*, *cr_str=None*, *use_sage_types=False*)

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- `cr_str` - (default: False) if True, then print a carriage return after each comma when calling `str()` on this sequence.
- **`use_sage_types` – (default: False) if True, coerce the**
  built-in Python numerical types int, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<class 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for “Sequence”, which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
```

Note that assignment coerces if possible,:

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.:

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v) == hash(tuple(range(10)))
True
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.:

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.:

```
sage: v = Sequence([1, 7, 6, GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
```

class sage.structure.sequence.**Sequence_generic**(*x*, *universe=None*, *check=True*, *immutable=False*, *cr=False*, *cr_str=None*, *use_sage_types=False*)

Bases: *SageObject*, list

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- x - a list or tuple instance
- universe - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- check – (default: True) whether to coerce the elements of x into the universe
- immutable - (default: True) whether or not this sequence is immutable
- cr - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- **use_sage_types – (default: False) if True, coerce the**
  built-in Python numerical types int, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<class 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for "Sequence", which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
```

Note that assignment coerces if possible,

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
```

(continues on next page)

(continued from previous page)

```
Rational Field
sage: v[3] = 2/3
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v) == hash(tuple(range(10)))
True
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.

```
sage: v = Sequence([1, 7, 6, GF(5)(3)]); v
[1, 2, 1, 3]
```

(continues on next page)

(continued from previous page)

```
sage: v.universe()
Finite Field of size 5
```

append(*x*)

EXAMPLES:

```
sage: v = Sequence([1,2,3,4], immutable=True)
sage: v.append(34)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: v = Sequence([1/3,2,3,4])
sage: v.append(4)
sage: type(v[4])
<class 'sage.rings.rational.Rational'>
```

extend(*iterable*)

Extend list by appending elements from the iterable.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.extend(range(4))
sage: B
[1, 2, 3, 0, 1, 2, 3]
```

insert(*index, object*)

Insert object before index.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.insert(10, 5)
sage: B
[1, 2, 3, 5]
```

is_immutable()

Return True if this object is immutable (can not be changed) and False if it is not.

To make this object immutable use `set_immutable()`.

EXAMPLES:

```
sage: v = Sequence([1, 2, 3, 4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

EXAMPLES:

```
sage: a = Sequence([1, 2/3, -2/5])
sage: a.is_mutable()
True
sage: a[0] = 100
sage: type(a[0])
<class 'sage.rings.rational.Rational'>
sage: a.set_immutable()
sage: a[0] = 50
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
sage: a.is_mutable()
False
```

pop(*index=-1*)

Remove and return item at index (default last)

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.pop(1)
2
sage: B
[1, 3]
```

remove(*value*)

Remove first occurrence of value

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.remove(2)
sage: B
[1, 3]
```

reverse()

Reverse the elements of self, in place.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.reverse(); B
[3, 2, 1]
```

set_immutable()

Make this object immutable, so it can never again be changed.

EXAMPLES:

```
sage: v = Sequence([1, 2, 3, 4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
```

(continues on next page)

(continued from previous page)

```
sage: v[3] = 7
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

sort(*key=None*, *reverse=False*)

Sort this list *IN PLACE*.

INPUT:

- `key` - see Python `list sort`
- `reverse` - see Python `list sort`

EXAMPLES:

```
sage: B = Sequence([3,2,1/5])
sage: B.sort()
sage: B
[1/5, 2, 3]
sage: B.sort(reverse=True); B
[3, 2, 1/5]
```

universe()

Return the universe of the sequence.

EXAMPLES:

```
sage: Sequence([1, 2/3, -2/5]).universe()
Rational Field
sage: Sequence([1, 2/3, '-2/5']).universe()
Category of objects
```

sage.structure.sequence.**seq**(*x*, *universe=None*, *check=True*, *immutable=False*, *cr=False*, *cr_str=None*, *use_sage_types=False*)

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- `cr_str` - (default: False) if True, then print a carriage return after each comma when calling `str()` on this sequence.
- **`use_sage_types` – (default: False) if True, coerce the**
  built-in Python numerical types int, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<class 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the built-in Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for "Sequence", which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
```

Note that assignment coerces if possible,:

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.:

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v) == hash(tuple(range(10)))
True
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)        # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))     # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.:

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.:

```
sage: v = Sequence([1, 7, 6, GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
```

4.5 Set factories

A *set factory* F is a device for constructing some `Parent` P that models subsets of a big set S. Typically, each such parent is constructed as the subset of S of all elements satisfying a certain collection of constraints $cons$. In such a hierarchy of subsets, one needs an easy and flexible control on how elements are constructed. For example, one may want to construct the elements of P in some subclass of the class of the elements of S. On other occasions, one also often needs P to be a facade parent, whose elements are represented as elements of S (see `FacadeSets`).

The role of a set factory is twofold:

- *Manage a database* of constructors for the different parents $P = F(cons)$ depending on the various kinds of constraints $cons$. Note: currently there is no real support for that. We are gathering use cases before fixing the interface.
- Ensure that the elements $e = P(...)$ created by the different parents follows a consistent policy concerning their *class and parent*.

Basic usage: constructing parents through a factory

The file `sage.structure.set_factories_example` shows an example of a `SetFactory` together with typical implementation. Note that the written code is intentionally kept minimal, many things and in particular several iterators could be written in a more efficient way.

Consider the set S of couples (x, y) with x and y in $I := \{0, 1, 2, 3, 4\}$. We represent an element of S as a 2-elements tuple, wrapped in a class `XYPair` deriving from `ElementWrapper`. You can create a `XYPair` with any `Parent`:

```
sage: from sage.structure.set_factories import *
sage: from sage.structure.set_factories_example import *
sage: p = XYPair(Parent(), (0,1)); p
(0, 1)
```

Now, given $(a, b) \in S$ we want to consider the following subsets of S

$$
\begin{aligned}
S_a &:= \{(x, y) \in S \mid x = a\},\\
S^b &:= \{(x, y) \in S \mid y = b\},\\
S_a^b &:= \{(x, y) \in S \mid x = a, y = b\}.
\end{aligned}
$$

The constraints considered here are admittedly trivial. In a realistic example, there would be much more of them. And for some sets of constraints no good enumeration algorithms would be known.

In Sage, those sets are constructed by passing the constraints to the factory. We first create the set with no constraints at all:

```
sage: XYPairs
Factory for XY pairs
sage: S = XYPairs(); S.list()
[(0, 0), (1, 0), ..., (4, 0), (0, 1), (1, 1), ..., (3, 4), (4, 4)]
sage: S.cardinality()
25
```

Let us construct S_2, S^3 and S_2^3:

```
sage: Sx2 = XYPairs(x=2); Sx2.list()
[(2, 0), (2, 1), (2, 2), (2, 3), (2, 4)]
sage: Sy3 = XYPairs(y=3); Sy3.list()
[(0, 3), (1, 3), (2, 3), (3, 3), (4, 3)]
sage: S23 = XYPairs(x=2, y=3); S23.list()
[(2, 3)]
```

Set factories provide an alternative way to build subsets of an already constructed set: each set constructed by a factory has a method `subset()` which accept new constraints. Sets constructed by the factory or the `subset()` methods are identical:

```
sage: Sx2s = S.subset(x=2); Sx2 is Sx2s
True
sage: Sx2.subset(y=3) is S23
True
```

It is not possible to change an already given constraint:

```
sage: S23.subset(y=5)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Duplicate value for constraints 'y': was 3 now 5
```

Constructing custom elements: policies

We now come to the point of factories: constructing custom elements. The writer of *XYPairs()* decided that, by default, the parents Sx2, Sy3 and S23 are facade for parent S. This means that each element constructed by those subsets behaves as if they where directly constructed by S itself:

```
sage: Sx2.an_element().parent()
AllPairs
sage: el = Sx2.an_element()
sage: el.parent() is S
True
sage: type(el) is S.element_class
True
```

This is not always desirable. The device which decides how to construct an element is called a *policy* (see *SetFactoryPolicy*). Each factory should have a default policy. Here is the policy of *XYPairs()*:

```
sage: XYPairs._default_policy
Set factory policy for <class 'sage.structure.set_factories_example.XYPair'> with parent␣
↪AllPairs[=Factory for XY pairs(())]
```

This means that with the current policy, the parent builds elements with class XYPair and parent AllPairs which is itself constructed by calling the factory *XYPairs()* with constraints (). There is a lot of flexibility to change that. We now illustrate how to make a few different choices.

1 - In a first use case, we want to add some methods to the constructed elements. As illustration, we add here a new method sum which returns $x + y$. We therefore create a new class for the elements which inherits from *XYPair*:

```
sage: class NewXYPair(XYPair):
....:     def sum(self):
....:         return sum(self.value)
```

Here is an instance of this class (with a dummy parent):

```
sage: el = NewXYPair(Parent(), (2,3))
sage: el.sum()
5
```

We now want to have subsets generating those new elements while still having a single real parent (the one with no constraint) for each element. The corresponding policy is called *TopMostParentPolicy*. It takes three parameters:

- the factory;
- the parameters for void constraint;
- the class used for elements.

Calling the factory with this policy returns a new set which builds its elements with the new policy:

```
sage: new_policy = TopMostParentPolicy(XYPairs, (), NewXYPair)
sage: NewS = XYPairs(policy=new_policy)
```

(continues on next page)

(continued from previous page)

```
sage: el = NewS.an_element(); el
(0, 0)
sage: el.sum()
0
sage: el.parent() is NewS
True
sage: isinstance(el, NewXYPair)
True
```

Newly constructed subsets inherit the policy:

```
sage: NewS2 = NewS.subset(x=2)
sage: el2 = NewS2.an_element(); el2
(2, 0)
sage: el2.sum()
2
sage: el2.parent() is NewS
True
```

2 - In a second use case, we want the elements to remember which parent created them. The corresponding policy is called *SelfParentPolicy*. It takes only two parameters:

- the factory;
- the class used for elements.

Here is an example:

```
sage: selfpolicy = SelfParentPolicy(XYPairs, NewXYPair)
sage: SelfS = XYPairs(policy=selfpolicy)
sage: el = SelfS.an_element()
sage: el.parent() is SelfS
True
```

Now all subsets are the parent of the elements that they create:

```
sage: SelfS2 = SelfS.subset(x=2)
sage: el2 = SelfS2.an_element()
sage: el2.parent() is NewS
False
sage: el2.parent() is SelfS2
True
```

3 - Finally, a common use case is to construct simple python object which are not Sage `sage.structure.Element`. As an example, we show how to build a parent `TupleS` which construct pairs as tuple. The corresponding policy is called *BareFunctionPolicy*. It takes two parameters:

- the factory;
- the function called to construct the elements.

Here is how to do it:

```
sage: cons = lambda t, check: tuple(t) # ignore the check parameter
sage: tuplepolicy = BareFunctionPolicy(XYPairs, cons)
sage: P = XYPairs(x=2, policy=tuplepolicy)
```

(continues on next page)

(continued from previous page)

```
sage: P.list()
[(2, 0), (2, 1), (2, 2), (2, 3), (2, 4)]
sage: el = P.an_element()
sage: type(el)
<... 'tuple'>
```

Here are the currently implemented policies:

- *FacadeParentPolicy*: reuse an existing parent together with its element_class
- *TopMostParentPolicy*: use a parent created by the factory itself and provide a class `Element` for it. In this case, we need to specify the set of constraints which build this parent taking the ownership of all elements and the class which will be used for the `Element`.
- *SelfParentPolicy*: provide systematically `Element` and element_class and ensure that the parent is `self`.
- *BareFunctionPolicy*: instead of calling a class constructor element are passed to a function provided to the policy.

Todo: Generalize *TopMostParentPolicy* to be able to have several topmost parents.

Technicalities: how policies inform parents

Parents built from factories should inherit from *ParentWithSetFactory*. This class provide a few methods related to factories and policies. The `__init__` method of *ParentWithSetFactory* must be provided with the set of constraints and the policy. A parent built from a factory must create elements through a call to the method `_element_constructor_`. The current way in which policies inform parents how to builds their elements is set by a few attributes. So the class must accept attribute adding. The precise details of which attributes are set may be subject to change in the future.

How to write a set factory

See also:

set_factories_example for an example of a factory.

Here are the specifications:

A parent built from a factory should

- *inherit* from *ParentWithSetFactory*. It should accept a `policy` argument and pass it verbatim to the `__init__` method of *ParentWithSetFactory* together with the set of constraints;
- *create its elements* through calls to the method `_element_constructor_`;
- *define a method* *ParentWithSetFactory.check_element* which checks if a built element indeed belongs to it. The method should accept an extra keyword parameter called `check` specifying which level of check should be performed. It will only be called when `bool(check)` evaluates to `True`.

The constructor of the elements of a parent from a factory should:

- receive the parent as first mandatory argument;
- accept an extra optional keyword parameter called `check` which is meant to tell if the input must be checked or not. The precise meaning of `check` is intentionally left vague. The only intent is that if `bool(check)` evaluates to `False`, no check is performed at all.

A factory should

- *define a method* `__call__` which is responsible for calling the appropriate parent constructor given the constraints;
- *define a method* overloading `SetFactory.add_constraints()` which is responsible of computing the union of two sets of constraints;
- *optionally* define a method or an attribute `_default_policy` passed to the `ParentWithSetFactory` if no policy is given to the factory.

Todo: There is currently no support for dealing with sets of constraints. The set factory and the parents must cooperate to consistently handle them. More support, together with a generic mechanism to select the appropriate parent class from the constraints, will be added as soon as we have gathered sufficiently enough use-cases.

AUTHORS:

- Florent Hivert (2011-2012): initial revision

class `sage.structure.set_factories.`**`BareFunctionPolicy`**(*factory*, *constructor*)

Bases: `SetFactoryPolicy`

Policy where element are constructed using a bare function.

INPUT:

- `factory` – an instance of `SetFactory`
- `constructor` – a function

Given a factory `F` and a function `c`, returns a policy for parent `P` creating element using the function `f`.

