Sage Reference Manual: Coercion

Release 7.3

The Sage Development Team

Aug 05, 2016

CHAPTER

ONE

PRELIMINARIES

1.1 What is coercion all about?

The primary goal of coercion is to be able to transparently do arithmetic, comparisons, etc. between elements of distinct sets.

As a concrete example, when one writes $1 + 1/2$ one wants to perform arithmetic on the operands as rational numbers, despite the left being an integer. This makes sense given the obvious and natural inclusion of the integers into the rational numbers. The goal of the coercion system is to facilitate this (and more complicated arithmetic) without having to explicitly map everything over into the same domain, and at the same time being strict enough to not resolve ambiguity or accept nonsense. Here are some examples:

```
sage: 1 + 1/2
3/2
sage: R.<x,y> = ZZ[]
sage: R
Multivariate Polynomial Ring in x, y over Integer Ring
sage: parent(x)
Multivariate Polynomial Ring in x, y over Integer Ring
sage: parent(1/3)
Rational Field
sage: x+1/3
x + 1/3
sage: parent(x+1/3)
Multivariate Polynomial Ring in x, y over Rational Field

sage: GF(5)(1) + CC(I)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Finite Field of size 5' and
↪'Complex Field with 53 bits of precision'
```

1.2 Parents and Elements

Parents are objects in concrete categories, and Elements are their members. Parents are first-class objects. Most things in Sage are either parents or have a parent. Typically whenever one sees the word *Parent* one can think *Set*. Here are some examples:

```
sage: parent(1)
Integer Ring
sage: parent(1) is ZZ
```

```
True
sage: ZZ
Integer Ring
sage: parent(1.50000000000000000000000000000000000000)
Real Field with 120 bits of precision
sage: parent(x)
Symbolic Ring
sage: x^sin(x)
x^sin(x)
sage: R.<t> = Qp(5)[]
sage: f = t^3-5; f
(1 + O(5^20))*t^3 + (4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + 4*5^6 + 4*5^7 + 4*5^8 +␣
↪4*5^9 + 4*5^10 + 4*5^11 + 4*5^12 + 4*5^13 + 4*5^14 + 4*5^15 + 4*5^16 + 4*5^17 + 4*5^
↪18 + 4*5^19 + 4*5^20 + O(5^21))
sage: parent(f)
Univariate Polynomial Ring in t over 5-adic Field with capped relative precision 20
sage: f = EllipticCurve('37a').lseries().taylor_series(10); f
0.990010459847588 + 0.0191338632530789*z - 0.0197489006172923*z^2 + 0.
↪0137240085327618*z^3 - 0.00703880791607153*z^4 + 0.00280906165766519*z^5 + O(z^6)  ␣
↪        # 32-bit
0.997997869801216 + 0.00140712894524925*z - 0.00049812761096009 7*z^2 + 0.
↪00011883559666595 6*z^3 - 0.00002159065224427 07*z^4 + (3.20363155418419e-6)*z^5 +␣
↪O(z^6)  # 64-bit
sage: parent(f)
Power Series Ring in z over Complex Field with 53 bits of precision
```

There is an important distinction between Parents and types:

```
sage: a = GF(5).random_element()
sage: b = GF(7).random_element()
sage: type(a)
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: type(b)
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: type(a) == type(b)
True
sage: parent(a)
Finite Field of size 5
sage: parent(a) == parent(b)
False
```

However, non-Sage objects don't really have parents, but we still want to be able to reason with them, so their type is used instead:

```
sage: a = int(10)
sage: parent(a)
<type 'int'>
```

In fact, under the hood, a special kind of parent "The set of all Python objects of type T" is used in these cases.

Note that parents are **not** always as tight as possible.

```
sage: parent(1/2)
Rational Field
sage: parent(2/1)
Rational Field
```

1.3 Maps between Parents

Many parents come with maps to and from other parents.

Sage makes a distinction between being able to **convert** between various parents, and **coerce** between them. Conversion is explicit and tries to make sense of an object in the target domain if at all possible. It is invoked by calling:

```
sage: ZZ(5)
5
sage: ZZ(10/5)
2
sage: QQ(10)
10
sage: parent(QQ(10))
Rational Field
sage: a = GF(5)(2); a
2
sage: parent(a)
Finite Field of size 5
sage: parent(ZZ(a))
Integer Ring
sage: GF(71)(1/5)
57
sage: ZZ(1/2)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

Conversions need not be canonical (they may for example involve a choice of lift) or even make sense mathematically (e.g. constructions of some kind).

```
sage: ZZ("123")
123
sage: ZZ(GF(5)(14))
4
sage: ZZ['x']([4,3,2,1])
x^3 + 2*x^2 + 3*x + 4
sage: a = Qp(5, 10)(1/3); a
2 + 3*5 + 5^2 + 3*5^3 + 5^4 + 3*5^5 + 5^6 + 3*5^7 + 5^8 + 3*5^9 + O(5^10)
sage: ZZ(a)
6510417
```

On the other hand, Sage has the notion of a **coercion**, which is a canonical morphism (occasionally up to a conventional choice made by developers) between parents. A coercion from one parent to another **must** be defined on the whole domain, and always succeeds. As it may be invoked implicitly, it should be obvious and natural (in both the mathematically rigorous and colloquial sense of the word). Up to inescapable rounding issues that arise with inexact representations, these coercion morphisms should all commute. In particular, if there are coercion maps $A \to B$ and $B \to A$, then their composites must be the identity maps.

Coercions can be discovered via the `Parent.has_coerce_map_from()` method, and if needed explicitly invoked with the `Parent.coerce()` method:

```
sage: QQ.has_coerce_map_from(ZZ)
True
sage: QQ.has_coerce_map_from(RR)
False
sage: ZZ['x'].has_coerce_map_from(QQ)
False
```

```
sage: ZZ['x'].has_coerce_map_from(ZZ)
True
sage: ZZ['x'].coerce(5)
5
sage: ZZ['x'].coerce(5).parent()
Univariate Polynomial Ring in x over Integer Ring
sage: ZZ['x'].coerce(5/1)
Traceback (most recent call last):
...
TypeError: no canonical coercion from Rational Field to Univariate Polynomial Ring in␣
↪x over Integer Ring
```

CHAPTER

TWO

BASIC ARITHMETIC RULES

Suppose we want to add two element, a and b, whose parents are A and B respectively. When we type `a+b` then

1. If A `is` B, call a._add_(b)
2. If there is a coercion $\phi : B \to A$, call a._add_(ϕ (b))
3. If there is a coercion $\phi : A \to B$, call ϕ (a)._add_(b)
4. Look for Z such that there is a coercion $\phi_A : A \to Z$ and $\phi_B : B \to Z$, call ϕ_A (a)._add_(ϕ_B (b))

These rules are evaluated in order; therefore if there are coercions in both directions, then the parent of a._add_b is A – the parent of the left-hand operand is used in such cases.

The same rules are used for subtraction, multiplication, and division. This logic is embedded in a coercion model object, which can be obtained and queried.

```
sage: parent(1 + 1/2)
Rational Field
sage: cm = sage.structure.element.get_coercion_model(); cm
<sage.structure.coerce.CoercionModel_cache_maps object at ...>
sage: cm.explain(ZZ, QQ)
Coercion on left operand via
   Natural morphism:
     From: Integer Ring
     To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZ['x','y'], QQ['x'])
Coercion on left operand via
   Conversion map:
     From: Multivariate Polynomial Ring in x, y over Integer Ring
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Coercion on right operand via
   Conversion map:
     From: Univariate Polynomial Ring in x over Rational Field
     To:   Multivariate Polynomial Ring in x, y over Rational Field
Arithmetic performed after coercions.
Result lives in Multivariate Polynomial Ring in x, y over Rational Field
Multivariate Polynomial Ring in x, y over Rational Field
```

The coercion model can be used directly for any binary operation (callable taking two arguments).

```
sage: cm.bin_op(77, 9, gcd)
1
```

There are also **actions** in the sense that a field K acts on a module over K, or a permutation group acts on a set. These are discovered between steps 1 and 2 above.

```
sage: cm.explain(ZZ['x'], ZZ, operator.mul)
Action discovered.
   Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x␣
↪over Integer Ring
Result lives in Univariate Polynomial Ring in x over Integer Ring
Univariate Polynomial Ring in x over Integer Ring

sage: cm.explain(ZZ['x'], ZZ, operator.div)
Action discovered.
   Right inverse action by Rational Field on Univariate Polynomial Ring in x over␣
↪Integer Ring
   with precomposition on right by Natural morphism:
     From: Integer Ring
     To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

sage: f = QQ.coerce_map_from(ZZ)
sage: f(3).parent()
Rational Field
```

Note that by trac ticket #14711 Sage's coercion system uses maps with weak references to the domain. Such maps should only be used internally, and so a copy should be used instead (unless one knows what one is doing):

```
sage: QQ._internal_coerce_map_from(int)
(map internal to coercion system -- copy before use)
Native morphism:
  From: Set of Python objects of type 'int'
  To:   Rational Field
sage: copy(QQ._internal_coerce_map_from(int))
Native morphism:
 From: Set of Python objects of type 'int'
 To:   Rational Field
```

Note that the user-visible method (without underscore) automates this copy:

```
sage: copy(QQ.coerce_map_from(int))
Native morphism:
 From: Set of Python objects of type 'int'
 To:   Rational Field
```

```
sage: QQ.has_coerce_map_from(RR)
False
sage: QQ['x'].get_action(QQ)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over␣
↪Rational Field
sage: QQ2 = QQ^2
sage: (QQ2).get_action(QQ)
Right scalar multiplication by Rational Field on Vector space of dimension 2 over␣
↪Rational Field
sage: QQ['x'].get_action(RR)
Right scalar multiplication by Real Field with 53 bits of precision on Univariate␣
↪Polynomial Ring in x over Rational Field
```

CHAPTER

THREE

HOW TO IMPLEMENT

3.1 Methods to implement

- Arithmetic on Elements: `_add_` , `_sub_` , `_mul_` , `_div_`

  This is where the binary arithmetic operators should be implemented. Unlike Python's `__add__` , both operands are *guaranteed* to have the same Parent at this point.

- Coercion for Parents: `_coerce_map_from_`

  Given two parents R and S, `R._coerce_map_from_(S)` is called to determine if there is a coercion $\phi : S \to R$. Note that the function is called on the potential codomain. To indicate that there is no coercion from S to R (self), return `False` or `None` . This is the default behavior. If there is a coercion, return `True` (in which case an morphism using `R._element_constructor_` will be created) or an actual `Morphism` object with S as the domain and R as the codomain.

- Actions for Parents: `_get_action_` or `_rmul_` , `_lmul_` , `_r_action_` , `_l_action_`

  Suppose one wants R to act on S. Some examples of this could be $R = \mathbf{Q}$, $S = \mathbf{Q}[x]$ or $R = \mathrm{Gal}(S/\mathbf{Q})$ where S is a number field. There are several ways to implement this:

  - If R is the base of S (as in the first example), simply implement `_rmul_` and/or `_lmul_` on the Elements of S. In this case `r * s` gets handled as `s._rmul_(r)` and `s * r` as `s._lmul_(r)` . The argument to `_rmul_` and `_lmul_` are *guaranteed* to be Elements of the base of S (with coercion happening beforehand if necessary).
  - If R acts on S, one can alternatively define the methods `_r_action_` and/or `_l_action_` on the Elements of R. There is no constraint on the type or parents of objects passed to these methods; raise a `TypeError` or `ValueError` if the wrong kind of object is passed in to indicate the action is not appropriate here.
  - If either R acts on S *or* S acts on R, one may implement `R._get_action_` to return an actual *Action* object to be used. This is how non-multiplicative actions must be implemented, and is the most powerful (and completed) way to do things.

- Element conversion/construction for Parents: use `_element_constructor_` **not** `__call__`

  The `Parent.__call__()` method dispatches to `_element_constructor_` . When someone writes `R(x,...)` , this is the method that eventually gets called in most cases. See the documentation on the `__call__` method below.

Parents may also call the `self._populate_coercion_lists_` method in their `__init__` functions to pass any callable for use instead of `_element_constructor_` , provide a list of Parents with coercions to self (as an alternative to implementing `_coerce_map_from_`), provide special construction methods (like `_integer_` for ZZ), etc. This also allows one to specify a single coercion embedding *out* of self (whereas the rest of the coercion functions all specify maps *into* self). There is extensive documentation in the docstring of the `_populate_coercion_lists_` method.

3.2 Example

Sometimes a simple example is worth a thousand words. Here is a minimal example of setting up a simple Ring that handles coercion. (It is easy to imagine much more sophisticated and powerful localizations, but that would obscure the main points being made here.)

```
class Localization(Ring):
    def __init__(self, primes):
        """
        Localization of `\ZZ` away from primes.
        """
        Ring.__init__(self, base=ZZ)
        self._primes = primes
        self._populate_coercion_lists_()

    def _repr_(self):
        """
        How to print self.
        """
        return "%s localized at %s" % (self.base(), self._primes)

    def _element_constructor_(self, x):
        """
        Make sure x is a valid member of self, and return the constructed element.
        """
        if isinstance(x, LocalizationElement):
            x = x._value
        else:
            x = QQ(x)
        for p, e in x.denominator().factor():
            if p not in self._primes:
                raise ValueError("Not integral at %s" % p)
        return LocalizationElement(self, x)

    def _coerce_map_from_(self, S):
        """
        The only things that coerce into this ring are:

        - the integer ring

        - other localizations away from fewer primes
        """
        if S is ZZ:
            return True
        elif isinstance(S, Localization):
            return all(p in self._primes for p in S._primes)


class LocalizationElement(RingElement):

    def __init__(self, parent, x):
        RingElement.__init__(self, parent)
        self._value = x


    # We're just printing out this way to make it easy to see what's going on in the␣
↪examples.
```

```
    def _repr_(self):
        return "LocalElt(%s)" % self._value

    # Now define addition, subtraction, and multiplication of elements.
    # Note that left and right always have the same parent.

    def _add_(left, right):
        return LocalizationElement(left.parent(), left._value + right._value)

    def _sub_(left, right):
        return LocalizationElement(left.parent(), left._value - right._value)

    def _mul_(left, right):
        return LocalizationElement(left.parent(), left._value * right._value)

    # The basering was set to ZZ, so c is guaranteed to be in ZZ

    def _rmul_(self, c):
        return LocalizationElement(self.parent(), c * self._value)

    def _lmul_(self, c):
        return LocalizationElement(self.parent(), self._value * c)
```

That's all there is to it. Now we can test it out:

```
sage: R = Localization([2]); R
Integer Ring localized at [2]
sage: R(1)
LocalElt(1)
sage: R(1/2)
LocalElt(1/2)
sage: R(1/3)
Traceback (most recent call last):
...
ValueError: Not integral at 3

sage: R.coerce(1)
LocalElt(1)
sage: R.coerce(1/4)
Traceback (click to the left for traceback)
...
TypeError: no cannonical coercion from Rational Field to Integer Ring localized at [2]

sage: R(1/2) + R(3/4)
LocalElt(5/4)
sage: R(1/2) + 5
LocalElt(11/2)
sage: 5 + R(1/2)
LocalElt(11/2)
sage: R(1/2) + 1/7
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Integer Ring localized at [2]' and
 ↪'Rational Field'
sage: R(3/4) * 7
LocalElt(21/4)

sage: R.get_action(ZZ)
```

```
Right scalar multiplication by Integer Ring on Integer Ring localized at [2]
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.explain(R, ZZ, operator.add)
Coercion on right operand via
   Conversion map:
     From: Integer Ring
     To:   Integer Ring localized at [2]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

sage: cm.explain(R, ZZ, operator.mul)
Action discovered.
   Right scalar multiplication by Integer Ring on Integer Ring localized at [2]
Result lives in Integer Ring localized at [2]
Integer Ring localized at [2]

sage: R6 = Localization([2,3]); R6
Integer Ring localized at [2, 3]
sage: R6(1/3) - R(1/2)
LocalElt(-1/6)
sage: parent(R6(1/3) - R(1/2))
Integer Ring localized at [2, 3]

sage: R.has_coerce_map_from(ZZ)
True
sage: R.coerce_map_from(ZZ)
Conversion map:
 From: Integer Ring
 To:   Integer Ring localized at [2]

sage: R6.coerce_map_from(R)
Conversion map:
 From: Integer Ring localized at [2]
 To:   Integer Ring localized at [2, 3]

sage: R6.coerce(R(1/2))
LocalElt(1/2)

sage: cm.explain(R, R6, operator.mul)
Coercion on left operand via
   Conversion map:
     From: Integer Ring localized at [2]
     To:   Integer Ring localized at [2, 3]
Arithmetic performed after coercions.
Result lives in Integer Ring localized at [2, 3]
Integer Ring localized at [2, 3]
```

3.3 Provided Methods

- `__call__`

  This provides a consistent interface for element construction. In particular, it makes sure that conversion always gives the same result as coercion, if a coercion exists. (This used to be violated for some Rings in Sage as the code for conversion and coercion got edited separately.) Let R be a Parent and assume the user types R(x), where x has parent X. Roughly speaking, the following occurs:

1. If X `is` R, return x (*)
2. If there is a coercion $f : X \to R$, return $f(x)$
3. If there is a coercion $f : R \to X$, try to return $f^{-1}(x)$
4. Return `R._element_constructor_(x)` (**)

Keywords and extra arguments are passed on. The result of all this logic is cached.

(*) Unless there is a "copy" keyword like R(x, copy=False)

(**) Technically, a generic morphism is created from X to R, which may use magic methods like `_integer_` or other data provided by `_populate_coercion_lists_` .

- `coerce`

  Coerces elements into self, raising a type error if there is no coercion map.

- `coerce_map_from`, `convert_map_from`

  Returns an actual `Morphism` object to coerce/convert from another Parent to self. Barring direct construction of elements of R, `R.convert_map_from(S)` will provide a callable Python object which is the fastest way to convert elements of S to elements of R. From Cython, it can be invoked via the cdef `_call_` method.

- `has_coerce_map_from`

  Returns `True` or `False` depending on whether or not there is a coercion. `R.has_coerce_map_from(S)` is shorthand for `R.coerce_map_from(S) is not None`

- `get_action`

  This will unwind all the `_rmul_`, `_lmul_`, `_r_action_`, `_l_action_`, `...` methods to provide an actual `Action` object, if one exists.

CHAPTER

FOUR

DISCOVERING NEW PARENTS

New parents are discovered using an algorithm in sage/category/pushout.py. The fundamental idea is that most Parents in Sage are constructed from simpler objects via various functors. These are accessed via the `construction()` method, which returns a (simpler) Parent along with a functor with which one can create self.

```
sage: CC.construction()
(AlgebraicClosureFunctor, Real Field with 53 bits of precision)
sage: RR.construction()
(Completion[+Infinity], Rational Field)
sage: QQ.construction()
(FractionField, Integer Ring)
sage: ZZ.construction()  # None

sage: Qp(5).construction()
(Completion[5], Rational Field)
sage: QQ.completion(5, 100, {})
5-adic Field with capped relative precision 100
sage: c, R = RR.construction()
sage: a = CC.construction()[0]
sage: a.commutes(c)
False
sage: RR == c(QQ)
True

sage: sage.categories.pushout.construction_tower(Frac(CDF['x']))
[(None,
 Fraction Field of Univariate Polynomial Ring in x over Complex Double Field),
(FractionField, Univariate Polynomial Ring in x over Complex Double Field),
(Poly[x], Complex Double Field),
(AlgebraicClosureFunctor, Real Double Field),
(Completion[+Infinity], Rational Field),
(FractionField, Integer Ring)]
```

Given Parents R and S, such that there is no coercion either from R to S or from S to R, one can find a common Z with coercions $R \to Z$ and $S \to Z$ by considering the sequence of construction functors to get from a common ancestor to both R and S. We then use a *heuristic* algorithm to interleave these constructors in an attempt to arrive at a suitable Z (if one exists). For example:

```
sage: ZZ['x'].construction()
(Poly[x], Integer Ring)
sage: QQ.construction()
(FractionField, Integer Ring)
sage: sage.categories.pushout.pushout(ZZ['x'], QQ)
Univariate Polynomial Ring in x over Rational Field
```

```
sage: sage.categories.pushout.pushout(ZZ['x'], QQ).construction()
(Poly[x], Rational Field)
```

The common ancestor is Z and our options for Z are $\mathrm{Frac}(\mathbf{Z}[x])$ or $\mathrm{Frac}(\mathbf{Z})[x]$. In Sage we choose the later, treating the fraction field functor as binding "more tightly" than the polynomial functor, as most people agree that $\mathbf{Q}[x]$ is the more natural choice. The same procedure is applied to more complicated Parents, returning a new Parent if one can be unambiguously determined.

```
sage: sage.categories.pushout.pushout(Frac(ZZ['x,y,z']), QQ['z, t'])
Univariate Polynomial Ring in t over Fraction Field of Multivariate Polynomial Ring␣
↪in x, y, z over Rational Field
```

CHAPTER

FIVE

MODULES

5.1 The Coercion Model

The coercion model manages how elements of one parent get related to elements of another. For example, the integer 2 can canonically be viewed as an element of the rational numbers. (The Parent of a non-element is its Python type.)

```
sage: ZZ(2).parent()
Integer Ring
sage: QQ(2).parent()
Rational Field
```

The most prominent role of the coercion model is to make sense of binary operations between elements that have distinct parents. It does this by finding a parent where both elements make sense, and doing the operation there. For example:

```
sage: a = 1/2; a.parent()
Rational Field
sage: b = ZZ['x'].gen(); b.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: a+b
x + 1/2
sage: (a+b).parent()
Univariate Polynomial Ring in x over Rational Field
```

If there is a coercion (see below) from one of the parents to the other, the operation is always performed in the codomain of that coercion. Otherwise a reasonable attempt to create a new parent with coercion maps from both original parents is made. The results of these discoveries are cached. On failure, a TypeError is always raised.

Some arithmetic operations (such as multiplication) can indicate an action rather than arithmetic in a common parent. For example:

```
sage: E = EllipticCurve('37a')
sage: P = E(0,0)
sage: 5*P
(1/4 : -5/8 : 1)
```

where there is action of $\mathbf{Z}$ on the points of E given by the additive group law. Parents can specify how they act on or are acted upon by other parents.

There are two kinds of ways to get from one parent to another, coercions and conversions.

Coercions are canonical (possibly modulo a finite number of deterministic choices) morphisms, and the set of all coercions between all parents forms a commuting diagram (modulo possibly rounding issues). $\mathbf{Z} \to \mathbf{Q}$ is an example of a coercion. These are invoked implicitly by the coercion model.

Conversions try to construct an element out of their input if at all possible. Examples include sections of coercions, creating an element from a string or list, etc. and may fail on some inputs of a given type while succeeding on others (i.e. they may not be defined on the whole domain). Conversions are always explicitly invoked, and never used by the coercion model to resolve binary operations.

For more information on how to specify coercions, conversions, and actions, see the documentation for Parent.

class sage.structure.coerce. **CoercionModel_cache_maps**

Bases: sage.structure.element.CoercionModel

See also sage.categories.pushout

EXAMPLES:

```
sage: f = ZZ['t','x'].0 + QQ['x'].0 + CyclotomicField(13).gen(); f
t + x + (zeta13)
sage: f.parent()
Multivariate Polynomial Ring in t, x over Cyclotomic Field of order 13 and degree␣
↪12
sage: ZZ['x','y'].0 + ~Frac(QQ['y']).0
(x*y + 1)/y
sage: MatrixSpace(ZZ['x'], 2, 2)(2) + ~Frac(QQ['x']).0
[(2*x + 1)/x           0]
[          0 (2*x + 1)/x]
sage: f = ZZ['x,y,z'].0 + QQ['w,x,z,a'].0; f
w + x
sage: f.parent()
Multivariate Polynomial Ring in w, x, y, z, a over Rational Field
sage: ZZ['x,y,z'].0 + ZZ['w,x,z,a'].1
2*x
```

TESTS:

Check that trac ticket #8426 is fixed (see also trac ticket #18076):