EXAMPLES:

```
sage: from sage.structure.set_factories import BareFunctionPolicy
sage: from sage.structure.set_factories_example import XYPairs
sage: cons = lambda t, check: tuple(t) # ignore the check parameter
sage: tuplepolicy = BareFunctionPolicy(XYPairs, cons)
sage: P = XYPairs(x=2, policy=tuplepolicy)
sage: el = P.an_element()
sage: type(el)
<... 'tuple'>
```

`element_constructor_attributes`(*constraints*)

Return the element constructor attributes as per `SetFactoryPolicy.element_constructor_attributes()`.

INPUT:

- `constraints` – a bunch of constraints

class `sage.structure.set_factories.`**`FacadeParentPolicy`**(*factory*, *parent*)

Bases: `SetFactoryPolicy`

Policy for facade parent.

INPUT:

- `factory` – an instance of `SetFactory`
- `parent` – an instance of `Parent`

Given a factory F and a class E, returns a policy for parent P creating elements as if they were created by `parent`.

EXAMPLES:

```
sage: from sage.structure.set_factories import SelfParentPolicy, FacadeParentPolicy
sage: from sage.structure.set_factories_example import XYPairs, XYPair
```

We create a custom standard parent P:

```
sage: selfpolicy = SelfParentPolicy(XYPairs, XYPair)
sage: P = XYPairs(x=2, policy=selfpolicy)
sage: pol = FacadeParentPolicy(XYPairs, P)
sage: P2 = XYPairs(x=2, y=3, policy=pol)
sage: el = P2.an_element()
sage: el.parent() is P
True
sage: type(el) is P.element_class
True
```

If `parent` is itself a facade parent, then transitivity is correctly applied:

```
sage: P =  XYPairs()
sage: P2 = XYPairs(x=2)
sage: P2.category()
Category of facade finite enumerated sets
sage: pol = FacadeParentPolicy(XYPairs, P)
sage: P23 = XYPairs(x=2, y=3, policy=pol)
sage: el = P2.an_element()
sage: el.parent() is P
True
sage: type(el) is P.element_class
True
```

element_constructor_attributes(*constraints*)

Return the element constructor attributes as per *SetFactoryPolicy.element_constructor_attributes()*.

INPUT:

- `constraints` – a bunch of constraints

class sage.structure.set_factories.**ParentWithSetFactory**(*constraints*, *policy*, *category=None*)

Bases: *Parent*

Abstract class for parent belonging to a set factory.

INPUT:

- `constraints` – a set of constraints
- `policy` – the policy for element construction
- `category` – the category of the parent (default to `None`)

Depending on the constraints and the policy, initialize the parent in a proper category to set up element construction.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs, PairsX_
sage: P = PairsX_(3, XYPairs._default_policy)
sage: P is XYPairs(3)
True
sage: P.category()
Category of facade finite enumerated sets
```

check_element(*x*, *check*)

Check that x verifies the constraints of self.

INPUT:

- x – an instance of self.element_class.
- check – the level of checking to be performed (usually a boolean).

This method may assume that x was properly constructed by self or a possible super-set of self for which self is a facade. It should return nothing if x verifies the constraints and raise a ValueError explaining which constraints x fails otherwise.

The method should accept an extra parameter check specifying which level of check should be performed. It will only be called when bool(check) evaluates to True.

Todo: Should we always call check element and let it decide which check has to be performed ?

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: S = XYPairs()
sage: el = S((2,3))
sage: S.check_element(el, True)
sage: XYPairs(x=2).check_element(el, True)
sage: XYPairs(x=3).check_element(el, True)
Traceback (most recent call last):
...
ValueError: Wrong first coordinate
sage: XYPairs(y=4).check_element(el, True)
Traceback (most recent call last):
...
ValueError: Wrong second coordinate
```

constraints()

Return the constraints defining self.

Note: Currently there is no specification on how constraints are passed as arguments.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: XYPairs().constraints()
()
sage: XYPairs(x=3).constraints()
(3, None)
```

(continues on next page)

(continued from previous page)

```
sage: XYPairs(y=2).constraints()
(None, 2)
```

facade_policy()

Return the policy for parent facade for `self`.

EXAMPLES:

```
sage: from sage.structure.set_factories import SelfParentPolicy
sage: from sage.structure.set_factories_example import XYPairs, XYPair
```

We create a custom standard parent P:

```
sage: selfpolicy = SelfParentPolicy(XYPairs, XYPair)
sage: P = XYPairs(x=2, policy=selfpolicy)
sage: P.facade_policy()
Set factory policy for facade parent {(2, b) | b in range(5)}
```

Now passing `P.facade_policy()` creates parent which are facade for P:

```
sage: P3 = XYPairs(x=2, y=3, policy=P.facade_policy())
sage: P3.facade_for() == (P,)
True
sage: el = P3.an_element()
sage: el.parent() is P
True
```

factory()

Return the factory which built `self`.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: XYPairs().factory() is XYPairs
True
sage: XYPairs(x=3).factory() is XYPairs
True
```

policy()

Return the policy used when `self` was created.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: XYPairs().policy()
Set factory policy for <class 'sage.structure.set_factories_example.XYPair'>␣
↪with parent AllPairs[=Factory for XY pairs(())]
sage: XYPairs(x=3).policy()
Set factory policy for <class 'sage.structure.set_factories_example.XYPair'>␣
↪with parent AllPairs[=Factory for XY pairs(())]
```

subset(**args*, ***options*)

Return a subset of `self` by adding more constraints.

Note: Currently there is no specification on how constraints are passed as arguments.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: S = XYPairs()
sage: S3 = S.subset(x=3)
sage: S3.list()
[(3, 0), (3, 1), (3, 2), (3, 3), (3, 4)]
```

class sage.structure.set_factories.**SelfParentPolicy**(*factory*, *Element*)

Bases: *SetFactoryPolicy*

Policy where each parent is a standard parent.

INPUT:

- factory – an instance of *SetFactory*
- Element – a subclass of *Element*

Given a factory F and a class E, returns a policy for parent P creating elements in class E and parent P itself.

EXAMPLES:

```
sage: from sage.structure.set_factories import SelfParentPolicy
sage: from sage.structure.set_factories_example import XYPairs, XYPair, Pairs_Y
sage: pol = SelfParentPolicy(XYPairs, XYPair)
sage: S = Pairs_Y(3, pol)
sage: el = S.an_element()
sage: el.parent() is S
True

sage: class Foo(XYPair): pass
sage: pol = SelfParentPolicy(XYPairs, Foo)
sage: S = Pairs_Y(3, pol)
sage: el = S.an_element()
sage: isinstance(el, Foo)
True
```

element_constructor_attributes(*constraints*)

Return the element constructor attributes as per *SetFactoryPolicy.element_constructor_attributes()*

INPUT:

- constraints – a bunch of constraints

class sage.structure.set_factories.**SetFactory**

Bases: *UniqueRepresentation*, *SageObject*

This class is currently just a stub that we will be using to add more structures on factories.

add_constraints(*cons*, **args*, ***opts*)

Add constraints to the set of constraints $cons$.

Should return a set of constraints.

Note: Currently there is no specification on how constraints are passed as arguments.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: XYPairs.add_constraints((3,),((None, 2), {}))
(3, 2)

sage: XYPairs.add_constraints((3,),((None, None), {'y': 2}))
(3, 2)
```

class sage.structure.set_factories.**SetFactoryPolicy**(*factory*)

Bases: *UniqueRepresentation*, *SageObject*

Abstract base class for policies.

A policy is a device which is passed to a parent inheriting from *ParentWithSetFactory* in order to set-up the element construction framework.

INPUT:

- factory – a *SetFactory*

Warning: This class is a base class for policies, one should not try to create instances.

element_constructor_attributes(*constraints*)

Element constructor attributes.

INPUT:

- constraints – a bunch of constraints

Should return the attributes that are prerequisite for element construction. This is coordinated with ParentWithSetFactory._element_constructor_(). Currently two standard attributes are provided in *facade_element_constructor_attributes()* and *self_element_constructor_attributes()*. You should return the one needed depending on the given constraints.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs, XYPair
sage: pol = XYPairs._default_policy
sage: pol.element_constructor_attributes(())
{'Element': <class 'sage.structure.set_factories_example.XYPair'>,
 '_parent_for': 'self'}
sage: pol.element_constructor_attributes((1))
{'_facade_for': AllPairs,
 '_parent_for': AllPairs,
 'element_class': <class 'sage.structure.set_factories_example.AllPairs_with_
↪category.element_class'>}
```

facade_element_constructor_attributes(*parent*)

Element Constructor Attributes for facade parent.

The list of attributes which must be set during the init of a facade parent with factory.

INPUT:

- `parent` – the actual parent for the elements

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs, XYPair
sage: pol = XYPairs._default_policy
sage: pol.facade_element_constructor_attributes(XYPairs())
{'_facade_for': AllPairs,
 '_parent_for': AllPairs,
 'element_class': <class 'sage.structure.set_factories_example.AllPairs_with_
↪category.element_class'>}
```

factory()

Return the factory for `self`.

EXAMPLES:

```
sage: from sage.structure.set_factories import SetFactoryPolicy,␣
↪SelfParentPolicy
sage: from sage.structure.set_factories_example import XYPairs, XYPair
sage: XYPairs._default_policy.factory()
Factory for XY pairs
sage: XYPairs._default_policy.factory() is XYPairs
True
```

self_element_constructor_attributes(*Element*)

Element Constructor Attributes for non facade parent.

The list of attributes which must be set during the init of a non facade parent with factory.

INPUT:

- `Element` – the class used for the elements

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs, XYPair
sage: pol = XYPairs._default_policy
sage: pol.self_element_constructor_attributes(XYPair)
{'Element': <class 'sage.structure.set_factories_example.XYPair'>,
 '_parent_for': 'self'}
```

class `sage.structure.set_factories.`**TopMostParentPolicy**(*factory*, *top_constraints*, *Element*)

Bases: *SetFactoryPolicy*

Policy where the parent of the elements is the topmost parent.

INPUT:

- `factory` – an instance of *SetFactory*
- `top_constraints` – the empty set of constraints.
- `Element` – a subclass of *Element*

Given a factory `F` and a class `E`, returns a policy for parent `P` creating element in class `E` and parent `factory(*top_constraints, policy)`.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs, XYPair
sage: P = XYPairs(); P.policy()
Set factory policy for <class 'sage.structure.set_factories_example.XYPair'> with
↪parent AllPairs[=Factory for XY pairs(())]
```

element_constructor_attributes(*constraints*)

Return the element constructor attributes as per *SetFactoryPolicy.element_constructor_attributes()*.

INPUT:

- `constraints` – a bunch of constraints

4.6 An example of set factory

The goal of this module is to exemplify the use of set factories. Note that the code is intentionally kept minimal; many things and in particular several iterators could be written in a more efficient way.

See also:

set_factories for an introduction to set factories, their specifications, and examples of their use and implementation based on this module.

We describe here a factory used to construct the set S of couples (x, y) with x and y in $I := \{0, 1, 2, 3, 4\}$, together with the following subsets, where $(a, b) \in S$

$$S_a := \{(x, y) \in S \mid x = a\},$$
$$S^b := \{(x, y) \in S \mid y = b\},$$
$$S_a^b := \{(x, y) \in S \mid x = a, y = b\}.$$

class sage.structure.set_factories_example.**AllPairs**(*policy*)

Bases: *ParentWithSetFactory*, `DisjointUnionEnumeratedSets`

This parent shows how one can use set factories together with `DisjointUnionEnumeratedSets`.

It is constructed as the disjoint union (`DisjointUnionEnumeratedSets`) of *Pairs_Y* parents:

$$S := \bigcup_{i=0,1,\ldots,4} S^y$$

> **Warning:** When writing a parent `P` as a disjoint union of a family of parents `P_i`, the parents `P_i` must be constructed as facade parents for `P`. As a consequence, it should be passed `P.facade_policy()` as policy argument. See the source code of *pairs_y()* for an example.

check_element(*el*, *check*)

pairs_y(*letter*)

Construct the parent for the disjoint union

Construct a parent in *Pairs_Y* as a facade parent for `self`.

This is an internal function which should be hidden from the user (typically under the name `_pairs_y`. We put it here for documentation.

class sage.structure.set_factories_example.**PairsX_**(*x*, *policy*)

Bases: *ParentWithSetFactory*, *UniqueRepresentation*

The set of pairs $(x, 0), (x, 1), ..., (x, 4)$.

an_element()

check_element(*el*, *check*)

class sage.structure.set_factories_example.**Pairs_Y**(*y*, *policy*)

Bases: *ParentWithSetFactory*, `DisjointUnionEnumeratedSets`

The set of pairs $(0, y), (1, y), ..., (4, y)$.

It is constructed as the disjoint union (`DisjointUnionEnumeratedSets`) of *SingletonPair* parents:

$$S^y := \bigcup_{i=0,1,\ldots,4} S_i^y$$

See also:

AllPairs for how to properly construct `DisjointUnionEnumeratedSets` using *ParentWithSetFactory*.

an_element()

check_element(*el*, *check*)

single_pair(*letter*)

Construct the singleton pair parent

Construct a singleton pair for `(self.y, letter)` as a facade parent for `self`.

See also:

AllPairs for how to properly construct `DisjointUnionEnumeratedSets` using *ParentWithSetFactory*.

class sage.structure.set_factories_example.**SingletonPair**(*x*, *y*, *policy*)

Bases: *ParentWithSetFactory*, *UniqueRepresentation*

check_element(*el*, *check*)

class sage.structure.set_factories_example.**XYPair**(*parent*, *value*, *check=True*)

Bases: *ElementWrapper*

A class for Elements (x, y) with x and y in $\{0, 1, 2, 3, 4\}$.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPair
sage: p = XYPair(Parent(), (0,1)); p
(0, 1)
sage: p = XYPair(Parent(), (0,8))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: numbers must be in range(5)
```

sage.structure.set_factories_example.**XYPairs**(*x=None*, *y=None*, *policy=None*)

Construct the subset from constraints.

Consider the set S of couples (x, y) with x and y in $I := \{0, 1, 2, 3, 4\}$. Returns the subsets of element of S satisfying some constraints.