```
sage: import numpy
sage: x = polygen(RR)
sage: numpy.float32('1.5') * x
1.50000000000000*x
sage: x * numpy.float32('1.5')
1.50000000000000*x
sage: p = x**3 + 2*x - 1
sage: p(numpy.float('1.2'))
3.12800000000000
sage: p(numpy.int('2'))
11.0000000000000
```

This used to fail (see trac ticket #18076):

```
sage: 1/3 + numpy.int8('12')
37/3
sage: -2/3 + numpy.int16('-2')
-8/3
sage: 2/5 + numpy.uint8('2')
12/5
```

The numpy types do not interact well with the Sage coercion framework. More precisely, if a numpy type is the first operand in a binary operation then this operation is done in numpy. The result is hence a numpy type:

```
sage: numpy.uint8('2') + 3
5
sage: type(_)
<type 'numpy.int32'>  # 32-bit
<type 'numpy.int64'>  # 64-bit

sage: numpy.int8('12') + 1/3
12.333333333333334
sage: type(_)
<type 'numpy.float64'>
```

AUTHOR:

- Robert Bradshaw

analyse (*xp*, *yp*, *op='mul'*)

Emulate the process of doing arithmetic between xp and yp, returning a list of steps and the parent that the result will live in. The `explain` function is easier to use, but if one wants access to the actual morphism and action objects (rather than their string representations) then this is the function to use.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: GF7 = GF(7)
sage: steps, res = cm.analyse(GF7, ZZ)
sage: steps
['Coercion on right operand via', Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 7, 'Arithmetic performed after coercions.']
sage: res
Finite Field of size 7
sage: f = steps[1]; type(f)
<type 'sage.rings.finite_rings.integer_mod.Integer_to_IntegerMod'>
sage: f(100)
2
```

bin_op (*x*, *y*, *op*)

Execute the operation op on x and y. It first looks for an action corresponding to op, and failing that, it tries to coerces x and y into a common parent and calls op on them.

If it cannot make sense of the operation, a TypeError is raised.

INPUT:

- `x` - the left operand
- `y` - the right operand
- `op` - a python function taking 2 arguments

Note: op is often an arithmetic operation, but need not be so.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.bin_op(1/2, 5, operator.mul)
5/2
```

The operator can be any callable:

```
sage: R.<x> = ZZ['x']
sage: cm.bin_op(x^2-1, x+1, gcd)
x + 1
```

Actions are detected and performed:

```
sage: M = matrix(ZZ, 2, 2, range(4))
sage: V = vector(ZZ, [5,7])
sage: cm.bin_op(M, V, operator.mul)
(7, 31)
```

TESTS:

```
sage: class Foo:
...       def __rmul__(self, left):
...           return 'hello'
...
sage: H = Foo()
sage: print(int(3)*H)
hello
sage: print(Integer(3)*H)
hello
sage: print(H*3)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '*': '<type 'instance'>' and
 ↪'Integer Ring'

sage: class Nonsense:
...       def __init__(self, s):
...           self.s = s
...       def __repr__(self):
...           return self.s
...       def __mul__(self, x):
...           return Nonsense(self.s + chr(x%256))
...       __add__ = __mul__
...       def __rmul__(self, x):
...           return Nonsense(chr(x%256) + self.s)
...       __radd__ = __rmul__
...
sage: a = Nonsense('blahblah')
sage: a*80
blahblahP
sage: 80*a
Pblahblah
sage: a+80
blahblahP
sage: 80+a
Pblahblah
```

canonical_coercion (*x*, *y*)

Given two elements x and y, with parents S and R respectively, find a common parent Z such that there are coercions $f : S \mapsto Z$ and $g : R \mapsto Z$ and return $f(x), g(y)$ which will have the same parent.

Raises a type error if no such Z can be found.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.canonical_coercion(mod(2, 10), 17)
(2, 7)
sage: x, y = cm.canonical_coercion(1/2, matrix(ZZ, 2, 2, range(4)))
sage: x
[1/2   0]
[  0 1/2]
sage: y
[0 1]
[2 3]
sage: parent(x) is parent(y)
True
```

There is some support for non-Sage datatypes as well:

```
sage: x, y = cm.canonical_coercion(int(5), 10)
sage: type(x), type(y)
(<type 'sage.rings.integer.Integer'>, <type 'sage.rings.integer.Integer'>)


sage: x, y = cm.canonical_coercion(int(5), complex(3))
sage: type(x), type(y)
(<type 'complex'>, <type 'complex'>)

sage: class MyClass:
...       def _sage_(self):
...           return 13
sage: a, b = cm.canonical_coercion(MyClass(), 1/3)
sage: a, b
(13, 1/3)
sage: type(a)
<type 'sage.rings.rational.Rational'>
```

We also make an exception for 0, even if **Z** does not map in:

```
sage: canonical_coercion(vector([1, 2, 3]), 0)
((1, 2, 3), (0, 0, 0))
sage: canonical_coercion(GF(5)(0), float(0))
(0, 0)
```

coercion_maps (R, S)

Give two parents R and S, return a pair of coercion maps $f : R \to Z$ and $g : S \to Z$, if such a Z can be found.

In the (common) case that $R = Z$ or $S = Z$ then `None` is returned for f or g respectively rather than constructing (and subsequently calling) the identity morphism.

If no suitable f, g can be found, a single `None` is returned. This result is cached.

Note: By trac ticket #14711, coerce maps should be copied when using them outside of the coercion system, because they may become defunct by garbage collection.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: f, g = cm.coercion_maps(ZZ, QQ)
sage: print(copy(f))
```

```
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print(g)
None

sage: ZZx = ZZ['x']
sage: f, g = cm.coercion_maps(ZZx, QQ)
sage: print(f)
(map internal to coercion system -- copy before use)
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
sage: print(g)
(map internal to coercion system -- copy before use)
Polynomial base injection morphism:
  From: Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field

sage: K = GF(7)
sage: cm.coercion_maps(QQ, K) is None
True
```

Note that to break symmetry, if there is a coercion map in both directions, the parent on the left is used:

```
sage: V = QQ^3
sage: W = V.__class__(QQ, 3)
sage: V == W
True
sage: V is W
False
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.coercion_maps(V, W)
(None, (map internal to coercion system -- copy before use)
Conversion map:
  From: Vector space of dimension 3 over Rational Field
  To:   Vector space of dimension 3 over Rational Field)
sage: cm.coercion_maps(W, V)
(None, (map internal to coercion system -- copy before use)
Conversion map:
  From: Vector space of dimension 3 over Rational Field
  To:   Vector space of dimension 3 over Rational Field)
sage: v = V([1,2,3])
sage: w = W([1,2,3])
sage: parent(v+w) is V
True
sage: parent(w+v) is W
True
```

TESTS:

We check that with trac ticket #14058, parents are still eligible for garbage collection after being involved in binary operations:

```
sage: import gc
sage: gc.collect() #random
852
sage: T=type(GF(2))
```

```
sage: N0=len(list(o for o in gc.get_objects() if type(o) is T))
sage: L=[ZZ(1)+GF(p)(1) for p in prime_range(2,50)]
sage: N1=len(list(o for o in gc.get_objects() if type(o) is T))
sage: N1 > N0
True
sage: del L
sage: gc.collect() #random
3939
sage: N2=len(list(o for o in gc.get_objects() if type(o) is T))
sage: N2-N0
0
```

common_parent (**args*)

Computes a common parent for all the inputs. It's essentially an n-ary canonical coercion except it can operate on parents rather than just elements.

INPUT:

- `args` – a set of elements and/or parents

OUTPUT:

A `Parent` into which each input should coerce, or raises a `TypeError` if no such `Parent` can be found.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.common_parent(ZZ, QQ)
Rational Field
sage: cm.common_parent(ZZ, QQ, RR)
Real Field with 53 bits of precision
sage: ZZT = ZZ[['T']]
sage: QQT = QQ['T']
sage: cm.common_parent(ZZT, QQT, RDF)
Power Series Ring in T over Real Double Field
sage: cm.common_parent(4r, 5r)
<type 'int'>
sage: cm.common_parent(int, float, ZZ)
<type 'float'>
sage: real_fields = [RealField(prec) for prec in [10,20..100]]
sage: cm.common_parent(*real_fields)
Real Field with 10 bits of precision
```

There are some cases where the ordering does matter, but if a parent can be found it is always the same:

```
sage: QQxy = QQ['x,y']
sage: QQyz = QQ['y,z']
sage: cm.common_parent(QQxy, QQyz) == cm.common_parent(QQyz, QQxy)
True
sage: QQzt = QQ['z,t']
sage: cm.common_parent(QQxy, QQyz, QQzt)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
sage: cm.common_parent(QQxy, QQzt, QQyz)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Multivariate␣
↪Polynomial Ring in x, y over Rational Field' and 'Multivariate Polynomial␣
↪Ring in z, t over Rational Field'
```

discover_action (*R*, *S*, *op*, *r=None*, *s=None*)

INPUT

- `R` - the left Parent (or type)
- `S` - the right Parent (or type)
- `op` - the operand, typically an element of the operator module
- `r` - (optional) element of R
- `s` - (optional) element of S.

OUTPUT:

- An action A such that s op r is given by A(s,r).

The steps taken are illustrated below.

EXAMPLES:

```
sage: P.<x> = ZZ['x']
sage: P.get_action(ZZ)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: ZZ.get_action(P) is None
True
sage: cm = sage.structure.element.get_coercion_model()
```

If R or S is a Parent, ask it for an action by/on R:

```
sage: cm.discover_action(ZZ, P, operator.mul)
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x␣
↪over Integer Ring
```

If R or S a type, recursively call get_action with the Sage versions of R and/or S:

```
sage: cm.discover_action(P, int, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Integer Ring
with precomposition on right by Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring
```

If op in an inplace operation, look for the non-inplace action:

```
sage: cm.discover_action(P, ZZ, operator.imul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Integer Ring
```

If op is division, look for action on right by inverse:

```
sage: cm.discover_action(P, ZZ, operator.div)
Right inverse action by Rational Field on Univariate Polynomial Ring in x␣
↪over Integer Ring
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

Check that trac ticket #17740 is fixed:

```
sage: R = GF(5)['x']
sage: cm.discover_action(R, ZZ, operator.div)
Right inverse action by Finite Field of size 5 on Univariate Polynomial Ring␣
↪in x over Finite Field of size 5
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 5
sage: cm.bin_op(R.gen(), 7, operator.div).parent()
Univariate Polynomial Ring in x over Finite Field of size 5
```

Check that trac ticket #18221 is fixed:

```
sage: F.<x> = FreeAlgebra(QQ)
sage: x / 2
1/2*x
sage: cm.discover_action(F, ZZ, operator.div)
Right inverse action by Rational Field on Free Algebra on 1 generators (x,)␣
↪over Rational Field
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

discover_coercion (*R*, *S*)

This actually implements the finding of coercion maps as described in the `coercion_maps` method.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
```

If R is S, then two identity morphisms suffice:

```
sage: cm.discover_coercion(SR, SR)
(None, None)
```

If there is a coercion map either direction, use that:

```
sage: cm.discover_coercion(ZZ, QQ)
((map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field, None)
sage: cm.discover_coercion(RR, QQ)
(None, (map internal to coercion system -- copy before use)
 Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

Otherwise, try and compute an appropriate cover:

```
sage: ZZxy = ZZ['x,y']
sage: cm.discover_coercion(ZZxy, RDF)
((map internal to coercion system -- copy before use)
Call morphism:
  From: Multivariate Polynomial Ring in x, y over Integer Ring
  To:   Multivariate Polynomial Ring in x, y over Real Double Field,
 Polynomial base injection morphism:
  From: Real Double Field
  To:   Multivariate Polynomial Ring in x, y over Real Double Field)
```

Sometimes there is a reasonable "cover," but no canonical coercion:

```
sage: sage.categories.pushout.pushout(QQ, QQ^3)
Vector space of dimension 3 over Rational Field
sage: print(cm.discover_coercion(QQ, QQ^3))
None
```

division_parent (*parent*)

Deduces where the result of division in parent lies by calculating the inverse of `parent.one()` or `parent.an_element()`.