INPUT:

- x=a – where a is an integer (default to None).
- y=b – where b is an integer (default to None).
- policy – the policy passed to the created set.

See also:

set_factories.SetFactoryPolicy

EXAMPLES:

Let us first create the set factory:

```
sage: from sage.structure.set_factories_example import XYPairsFactory
sage: XYPairs = XYPairsFactory()
```

One can then use the set factory to construct a set:

```
sage: P = XYPairs(); P.list()
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (0,
↪ 2), (1, 2), (2, 2), (3, 2), (4, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (0,␣
↪4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

Note: This function is actually the `__call__` method of *XYPairsFactory*.

class sage.structure.set_factories_example.**XYPairsFactory**

Bases: *SetFactory*

An example of set factory, for sets of pairs of integers.

See also:

set_factories for an introduction to set factories.

add_constraints(*cons*, *args_opts*)

Add constraints to the set cons as per *SetFactory.add_constraints*.

This is a crude implementation for the sake of the demonstration which should not be taken as an example.

EXAMPLES:

```
sage: from sage.structure.set_factories_example import XYPairs
sage: XYPairs.add_constraints((3,None), ((2,), {}))
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: Duplicate value for constraints 'x': was 3 now 2
sage: XYPairs.add_constraints((), ((2,), {}))
(2, None)
sage: XYPairs.add_constraints((), ((2,), {'y':3}))
(2, 3)
```

CHAPTER

FIVE

USE OF HEURISTIC AND PROBABILISTIC ALGORITHMS

5.1 Global proof preferences

class sage.structure.proof.proof.**WithProof**(*subsystem*, *t*)

Bases: object

Use WithProof to temporarily set the value of one of the proof systems for a block of code, with a guarantee that it will be set back to how it was before after the block is done, even if there is an error.

EXAMPLES:

This would hang "forever" if attempted with proof=True:

```
sage: proof.arithmetic(True)
sage: with proof.WithProof('arithmetic', False):                              #␣
↪needs sage.libs.pari
....:     print((10^1000 + 453).is_prime())
....:     print(1/0)
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
sage: proof.arithmetic()
True
```

sage.structure.proof.proof.**get_flag**(*t=None*, *subsystem=None*)

Used for easily determining the correct proof flag to use.

EXAMPLES:

```
sage: from sage.structure.proof.proof import get_flag
sage: get_flag(False)
False
sage: get_flag(True)
True
sage: get_flag()
True
sage: proof.all(False)
sage: get_flag()
False
```

5.2 Whether or not computations are provably correct by default

CHAPTER

SIX

UTILITIES

6.1 Cython-like rich comparisons in Python

With "rich comparisons", we mean the Python 3 comparisons which are usually implemented in Python using methods like `__eq__` and `__lt__`. Internally in Python, there is only one rich comparison slot `tp_richcompare`. The actual operator is passed as an integer constant (defined in this module as `op_LT`, `op_LE`, `op_EQ`, `op_NE`, `op_GT`, `op_GE`).

Cython exposes rich comparisons in `cdef` classes as the `__richcmp__` special method. The Sage coercion model also supports rich comparisons this way: for two instances `x` and `y` of *Element*, `x._richcmp_(y, op)` is called when the user does something like `x <= y` (possibly after coercion if `x` and `y` have different parents).

Various helper functions exist to make it easier to implement rich comparison: the most important one is the *richcmp()* function. This is analogous to the Python 2 function `cmp()` but implements rich comparison, with the comparison operator (e.g. `op_GE`) as third argument. There is also *richcmp_not_equal()* which is like *richcmp()* but it is optimized assuming that the compared objects are not equal.

The functions *rich_to_bool()* and *rich_to_bool_sgn()* can be used to convert results of `cmp()` (i.e. -1, 0 or 1) to a boolean `True`/`False` for rich comparisons.

AUTHORS:

- Jeroen Demeyer

`sage.structure.richcmp.`**`revop`**`(`*op*`)`

Return the reverse operation of `op`.

For example, <= becomes >=, etc.

EXAMPLES:

```
sage: from sage.structure.richcmp import revop
sage: [revop(i) for i in range(6)]
[4, 5, 2, 3, 0, 1]
```

`sage.structure.richcmp.`**`rich_to_bool`**`(`*op*, *c*`)`

Return the corresponding `True` or `False` value for a rich comparison, given the result of an old-style comparison.

INPUT:

- `op` – a rich comparison operation (e.g. `Py_EQ`)
- `c` – the result of an old-style comparison: -1, 0 or 1.

OUTPUT: 1 or 0 (corresponding to `True` and `False`)

See also:

rich_to_bool_sgn() if `c` could be outside the [-1, 0, 1] range.

EXAMPLES:

```
sage: from sage.structure.richcmp import (rich_to_bool,
....:    op_EQ, op_NE, op_LT, op_LE, op_GT, op_GE)
sage: for op in (op_LT, op_LE, op_EQ, op_NE, op_GT, op_GE):
....:     for c in (-1,0,1):
....:         print(rich_to_bool(op, c))
True False False
True True False
False True False
True False True
False False True
False True True
```

Indirect tests using integers:

```
sage: 0 < 5, 5 < 5, 5 < -8
(True, False, False)
sage: 0 <= 5, 5 <= 5, 5 <= -8
(True, True, False)
sage: 0 >= 5, 5 >= 5, 5 >= -8
(False, True, True)
sage: 0 > 5, 5 > 5, 5 > -8
(False, False, True)
sage: 0 == 5, 5 == 5, 5 == -8
(False, True, False)
sage: 0 != 5, 5 != 5, 5 != -8
(True, False, True)
```

sage.structure.richcmp.**rich_to_bool_sgn**(*op*, *c*)

Same as `rich_to_bool`, but allow any $c < 0$ and $c > 0$ instead of only -1 and 1.

Note: This is in particular needed for `mpz_cmp()`.

sage.structure.richcmp.**richcmp**(*x*, *y*, *op*)

Return the result of the rich comparison of `x` and `y` with operator `op`.

INPUT:

- `x`, `y` – arbitrary Python objects
- `op` – comparison operator (one of `op_LT`, ``op_LE`, `op_EQ`, `op_NE`, `op_GT`, `op_GE`).

EXAMPLES:

```
sage: from sage.structure.richcmp import *
sage: richcmp(3, 4, op_LT)
True
sage: richcmp(x, x^2, op_EQ)                                        #␣
↪needs sage.symbolic
x == x^2
```

The two examples above are completely equivalent to `3 < 4` and `x == x^2`. For this reason, it only makes sense in practice to call `richcmp` with a non-constant value for `op`.

We can write a custom `Element` class which shows a more realistic example of how to use this:

```
sage: from sage.structure.element import Element
sage: class MyElement(Element):
....:     def __init__(self, parent, value):
....:         Element.__init__(self, parent)
....:         self.v = value
....:     def _richcmp_(self, other, op):
....:         return richcmp(self.v, other.v, op)
sage: P = Parent()
sage: x = MyElement(P, 3)
sage: y = MyElement(P, 3)
sage: x < y
False
sage: x == y
True
sage: x > y
False
```

sage.structure.richcmp.**richcmp_by_eq_and_lt**(*eq_attr*, *lt_attr*)

Create a rich comparison method for a partial order, where the order is specified by methods called `eq_attr` and `lt_attr`.

INPUT when creating the method:

- `eq_attr` – attribute name for equality comparison
- `lt_attr` – attribute name for less-than comparison

INPUT when calling the method:

- `self` – objects having methods `eq_attr` and `lt_attr`
- `other` – arbitrary object. If it does have `eq_attr` and `lt_attr` methods, these are used for the comparison. Otherwise, the comparison is undefined.
- `op` – a rich comparison operation (e.g. `op_EQ`)

Note: For efficiency, identical objects (when `self is other`) always compare equal.

Note: The order is partial, so `x <= y` is implemented as `x == y or x < y`. It is not required that this is the negation of `y < x`.

Note: This function is intended to be used as a method `_richcmp_` in a class derived from *`sage.structure.element.Element`* or a method `__richcmp__` in a class using *`richcmp_method()`*.

EXAMPLES:

```
sage: from sage.structure.richcmp import richcmp_by_eq_and_lt
sage: from sage.structure.element import Element

sage: class C(Element):
....:     def __init__(self, a, b):
....:         super().__init__(ZZ)
```

(continues on next page)

(continued from previous page)

```
....:         self.a = a
....:         self.b = b
....:     _richcmp_ = richcmp_by_eq_and_lt("eq", "lt")
....:     def eq(self, other):
....:         return self.a == other.a and self.b == other.b
....:     def lt(self, other):
....:         return self.a < other.a and self.b < other.b

sage: x = C(1,2); y = C(2,1); z = C(3,3)

sage: x == x, x <= x, x == C(1,2), x <= C(1,2)  # indirect doctest
(True, True, True, True)
sage: y == z, y != z
(False, True)

sage: x < y, y < x, x > y, y > x, x <= y, y <= x, x >= y, y >= x
(False, False, False, False, False, False, False, False)
sage: y < z, z < y, y > z, z > y, y <= z, z <= y, y >= z, z >= y
(True, False, False, True, True, False, False, True)
sage: z < x, x < z, z > x, x > z, z <= x, x <= z, z >= x, x >= z
(False, True, True, False, False, True, True, False)
```

A simple example using `richcmp_method`:

```
sage: from sage.structure.richcmp import richcmp_method, richcmp_by_eq_and_lt
sage: @richcmp_method
....: class C():
....:     __richcmp__ = richcmp_by_eq_and_lt("_eq", "_lt")
....:     def _eq(self, other):
....:         return True
....:     def _lt(self, other):
....:         return True
sage: a = C(); b = C()
sage: a == b
True
sage: a > b  # Calls b._lt(a)
True
sage: class X(): pass
sage: x = X()
sage: a == x  # Does not call a._eq(x) because x does not have _eq
False
```

sage.structure.richcmp.**richcmp_item**(*x*, *y*, *op*)

This function is meant to implement lexicographic rich comparison of sequences (lists, vectors, polynomials, …). The inputs `x` and `y` are corresponding items of such lists which should compared.

INPUT:

- `x`, `y` – arbitrary Python objects. Typically, these are `X[i]` and `Y[i]` for sequences `X` and `Y`.
- `op` – comparison operator (one of `op_LT`, ``op_LE`, `op_EQ`, `op_NE`, `op_GT`, `op_GE`)

OUTPUT:

Assuming that `x = X[i]` and `y = Y[i]`:

- if the comparison `X {op} Y` (where `op` is the given operation) could not be decided yet (i.e. we should compare the next items in the list): return `NotImplemented`
- otherwise, if the comparison `X {op} Y` could be decided: return `x {op} y`, which should then also be the result for `X {op} Y`.

Note: Since `x {op} y` cannot return `NotImplemented`, the two cases above are mutually exclusive.

The semantics of the comparison is different from Python lists or tuples in the case that the order is not total. Assume that `A` and `B` are lists whose rich comparison is implemented using `richcmp_item` (as in EXAMPLES below). Then

- `A == B` iff `A[i] == B[i]` for all indices i.
- `A != B` iff `A[i] != B[i]` for some index i.
- `A < B` iff `A[i] < B[i]` for some index i and for all $j < i$, `A[j] <= B[j]`.
- `A <= B` iff `A < B` or `A[i] <= B[i]` for all i.
- `A > B` iff `A[i] > B[i]` for some index i and for all $j < i$, `A[j] >= B[j]`.
- `A >= B` iff `A > B` or `A[i] >= B[i]` for all i.

See below for a detailed description of the exact semantics of `richcmp_item` in general.

EXAMPLES:

```
sage: from sage.structure.richcmp import *
sage: @richcmp_method
....: class Listcmp(list):
....:     def __richcmp__(self, other, op):
....:         for i in range(len(self)):  # Assume equal lengths
....:             res = richcmp_item(self[i], other[i], op)
....:             if res is not NotImplemented:
....:                 return res
....:         return rich_to_bool(op, 0)  # Consider the lists to be equal
sage: a = Listcmp([0, 1, 3])
sage: b = Listcmp([0, 2, 1])
sage: a == a
True
sage: a != a
False
sage: a < a
False
sage: a <= a
True
sage: a > a
False
sage: a >= a
True
sage: a == b, b == a
(False, False)
sage: a != b, b != a
(True, True)
sage: a < b, b > a
(True, True)
```

(continues on next page)

(continued from previous page)

```
sage: a <= b, b >= a
(True, True)
sage: a > b, b < a
(False, False)
sage: a >= b, b <= a
(False, False)
```

The above tests used a list of integers, where the result of comparisons are the same as for Python lists.

If we want to see the difference, we need more general entries in the list. The comparison rules are made to be consistent with setwise operations. If A and B are sets, we define `A {op} B` to be true if `a {op} B` is true for every a in A and b in B. Interval comparisons are a special case of this. For lists of non-empty(!) sets, we want that `[A1, A2] {op} [B1, B2]` is true if and only if `[a1, a2] {op} [b1, b2]` is true for all elements. We verify this:

```
sage: @richcmp_method
....: class Setcmp(tuple):
....:     def __richcmp__(self, other, op):
....:         return all(richcmp(x, y, op) for x in self for y in other)
sage: sym = {op_EQ: "==", op_NE: "!=", op_LT: "<", op_GT: ">", op_LE: "<=", op_GE:
↪">="}
sage: for A1 in Set(range(4)).subsets():  # long time
....:     if not A1: continue
....:     for B1 in Set(range(4)).subsets():
....:         if not B1: continue
....:         for A2 in Set(range(4)).subsets():
....:             if not A2: continue
....:             for B2 in Set(range(3)).subsets():
....:                 if not B2: continue
....:                 L1 = Listcmp([Setcmp(A1), Setcmp(A2)])
....:                 L2 = Listcmp([Setcmp(B1), Setcmp(B2)])
....:                 for op in range(6):
....:                     reslist = richcmp(L1, L2, op)
....:                     reselt = all(richcmp([a1, a2], [b1, b2], op) for a1 in A1␣
↪for a2 in A2 for b1 in B1 for b2 in B2)
....:                     assert reslist is reselt
```

EXACT SEMANTICS:

Above, we only described how `richcmp_item` behaves when it is used to compare sequences. Here, we specify the exact semantics. First of all, recall that the result of `richcmp_item(x, y, op)` is either `NotImplemented` or `x {op} y`.

- if op is ==: return `NotImplemented` if `x == y`. If `x == y` is false, then return `x == y`.
- if op is !=: return `NotImplemented` if not `x != y`. If `x != y` is true, then return `x != y`.
- if op is <: return `NotImplemented` if `x == y`. If `x < y` or not `x <= y`, return `x < y`. Otherwise (if both `x == y` and `x < y` are false but `x <= y` is true), return `NotImplemented`.
- if op is <=: return `NotImplemented` if `x == y`. If `x < y` or not `x <= y`, return `x <= y`. Otherwise (if both `x == y` and `x < y` are false but `x <= y` is true), return `NotImplemented`.
- the > and >= operators are analogous to < and <=.

sage.structure.richcmp.**richcmp_method**(*cls*)

Class decorator to implement rich comparison using the special method `__richcmp__` (analogous to Cython) instead of the 6 methods `__eq__` and friends.

This changes the class in-place and returns the given class.

EXAMPLES:

```
sage: from sage.structure.richcmp import *
sage: sym = {op_EQ: "==", op_NE: "!=", op_LT: "<", op_GT: ">", op_LE: "<=", op_GE:
↪">="}
sage: @richcmp_method
....: class A(str):
....:     def __richcmp__(self, other, op):
....:         print("%s %s %s" % (self, sym[op], other))
sage: A("left") < A("right")
left < right
sage: object() <= A("right")
right >= <object object at ...>
```

We can call this comparison with the usual Python special methods:

```
sage: x = A("left"); y = A("right")
sage: x.__eq__(y)
left == right
sage: A.__eq__(x, y)
left == right
```

Everything still works in subclasses:

```
sage: class B(A):
....:     pass
sage: x = B("left"); y = B("right")
sage: x != y
left != right
sage: x.__ne__(y)
left != right
sage: B.__ne__(x, y)
left != right
```

We can override `__richcmp__` with standard Python rich comparison methods and conversely:

```
sage: class C(A):
....:     def __ne__(self, other):
....:         return False
sage: C("left") != C("right")
False
sage: C("left") == C("right")  # Calls __eq__ from class A
left == right

sage: class Base():
....:     def __eq__(self, other):
....:         return False
sage: @richcmp_method
....: class Derived(Base):
```

(continues on next page)

(continued from previous page)

```
....:     def __richcmp__(self, other, op):
....:         return True
sage: Derived() == Derived()
True
```

sage.structure.richcmp.**richcmp_not_equal**(*x*, *y*, *op*)

Like `richcmp(x, y, op)` but assuming that x is not equal to y.

INPUT:

- `op` – a rich comparison operation (e.g. `Py_EQ`)

OUTPUT:

If `op` is not `op_EQ` or `op_NE`, the result of `richcmp(x, y, op)`. If `op` is `op_EQ`, return `False`. If `op` is `op_NE`, return `True`.

This is useful to compare lazily two objects A and B according to 2 (or more) different parameters, say width and height for example. One could use:

```
return richcmp((A.width(), A.height()), (B.width(), B.height()), op)
```

but this will compute both width and height in all cases, even if A.width() and B.width() are enough to decide the comparison.

Instead one can do:

```
wA = A.width()
wB = B.width()
if wA != wB:
    return richcmp_not_equal(wA, wB, op)
return richcmp(A.height(), B.height(), op)
```

The difference with `richcmp` is that `richcmp_not_equal` assumes that its arguments are not equal, which is excluding the case where the comparison cannot be decided so far, without knowing the rest of the parameters.

EXAMPLES:

```
sage: from sage.structure.richcmp import (richcmp_not_equal,
....:    op_EQ, op_NE, op_LT, op_LE, op_GT, op_GE)
sage: for op in (op_LT, op_LE, op_EQ, op_NE, op_GT, op_GE):
....:     print(richcmp_not_equal(3, 4, op))
True
True
False
True
False
False
sage: for op in (op_LT, op_LE, op_EQ, op_NE, op_GT, op_GE):
....:     print(richcmp_not_equal(5, 4, op))
False
False
False
True
True
True
```

6.2 Unique Representation

Abstract classes for cached and unique representation behavior.

See also:

sage.structure.factory.UniqueFactory

AUTHORS:

- Nicolas M. Thiery (2008): Original version.
- Simon A. King (2013-02): Separate cached and unique representation.
- Simon A. King (2013-08): Extended documentation.

6.2.1 What is a cached representation?

Instances of a class have a *cached representation behavior* when several instances constructed with the same arguments share the same memory representation. For example, calling twice:

```
sage: G = SymmetricGroup(6)                                                    #␣
↪needs sage.groups
sage: H = SymmetricGroup(6)                                                    #␣
↪needs sage.groups
```

to create the symmetric group on six elements gives back the same object:

```
sage: G is H                                                                   #␣
↪needs sage.groups
True
```

This is a standard design pattern. Besides saving memory, it allows for sharing cached data (say representation theoretical information about a group). And of course a look-up in the cache is faster than the creation of a new object.

Implementing a cached representation

Sage provides two standard ways to create a cached representation: *CachedRepresentation* and *UniqueFactory*. Note that, in spite of its name, *UniqueFactory* does not ensure *unique* representation behaviour, which will be explained below.

Using CachedRepresentation

It is often very easy to use *CachedRepresentation*: One simply writes a Python class and adds *CachedRepresentation* to the list of base classes. If one does so, then the arguments used to create an instance of this class will by default also be used as keys for the cache:

```
sage: from sage.structure.unique_representation import CachedRepresentation
sage: class C(CachedRepresentation):
....:     def __init__(self, a, b=0):
....:         self.a = a
....:         self.b = b
....:     def __repr__(self):
....:         return "C(%s, %s)"%(self.a, self.b)
```

(continues on next page)

(continued from previous page)

```
sage: a = C(1)
sage: a is C(1)
True
```

In addition, pickling just works, provided that Python is able to look up the class. Hence, in the following two lines, we explicitly put the class into the `__main__` module. This is needed in doctests, but not in an interactive session:

```
sage: import __main__
sage: __main__.C = C
sage: loads(dumps(a)) is a
True
```

Often, this very easy approach is sufficient for applications. However, there are some pitfalls. Since the arguments are used for caching, all arguments must be hashable, i.e., must be valid as dictionary keys:

```
sage: C((1,2))
C((1, 2), 0)
sage: C([1,2])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

In addition, equivalent ways of providing the arguments are *not* automatically normalised when forming the cache key, and hence different but equivalent arguments may yield distinct instances:

```
sage: C(1) is C(1,0)
False
sage: C(1) is C(a=1)
False
sage: repr(C(1)) == repr(C(a=1))
True
```

It should also be noted that the arguments are compared by equality, not by identity. This is often desired, but can imply subtle problems. For example, since `C(1)` already is in the cache, and since the unit elements in different finite fields are all equal to the integer one, we find:

```
sage: GF(5)(1) == 1 == GF(3)(1)
True
sage: C(1) is C(GF(3)(1)) is C(GF(5)(1))
True
```

But `C(2)` is not in the cache, and the number two is not equal in different finite fields (i. e., `GF(5)(2) == GF(3)(2)` returns as `False`), even though it is equal to the number two in the ring of integers (`GF(5)(2) == 2 == GF(3)(2)` returns as `True`; equality is not transitive when comparing elements of *distinct* algebraic structures!!). Hence, we have:

```
sage: GF(5)(2) == GF(3)(2)
False
sage: C(GF(3)(2)) is C(GF(5)(2))
False
```

Normalising the arguments

CachedRepresentation uses the metaclass ClasscallMetaclass. Its `__classcall__` method is a WeakCachedFunction. This function creates an instance of the given class using the given arguments, unless it finds the result in the cache. This has the following implications:

- The arguments must be valid dictionary keys (i.e., they must be hashable; see above).
- It is a weak cache, hence, if the user does not keep a reference to the resulting instance, then it may be removed from the cache during garbage collection.
- It is possible to preprocess the input arguments by implementing a `__classcall__` or a `__classcall_private__` method, but in order to benefit from caching, `CachedRepresentation.__classcall__()` should at some point be called.

Note: For technical reasons, it is needed that `__classcall__` respectively `__classcall_private__` are "static methods", i.e., they are callable objects that do not bind to an instance or class. For example, a `cached_function` can be used here, because it is callable, but does not bind to an instance or class, because it has no `__get__()` method. A usual Python function, however, has a `__get__()` method and would thus under normal circumstances bind to an instance or class, and thus the instance or class would be passed to the function as the first argument. To prevent a callable object from being bound to the instance or class, one can prepend the `@staticmethod` decorator to the definition; see `staticmethod`.

For more on Python's `__get__()` method, see: https://docs.python.org/2/howto/descriptor.html

Warning: If there is preprocessing, then the preprocessed arguments passed to `CachedRepresentation.__classcall__()` must be invariant under the preprocessing. That is to say, preprocessing the input arguments twice must have the same effect as preprocessing the input arguments only once. That is to say, the preprocessing must be idempotent.

The reason for this warning lies in the way pickling is implemented. If the preprocessed arguments are passed to `CachedRepresentation.__classcall__()`, then the resulting instance will store the *preprocessed* arguments in some attribute, and will use them for pickling. If the pickle is unpickled, then preprocessing is applied to the preprocessed arguments—and this second round of preprocessing must not change the arguments further, since otherwise a different instance would be created.

We illustrate the warning by an example. Imagine that one has instances that are created with an integer-valued argument, but only depend on the *square* of the argument. It would be a mistake to square the given argument during preprocessing:

```
sage: class WrongUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super().__classcall__(cls, n^2)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "Something(%d)"%self.n
sage: import __main__
sage: __main__.WrongUsage = WrongUsage # This is only needed in doctests
sage: w = WrongUsage(3); w
Something(9)
```

(continues on next page)

(continued from previous page)

```
sage: w._reduction
(<class '__main__.WrongUsage'>, (9,), {})
```

Indeed, the reduction data are obtained from the preprocessed argument. By consequence, if the resulting instance is pickled and unpickled, the argument gets squared *again*:

```
sage: loads(dumps(w))
Something(81)
```

Instead, the preprocessing should only take the absolute value of the given argument, while the squaring should happen inside of the `__init__` method, where it won't mess with the cache:

```
sage: class BetterUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super().__classcall__(cls, abs(n))
....:     def __init__(self, n):
....:         self.n = n^2
....:     def __repr__(self):
....:         return "SomethingElse(%d)"%self.n
sage: __main__.BetterUsage = BetterUsage # This is only needed in doctests
sage: b = BetterUsage(3); b
SomethingElse(9)
sage: loads(dumps(b)) is b
True
sage: b is BetterUsage(-3)
True
```

In our next example, we create a cached representation class `C` that returns an instance of a sub-class `C1` or `C2` depending on the given arguments. This is implemented in a static `__classcall_private__` method of `C`, letting it choose the sub-class according to the given arguments. Since a `__classcall_private__` method will be ignored on sub-classes, the caching of `CachedRepresentation` is available to both `C1` and `C2`. But for illustration, we overload the static `__classcall__` method on `C2`, doing some argument preprocessing. We also create a sub-class `C2b` of `C2`, demonstrating that the `__classcall__` method is used on the sub-class (in contrast to a `__classcall_private__` method!).

```
sage: class C(CachedRepresentation):
....:     @staticmethod
....:     def __classcall_private__(cls, n, implementation=0):
....:         if not implementation:
....:             return C.__classcall__(cls, n)
....:         if implementation==1:
....:             return C1(n)
....:         if implementation>1:
....:             return C2(n,implementation)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "C(%d, 0)"%self.n
sage: class C1(C):
....:     def __repr__(self):
....:         return "C1(%d)"%self.n
sage: class C2(C):
```

(continues on next page)

(continued from previous page)

```
....:     @staticmethod
....:     def __classcall__(cls, n, implementation=0):
....:         if implementation:
....:             return super().__classcall__(cls, (n,)*implementation)
....:         return super().__classcall__(cls, n)
....:     def __init__(self, t):
....:         self.t = t
....:     def __repr__(self):
....:         return "C2(%s)"%repr(self.t)
sage: class C2b(C2):
....:     def __repr__(self):
....:         return "C2b(%s)"%repr(self.t)
sage: __main__.C2 = C2       # not needed in an interactive session
sage: __main__.C2b = C2b
```

In the above example, `C` drops the argument `implementation` if it evaluates to `False`, and since the cached `__classcall__` is called in this case, we have:

```
sage: C(1)
C(1, 0)
sage: C(1) is C(1,0)
True
sage: C(1) is C(1,0) is C(1,None) is C(1,[])
True
```

(Note that we were able to bypass the issue of arguments having to be hashable by catching the empty list `[]` during preprocessing in the `__classcall_private__` method. Similarly, unhashable arguments can be made hashable – e. g., lists normalized to tuples – in the `__classcall_private__` method before they are further delegated to `__classcall__`. See `TCrystal` for an example.)

If we call `C1` directly or if we provide `implementation=1` to `C`, we obtain an instance of `C1`. Since it uses the `__classcall__` method inherited from *`CachedRepresentation`*, the resulting instances are cached:

```
sage: C1(2)
C1(2)
sage: C(2, implementation=1)
C1(2)
sage: C(2, implementation=1) is C1(2)
True
```

The class `C2` preprocesses the input arguments. Instances can, again, be obtained directly or by calling `C`:

```
sage: C(1, implementation=3)
C2((1, 1, 1))
sage: C(1, implementation=3) is C2(1,3)
True
```

The argument preprocessing of `C2` is inherited by `C2b`, since `__classcall__` and not `__classcall_private__` is used. Pickling works, since the preprocessing of arguments is idempotent:

```
sage: c2b = C2b(2,3); c2b
C2b((2, 2, 2))
sage: loads(dumps(c2b)) is c2b
True
```

Using `UniqueFactory`

For creating a cached representation using a factory, one has to

- create a class *separately* from the factory. This class **must** inherit from `object`. Its instances **must** allow attribute assignment.
- write a method `create_key` (or `create_key_and_extra_args`) that creates the cache key from the given arguments.
- write a method `create_object` that creates an instance of the class from a given cache key.
- create an instance of the factory with a name that allows to conclude where it is defined.