The result is cached.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.division_parent(ZZ)
Rational Field
sage: cm.division_parent(QQ)
Rational Field
sage: ZZx = ZZ['x']
sage: cm.division_parent(ZZx)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: K = GF(41)
sage: cm.division_parent(K)
Finite Field of size 41
sage: Zmod100 = Integers(100)
sage: cm.division_parent(Zmod100)
Ring of integers modulo 100
sage: S5 = SymmetricGroup(5)
sage: cm.division_parent(S5)
Symmetric group of order 5! as a permutation group
```

exception_stack ( )

Returns the list of exceptions that were caught in the course of executing the last binary operation. Useful for diagnosis when user-defined maps or actions raise exceptions that are caught in the course of coercion detection.

If all went well, this should be the empty list. If things aren't happening as you expect, this is a good place to check. See also `coercion_traceback()`.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.record_exceptions()
sage: 1/2 + 2
5/2
sage: cm.exception_stack()
[]
sage: 1/2 + GF(3)(2)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Rational Field' and
↪'Finite Field of size 3'
```

Now see what the actual problem was:

```
sage: import traceback
sage: cm.exception_stack()
```

```
['Traceback (most recent call last):...', 'Traceback (most recent call last):.
↪..']
sage: print(cm.exception_stack()[-1])
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Rational␣
↪Field' and 'Finite Field of size 3'
```

This is typically accessed via the `coercion_traceback()` function.

```
sage: coercion_traceback()
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Rational␣
↪Field' and 'Finite Field of size 3'
```

explain (*xp*, *yp*, *op='mul'*, *verbosity=2*)

This function can be used to understand what coercions will happen for an arithmetic operation between xp and yp (which may be either elements or parents). If the parent of the result can be determined then it will be returned.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()

sage: cm.explain(ZZ, ZZ)
Identical parents, arithmetic performed immediately.
Result lives in Integer Ring
Integer Ring

sage: cm.explain(QQ, int)
Coercion on right operand via
    Native morphism:
      From: Set of Python objects of type 'int'
      To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: R = ZZ['x']
sage: cm.explain(R, QQ)
Action discovered.
    Right scalar multiplication by Rational Field on Univariate Polynomial␣
↪Ring in x over Integer Ring
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(ZZ['x'], QQ, operator.add)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Coercion on right operand via
    Polynomial base injection morphism:
```

```
      From: Rational Field
      To:   Univariate Polynomial Ring in x over Rational Field
Arithmetic performed after coercions.
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Sometimes with non-sage types there is not enough information to deduce what will actually happen:

```
sage: R100 = RealField(100)
sage: cm.explain(R100, float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(R100(1) + float(1))
<type 'float'>
sage: cm.explain(QQ, float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(QQ(1) + float(1))
<type 'float'>
```

Special care is taken to deal with division:

```
sage: cm.explain(ZZ, ZZ, operator.div)
Identical parents, arithmetic performed immediately.
Result lives in Rational Field
Rational Field

sage: ZZx = ZZ['x']
sage: QQx = QQ['x']
sage: cm.explain(ZZx, QQx, operator.div)
Coercion on left operand via
    Ring morphism:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
      Defn: Induced from base ring by
            Natural morphism:
              From: Integer Ring
              To:   Rational Field
Arithmetic performed after coercions.
Result lives in Fraction Field of Univariate Polynomial Ring in x over␣
↪Rational Field
Fraction Field of Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(int, ZZ, operator.div)
Coercion on left operand via
    Native morphism:
      From: Set of Python objects of type 'int'
      To:   Integer Ring
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZx, ZZ, operator.div)
Action discovered.
    Right inverse action by Rational Field on Univariate Polynomial Ring in x␣
↪over Integer Ring
    with precomposition on right by Natural morphism:
      From: Integer Ring
```

```
      To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Note: This function is accurate only in so far as analyse is kept in sync with the *bin_op()* and *canonical_coercion()* which are kept separate for maximal efficiency.

get_action (*R*, *S*, *op*, *r=None*, *s=None*)

Get the action of R on S or S on R associated to the operation op.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: ZZx = ZZ['x']
sage: cm.get_action(ZZx, ZZ, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: cm.get_action(ZZx, ZZ, operator.imul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: cm.get_action(ZZx, QQ, operator.mul)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring␣
↪in x over Integer Ring
sage: QQx = QQ['x']
sage: cm.get_action(QQx, int, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in␣
↪x over Rational Field
with precomposition on right by Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring

sage: A = cm.get_action(QQx, ZZ, operator.div); A
Right inverse action by Rational Field on Univariate Polynomial Ring in x␣
↪over Rational Field
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: x = QQx.gen()
sage: A(x+10, 5)
1/5*x + 2
```

get_cache ( )

This returns the current cache of coercion maps and actions, primarily useful for debugging and introspection.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.canonical_coercion(1,2/3)
(1, 2/3)
sage: maps, actions = cm.get_cache()
```

Now let us see what happens when we do a binary operations with an integer and a rational:

```
sage: left_morphism_ref, right_morphism_ref = maps[ZZ, QQ]
```

Note that by trac ticket #14058 the coercion model only stores a weak reference to the coercion maps in this case:

```
sage: left_morphism_ref
<weakref at ...; to 'sage.rings.rational.Z_to_Q' at ...
(RingHomset_generic_with_category._abstract_element_class)>
```

Moreover, the weakly referenced coercion map uses only a weak reference to the codomain:

```
sage: left_morphism_ref()
(map internal to coercion system -- copy before use)
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

To get an actual valid map, we simply copy the weakly referenced coercion map:

```
sage: print(copy(left_morphism_ref()))
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print(right_morphism_ref)
None
```

We can see that it coerces the left operand from an integer to a rational, and doesn't do anything to the right.

Now for some actions:

```
sage: R.<x> = ZZ['x']
sage: 1/2 * x
1/2*x
sage: maps, actions = cm.get_cache()
sage: act = actions[QQ, R, operator.mul]; act
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: act.actor()
Rational Field
sage: act.domain()
Univariate Polynomial Ring in x over Integer Ring
sage: act.codomain()
Univariate Polynomial Ring in x over Rational Field
sage: act(1/5, x+10)
1/5*x + 2
```

record_exceptions (*value=True*)

Enables (or disables) recording of the exceptions suppressed during arithmetic.

Each time that record_exceptions is called (either enabling or disabling the record), the exception_stack is cleared.

TESTS:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.record_exceptions()
sage: cm._test_exception_stack()
sage: cm.exception_stack()
['Traceback (most recent call last):\n  File "sage/structure/coerce.pyx",␣
↪line ...TypeError: just a test']
```

```
sage: cm.record_exceptions(False)
sage: cm._test_exception_stack()
sage: cm.exception_stack()
[]
```

reset_cache (*lookup_dict_size=127*, *lookup_dict_threshold=0.75*)

Clear the coercion cache.

This should have no impact on the result of arithmetic operations, as the exact same coercions and actions will be re-discovered when needed.

It may be useful for debugging, and may also free some memory.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: len(cm.get_cache()[0])     # random
42
sage: cm.reset_cache()
sage: cm.get_cache()
({}, {})
```

richcmp (*x*, *y*, *op*)

Given two arbitrary objects `x` and `y` , coerce them to a common parent and compare them using rich comparison operator `op` .

EXAMPLES:

```
sage: from sage.structure.element import get_coercion_model
sage: from sage.structure.sage_object import op_LT, op_LE, op_EQ, op_NE, op_
↪GT, op_GE
sage: richcmp = get_coercion_model().richcmp
sage: richcmp(None, None, op_EQ)
True
sage: richcmp(None, 1, op_LT)
True
sage: richcmp("hello", None, op_LE)
False
sage: richcmp(-1, 1, op_GE)
False
sage: richcmp(int(1), float(2), op_GE)
False
```

If there is no coercion, compare types:

```
sage: x = QQ.one(); y = GF(2).one()
sage: richcmp(x, y, op_EQ)
False
sage: richcmp(x, y, op_NE)
True
sage: richcmp(x, y, op_LT if cmp(type(x), type(y)) == -1 else op_GT)
True
```

verify_action (*action*, *R*, *S*, *op*, *fix=True*)

Verify that `action` takes an element of R on the left and S on the right, raising an error if not.

This is used for consistency checking in the coercion model.

EXAMPLES:

```
sage: R.<x> = ZZ['x']
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.verify_action(R.get_action(QQ), R, QQ, operator.mul)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring␣
↪in x over Integer Ring
sage: cm.verify_action(R.get_action(QQ), RDF, R, operator.mul)
Traceback (most recent call last):
...
RuntimeError: There is a BUG in the coercion model:
    Action found for R <built-in function mul> S does not have the correct␣
↪domains
    R = Real Double Field
    S = Univariate Polynomial Ring in x over Integer Ring
    (should be Univariate Polynomial Ring in x over Integer Ring, Rational␣
↪Field)
    action = Right scalar multiplication by Rational Field on Univariate␣
↪Polynomial Ring in x over Integer Ring (<type 'sage.structure.coerce_
↪actions.RightModuleAction'>)
```

verify_coercion_maps (*R*, *S*, *homs*, *fix=False*)

Make sure this is a valid pair of homomorphisms from R and S to a common parent. This function is used to protect the user against buggy parents.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: homs = QQ.coerce_map_from(ZZ), None
sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
True
sage: homs = QQ.coerce_map_from(ZZ), RR.coerce_map_from(QQ)
sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
Traceback (most recent call last):
...
RuntimeError: ('BUG in coercion model, codomains must be identical', Natural␣
↪morphism:
  From: Integer Ring
  To:   Rational Field, Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

sage.structure.coerce. **is_numpy_type** (*t*)

Return `True` if and only if *t* is a type whose name starts with `numpy`.

EXAMPLES:

```
sage: from sage.structure.coerce import is_numpy_type
sage: import numpy
sage: is_numpy_type(numpy.int16)
True
sage: is_numpy_type(numpy.floating)
True
sage: is_numpy_type(numpy.float)  # Alias for Python float
False
sage: is_numpy_type(numpy.ndarray)
True
sage: is_numpy_type(numpy.matrix)
True
sage: is_numpy_type(int)
```

```
False
sage: is_numpy_type(Integer)
False
sage: is_numpy_type(Sudoku)
False
sage: is_numpy_type(None)
False
```

TESTS:

This used to crash Sage (trac ticket #20715):

```
sage: is_numpy_type(object)
False
sage: 1 + object()
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Integer Ring' and '<type
↪'object'>'
```

sage.structure.coerce. **py_scalar_parent** (*py_type*)

Returns the Sage equivalent of the given python type, if one exists. If there is no equivalent, return None.

EXAMPLES:

```
sage: from sage.structure.coerce import py_scalar_parent
sage: py_scalar_parent(int)
Integer Ring
sage: py_scalar_parent(long)
Integer Ring
sage: py_scalar_parent(float)
Real Double Field
sage: py_scalar_parent(complex)
Complex Double Field
sage: py_scalar_parent(bool)
Integer Ring
sage: py_scalar_parent(dict),
(None,)

sage: import numpy
sage: py_scalar_parent(numpy.int16)
Integer Ring
sage: py_scalar_parent(numpy.int32)
Integer Ring
sage: py_scalar_parent(numpy.uint64)
Integer Ring

sage: py_scalar_parent(numpy.float)
Real Double Field
sage: py_scalar_parent(numpy.double)
Real Double Field

sage: py_scalar_parent(numpy.complex)
Complex Double Field
```

sage.structure.coerce. **py_scalar_to_element** (*x*)

Convert `x` to a Sage `Element` if possible.

If `x` was already an `Element` or if there is no obvious conversion possible, just return `x` itself.

EXAMPLES:

```
sage: from sage.structure.coerce import py_scalar_to_element
sage: x = py_scalar_to_element(42)
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(int(42))
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(long(42))
sage: x, parent(x)
(42, Integer Ring)
sage: x = py_scalar_to_element(float(42))
sage: x, parent(x)
(42.0, Real Double Field)
sage: x = py_scalar_to_element(complex(42))
sage: x, parent(x)
(42.0, Complex Double Field)
sage: py_scalar_to_element('hello')
'hello'
```

Note that bools are converted to 0 or 1:

```
sage: py_scalar_to_element(False), py_scalar_to_element(True)
(0, 1)
```

Test compatibility with *py_scalar_parent()* :

```
sage: from sage.structure.coerce import py_scalar_parent
sage: elt = [True, int(42), long(42), float(42), complex(42)]
sage: for x in elt:
....:     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()

sage: import numpy
sage: elt = [numpy.int8('-12'),  numpy.uint8('143'),
....:        numpy.int16('-33'), numpy.uint16('122'),
....:        numpy.int32('-19'), numpy.uint32('44'),
....:        numpy.int64('-3'),  numpy.uint64('552'),
....:        numpy.float16('-1.23'), numpy.float32('-2.22'),
....:        numpy.float64('-3.412'), numpy.complex64(1.2+I),
....:         numpy.complex128(-2+I)]
sage: for x in elt:
....:     assert py_scalar_parent(type(x)) == py_scalar_to_element(x).parent()
```

5.2 Coerce actions

class sage.structure.coerce_actions. **ActOnAction**
Bases: *sage.structure.coerce_actions.GenericAction*

Class for actions defined via the _act_on_ method.

class sage.structure.coerce_actions. **ActedUponAction**
Bases: *sage.structure.coerce_actions.GenericAction*

Class for actions defined via the _acted_upon_ method.

class sage.structure.coerce_actions. **GenericAction**

Bases: *sage.categories.action.Action*

TESTS:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see trac ticket #18157).

```
sage: M = MatrixSpace(ZZ,2)
sage: sage.structure.coerce_actions.ActedUponAction(M, Cusps, True)
Left action by Full MatrixSpace of 2 by 2 dense matrices over Integer Ring on Set␣
↪P^1(QQ) of all cusps

sage: Z6 = Zmod(6)
sage: sage.structure.coerce_actions.GenericAction(QQ, Z6, True)
Traceback (most recent call last):
...
NotImplementedError: Action not implemented.
```

This will break if we tried to use it:

```
sage: sage.structure.coerce_actions.GenericAction(QQ, Z6, True, check=False)
Left action by Rational Field on Ring of integers modulo 6
```

codomain ( )

Returns the "codomain" of this action, i.e. the Parent in which the result elements live. Typically, this should be the same as the acted upon set.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see trac ticket #18157).

```
sage: M = MatrixSpace(ZZ,2)
sage: A = sage.structure.coerce_actions.ActedUponAction(M, Cusps, True)
sage: A.codomain()
Set P^1(QQ) of all cusps

sage: S3 = SymmetricGroup(3)
sage: QQxyz = QQ['x,y,z']
sage: A = sage.structure.coerce_actions.ActOnAction(S3, QQxyz, False)
sage: A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

class sage.structure.coerce_actions. **IntegerMulAction**

Bases: *sage.categories.action.Action*

This class implements the action $n \cdot a = a + a + \cdots + a$ via repeated doubling.

Both addition and negation must be defined on the set M.

NOTE:

This class is used internally in Sage's coercion model. Outside of the coercion model, special precautions are needed to prevent domains of the action from being garbage collected.

INPUT:

- An integer ring, ZZ

- A ZZ module M
- Optional: An element m of M

EXAMPLES:

```
sage: from sage.structure.coerce_actions import IntegerMulAction
sage: R.<x> = QQ['x']
sage: act = IntegerMulAction(ZZ, R)
sage: act(5, x)
5*x
sage: act(0, x)
0
sage: act(-3, x-1)
-3*x + 3
```

class sage.structure.coerce_actions. **LAction**
Bases: *sage.categories.action.Action*

Action calls _l_action of the actor.

class sage.structure.coerce_actions. **LeftModuleAction**
Bases: *sage.structure.coerce_actions.ModuleAction*

class sage.structure.coerce_actions. **ModuleAction**
Bases: *sage.categories.action.Action*

Module action.

See also:

This is an abstract class, one must actually instantiate a *LeftModuleAction* or a *RightModuleAction* .

INPUT:

- G – the actor, an instance of Parent .
- S – the object that is acted upon.
- g – optional, an element of G .
- a – optional, an element of S .
- check – if True (default), then there will be no consistency tests performed on sample elements.

NOTE:

By default, the sample elements of S and G are obtained from an_element() , which relies on the implementation of an _an_element_() method. This is not always available. But usually, the action is only needed when one already *has* two elements. Hence, by trac ticket #14249, the coercion model will pass these two elements the the *ModuleAction* constructor.

The actual action is implemented by the _rmul_ or _lmul_ function on its elements. We must, however, be very particular about what we feed into these functions, because they operate under the assumption that the inputs lie exactly in the base ring and may segfault otherwise. Thus we handle all possible base extensions manually here.

codomain ( )
The codomain of self, which may or may not be equal to the domain.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see trac ticket #18157).

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: ZZxyz = ZZ['x,y,z']
sage: A = LeftModuleAction(QQ, ZZxyz)
sage: A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

domain ( )

The domain of self, which is the module that is being acted on.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see trac ticket #18157).

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: ZZxyz = ZZ['x,y,z']
sage: A = LeftModuleAction(QQ, ZZxyz)
sage: A.domain()
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

class sage.structure.coerce_actions. **PyScalarAction**

Bases: *sage.categories.action.Action*

class sage.structure.coerce_actions. **RAction**

Bases: *sage.categories.action.Action*

Action calls _r_action of the actor.

class sage.structure.coerce_actions. **RightModuleAction**

Bases: *sage.structure.coerce_actions.ModuleAction*

sage.structure.coerce_actions. **detect_element_action** (*X*, *Y*, *X_on_left*, *X_el=None*, *Y_el=None*)

Returns an action of X on Y or Y on X as defined by elements X, if any.

EXAMPLES:

Note that coerce actions should only be used inside of the coercion model. For this test, we need to strongly reference the domains, for otherwise they could be garbage collected, giving rise to random errors (see trac ticket #18157).

```
sage: from sage.structure.coerce_actions import detect_element_action
sage: ZZx = ZZ['x']
sage: M = MatrixSpace(ZZ,2)
sage: detect_element_action(ZZx, ZZ, False)
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x␣
↪over Integer Ring
sage: detect_element_action(ZZx, QQ, True)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x␣
↪over Integer Ring
sage: detect_element_action(Cusps, M, False)
Left action by Full MatrixSpace of 2 by 2 dense matrices over Integer Ring on Set␣
↪P^1(QQ) of all cusps
sage: detect_element_action(Cusps, M, True),
(None,)
```

```
sage: detect_element_action(ZZ, QQ, True),
(None,)
```

TESTS:

This test checks that the issue in trac ticket #7718 has been fixed:

```
sage: class MyParent(Parent):
....:     def an_element(self):
....:         pass
....:
sage: A = MyParent()
sage: detect_element_action(A, ZZ, True)
Traceback (most recent call last):
...
RuntimeError: an_element() for <class '__main__.MyParent'> returned None
```

5.3 Coerce maps

class sage.structure.coerce_maps. **CCallableConvertMap_class**
Bases: sage.categories.map.Map

class sage.structure.coerce_maps. **CallableConvertMap**
Bases: sage.categories.map.Map

This lets one easily create maps from any callable object.

This is especially useful to create maps from bound methods.

EXAMPLES:

```
sage: from sage.structure.coerce_maps import CallableConvertMap
sage: def foo(P, x): return x/2
sage: f = CallableConvertMap(ZZ, QQ, foo)
sage: f(3)
3/2
sage: f
Conversion via foo map:
  From: Integer Ring
  To:   Rational Field
```

Create a homomorphism from $\mathbf{R}$ to $\mathbf{R}^+$ viewed as additive groups.

```
sage: f = CallableConvertMap(RR, RR, exp, parent_as_first_arg=False)
sage: f(0)
1.00000000000000
sage: f(1)
2.71828182845905
sage: f(-3)
0.0497870683678639
```

class sage.structure.coerce_maps. **DefaultConvertMap**
Bases: sage.categories.map.Map

This morphism simply calls the codomain's element_constructor method, passing in the codomain as the first argument.

class sage.structure.coerce_maps. **DefaultConvertMap_unique**
Bases: *sage.structure.coerce_maps.DefaultConvertMap*

This morphism simply defers action to the codomain's element_constructor method, WITHOUT passing in the codomain as the first argument.

This is used for creating elements that don't take a parent as the first argument to their __init__ method, for example, Integers, Rationals, Algebraic Reals... all have a unique parent. It is also used when the element_constructor is a bound method (whose self argument is assumed to be bound to the codomain).

class sage.structure.coerce_maps. **ListMorphism**
Bases: sage.categories.map.Map

class sage.structure.coerce_maps. **NamedConvertMap**
Bases: sage.categories.map.Map

This is used for creating a elements via the _xxx_ methods.

For example, many elements implement an _integer_ method to convert to ZZ, or a _rational_ method to convert to QQ.

method_name

class sage.structure.coerce_maps. **TryMap**
Bases: sage.categories.map.Map

TESTS:

```
sage: sage.structure.coerce_maps.TryMap(RDF.coerce_map_from(QQ), RDF.coerce_map_
↪from(ZZ))
Traceback (most recent call last):
...
TypeError: incorrectly matching parent
```

sage.structure.coerce_maps. **test_CCallableConvertMap** (*domain*, *name=None*)
For testing CCallableConvertMap_class.

TESTS:

```
sage: from sage.structure.coerce_maps import test_CCallableConvertMap
sage: f = test_CCallableConvertMap(ZZ, 'test'); f
Conversion via c call 'test' map:
  From: Integer Ring
  To:   Integer Ring
sage: f(3)
24
sage: f(9)
720
```

5.4 Coercion via Construction Functors

class sage.categories.pushout. **AlgebraicClosureFunctor**
Bases: *sage.categories.pushout.ConstructionFunctor*

Algebraic Closure.

EXAMPLE:

```
sage: F = CDF.construction()[0]
sage: F(QQ)
Algebraic Field
sage: F(RR)
Complex Field with 53 bits of precision
sage: F(F(QQ)) is F(QQ)
True
```

merge (*other*)

Mathematically, Algebraic Closure subsumes Algebraic Extension. However, it seems that people do want to work with algebraic extensions of `RR` . Therefore, we do not merge with algebraic extension.

TEST:

```
sage: K.<a>=NumberField(x^3+x^2+1)
sage: CDF.construction()[0].merge(K.construction()[0]) is None
True
sage: CDF.construction()[0].merge(CDF.construction()[0])
AlgebraicClosureFunctor
```

class sage.categories.pushout. **AlgebraicExtensionFunctor** (*polys*, *names*, *embeddings=None*, *structures=None*, *cyclotomic=None*, ***kwds*)

Bases: *sage.categories.pushout.ConstructionFunctor*

Algebraic extension (univariate polynomial ring modulo principal ideal).

EXAMPLE:

```
sage: K.<a> = NumberField(x^3+x^2+1)
sage: F = K.construction()[0]
sage: F(ZZ['t'])
Univariate Quotient Polynomial Ring in a over Univariate Polynomial Ring in t␣
↪over Integer Ring with modulus a^3 + a^2 + 1
```

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

```
sage: F(CC)
Univariate Quotient Polynomial Ring in a over Complex Field with 53 bits of␣
↪precision with modulus a^3 + a^2 + 1.00000000000000
sage: F(RR)
Univariate Quotient Polynomial Ring in a over Real Field with 53 bits of␣
↪precision with modulus a^3 + a^2 + 1.00000000000000
```

Note that the construction functor of a number field applied to the integers returns an order (not necessarily maximal) of that field, similar to the behaviour of `ZZ.extension(...)` :

```
sage: F(ZZ)
Order in Number Field in a with defining polynomial x^3 + x^2 + 1
```

This also holds for non-absolute number fields:

```
sage: K.<a,b> = NumberField([x^3+x^2+1,x^2+x+1])
sage: F = K.construction()[0]
sage: O = F(ZZ); O
Relative Order in Number Field in a with defining polynomial x^3 + x^2 + 1 over␣
↪its base field
```