An example:

```
sage: class C():
....:     def __init__(self, t):
....:         self.t = t
....:     def __repr__(self):
....:         return "C%s"%repr(self.t)
sage: from sage.structure.factory import UniqueFactory
sage: class MyFactory(UniqueFactory):
....:     def create_key(self, n, m=None):
....:         if isinstance(n, (tuple,list)) and m is None:
....:             return tuple(n)
....:         return (n,)*m
....:     def create_object(self, version, key, **extra_args):
....:         # We ignore version and extra_args
....:         return C(key)
```

Now, we define an instance of the factory, stating that it can be found under the name `"F"` in the `__main__` module. By consequence, pickling works:

```
sage: F = MyFactory("__main__.F")
sage: __main__.F = F                # not needed in an interactive session
sage: loads(dumps(F)) is F
True
```

We can now create *cached* instances of `C` by calling the factory. The cache only takes into account the key computed with the method `create_key` that we provided. Hence, different given arguments may result in the same instance. Note that, again, the cache is weak, hence, the instance might be removed from the cache during garbage collection, unless an external reference is preserved.

```
sage: a = F(1, 2); a
C(1, 1)
sage: a is F((1,1))
True
```

If the class of the returned instances is a sub-class of `object`, and **if** the resulting instance allows attribute assignment, then pickling of the resulting instances is automatically provided for, and respects the cache.

```
sage: loads(dumps(a)) is a
True
```

This is because an attribute is stored that explains how the instance was created:

```
sage: a._factory_data
(<__main__.MyFactory object at ...>, (...), (1, 1), {})
```

Note: If a class is used that does not inherit from `object` then unique pickling is *not* provided.

Caching is only available if the factory is called. If an instance of the class is directly created, then the cache is not used:

```
sage: C((1,1))
C(1, 1)
sage: C((1,1)) is a
False
```

Comparing the two ways of implementing a cached representation

In this sub-section, we discuss advantages and disadvantages of the two ways of implementing a cached representation, depending on the type of application.

Simplicity and transparency

In many cases, turning a class into a cached representation requires nothing more than adding *CachedRepresentation* to the list of base classes of this class. This is, of course, a very easy and convenient way. Writing a factory would involve a lot more work.

If preprocessing of the arguments is needed, then we have seen how to do this by a `__classcall_private__` or `__classcall__` method. But these are double underscore methods and hence, for example, invisible in the automatically created reference manual. Moreover, preprocessing *and* caching are implemented in the same method, which might be confusing. In a unique factory, these two tasks are cleanly implemented in two separate methods. With a factory, it is possible to create the resulting instance by arguments that are different from the key used for caching. This is significantly restricted with CachedRepresentation due to the requirement that argument preprocessing be idempotent.

Hence, if advanced preprocessing is needed, then *UniqueFactory* might be easier and more transparent to use than *CachedRepresentation*.

Class inheritance

Using *CachedRepresentation* has the advantage that one has a class and creates cached instances of this class by the usual Python syntax:

```
sage: G = SymmetricGroup(6)                                                     # needs sage.groups
sage: issubclass(SymmetricGroup, sage.structure.unique_representation.CachedRepresentation)     # needs sage.groups
True
sage: isinstance(G, SymmetricGroup)                                             # needs sage.groups
True
```

In contrast, a factory is just a callable object that returns something that has absolutely nothing to do with the factory, and may in fact return instances of quite different classes:

```
sage: isinstance(GF, sage.structure.factory.UniqueFactory)
True
sage: K5 = GF(5)
sage: type(K5)
<class 'sage.rings.finite_rings.finite_field_prime_modn.FiniteField_prime_modn_with_
↪category'>

sage: # needs sage.rings.finite_rings
sage: K25 = GF(25, 'x')
sage: type(K25)                                                                  #␣
↪needs sage.libs.linbox
<class 'sage.rings.finite_rings.finite_field_givaro.FiniteField_givaro_with_category'>
sage: Kp = GF(next_prime_power(1000000)^2, 'x')
sage: type(Kp)
<class 'sage.rings.finite_rings.finite_field_pari_ffelt.FiniteField_pari_ffelt_with_
↪category'>
```

This can be confusing to the user. Namely, the user might determine the class of an instance and try to create further instances by calling the class rather than the factory—which is a mistake since it works around the cache (and also since the class might be more restrictive than the factory – i. e., the type of `K5` in the above doctest cannot be called on a prime power which is not a prime). This mistake can more easily be avoided by using *`CachedRepresentation`*.

We have seen above that one can easily create new cached-representation classes by subclassing an existing cached-representation class, even making use of an existing argument preprocess. This would be much more complicated with a factory. Namely, one would need to rewrite old factories making them aware of the new classes, and/or write new factories for the new classes.

Python versus extension classes

`CachedRepresentation` uses a metaclass, namely `ClasscallMetaclass`. Hence, it can currently not be a Cython extension class. Moreover, it is supposed to be used by providing it as a base class. But in typical applications, one also has another base class, say, *`Parent`*. Hence, one would like to create a class with at least two base classes, which is currently impossible in Cython extension classes.

In other words, when using *`CachedRepresentation`*, one must work with Python classes. These can be defined in Cython code (`.pyx` files) and can thus benefit from Cython's speed inside of their methods, but they must not be `cdef class` and can thus not use `cdef` attributes or methods.

Such restrictions do not exist when using a factory. However, if attribute assignment does not work, then the automatic pickling provided by *`UniqueFactory`* will not be available.

6.2.2 What is a unique representation?

Instances of a class have a *unique instance behavior* when instances of this class evaluate equal if and only if they are identical. Sage provides the base class `WithEqualityById`, which provides comparison by identity and a hash that is determined by the memory address of the instance. Both the equality test and the hash are implemented in Cython and are very fast, even when one has a Python class inheriting from `WithEqualityById`.

In many applications, one wants to combine unique instance and cached representation behaviour. This is called *unique representation* behaviour. We have seen above that symmetric groups have a *cached* representation behaviour. However, they do not show the *unique* representation behaviour, since they are equal to groups created in a totally different way, namely to subgroups:

```
sage: # needs sage.groups
sage: G = SymmetricGroup(6)
sage: G3 = G.subgroup([G((1,2,3,4,5,6)), G((1,2))])
sage: G is G3
False
sage: type(G) == type(G3)
False
sage: G == G3
True
```

The unique representation behaviour can conveniently be implemented with a class that inherits from *UniqueRepresentation*: By adding *UniqueRepresentation* to the base classes, the class will simultaneously inherit from *CachedRepresentation* and from WithEqualityById.

For example, a symmetric function algebra is uniquely determined by the base ring. Thus, it is reasonable to use *UniqueRepresentation* in this case:

```
sage: isinstance(SymmetricFunctions(CC), SymmetricFunctions)                    #␣
↪needs sage.combinat
True
sage: issubclass(SymmetricFunctions, UniqueRepresentation)                      #␣
↪needs sage.combinat
True
```

UniqueRepresentation differs from *CachedRepresentation* only by adding WithEqualityById as a base class. Hence, the above examples of argument preprocessing work for *UniqueRepresentation* as well.

Note that a cached representation created with *UniqueFactory* does *not* automatically provide unique representation behaviour, in spite of its name! Hence, for unique representation behaviour, one has to implement hash and equality test accordingly, for example by inheriting from WithEqualityById.

class sage.structure.unique_representation.**CachedRepresentation**

Bases: object

Classes derived from CachedRepresentation inherit a weak cache for their instances.

Note: If this class is used as a base class, then instances are (weakly) cached, according to the arguments used to create the instance. Pickling is provided, of course by using the cache.

Note: Using this class, one can have arbitrary hash and comparison. Hence, *unique* representation behaviour is *not* provided.

See also:

UniqueRepresentation, *unique_representation*

EXAMPLES:

Providing a class with a weak cache for the instances is easy: Just inherit from *CachedRepresentation*:

```
sage: from sage.structure.unique_representation import CachedRepresentation
sage: class MyClass(CachedRepresentation):
....:     # all the rest as usual
....:     pass
```

We start with a simple class whose constructor takes a single value as argument (TODO: find a more meaningful example):

```
sage: class MyClass(CachedRepresentation):
....:     def __init__(self, value):
....:         self.value = value
....:     def __eq__(self, other):
....:         if type(self) != type(other):
....:             return False
....:         return self.value == other.value
```

Two coexisting instances of `MyClass` created with the same argument data are guaranteed to share the same identity. Since github issue #12215, this is only the case if there is some strong reference to the returned instance, since otherwise it may be garbage collected:

```
sage: x = MyClass(1)
sage: y = MyClass(1)
sage: x is y               # There is a strong reference
True
sage: z = MyClass(2)
sage: x is z
False
```

In particular, modifying any one of them modifies the other (reference effect):

```
sage: x.value = 3
sage: x.value, y.value
(3, 3)
sage: y.value = 1
sage: x.value, y.value
(1, 1)
```

The arguments can consist of any combination of positional or keyword arguments, as taken by a usual `__init__` function. However, all values passed in should be hashable:

```
sage: MyClass(value = [1,2,3])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Argument preprocessing

Sometimes, one wants to do some preprocessing on the arguments, to put them in some canonical form. The following example illustrates how to achieve this; it takes as argument any iterable, and canonicalizes it into a tuple (which is hashable!):

```
sage: class MyClass2(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, iterable):
....:         t = tuple(iterable)
....:         return super().__classcall__(cls, t)
....:
....:     def __init__(self, value):
```

(continues on next page)

(continued from previous page)

```
....:         self.value = value
sage: x = MyClass2([1,2,3])
sage: y = MyClass2(tuple([1,2,3]))
sage: z = MyClass2(i for i in [1,2,3])
sage: x.value
(1, 2, 3)
sage: x is y, y is z
(True, True)
```

A similar situation arises when the constructor accepts default values for some of its parameters. Alas, the obvious implementation does not work:

```
sage: class MyClass3(CachedRepresentation):
....:     def __init__(self, value = 3):
....:         self.value = value
sage: MyClass3(3) is MyClass3()
False
```

Instead, one should do:

```
sage: class MyClass3(UniqueRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, value = 3):
....:         return super().__classcall__(cls, value)
....:
....:     def __init__(self, value):
....:         self.value = value
sage: MyClass3(3) is MyClass3()
True
```

A bit of explanation is in order. First, the call `MyClass2([1,2,3])` triggers a call to `MyClass2.__classcall__(MyClass2, [1,2,3])`. This is an extension of the standard Python behavior, needed by *CachedRepresentation*, and implemented by the `ClasscallMetaclass`. Then, `MyClass2.__classcall__` does the desired transformations on the arguments. Finally, it uses `super` to call the default implementation of `__classcall__` provided by *CachedRepresentation*. This one in turn handles the caching and, if needed, constructs and initializes a new object in the class using `__new__` and `__init__` as usual.

Constraints:

- `__classcall__()` is a staticmethod (like, implicitly, `__new__`)
- the preprocessing on the arguments should be idempotent. That is, if `MyClass2.__classcall__(<arguments>)` calls `CachedRepresentation.__classcall__(<preprocessed_arguments>)`, then `MyClass2.__classcall__(<preprocessed_arguments>)` should also result in a call to `CachedRepresentation.__classcall__(<preprocessed_arguments>)`.
- `MyClass2.__classcall__` should return the result of `CachedRepresentation.__classcall__()` without modifying it.

Other than that `MyClass2.__classcall__` may play any tricks, like acting as a factory and returning objects from other classes.

Warning: It is possible, but strongly discouraged, to let the `__classcall__` method of a class `C` return objects that are not instances of `C`. Of course, instances of a *subclass* of `C` are fine. Compare the examples in *unique_representation*.

We illustrate what is meant by an "idempotent" preprocessing. Imagine that one has instances that are created with an integer-valued argument, but only depend on the *square* of the argument. It would be a mistake to square the given argument during preprocessing:

```
sage: class WrongUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super().__classcall__(cls, n^2)
....:     def __init__(self, n):
....:         self.n = n
....:     def __repr__(self):
....:         return "Something(%d)"%self.n
sage: import __main__
sage: __main__.WrongUsage = WrongUsage # This is only needed in doctests
sage: w = WrongUsage(3); w
Something(9)
sage: w._reduction
(<class '__main__.WrongUsage'>, (9,), {})
```

Indeed, the reduction data are obtained from the preprocessed arguments. By consequence, if the resulting instance is pickled and unpickled, the argument gets squared *again*:

```
sage: loads(dumps(w))
Something(81)
```

Instead, the preprocessing should only take the absolute value of the given argument, while the squaring should happen inside of the `__init__` method, where it won't mess with the cache:

```
sage: class BetterUsage(CachedRepresentation):
....:     @staticmethod
....:     def __classcall__(cls, n):
....:         return super().__classcall__(cls, abs(n))
....:     def __init__(self, n):
....:         self.n = n^2
....:     def __repr__(self):
....:         return "SomethingElse(%d)"%self.n
sage: __main__.BetterUsage = BetterUsage # This is only needed in doctests
sage: b = BetterUsage(3); b
SomethingElse(9)
sage: loads(dumps(b)) is b
True
sage: b is BetterUsage(-3)
True
```

Cached representation and mutability

CachedRepresentation is primarily intended for implementing objects which are (at least semantically) immutable. This is in particular assumed by the default implementations of `copy` and `deepcopy`:

```
sage: copy(x) is x
True
sage: from copy import deepcopy
sage: deepcopy(x) is x
True
```

However, in contrast to *UniqueRepresentation*, using *CachedRepresentation* allows for a comparison that is not by identity:

```
sage: t = MyClass(3)
sage: z = MyClass(2)
sage: t.value = 2
```

Now `t` and `z` are non-identical, but equal:

```
sage: t.value == z.value
True
sage: t == z
True
sage: t is z
False
```

More on cached representation and identity

CachedRepresentation is implemented by means of a cache. This cache uses weak references in general, but strong references to the most recently created objects. Hence, when all other references to, say, `MyClass(1)` have been deleted, the instance is eventually deleted from memory (after enough other objects have been created to remove the strong reference to `MyClass(1)`). A later call to `MyClass(1)` reconstructs the instance from scratch:

```
sage: class SomeClass(UniqueRepresentation):
....:     def __init__(self, i):
....:         print("creating new instance for argument %s" % i)
....:         self.i = i
....:     def __del__(self):
....:         print("deleting instance for argument %s" % self.i)
sage: class OtherClass(UniqueRepresentation):
....:     def __init__(self, i):
....:         pass
sage: O = SomeClass(1)
creating new instance for argument 1
sage: O is SomeClass(1)
True
sage: O is SomeClass(2)
creating new instance for argument 2
False
sage: L = [OtherClass(i) for i in range(200)]
deleting instance for argument 2
sage: del O
```

(continues on next page)

(continued from previous page)

```
deleting instance for argument 1
sage: O = SomeClass(1)
creating new instance for argument 1
sage: del O
sage: del L
sage: L = [OtherClass(i) for i in range(200)]
deleting instance for argument 1
```

Cached representation and pickling

The default Python pickling implementation (by reconstructing an object from its class and dictionary, see "The pickle protocol" in the Python Library Reference) does not preserve cached representation, as Python has no chance to know whether and where the same object already exists.

`CachedRepresentation` tries to ensure appropriate pickling by implementing a `__reduce__` method returning the arguments passed to the constructor:

```
sage: import __main__           # Fake MyClass being defined in a python module
sage: __main__.MyClass = MyClass
sage: x = MyClass(1)
sage: loads(dumps(x)) is x
True
```

`CachedRepresentation` uses the `__reduce__` pickle protocol rather than `__getnewargs__` because the latter does not handle keyword arguments:

```
sage: x = MyClass(value = 1)
sage: x.__reduce__()
(<function unreduce at ...>, (<class '__main__.MyClass'>, (), {'value': 1}))
sage: x is loads(dumps(x))
True
```

Note: The default implementation of `__reduce__` in `CachedRepresentation` requires to store the constructor's arguments in the instance dictionary upon construction:

```
sage: x.__dict__
{'_reduction': (<class '__main__.MyClass'>, (), {'value': 1}), 'value': 1}
```

It is often easy in a derived subclass to reconstruct the constructor's arguments from the instance data structure. When this is the case, `__reduce__` should be overridden; automagically the arguments won't be stored anymore:

```
sage: class MyClass3(UniqueRepresentation):
....:     def __init__(self, value):
....:         self.value = value
....:
....:     def __reduce__(self):
....:         return (MyClass3, (self.value,))
sage: import __main__; __main__.MyClass3 = MyClass3  # Fake MyClass3 being defined␣
↪in a python module
sage: x = MyClass3(1)
sage: loads(dumps(x)) is x
```

(continues on next page)

(continued from previous page)

```
True
sage: x.__dict__
{'value': 1}
```

Migrating classes to `CachedRepresentation` and unpickling

We check that, when migrating a class to *CachedRepresentation*, older pickles can still be reasonably unpickled. Let us create a (new style) class, and pickle one of its instances:

```
sage: class MyClass4():
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined␣
↪in a python module
sage: pickle = dumps(MyClass4(1))
```

It can be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Now, we upgrade the class to derive from *UniqueRepresentation*, which inherits from *CachedRepresentation*:

```
sage: class MyClass4(UniqueRepresentation, object):
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined␣
↪in a python module
sage: __main__.MyClass4 = MyClass4
```

The pickle can still be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Note however that, for the reasons explained above, unique representation is not guaranteed in this case:

```
sage: y is MyClass4(1)
False
```

Todo: Illustrate how this can be fixed on a case by case basis.

Now, we redo the same test for a class deriving from SageObject:

```
sage: class MyClass4(SageObject):
....:     def __init__(self, value):
```

(continues on next page)

(continued from previous page)

```
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined␣
↪in a python module
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
....:     def __init__(self, value):
....:         self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)
sage: y.value
1
```

Caveat: unpickling instances of a formerly old-style class is not supported yet by default:

```
sage: class MyClass4:
....:     def __init__(self, value):
....:         self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined␣
↪in a python module
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
....:     def __init__(self, value):
....:         self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)  # todo: not implemented
sage: y.value            # todo: not implemented
1
```

Rationale for the current implementation

CachedRepresentation and derived classes use the `ClasscallMetaclass` of the standard Python type. The following example explains why.

We define a variant of `MyClass` where the calls to `__init__` are traced:

```
sage: class MyClass(CachedRepresentation):
....:     def __init__(self, value):
....:         print("initializing object")
....:         self.value = value
```

Let us create an object twice:

```
sage: x = MyClass(1)
initializing object
sage: z = MyClass(1)
```

As desired the `__init__` method was only called the first time, which is an important feature.

As far as we can tell, this is not achievable while just using `__new__` and `__init__` (as defined by type; see Section Basic Customization in the Python Reference Manual). Indeed, `__init__` is called systematically on the result of `__new__` whenever the result is an instance of the class.

Another difficulty is that argument preprocessing (as in the example above) cannot be handled by `__new__`, since the unprocessed arguments will be passed down to `__init__`.

class sage.structure.unique_representation.**UniqueRepresentation**

Bases: *CachedRepresentation*, WithEqualityById

Classes derived from UniqueRepresentation inherit a unique representation behavior for their instances.

See also:

unique_representation

EXAMPLES:

The short story: to construct a class whose instances have a unique representation behavior one just has to do:

```
sage: class MyClass(UniqueRepresentation):
....:     # all the rest as usual
....:     pass
```

Everything below is for the curious or for advanced usage.

What is unique representation?

Instances of a class have a *unique representation behavior* when instances evaluate equal if and only if they are identical (i.e., share the same memory representation), if and only if they were created using equal arguments. For example, calling twice:

```
sage: f = SymmetricFunctions(QQ)                                    #␣
↪needs sage.combinat sage.modules
sage: g = SymmetricFunctions(QQ)                                    #␣
↪needs sage.combinat sage.modules
```

to create the symmetric function algebra over $\mathbf{Q}$ actually gives back the same object:

```
sage: f == g                                                        #␣
↪needs sage.combinat sage.modules
True
sage: f is g                                                        #␣
↪needs sage.combinat sage.modules
True
```

This is a standard design pattern. It allows for sharing cached data (say representation theoretical information about a group) as well as for very fast hashing and equality testing. This behaviour is typically desirable for parents and categories. It can also be useful for intensive computations where one wants to cache all the operations on a small set of elements (say the multiplication table of a small group), and access this cache as quickly as possible.

UniqueRepresentation is very easy to use: a class just needs to derive from it, or make sure some of its super classes does. Also, it groups together the class and the factory in a single gadget:

```
sage: isinstance(SymmetricFunctions(CC), SymmetricFunctions)        #␣
↪needs sage.combinat sage.modules
True
sage: issubclass(SymmetricFunctions, UniqueRepresentation)          #␣
↪needs sage.combinat sage.modules
True
```

This nice behaviour is not available when one just uses a factory:

```
sage: isinstance(GF(7), GF)
Traceback (most recent call last):
...
TypeError: isinstance() arg 2 must be a type...

sage: isinstance(GF, sage.structure.factory.UniqueFactory)
True
```

In addition, *UniqueFactory* only provides the *cached* representation behaviour, but not the *unique* representation behaviour—the examples in *unique_representation* explain this difference.

On the other hand, the *UniqueRepresentation* class is more intrusive, as it imposes a behavior (and a metaclass) on all the subclasses. In particular, the unique representation behaviour is imposed on *all* subclasses (unless the `__classcall__` method is overloaded and not called in the subclass, which is not recommended). Its implementation is also more technical, which leads to some subtleties.

EXAMPLES:

We start with a simple class whose constructor takes a single value as argument. This pattern is similar to what is done in `sage.combinat.sf.sf.SymmetricFunctions`:

```
sage: class MyClass(UniqueRepresentation):
....:     def __init__(self, value):
....:         self.value = value
```

Two coexisting instances of `MyClass` created with the same argument data are guaranteed to share the same identity. Since github issue #12215, this is only the case if there is some strong reference to the returned instance, since otherwise it may be garbage collected:

```
sage: x = MyClass(1)
sage: y = MyClass(1)
sage: x is y               # There is a strong reference
True
sage: z = MyClass(2)
sage: x is z
False
```

In particular, modifying any one of them modifies the other (reference effect):

```
sage: x.value = 3
sage: x.value, y.value
(3, 3)
sage: y.value = 1
sage: x.value, y.value
(1, 1)
```

When comparing two instances of a unique representation with == or != comparison by identity is used:

```
sage: x == y
True
sage: x is y
True
sage: z = MyClass(2)
sage: x == z
```

(continues on next page)

(continued from previous page)

```
False
sage: x is z
False
sage: x != y
False
sage: x != z
True
```

A hash function equivalent to `object.__hash__()` is used, which is compatible with comparison by identity. However this means that the hash function may change in between Sage sessions, or even within the same Sage session.

```
sage: hash(x) == object.__hash__(x)
True
```

> **Warning:** It is possible to inherit from *UniqueRepresentation* and then overload comparison in a way that destroys the unique representation property. We strongly recommend against it! You should use *CachedRepresentation* instead.

Mixing super types and super classes

`sage.structure.unique_representation.`**`unreduce`**(*cls*, *args*, *keywords*)

Calls a class on the given arguments:

```
sage: sage.structure.unique_representation.unreduce(Integer, (1,), {})
1
```

Todo: should reuse something preexisting …

6.3 Factory for cached representations

See also:

sage.structure.unique_representation

Using a *UniqueFactory* is one way of implementing a *cached representation behaviour*. In spite of its name, using a *UniqueFactory* is not enough to ensure the *unique representation behaviour*. See *unique_representation* for a detailed explanation.

With a *UniqueFactory*, one can preprocess the given arguments. There is special support for specifying a subset of the arguments that serve as the unique key, so that still *all* given arguments are used to create a new instance, but only the specified subset is used to look up in the cache. Typically, this is used to construct objects that accept an optional `check=[True|False]` argument, but whose result should be unique regardless of said optional argument. (This use case should be handled with care, though: Any checking which isn't done in the `create_key` or `create_key_and_extra_args` method will be done only when a new object is generated, but not when a cached object is retrieved from cache. Consequently, if the factory is once called with `check=False`, a subsequent call with `check=True` cannot be expected to perform all checks unless these checks are all in the `create_key` or `create_key_and_extra_args` method.)

For a class derived from *CachedRepresentation*, argument preprocessing can be obtained by providing a custom static `__classcall__` or `__classcall_private__` method, but this seems less transparent. When argument preprocessing is not needed or the preprocess is not very sophisticated, then generally *CachedRepresentation* is much easier to use than a factory.

AUTHORS:

- Robert Bradshaw (2008): initial version.
- Simon King (2013): extended documentation.
- Julian Rueth (2014-05-09): use `_cache_key` if parameters are unhashable

class sage.structure.factory.**UniqueFactory**

Bases: *SageObject*

This class is intended to make it easy to cache objects.

It is based on the idea that the object is uniquely defined by a set of defining data (the key). There is also the possibility of some non-defining data (extra args) which will be used in initial creation, but not affect the caching.

Warning: This class only provides *cached representation behaviour*. Hence, using *UniqueFactory*, it is still possible to create distinct objects that evaluate equal. Unique representation behaviour can be added, for example, by additionally inheriting from `sage.misc.fast_methods.WithEqualityById`.

The objects created are cached (using weakrefs) based on their key and returned directly rather than re-created if requested again. Pickling is taken care of by the factory, and will return the same object for the same version of Sage, and distinct (but hopefully equal) objects for different versions of Sage.

Warning: The objects returned by a *UniqueFactory* must be instances of new style classes (hence, they must be instances of `object`) that must not only allow a weak reference, but must accept general attribute assignment. Otherwise, pickling won't work.

USAGE:

A *unique factory* provides a way to create objects from parameters (the type of these objects can depend on the parameters, and is often determined only at runtime) and to cache them by a certain key derived from these parameters, so that when the factory is being called again with the same parameters (or just with parameters which yield the same key), the object is being returned from cache rather than constructed anew.

An implementation of a unique factory consists of a factory class and an instance of this factory class.

The factory class has to be a class inheriting from `UniqueFactory`. Typically it only needs to implement *create_key()* (a method that creates a key from the given parameters, under which key the object will be stored in the cache) and *create_object()* (a method that returns the actual object from the key). Sometimes, one would also implement *create_key_and_extra_args()* (this differs from *create_key()* in allowing to also create some additional arguments from the given parameters, which arguments then get passed to *create_object()* and thus can have an effect on the initial creation of the object, but do *not* affect the key) or *other_keys()*. Other methods are not supposed to be overloaded.

The factory class itself cannot be called to create objects. Instead, an instance of the factory class has to be created first. For technical reasons, this instance has to be provided with a name that allows Sage to find its definition. Specifically, the name of the factory instance (or the full path to it, if it is not in the global namespace) has to be passed to the factory class as a string variable. So, if our factory class has been called `A` and is located in `sage/spam/battletoads.py`, then we need to define an instance (say, `B`) of `A` by writing `B = A("sage.spam.battletoads.B")` (or `B = A("B")` if this `B` will be imported into global namespace). This instance can then be used to create objects (by calling `B(*parameters)`).

Notice that the objects created by the factory don't inherit from the factory class. They *do* know about the factory that created them (this information, along with the keys under which this factory caches them, is stored in the `_factory_data` attributes of the objects), but not via inheritance.

EXAMPLES:

The below examples are rather artificial and illustrate particular aspects. For a "real-life" usage case of `UniqueFactory`, see the finite field factory in `sage.rings.finite_rings.finite_field_constructor`.

In many cases, a factory class is implemented by providing the two methods `create_key()` and `create_object()`. In our example, we want to demonstrate how to use "extra arguments" to choose a specific implementation, with preference given to an instance found in the cache, even if its implementation is different. Hence, we implement `create_key_and_extra_args()` rather than `create_key()`, putting the chosen implementation into the extra arguments. Then, in the `create_object()` method, we create and return instances of the specified implementation.

```
sage: from sage.structure.factory import UniqueFactory
sage: class MyFactory(UniqueFactory):
....:     def create_key_and_extra_args(self, *args, **kwds):
....:         return args, {'impl':kwds.get('impl', None)}
....:     def create_object(self, version, key, **extra_args):
....:         impl = extra_args['impl']
....:         if impl=='C':
....:             return C(*key)
....:         if impl=='D':
....:             return D(*key)
....:         return E(*key)
....:
```

Now we can create a factory instance. It is supposed to be found under the name `"F"` in the `"__main__"` module. Note that in an interactive session, `F` would automatically be in the `__main__` module. Hence, the second and third of the following four lines are only needed in doctests.

```
sage: F = MyFactory("__main__.F")
sage: import __main__
sage: __main__.F = F
sage: loads(dumps(F)) is F
True
```

Now we create three classes `C`, `D` and `E`. The first is a Cython extension-type class that does not allow weak references nor attribute assignment. The second is a Python class that is not derived from `object`. The third allows attribute assignment and is derived from `object`.

```
sage: cython("cdef class C: pass")                                          # needs sage.misc.cython
sage: class D:
....:     def __init__(self, *args):
....:         self.t = args
....:     def __repr__(self):
....:         return "D%s"%repr(self.t)
....:
sage: class E(D, object): pass
```

Again, being in a doctest, we need to put the class `D` into the `__main__` module, so that Python can find it:

```
sage: import __main__
sage: __main__.D = D
```

It is impossible to create an instance of C with our factory, since it does not allow weak references:

```
sage: F(1, impl='C')                                                    # needs sage.misc.cython
Traceback (most recent call last):
...
TypeError: cannot create weak reference to '....C' object
```

Let us try again, with a Cython class that does allow weak references. Now, creation of an instance using the factory works:

```
sage: cython(                                                           # needs sage.misc.cython
....: '''
....: cdef class C:
....:     cdef __weakref__
....: ''')
....:
sage: c = F(1, impl='C')                                                # needs sage.misc.cython
sage: isinstance(c, C)                                                  # needs sage.misc.cython
True
```

The cache is used when calling the factory again—even if it is suggested to use a different implementation. This is because the implementation is only considered an "extra argument" that does not count for the key.

```
sage: c is F(1, impl='C') is F(1, impl="D") is F(1)                    # needs sage.misc.cython
True
```

However, pickling and unpickling does not use the cache. This is because the factory has tried to assign an attribute to the instance that provides information on the key used to create the instance, but failed:

```
sage: loads(dumps(c)) is c                                              # needs sage.misc.cython
False
sage: hasattr(c, '_factory_data')                                       # needs sage.misc.cython
False
```

We have already seen that our factory will only take the requested implementation into account if the arguments used as key have not been used yet. So, we use other arguments to create an instance of class D:

```
sage: d = F(2, impl='D')
sage: isinstance(d, D)
True
```

The factory only knows about the pickling protocol used by new style classes. Hence, again, pickling and unpickling fails to use the cache, even though the "factory data" are now available (this is not the case on Python 3 which *only* has new style classes):

```
sage: loads(dumps(d)) is d
True
sage: d._factory_data
(<__main__.MyFactory object at ...>,
 (...),
 (2,),
 {'impl': 'D'})
```

Only when we have a new style class that can be weak referenced and allows for attribute assignment, everything works:

```
sage: e = F(3)
sage: isinstance(e, E)
True
sage: loads(dumps(e)) is e
True
sage: e._factory_data
(<__main__.MyFactory object at ...>,
 (...),
 (3,),
 {'impl': None})
```

create_key(**args*, ***kwds*)

Given the parameters (arguments and keywords), create a key that uniquely determines this object.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_key(1, 2, key=5)
(1, 2)
```

create_key_and_extra_args(**args*, ***kwds*)

Return a tuple containing the key (uniquely defining data) and any extra arguments (empty by default).

Defaults to `create_key()`.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_key_and_extra_args(1, 2, key=5)
((1, 2), {})
sage: GF.create_key_and_extra_args(3)
((3, ('x',), None, 'modn', 3, 1, True, None, None, None, True, False), {})
```

create_object(*version*, *key*, ***extra_args*)

Create the object from the key and extra arguments. This is only called if the object was not found in the cache.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.create_object(0, (1,2,3))
Making object (1, 2, 3)
<sage.structure.test_factory.A object at ...>
sage: test_factory('a')
```

(continues on next page)

(continued from previous page)

```
Making object ('a',)
<sage.structure.test_factory.A object at ...>
sage: test_factory('a') # NOT called again
<sage.structure.test_factory.A object at ...>
```

get_object(*version*, *key*, *extra_args*)

Returns the object corresponding to `key`, creating it with `extra_args` if necessary (for example, it isn't in the cache or it is unpickling from an older version of Sage).

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: a = test_factory.get_object(3.0, 'a', {}); a
Making object a
<sage.structure.test_factory.A object at ...>
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.0, 'a',
↪ {})
True
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.1, 'a',
↪ {})
Making object a
False
sage: test_factory.get_object(3.0, 'a', {}) is test_factory.get_object(3.0, 'b',
↪ {})
Making object b
False
```

get_version(*sage_version*)

This is provided to allow more or less granular control over pickle versioning. Objects pickled in the same version of Sage will unpickle to the same rather than simply equal objects. This can provide significant gains as arithmetic must be performed on objects with identical parents. However, if there has been an incompatible change (e.g. in element representation) we want the version number to change so coercion is forced between the two parents.

Defaults to the Sage version that is passed in, but coarser granularity can be provided.

EXAMPLES:

```
sage: from sage.structure.test_factory import test_factory
sage: test_factory.get_version((3,1,0))
(3, 1, 0)
```

other_keys(*key*, *obj*)

Sometimes during object creation, certain defaults are chosen which may result in a new (more specific) key. This allows the more specific key to be regarded as equivalent to the original key returned by *`create_key()`* for the purpose of lookup in the cache, and is used for pickling.

EXAMPLES:

The `GF` factory used to have a custom *`other_keys()`* method, but this was removed in github issue #16934:

```
sage: # needs sage.libs.linbox sage.ring.finite_rings
sage: key, _ = GF.create_key_and_extra_args(27, 'k'); key
(27, ('k',), x^3 + 2*x + 1, 'givaro', 3, 3, True, None, 'poly', True, True,␣
```

(continues on next page)

(continued from previous page)

```
↪True)
sage: K = GF.create_object(0, key); K
Finite Field in k of size 3^3
sage: GF.other_keys(key, K)
[]

sage: K = GF(7^40, 'a')                                                    #␣
↪needs sage.rings.finite_rings
sage: loads(dumps(K)) is K                                                 #␣
↪needs sage.rings.finite_rings
True
```

reduce_data(*obj*)

The results of this function can be returned from `__reduce__()`. This is here so the factory internals can change without having to re-write `__reduce__()` methods that use it.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.modules.free_module import FreeModuleFactory_with_standard_
↪basis as F
sage: V = F(ZZ, 5)
sage: factory, data = F.reduce_data(V)
sage: factory(*data)
Ambient free module of rank 5 over the principal ideal domain Integer Ring
sage: factory(*data) is V
True

sage: from sage.structure.test_factory import test_factory
sage: a = test_factory(1, 2)
Making object (1, 2)
sage: test_factory.reduce_data(a)
(<built-in function generic_factory_unpickle>,
 (<sage.structure.test_factory.UniqueFactoryTester object at ...>,
  (...),
  (1, 2),
  {}))
```

Note that the ellipsis `(...)` here stands for the Sage version.

sage.structure.factory.**generic_factory_reduce**(*self*, *proto*)

Used to provide a `__reduce__` method if one does not already exist.

EXAMPLES:

```
sage: V = QQ^6                                                             #␣
↪needs sage.modules
sage: sage.structure.factory.generic_factory_reduce(V, 1) == V.__reduce_ex__(1)   ␣
↪         # needs sage.modules
True
```

sage.structure.factory.**generic_factory_unpickle**(*factory*, **args*)

Method used for unpickling the object.

The unpickling mechanism needs a plain Python function to call. It takes a factory as the first argument, passes the rest of the arguments onto the factory's *UniqueFactory.get_object()* method.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.modules.free_module import FreeModuleFactory_with_standard_basis as␣
↪F
sage: V = F(ZZ, 5)
sage: func, data = F.reduce_data(V)
sage: func is sage.structure.factory.generic_factory_unpickle
True
sage: sage.structure.factory.generic_factory_unpickle(*data) is V
True
```

sage.structure.factory.**lookup_global**(*name*)

Used in unpickling the factory itself.

EXAMPLES:

```
sage: from sage.structure.factory import lookup_global
sage: lookup_global('ZZ')
Integer Ring
sage: lookup_global('sage.rings.integer_ring.ZZ')
Integer Ring
```

sage.structure.factory.**register_factory_unpickle**(*name*, *callable*)

Register a callable to handle the unpickling from an old *UniqueFactory* object.

UniqueFactory pickles use a global name through *generic_factory_unpickle()*, so the usual `register_unpickle_override()` cannot be used here.

See also:

generic_factory_unpickle()

6.4 Dynamic classes

Why dynamic classes?

The short answer:

- Multiple inheritance is a powerful tool for constructing new classes by combining preexisting building blocks.
- There is a combinatorial explosion in the number of potentially useful classes that can be produced this way.
- The implementation of standard mathematical constructions calls for producing such combinations automatically.
- Dynamic classes, i.e. classes created on the fly by the Python interpreter, are a natural mean to achieve this.

The long answer:

Say we want to construct a new class `MyPermutation` for permutations in a given set S (in Sage, S will be modelled by a parent, but we won't discuss this point here). First, we have to choose a data structure for the permutations, typically among the following:

- Stored by cycle type
- Stored by code

- Stored in list notation - C arrays of short ints (for small permutations) - python lists of ints (for huge permutations) - …
- Stored by reduced word
- Stored as a function
- …

Luckily, the Sage library provides (or will provide) classes implementing each of those data structures. Those classes all share a common interface (or possibly a common abstract base class). So we can just derive our class from the chosen one:

```
class MyPermutation(PermutationCycleType):
    ...
```

Then we may want to further choose a specific memory behavior (unique representation, copy-on-write) which (hopefully) can again be achieved by inheritance:

```
class MyPermutation(UniqueRepresentation, PermutationCycleType):
     ...
```

Finally, we may want to endow the permutations in S with further operations coming from the (algebraic) structure of S:

- group operations
- or just monoid operations (for a subset of permutations not stable by inverse)
- poset operations (for left/right/Bruhat order)
- word operations (searching for substrings, patterns, …)

Or any combination thereof. Now, our class typically looks like:

```
class MyPermutation(UniqueRepresentation, PermutationCycleType, PosetElement,␣
↪GroupElement):
     ...
```

Note the combinatorial explosion in the potential number of classes which can be created this way.

In practice, such classes will be used in mathematical constructions like:

```
SymmetricGroup(5).subset(... TODO: find a good example in the context above ...)
```

In such a construction, the structure of the result, and therefore the operations on its elements can only be determined at execution time. Let us take another standard construction:

```
A = cartesian_product( B, C )
```

Depending on the structure of B and C, and possibly on further options passed down by the user, A may be:

- an enumerated set
- a group
- an algebra
- a poset
- …

Or any combination thereof.

Hardcoding classes for all potential combinations would be at best tedious. Furthermore, this would require a cumbersome mechanism to lookup the appropriate class depending on the desired combination.

Instead, one may use the ability of Python to create new classes dynamically:

```
type("class name", tuple of base classes, dictionary of methods)
```

This paradigm is powerful, but there are some technicalities to address. The purpose of this library is to standardize its use within Sage, and in particular to ensure that the constructed classes are reused whenever possible (unique representation), and can be pickled.

Combining dynamic classes and Cython classes

Cython classes cannot inherit from a dynamic class (there might be some partial support for this in the future). On the other hand, such an inheritance can be partially emulated using `__getattr__()`. See `sage.categories.examples.semigroups_cython` for an example.

class `sage.structure.dynamic_class.`**`DynamicClasscallMetaclass`**

Bases: *`DynamicMetaclass`*, `ClasscallMetaclass`

class `sage.structure.dynamic_class.`**`DynamicInheritComparisonClasscallMetaclass`**

Bases: *`DynamicMetaclass`*, `InheritComparisonClasscallMetaclass`

class `sage.structure.dynamic_class.`**`DynamicInheritComparisonMetaclass`**

Bases: *`DynamicMetaclass`*, `InheritComparisonMetaclass`

class `sage.structure.dynamic_class.`**`DynamicMetaclass`**

Bases: `type`

A metaclass implementing an appropriate reduce-by-construction method

`sage.structure.dynamic_class.`**`M`**

alias of *`DynamicInheritComparisonClasscallMetaclass`*

class `sage.structure.dynamic_class.`**`TestClass`**

Bases: `object`

A class used for checking that introspection works

`bla``()`

bla ...

`sage.structure.dynamic_class.`**`dynamic_class`**`(`*name*, *bases*, *cls=None*, *reduction=None*, *doccls=None*, *prepend_cls_bases=True*, *cache=True*`)`

INPUT:

- `name` – a string
- `bases` – a tuple of classes
- `cls` – a class or `None`
- `reduction` – a tuple or `None`
- `doccls` – a class or `None`
- `prepend_cls_bases` – a boolean (default: `True`)
- `cache` – a boolean or `"ignore_reduction"` (default: `True`)

Constructs dynamically a new class C with name name, and bases bases. If cls is provided, then its methods will be inserted into C, and its bases will be prepended to bases (unless prepend_cls_bases is False).

The module, documentation and source instrospection is taken from doccls, or cls if doccls is None, or bases[0] if both are None (therefore bases should be non empty if cls` is ``None).

The constructed class can safely be pickled (assuming the arguments themselves can).

Unless cache is False, the result is cached, ensuring unique representation of dynamic classes.

See *sage.structure.dynamic_class* for a discussion of the dynamic classes paradigm, and its relevance to Sage.

EXAMPLES:

To setup the stage, we create a class Foo with some methods, cached methods, and lazy attributes, and a class Bar:

```
sage: from sage.misc.lazy_attribute import lazy_attribute
sage: from sage.misc.cachefunc import cached_function
sage: from sage.structure.dynamic_class import dynamic_class
sage: class Foo():
....:     "The Foo class"
....:     def __init__(self, x):
....:         self._x = x
....:     @cached_method
....:     def f(self):
....:         return self._x^2
....:     def g(self):
....:         return self._x^2
....:     @lazy_attribute
....:     def x(self):
....:         return self._x
sage: class Bar:
....:     def bar(self):
....:         return self._x^2
```

We now create a class FooBar which is a copy of Foo, except that it also inherits from Bar:

```
sage: FooBar = dynamic_class("FooBar", (Bar,), Foo)
sage: x = FooBar(3)
sage: x.f()
9
sage: x.f() is x.f()
True
sage: x.x
3
sage: x.bar()
9
sage: FooBar.__name__
'FooBar'
sage: FooBar.__module__
'__main__'

sage: Foo.__bases__
(<class 'object'>,)
sage: FooBar.__bases__
```

(continues on next page)

(continued from previous page)

```
(<class '__main__.Bar'>,)
sage: Foo.mro()
[<class '__main__.Foo'>, <class 'object'>]
sage: FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Bar'>, <class 'object'>]
```

If all the base classes have a zero `__dictoffset__`, the dynamic class also has a zero `__dictoffset__`. This means that the instances of the class don't have a `__dict__` (see github issue #23435):

```
sage: dyn = dynamic_class("dyn", (Integer,))
sage: dyn.__dictoffset__
0
```

Pickling

Dynamic classes are pickled by construction. Namely, upon unpickling, the class will be reconstructed by recalling dynamic_class with the same arguments:

```
sage: type(FooBar).__reduce__(FooBar)
(<function dynamic_class at ...>, ('FooBar', (<class '__main__.Bar'>,), <class '__
↪main__.Foo'>, None, None))
```

Technically, this is achieved by using a metaclass, since the Python pickling protocol for classes is to pickle by name:

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

The following (meaningless) example illustrates how to customize the result of the reduction:

```
sage: BarFoo = dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,)))
sage: type(BarFoo).__reduce__(BarFoo)
(<class 'str'>, (3,))
sage: loads(dumps(BarFoo))
'3'
```

Caching

By default, the built class is cached:

```
sage: dynamic_class("FooBar", (Bar,), Foo) is FooBar
True
sage: dynamic_class("FooBar", (Bar,), Foo, cache=True) is FooBar
True
```

and the result depends on the reduction:

```
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,))) is BarFoo
True
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,))) is BarFoo
False
```

With cache=False, a new class is created each time:

```
sage: FooBar1 = dynamic_class("FooBar", (Bar,), Foo, cache=False); FooBar1
<class '__main__.FooBar'>
sage: FooBar2 = dynamic_class("FooBar", (Bar,), Foo, cache=False); FooBar2
<class '__main__.FooBar'>
sage: FooBar1 is FooBar
False
sage: FooBar2 is FooBar1
False
```

With cache="ignore_reduction", the class does not depend on the reduction:

```
sage: BarFoo = dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (3,)), cache=
↪"ignore_reduction")
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,)), cache="ignore_
↪reduction") is BarFoo
True
```

In particular, the reduction used is that provided upon creating the first class:

```
sage: dynamic_class("BarFoo", (Foo,), Bar, reduction = (str, (2,)), cache="ignore_
↪reduction")._reduction
(<class 'str'>, (3,))
```

Warning: The behaviour upon creating several dynamic classes from the same data but with different values for cache option is currently left unspecified. In other words, for a given application, it is recommended to consistently use the same value for that option.

sage.structure.dynamic_class.**dynamic_class_internal**(*bases*, *cls=None*, *reduction=None*, *doccls=None*, *prepend_cls_bases=True*)

See sage.structure.dynamic_class.dynamic_class? for indirect doctests.

6.5 Mutability Cython Implementation

class sage.structure.mutability.**Mutability**

Bases: object

Class to mix in mutability feature.

EXAMPLES:

```
sage: class A(SageObject, Mutability):
....:     def __init__(self, val):
....:         self._val = val
....:     def change(self, val):
....:         self._require_mutable()
....:         self._val = val
....:     def __hash__(self):
....:         self._require_immutable()
....:         return hash(self._val)
```

(continues on next page)

(continued from previous page)

```
sage: a = A(4)
sage: a._val
4
sage: a.change(6); a._val
6
sage: hash(a)
Traceback (most recent call last):
...
ValueError: object is mutable; please make it immutable first
sage: a.set_immutable()
sage: a.change(4)
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead
sage: hash(a)
6
```

is_immutable()

Return `True` if this object is immutable (cannot be changed) and `False` if it is not.

To make this object immutable use self.set_immutable().

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

Return `True` if this object is mutable (can be changed) and `False` if it is not.

To make this object immutable use `self.set_immutable()`.

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_mutable()
True
sage: v.set_immutable()
sage: v.is_mutable()
False
```

set_immutable()

Make this object immutable, so it can never again be changed.

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

sage.structure.mutability.**require_immutable**(*f*)

A decorator that requires immutability for a method to be called.

Note: Objects whose methods use this decorator should have an attribute `_is_immutable`. Otherwise, the object is assumed to be mutable.

EXAMPLES:

```
sage: from sage.structure.mutability import require_mutable, require_immutable
sage: class A():
....:  def __init__(self, val):
....:      self._m = val
....:  @require_mutable
....:  def change(self, new_val):
....:      'change self'
....:      self._m = new_val
....:  @require_immutable
....:  def __hash__(self):
....:      'implement hash'
....:      return hash(self._m)
sage: a = A(5)
sage: a.change(6)
sage: hash(a)   # indirect doctest
Traceback (most recent call last):
...
ValueError: <class '__main__.A'> instance is mutable, <function ...__hash__ at ...>␣
↪must not be called
sage: a._is_immutable = True
sage: hash(a)
6
sage: a.change(7)
Traceback (most recent call last):
...
ValueError: <class '__main__.A'> instance is immutable, <function ...change at ...>␣
↪must not be called
sage: from sage.misc.sageinspect import sage_getdoc
sage: print(sage_getdoc(a.__hash__))
implement hash
```

AUTHORS:

- Simon King <simon.king@uni-jena.de>

sage.structure.mutability.**require_mutable**(*f*)

A decorator that requires mutability for a method to be called.

Note: Objects whose methods use this decorator should have an attribute `_is_immutable`. Otherwise, the object is assumed to be mutable.

EXAMPLES:

```
sage: from sage.structure.mutability import require_mutable, require_immutable
sage: class A():
....:     def __init__(self, val):
....:         self._m = val
....:     @require_mutable
....:     def change(self, new_val):
....:         'change self'
....:         self._m = new_val
....:     @require_immutable
....:     def __hash__(self):
....:         'implement hash'
....:         return hash(self._m)
sage: a = A(5)
sage: a.change(6)
sage: hash(a)
Traceback (most recent call last):
...
ValueError: <class '__main__.A'> instance is mutable, <function ...__hash__ at ...>
↪must not be called
sage: a._is_immutable = True
sage: hash(a)
6
sage: a.change(7)   # indirect doctest
Traceback (most recent call last):
...
ValueError: <class '__main__.A'> instance is immutable, <function ...change at ...>
↪must not be called
sage: from sage.misc.sageinspect import sage_getdoc
sage: print(sage_getdoc(a.change))
change self
```

AUTHORS:

- Simon King <simon.king@uni-jena.de>

CHAPTER

SEVEN

INTERNALS

7.1 Debug options for the sage.structure modules

EXAMPLES:

```
sage: from sage.structure.debug_options import debug
sage: debug.unique_parent_warnings
False
sage: debug.refine_category_hash_check
True
```

class sage.structure.debug_options.**DebugOptions_class**

Bases: object

refine_category_hash_check

unique_parent_warnings

7.2 Performance Test for Clone Protocol

see *sage.structure.list_clone.ClonableArray*

EXAMPLES:

```
sage: from sage.structure.list_clone_timings import *
sage: cmd =["",
....:     "e.__copy__()",
....:     "copy(e)",
....:     "e.clone()",
....:     "e.__class__(e.parent(), e._get_list())",
....:     "e.__class__(e.parent(), e[:])",
....:     "e.check()",
....:     "",
....:     "add1_internal(e)",
....:     "add1_immutable(e)",
....:     "add1_mutable(e)",
....:     "add1_with(e)",
....:     "",
....:     "cy_add1_internal(e)",
....:     "cy_add1_immutable(e)",
```

(continues on next page)

(continued from previous page)

```
....:     "cy_add1_mutable(e)",
....:     "cy_add1_with(e)"]
```

Various timings using a Cython class:

```
sage: size = 5
sage: e = IncreasingArrays()(range(size))
sage: # random
....: for p in cmd:
....:     print("{0:36} : ".format(p), end=""); timeit(p)
                                     :
e.__copy__()                         :  625 loops, best of 3: 446 ns per loop
copy(e)                              :  625 loops, best of 3: 1.94 µs per loop
e.clone()                            :  625 loops, best of 3: 736 ns per loop
e.__class__(e.parent(), e._get_list()) :  625 loops, best of 3: 1.34 µs per loop
e.__class__(e.parent(), e[:])        :  625 loops, best of 3: 1.35 µs per loop
e.check()                            :  625 loops, best of 3: 342 ns per loop
                                     :
add1_internal(e)                     :  625 loops, best of 3: 3.53 µs per loop
add1_immutable(e)                    :  625 loops, best of 3: 3.72 µs per loop
add1_mutable(e)                      :  625 loops, best of 3: 3.42 µs per loop
add1_with(e)                         :  625 loops, best of 3: 4.05 µs per loop
                                     :
cy_add1_internal(e)                  :  625 loops, best of 3: 752 ns per loop
cy_add1_immutable(e)                 :  625 loops, best of 3: 1.28 µs per loop
cy_add1_mutable(e)                   :  625 loops, best of 3: 861 ns per loop
cy_add1_with(e)                      :  625 loops, best of 3: 1.51 µs per loop
```

Various timings using a Python class:

```
sage: e = IncreasingArraysPy()(range(size))
sage: # random
....: for p in cmd: print("{0:36} : ".format(p), end=""); timeit(p)
                                     :
e.__copy__()                         :  625 loops, best of 3: 869 ns per loop
copy(e)                              :  625 loops, best of 3: 2.13 µs per loop
e.clone()                            :  625 loops, best of 3: 1.86 µs per loop
e.__class__(e.parent(), e._get_list()) :  625 loops, best of 3: 7.52 µs per loop
e.__class__(e.parent(), e[:])        :  625 loops, best of 3: 7.27 µs per loop
e.check()                            :  625 loops, best of 3: 4.02 µs per loop
                                     :
add1_internal(e)                     :  625 loops, best of 3: 9.34 µs per loop
add1_immutable(e)                    :  625 loops, best of 3: 9.91 µs per loop
add1_mutable(e)                      :  625 loops, best of 3: 12.6 µs per loop
add1_with(e)                         :  625 loops, best of 3: 15.9 µs per loop
                                     :
cy_add1_internal(e)                  :  625 loops, best of 3: 7.13 µs per loop
cy_add1_immutable(e)                 :  625 loops, best of 3: 6.95 µs per loop
cy_add1_mutable(e)                   :  625 loops, best of 3: 14.1 µs per loop
cy_add1_with(e)                      :  625 loops, best of 3: 17.5 µs per loop
```

class sage.structure.list_clone_timings.**IncreasingArraysPy**

Bases: *IncreasingArrays*

class Element

Bases: *ClonableArray*

A small class for testing *ClonableArray*: Increasing Lists

check()

Check that `self` is increasing.

EXAMPLES:

```
sage: from sage.structure.list_clone_timings import IncreasingArraysPy
sage: IncreasingArraysPy()([1,2,3]) # indirect doctest
[1, 2, 3]
sage: IncreasingArraysPy()([3,2,1]) # indirect doctest
Traceback (most recent call last):
...
ValueError: Lists is not increasing
```

sage.structure.list_clone_timings.**add1_immutable**(*bla*)

sage.structure.list_clone_timings.**add1_internal**(*bla*)

sage.structure.list_clone_timings.**add1_mutable**(*bla*)

sage.structure.list_clone_timings.**add1_with**(*bla*)

7.3 Cython Functions for Timing Clone Protocol

sage.structure.list_clone_timings_cy.**cy_add1_immutable**(*bla*)

sage.structure.list_clone_timings_cy.**cy_add1_internal**(*bla*)

sage.structure.list_clone_timings_cy.**cy_add1_mutable**(*bla*)

sage.structure.list_clone_timings_cy.**cy_add1_with**(*bla*)

7.4 Test of the `factory` module

class sage.structure.test_factory.**A**

Bases: `object`

class sage.structure.test_factory.**UniqueFactoryTester**

Bases: *UniqueFactory*

create_key(**args*, ***kwds*)

EXAMPLES:

```
sage: from sage.structure.test_factory import UniqueFactoryTester
sage: test_factory = UniqueFactoryTester('foo')
sage: test_factory.create_key(1, 2, 3)
(1, 2, 3)
```

create_object(*version*, *key*, ***extra_args*)

EXAMPLES:

```
sage: from sage.structure.test_factory import UniqueFactoryTester
sage: test_factory = UniqueFactoryTester('foo')
sage: test_factory.create_object('version', key=(1, 2, 4))
Making object (1, 2, 4)
<sage.structure.test_factory.A object at ...>
```

CHAPTER

EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

s

INDEX

Symbols

A

B

E

F

G

H

I

L

M

N

O

P

Q

R

S