```
sage: O.ambient() is K
True
```

expand ( )

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

EXAMPLES:

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

merge (*other*)

Merging with another *AlgebraicExtensionFunctor* .

INPUT:

`other` – Construction Functor.

OUTPUT:

- If `self==other` , `self` is returned.
- If `self` and `other` are simple extensions and both provide an embedding, then it is tested whether one of the number fields provided by the functors coerces into the other; the functor associated with the target of the coercion is returned. Otherwise, the construction functor associated with the pushout of the codomains of the two embeddings is returned, provided that it is a number field.
- If these two extensions are defined by Conway polynomials over finite fields, merges them into a single extension of degree the lcm of the two degrees.
- Otherwise, None is returned.

REMARK:

Algebraic extension with embeddings currently only works when applied to the rational field. This is why we use the admittedly strange rule above for merging.

EXAMPLES:

The following demonstrate coercions for finite fields using Conway or pseudo-Conway polynomials:

```
sage: k = GF(3^2, prefix='z'); a = k.gen()
sage: l = GF(3^3, prefix='z'); b = l.gen()
sage: a + b # indirect doctest
z6^5 + 2*z6^4 + 2*z6^3 + z6^2 + 2*z6 + 1
```

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

```
sage: m = GF(3^5, prefix='z'); c = m.gen()
sage: (a+b)+c == a+(b+c) # indirect doctest
True
sage: from sage.categories.pushout import pushout
sage: n = pushout(k, l)
sage: o = pushout(l, m)
sage: q = pushout(n, o)
sage: q(o(b)) == q(n(b)) # indirect doctest
True
```

Coercion is also available for number fields:

```
sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^8-x^4+1, embedding=CDF.0)
sage: M1.<c1> = NumberField(x^2+x+1, embedding=b^4-1)
sage: M2.<c2> = NumberField(x^2+1, embedding=-b^6)
sage: M1.coerce_map_from(M2)
sage: M2.coerce_map_from(M1)
sage: c1+c2; parent(c1+c2)    #indirect doctest
-b^6 + b^4 - 1
Number Field in b with defining polynomial x^8 - x^4 + 1
sage: pushout(M1['x'],M2['x'])
Univariate Polynomial Ring in x over Number Field in b with defining␣
↪polynomial x^8 - x^4 + 1
```

In the previous example, the number field `L` becomes the pushout of `M1` and `M2` since both are provided with an embedding into `L`, *and* since `L` is a number field. If two number fields are embedded into a field that is not a numberfield, no merging occurs:

```
sage: K.<a> = NumberField(x^3-2, embedding=CDF(1/2*I*2^(1/3)*sqrt(3) - 1/2*2^
↪(1/3)))
sage: L.<b> = NumberField(x^6-2, embedding=1.1)
sage: L.coerce_map_from(K)
sage: K.coerce_map_from(L)
sage: pushout(K,L)
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', Number Field in a with␣
↪defining polynomial x^3 - 2, Number Field in b with defining polynomial x^6␣
↪- 2)
```

class sage.categories.pushout.**BlackBoxConstructionFunctor**(*box*)

Bases: *sage.categories.pushout.ConstructionFunctor*

Construction functor obtained from any callable object.

EXAMPLES:

```
sage: from sage.categories.pushout import BlackBoxConstructionFunctor
sage: FG = BlackBoxConstructionFunctor(gap)
sage: FS = BlackBoxConstructionFunctor(singular)
sage: FG
BlackBoxConstructionFunctor
sage: FG(ZZ)
Integers
sage: FG(ZZ).parent()
Gap
sage: FS(QQ['t'])
```

```
//   characteristic : 0
//   number of vars : 1
//        block   1 : ordering lp
//                  : names    t
//        block   2 : ordering C
sage: FG == FS
False
sage: FG == loads(dumps(FG))
True
```

class sage.categories.pushout. **CompletionFunctor** (*p*, *prec*, *extras=None*)

Bases: *sage.categories.pushout.ConstructionFunctor*

Completion of a ring with respect to a given prime (including infinity).

EXAMPLES:

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

TEST:

```
sage: R1.<a> = Zp(5,prec=20)[]
sage: R2 = Qp(5,prec=40)
sage: R2(1) + a
(1 + O(5^20))*a + (1 + O(5^40))
sage: 1/2 + a
(1 + O(5^20))*a + (3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^
↪8 + 2*5^9 + 2*5^10 + 2*5^11 + 2*5^12 + 2*5^13 + 2*5^14 + 2*5^15 + 2*5^16 + 2*5^
↪17 + 2*5^18 + 2*5^19 + O(5^20))
```

commutes (*other*)

Completion commutes with fraction fields.

EXAMPLE:

```
sage: F1 = Qp(5).construction()[0]
sage: F2 = QQ.construction()[0]
```

```
sage: F1.commutes(F2)
True
```

TEST:

The fraction field `R` in the example below has no completion method. But completion commutes with the fraction field functor, and so it is tried internally whether applying the construction functors in opposite order works. It does:

```
sage: P.<x> = ZZ[]
sage: C = P.completion(x).construction()[0]
sage: R = FractionField(P)
sage: hasattr(R,'completion')
False
sage: C(R) is Frac(C(P))
True
sage: F = R.construction()[0]
sage: (C*F)(ZZ['x']) is (F*C)(ZZ['x'])
True
```

The following was fixed in trac ticket #15329 (it used to result in an infinite recursion):

```
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7),RLF)
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', 7-adic Field with capped␣
↪relative precision 20, Real Lazy Field)
```

merge (*other*)

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

EXAMPLE:

```
sage: R1.<a> = Zp(5,prec=20)[]
sage: R2 = Qp(5,prec=40)
sage: R2(1)+a          # indirect doctest
(1 + O(5^20))*a + (1 + O(5^40))
sage: R3 = RealField(30)
sage: R4 = RealField(50)
sage: R3(1) + R4(1)    # indirect doctest
2.0000000
sage: (R3(1) + R4(1)).parent()
Real Field with 30 bits of precision
```

TESTS:

We check that #12353 has been resolved:

```
sage: RealIntervalField(53)(-1) > RR(1)
False
sage: RealIntervalField(54)(-1) > RR(1)
False
sage: RealIntervalField(54)(1) > RR(-1)
True
```

```
sage: RealIntervalField(53)(1) > RR(-1)
True
```

We check that various pushouts work:

```
sage: R0 = RealIntervalField(30)
sage: R1 = RealIntervalField(30, sci_not=True)
sage: R2 = RealIntervalField(53)
sage: R3 = RealIntervalField(53, sci_not = True)
sage: R4 = RealIntervalField(90)
sage: R5 = RealIntervalField(90, sci_not = True)
sage: R6 = RealField(30)
sage: R7 = RealField(30, sci_not=True)
sage: R8 = RealField(53, rnd = 'RNDD')
sage: R9 = RealField(53, sci_not = True, rnd = 'RNDZ')
sage: R10 = RealField(53, sci_not = True)
sage: R11 = RealField(90, sci_not = True, rnd = 'RNDZ')
sage: Rlist = [R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11]
sage: from sage.categories.pushout import pushout
sage: pushouts =␣
↪[R0,R0,R0,R1,R0,R1,R0,R1,R0,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R1,R0,R1,R2,R2,R2,R3
sage: all([R is S for R, S in zip(pushouts, [pushout(a, b) for a in Rlist for␣
↪b in Rlist])])
True
```

```
sage: P0 = ZpFM(5, 10)
sage: P1 = ZpFM(5, 20)
sage: P2 = ZpCR(5, 10)
sage: P3 = ZpCR(5, 20)
sage: P4 = ZpCA(5, 10)
sage: P5 = ZpCA(5, 20)
sage: P6 = Qp(5, 10)
sage: P7 = Qp(5, 20)
sage: Plist = [P2,P3,P4,P5,P6,P7]
sage: from sage.categories.pushout import pushout
sage: pushouts =␣
↪[P2,P3,P4,P5,P6,P7,P3,P3,P5,P5,P7,P7,P4,P5,P4,P5,P6,P7,P5,P5,P5,P5,P7,P7,P6,P7,P6,P7,P6,P7
sage: all([P is Q for P, Q in zip(pushouts, [pushout(a, b) for a in Plist for␣
↪b in Plist])])
True
```

class sage.categories.pushout. **CompositeConstructionFunctor** (**args*)

Bases: *sage.categories.pushout.ConstructionFunctor*

A Construction Functor composed by other Construction Functors.

INPUT:

`F1,F2,...` : A list of Construction Functors. The result is the composition `F1` followed by `F2` followed by ...

EXAMPLES:

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

```
True
sage: F == CompositeConstructionFunctor(*F.all)
True
sage: F(GF(2)['t'])
Univariate Polynomial Ring in y over Fraction Field of Univariate Polynomial Ring␣
↪in x over Fraction Field of Univariate Polynomial Ring in t over Finite Field␣
↪of size 2 (using NTL)
```

expand ( )

Return expansion of a CompositeConstructionFunctor.

NOTE:

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

EXAMPLES:

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

class sage.categories.pushout. **ConstructionFunctor**

Bases: sage.categories.functor.Functor

Base class for construction functors.

A construction functor is a functorial algebraic construction, such as the construction of a matrix ring over a given ring or the fraction field of a given ring.

In addition to the class `Functor`, construction functors provide rules for combining and merging constructions. This is an important part of Sage's coercion model, namely the pushout of two constructions: When a polynomial `p` in a variable `x` with integer coefficients is added to a rational number `q`, then Sage finds that the parents `ZZ['x']` and `QQ` are obtained from `ZZ` by applying a polynomial ring construction respectively the fraction field construction. Each construction functor has an attribute `rank`, and the rank of the polynomial ring construction is higher than the rank of the fraction field construction. This means that the pushout of `QQ` and `ZZ['x']`, and thus a common parent in which `p` and `q` can be added, is `QQ['x']`, since the construction functor with a lower rank is applied first.

```
sage: F1, R = QQ.construction()
sage: F1
FractionField
sage: R
Integer Ring
sage: F2, R = (ZZ['x']).construction()
sage: F2
Poly[x]
sage: R
Integer Ring
sage: F3 = F2.pushout(F1)
sage: F3
Poly[x](FractionField(...))
sage: F3(R)
Univariate Polynomial Ring in x over Rational Field
sage: from sage.categories.pushout import pushout
sage: P.<x> = ZZ[]
```

```
sage: pushout(QQ,P)
Univariate Polynomial Ring in x over Rational Field
sage: ((x+1) + 1/2).parent()
Univariate Polynomial Ring in x over Rational Field
```

When composing two construction functors, they are sometimes merged into one, as is the case in the Quotient construction:

```
sage: Q15, R = (ZZ.quo(15*ZZ)).construction()
sage: Q15
QuotientFunctor
sage: Q35, R = (ZZ.quo(35*ZZ)).construction()
sage: Q35
QuotientFunctor
sage: Q15.merge(Q35)
QuotientFunctor
sage: Q15.merge(Q35)(ZZ)
Ring of integers modulo 5
```

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

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

common_base (*other_functor*, *self_bases*, *other_bases*)

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

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

INPUT:

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

OUTPUT:

Nothing, since a *CoercionException* is raised.

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

TESTS:

```
sage: from sage.categories.pushout import pushout
sage: pushout(QQ, cartesian_product([ZZ]))  # indirect doctest
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
FractionField(Integer Ring) and The cartesian_product functorial␣
↪construction(Integer Ring).
```

commutes (*other*)

Determine whether `self` commutes with another construction functor.

NOTE:

By default, `False` is returned in all cases (even if the two functors are the same, since in this case *merge()* will apply anyway). So far there is no construction functor that overloads this method. Anyway, this method only becomes relevant if two construction functors have the same rank.

EXAMPLES:

```
sage: F = QQ.construction()[0]
sage: P = ZZ['t'].construction()[0]
sage: F.commutes(P)
False
sage: P.commutes(F)
False
sage: F.commutes(F)
False
```

expand ( )

Decompose `self` into a list of construction functors.

NOTE:

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

EXAMPLE:

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

merge (*other*)

Merge `self` with another construction functor, or return None.

NOTE:

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

EXAMPLES:

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

pushout (*other*)

Composition of two construction functors, ordered by their ranks.

NOTE:

- This method seems not to be used in the coercion model.
- By default, the functor with smaller rank is applied first.

TESTS:

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

class sage.categories.pushout. **FractionField**

Bases: *sage.categories.pushout.ConstructionFunctor*

Construction functor for fraction fields.

EXAMPLE:

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

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

Bases: *sage.categories.pushout.ConstructionFunctor*

A construction functor that is the identity functor.

TESTS:

```
sage: from sage.categories.pushout import IdentityConstructionFunctor
sage: I = IdentityConstructionFunctor()
sage: I(RR) is RR
True
sage: I == loads(dumps(I))
True
```

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

Bases: *sage.categories.pushout.ConstructionFunctor*

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

AUTHOR:

– Simon King

This construction functor is used to provide uniqueness of infinite polynomial rings as parent structures. As usual, the construction functor allows for constructing pushouts.

Another purpose is to avoid name conflicts of variables of the to-be-constructed infinite polynomial ring with variables of the base ring, and moreover to keep the internal structure of an Infinite Polynomial Ring as simple as possible: If variables $v_1, ..., v_n$ of the given base ring generate an *ordered* sub-monoid of the monomials of the ambient Infinite Polynomial Ring, then they are removed from the base ring and merged with the generators of the ambient ring. However, if the orders don't match, an error is raised, since there was a name conflict without merging.

EXAMPLES:

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

Apparently the variables a_1, a_3 of the polynomial ring are merged with the variables $a_0, a_1, a_2, ...$ of the infinite polynomial ring; indeed, they form an ordered sub-structure. However, if the polynomial ring was given a different ordering, merging would not be allowed, resulting in a name conflict:

```
sage: A.construction()[0]*PolynomialRing(QQ,names=['x','y','a_3','a_1']).
↪construction()[0]
Traceback (most recent call last):
...
CoercionException: Incompatible term orders lex, degrevlex
```

In an infinite polynomial ring with generator a_*, the variable a_3 will always be greater than the variable a_1. Hence, the orders are incompatible in the next example as well:

```
sage: A.construction()[0]*PolynomialRing(QQ,names=['x','y','a_1','a_3'], order=
↪'lex').construction()[0]
Traceback (most recent call last):
```

```
...
CoercionException: Overlapping variables (('a', 'b'),['a_1', 'a_3']) are␣
↪incompatible
```

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

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

Since the construction functors are actually used to construct infinite polynomial rings, the following result is no surprise:

```
sage: C.<a,b> = InfinitePolynomialRing(B); C
Infinite polynomial ring in a, b over Multivariate Polynomial Ring in x, y over␣
↪Rational Field
```

There is also an overlap in the next example:

```
sage: X.<w,x,y> = InfinitePolynomialRing(ZZ)
sage: Y.<x,y,z> = InfinitePolynomialRing(QQ)
```

X and Y have an overlapping generators x_*, y_*. Since the default lexicographic order is used in both rings, it gives rise to isomorphic sub-monoids in both X and Y. They are merged in the pushout, which also yields a common parent for doing arithmetic:

```
sage: P = sage.categories.pushout.pushout(Y,X); P
Infinite polynomial ring in w, x, y, z over Rational Field
sage: w[2]+z[3]
w_2 + z_3
sage: _.parent() is P
True
```

expand ( )

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

EXAMPLES:

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

merge (*other*)

Merge two construction functors of infinite polynomial rings, regardless of monomial order and implementation.

The purpose is to have a pushout (and thus, arithmetic) even in cases when the parents are isomorphic as rings, but not as ordered rings.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ,order='degrevlex')
sage: X.construction()
[InfPoly{[x,y], "lex", "sparse"}, Rational Field]
sage: Y.construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
sage: Y.construction()[0].merge(Y.construction()[0])
InfPoly{[x,y], "degrevlex", "dense"}
sage: y[3] + X(x[2])
x_2 + y_3
sage: _.parent().construction()
[InfPoly{[x,y], "degrevlex", "dense"}, Rational Field]
```

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

Bases: *sage.categories.pushout.ConstructionFunctor*

Construction functor for Laurent polynomial rings.

EXAMPLES:

```
sage: L.<t> = LaurentPolynomialRing(ZZ)
sage: F = L.construction()[0]
sage: F
LaurentPolynomialFunctor
sage: F(QQ)
Univariate Laurent Polynomial Ring in t over Rational Field
sage: K.<x> = LaurentPolynomialRing(ZZ)
sage: F(K)
Univariate Laurent Polynomial Ring in t over Univariate Laurent Polynomial Ring␣
↪in x over Integer Ring
sage: P.<x,y> = ZZ[]
sage: f = P.hom([x+2*y,3*x-y],P)
sage: F(f)
Ring endomorphism of Univariate Laurent Polynomial Ring in t over Multivariate␣
↪Polynomial Ring in x, y over Integer Ring
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer␣
↪Ring
          Defn: x |--> x + 2*y
                y |--> 3*x - y
sage: F(f)(x*F(P).gen()^-2+y*F(P).gen()^3)
(x + 2*y)*t^-2 + (3*x - y)*t^3
```

merge (*other*)

Two Laurent polynomial construction functors merge if the variable names coincide. The result is multivariate if one of the arguments is multivariate.

EXAMPLE:

```
sage: from sage.categories.pushout import LaurentPolynomialFunctor
sage: F1 = LaurentPolynomialFunctor('t')
```

```
sage: F2 = LaurentPolynomialFunctor('t', multi_variate=True)
sage: F1.merge(F2)
LaurentPolynomialFunctor
sage: F1.merge(F2)(LaurentPolynomialRing(GF(2),'a'))
Multivariate Laurent Polynomial Ring in a, t over Finite Field of size 2
sage: F1.merge(F1)(LaurentPolynomialRing(GF(2),'a'))
Univariate Laurent Polynomial Ring in t over Univariate Laurent Polynomial␣
↪Ring in a over Finite Field of size 2
```

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

Bases: *sage.categories.pushout.ConstructionFunctor*

A construction functor for matrices over rings.

EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,2, 3)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: MS = MatrixSpace(ZZ,2)
sage: F = MS.construction()[0]; F
MatrixFunctor
sage: P.<x,y> = QQ[]
sage: R = F(P); R
Full MatrixSpace of 2 by 2 dense matrices over Multivariate Polynomial Ring in x,␣
↪y over Rational Field
sage: f = P.hom([x+y,x-y],P); F(f)
Ring endomorphism of Full MatrixSpace of 2 by 2 dense matrices over Multivariate␣
↪Polynomial Ring in x, y over Rational Field
  Defn: Induced from base ring by
        Ring endomorphism of Multivariate Polynomial Ring in x, y over Rational␣
↪Field
          Defn: x |--> x + y
                y |--> x - y
sage: M = R([x,y,x*y,x+y])
sage: F(f)(M)
[    x + y     x - y]
[x^2 - y^2       2*x]
```

merge (*other*)

Merging is only happening if both functors are matrix functors of the same dimension. The result is sparse if and only if both given functors are sparse.

EXAMPLE:

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

```
sage: F3.merge(F3).is_sparse
True
```

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

Bases: *sage.categories.pushout.ConstructionFunctor*

A constructor for multivariate polynomial rings.

EXAMPLES:

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

expand ( )

Decompose `self` into a list of construction functors.

EXAMPLES:

```
sage: F = QQ['x,y,z,t'].construction()[0]; F
MPoly[x,y,z,t]
sage: F.expand()
[MPoly[t], MPoly[z], MPoly[y], MPoly[x]]
```

Now an actual use case:

```
sage: R.<x,y,z> = ZZ[]
sage: S.<z,t> = QQ[]
sage: x+t
x + t
sage: parent(x+t)
Multivariate Polynomial Ring in x, y, z, t over Rational Field
sage: T.<y,s> = QQ[]
sage: x + s
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Multivariate Polynomial␣
↪Ring in x, y, z over Integer Ring' and 'Multivariate Polynomial Ring in y,␣
↪s over Rational Field'
sage: R = PolynomialRing(ZZ, 'x', 500)
sage: S = PolynomialRing(GF(5), 'x', 200)
sage: R.gen(0) + S.gen(0)
2*x0
```

merge (*other*)

Merge `self` with another construction functor, or return None.

EXAMPLES:

```
sage: F = sage.categories.pushout.MultiPolynomialFunctor(['x','y'], None)
sage: G = sage.categories.pushout.MultiPolynomialFunctor(['t'], None)
sage: F.merge(G) is None
True
sage: F.merge(F)
MPoly[x,y]
```

class sage.categories.pushout. **MultivariateConstructionFunctor**

Bases: *sage.categories.pushout.ConstructionFunctor*

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

TESTS:

```
sage: from sage.categories.pushout import pushout
sage: A = cartesian_product((QQ['z'], QQ))
sage: B = cartesian_product((ZZ['t']['z'], QQ))
sage: pushout(A, B)
The Cartesian product of (Univariate Polynomial Ring in z over
Univariate Polynomial Ring in t over Rational Field,
Rational Field)
sage: A.construction()
(The cartesian_product functorial construction,
 (Univariate Polynomial Ring in z over Rational Field, Rational Field))
sage: pushout(A, B)
The Cartesian product of (Univariate Polynomial Ring in z over Univariate␣
↪Polynomial Ring in t over Rational Field, Rational Field)
```

common_base (*other_functor*, *self_bases*, *other_bases*)

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

INPUT:

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

OUTPUT:

A parent.

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

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

TESTS:

```
sage: from sage.categories.pushout import pushout
sage: pushout(cartesian_product([ZZ]), QQ)  # indirect doctest
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
```

```
The cartesian_product functorial construction(Integer Ring) and␣
↪FractionField(Integer Ring):
(Multivariate) functors are incompatible.
sage: pushout(cartesian_product([ZZ]), cartesian_product([ZZ, QQ]))  #␣
↪indirect doctest
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
The cartesian_product functorial construction(Integer Ring) and
The cartesian_product functorial construction(Integer Ring, Rational Field):
Functors need the same number of arguments.
```

class sage.categories.pushout. **PermutationGroupFunctor** (*gens*, *domain*)

Bases: *sage.categories.pushout.ConstructionFunctor*

EXAMPLES:

```
sage: from sage.categories.pushout import PermutationGroupFunctor
sage: PF = PermutationGroupFunctor([PermutationGroupElement([(1,2)])], [1,2]); PF
PermutationGroupFunctor[(1,2)]
```

gens ( )

EXAMPLES:

```
sage: P1 = PermutationGroup([[(1,2)]])
sage: PF, P = P1.construction()
sage: PF.gens()
[(1,2)]
```

merge (*other*)

Merge `self` with another construction functor, or return None.

EXAMPLES:

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

class sage.categories.pushout. **PolynomialFunctor** (*var*, *multi_variate=False*, *sparse=False*)

Bases: *sage.categories.pushout.ConstructionFunctor*

Construction functor for univariate polynomial rings.

EXAMPLE:

```
sage: P = ZZ['t'].construction()[0]
sage: P(GF(3))
Univariate Polynomial Ring in t over Finite Field of size 3
sage: P == loads(dumps(P))
True
sage: R.<x,y> = GF(5)[]
sage: f = R.hom([x+2*y,3*x-y],R)
sage: P(f)((x+y)*P(R).0)
(-x + y)*t
```

By trac ticket #9944, the construction functor distinguishes sparse and dense polynomial rings. Before, the following example failed:

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

merge (*other*)

Merge `self` with another construction functor, or return None.

NOTE:

Internally, the merging is delegated to the merging of multipolynomial construction functors. But in effect, this does the same as the default implementation, that returns `None` unless the to-be-merged functors coincide.

EXAMPLE:

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

class sage.categories.pushout. **QuotientFunctor** (*I*, *names=None*, *as_field=False*)

Bases: *sage.categories.pushout.ConstructionFunctor*

Construction functor for quotient rings.

NOTE:

The functor keeps track of variable names.

EXAMPLE:

```
sage: P.<x,y> = ZZ[]
sage: Q = P.quo([x^2+y^2]*P)
sage: F = Q.construction()[0]
sage: F(QQ['x','y'])
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal␣
↪(x^2 + y^2)
sage: F(QQ['x','y']) == QQ['x','y'].quo([x^2+y^2]*QQ['x','y'])
True
sage: F(QQ['x','y','z'])
Traceback (most recent call last):
...
CoercionException: Can not apply this quotient functor to Multivariate Polynomial␣
↪Ring in x, y, z over Rational Field
sage: F(QQ['y','z'])
```

```
Traceback (most recent call last):
...
TypeError: Could not find a mapping of the passed element to this ring.
```

merge (*other*)

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

EXAMPLE:

```
sage: P.<x> = QQ[]
sage: Q1 = P.quo([(x^2+1)^2*(x^2-3)])
sage: Q2 = P.quo([(x^2+1)^2*(x^5+3)])
sage: from sage.categories.pushout import pushout
sage: pushout(Q1,Q2)     # indirect doctest
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus␣
↪x^4 + 2*x^2 + 1
```

The following was fixed in trac ticket #8800:

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

class sage.categories.pushout. **SubspaceFunctor** (*basis*)

Bases: *sage.categories.pushout.ConstructionFunctor*

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

NOTE:

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

EXAMPLES:

```
sage: M = ZZ^3
sage: S = M.submodule([(1,2,3),(4,5,6)]); S
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: F = S.construction()[0]
sage: F(GF(2)^3)
Vector space of degree 3 and dimension 2 over Finite Field of size 2
User basis matrix:
[1 0 1]
[0 1 0]
```

merge (*other*)

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

EXAMPLE:

```
sage: M = GF(5)^3
sage: S1 = M.submodule([(1,2,3),(4,5,6)])
sage: S2 = M.submodule([(2,2,3)])
sage: F1 = S1.construction()[0]
sage: F2 = S2.construction()[0]
sage: F1.merge(F2)
SubspaceFunctor
sage: F1.merge(F2)(GF(5)^3) == S1+S2
```

```
True
sage: F1.merge(F2)(GF(5)['t']^3)
Free module of degree 3 and rank 3 over Univariate Polynomial Ring in t over␣
↪Finite Field of size 5
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

TEST:

```
sage: P.<t> = ZZ[]
sage: S1 = (ZZ^3).submodule([(1,2,3),(4,5,6)])
sage: S2 = (Frac(P)^3).submodule([(t,t^2,t^3+1),(4*t,0,1)])
sage: v = S1([0,3,6]) + S2([2,0,1/(2*t)]); v    # indirect doctest
(2, 3, (-12*t - 1)/(-2*t))
sage: v.parent()
Vector space of degree 3 and dimension 3 over Fraction Field of Univariate␣
↪Polynomial Ring in t over Integer Ring
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

class sage.categories.pushout.**VectorFunctor**(*n*, *is_sparse=False*, *inner_product_matrix=None*)

Bases: *sage.categories.pushout.ConstructionFunctor*

A construction functor for free modules over commutative rings.

EXAMPLE:

```
sage: F = (ZZ^3).construction()[0]
sage: F
VectorFunctor
sage: F(GF(2)['t'])
Ambient free module of rank 3 over the principal ideal domain Univariate␣
↪Polynomial Ring in t over Finite Field of size 2 (using NTL)
```

merge(*other*)

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

EXAMPLE:

Two modules without explicitly given inner product allow coercion:

```
sage: M1 = QQ^3
sage: P.<t> = ZZ[]
sage: M2 = FreeModule(P,3)
sage: M1([1,1/2,1/3]) + M2([t,t^2+t,3])      # indirect doctest
(t + 1, t^2 + t + 1/2, 10/3)
```

If only one summand has an explicit inner product, the result will be provided with it:

```
sage: M3 = FreeModule(P,3, inner_product_matrix = Matrix(3,3,range(9)))
sage: M1([1,1/2,1/3]) + M3([t,t^2+t,3])
(t + 1, t^2 + t + 1/2, 10/3)
sage: (M1([1,1/2,1/3]) + M3([t,t^2+t,3])).parent().inner_product_matrix()
```

```
[0 1 2]
[3 4 5]
[6 7 8]
```

If both summands have an explicit inner product (even if it is the standard inner product), then the products must coincide. The only difference between `M1` and `M4` in the following example is the fact that the default inner product was *explicitly* requested for `M4`. It is therefore not possible to coerce with a different inner product:

```
sage: M4 = FreeModule(QQ,3, inner_product_matrix = Matrix(3,3,1))
sage: M4 == M1
True
sage: M4.inner_product_matrix() == M1.inner_product_matrix()
True
sage: M4([1,1/2,1/3]) + M3([t,t^2+t,3])      # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+': 'Ambient quadratic space of␣
↪dimension 3 over Rational Field
Inner product matrix:
[1 0 0]
[0 1 0]
[0 0 1]' and 'Ambient free quadratic module of rank 3 over the integral␣
↪domain Univariate Polynomial Ring in t over Integer Ring
Inner product matrix:
[0 1 2]
[3 4 5]
[6 7 8]'
```

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

An auxiliary function that is used in *pushout()* and *pushout_lattice()*.

INPUT:

An object

OUTPUT:

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

EXAMPLE:

```
sage: from sage.categories.pushout import construction_tower
sage: construction_tower(MatrixSpace(FractionField(QQ['t']),2))
[(None, Full MatrixSpace of 2 by 2 dense matrices over Fraction Field of␣
↪Univariate Polynomial Ring in t over Rational Field), (MatrixFunctor, Fraction␣
↪Field of Univariate Polynomial Ring in t over Rational Field), (FractionField,␣
↪Univariate Polynomial Ring in t over Rational Field), (Poly[t], Rational␣
↪Field), (FractionField, Integer Ring)]
```

sage.categories.pushout. **expand_tower** (*tower*)

An auxiliary function that is used in *pushout()*.

INPUT:

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

OUTPUT:

A new construction tower with all the construction functors expanded.

EXAMPLE:

```
sage: from sage.categories.pushout import construction_tower, expand_tower
sage: construction_tower(QQ['x,y,z'])
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[x,y,z], Rational Field),
 (FractionField, Integer Ring)]
sage: expand_tower(construction_tower(QQ['x,y,z']))
[(None, Multivariate Polynomial Ring in x, y, z over Rational Field),
 (MPoly[z], Univariate Polynomial Ring in y over Univariate Polynomial Ring in x␣
↪over Rational Field),
 (MPoly[y], Univariate Polynomial Ring in x over Rational Field),
 (MPoly[x], Rational Field),
 (FractionField, Integer Ring)]
```

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

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

ALGORITHM:

This incorporates the idea of functors discussed at Sage Days 4. Every object R can be viewed as an initial object and a series of functors (e.g. polynomial, quotient, extension, completion, vector/matrix, etc.). Call the series of increasingly simple objects (with the associated functors) the "tower" of R. The construction method is used to create the tower.

Given two objects R and S, try to find a common initial object Z. If the towers of R and S meet, let Z be their join. Otherwise, see if the top of one coerces naturally into the other.

Now we have an initial object and two ordered lists of functors to apply. We wish to merge these in an unambiguous order, popping elements off the top of one or the other tower as we apply them to Z.

- If the functors are of distinct types, there is an absolute ordering given by the rank attribute. Use this.
- Otherwise:
  - If the tops are equal, we (try to) merge them.
  - If exactly one occurs lower in the other tower, we may unambiguously apply the other (hoping for a later merge).
  - If the tops commute, we can apply either first.
  - Otherwise fail due to ambiguity.

The algorithm assumes by default that when a construction F is applied to an object X, the object $F(X)$ admits a coercion map from X. However, the algorithm can also handle the case where $F(X)$ has a coercion map *to* X instead. In this case, the attribute `coercion_reversed` of the class implementing F should be set to `True` .

EXAMPLES:

Here our "towers" are $R = Complete_7(Frac(\mathbf{Z}))$ and $Frac(Poly_x(\mathbf{Z}))$, which give us $Frac(Poly_x(Complete_7(Frac(\mathbf{Z}))))$:

```
sage: from sage.categories.pushout import pushout
sage: pushout(Qp(7), Frac(ZZ['x']))
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped␣
↪relative precision 20
```

Note we get the same thing with

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

Note that polynomial variable ordering must be unambiguously determined.

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

Some other examples:

```
sage: pushout(Zp(7)['y'], Frac(QQ['t'])['x,y,z'])
Multivariate Polynomial Ring in x, y, z over Fraction Field of Univariate␣
↪Polynomial Ring in t over 7-adic Field with capped relative precision 20
sage: pushout(ZZ['x,y,z'], Frac(ZZ['x'])['y'])
Multivariate Polynomial Ring in y, z over Fraction Field of Univariate Polynomial␣
↪Ring in x over Integer Ring
sage: pushout(MatrixSpace(RDF, 2, 2), Frac(ZZ['x']))
Full MatrixSpace of 2 by 2 dense matrices over Fraction Field of Univariate␣
↪Polynomial Ring in x over Real Double Field
sage: pushout(ZZ, MatrixSpace(ZZ[['x']], 3, 3))
Full MatrixSpace of 3 by 3 dense matrices over Power Series Ring in x over␣
↪Integer Ring
sage: pushout(QQ['x,y'], ZZ[['x']])
Univariate Polynomial Ring in y over Power Series Ring in x over Rational Field
sage: pushout(Frac(ZZ['x']), QQ[['x']])
Laurent Series Ring in x over Rational Field
```

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

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

sage: V = (QQ^3).span([[1, 2, 3/4]])
sage: A = ZZ^3
sage: pushout(A, V)
Vector space of dimension 3 over Rational Field
sage: B = A.span([[0, 0, 2/3]])
sage: pushout(B, V)
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 0]
[0 0 1]
```

Some more tests with `coercion_reversed = True`:

```
sage: from sage.categories.pushout import ConstructionFunctor
sage: class EvenPolynomialRing(type(QQ['x'])):
....:     def __init__(self, base, var):
....:         super(EvenPolynomialRing, self).__init__(base, var)
....:         self.register_embedding(base[var])
....:     def __repr__(self):
....:         return "Even Power " + super(EvenPolynomialRing, self).__repr__()
....:     def construction(self):
....:         return EvenPolynomialFunctor(), self.base()[self.variable_name()]
....:     def _coerce_map_from_(self, R):
....:         return self.base().has_coerce_map_from(R)
....:
sage: class EvenPolynomialFunctor(ConstructionFunctor):
....:     rank = 10
....:     coercion_reversed = True
....:     def __init__(self):
....:         ConstructionFunctor.__init__(self, Rings(), Rings())
....:     def _apply_functor(self, R):
....:         return EvenPolynomialRing(R.base(), R.variable_name())
....:
sage: pushout(EvenPolynomialRing(QQ, 'x'), ZZ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), QQ)
Even Power Univariate Polynomial Ring in x over Rational Field
sage: pushout(EvenPolynomialRing(QQ, 'x'), RR)
Even Power Univariate Polynomial Ring in x over Real Field with 53 bits of␣
↪precision

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

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

sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR^2)
Ambient free module of rank 2 over the principal ideal domain Even Power␣
↪Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: pushout(EvenPolynomialRing(QQ, 'x')^2, RR['x']^2)
Ambient free module of rank 2 over the principal ideal domain Univariate␣
↪Polynomial Ring in x over Real Field with 53 bits of precision
```

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

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

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

```
sage: class GPolynomialRing(Parent):
....:     def __init__(self, coefficients, var, exponents):
....:         self.coefficients = coefficients
....:         self.var = var
....:         self.exponents = exponents
....:         super(GPolynomialRing, self).__init__(category=Rings())
....:     def _repr_(self):
....:         return 'Generalized Polynomial Ring in %s^(%s) over %s' % (
....:                self.var, self.exponents, self.coefficients)
....:     def construction(self):
....:         return GPolynomialFunctor(self.var, self.exponents), self.
↪coefficients
....:     def _coerce_map_from_(self, R):
....:         return self.coefficients.has_coerce_map_from(R)
```

and

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

We can construct a parent now in two different ways:

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

Since the construction

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

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

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

```
  over Multivariate Polynomial Ring in a, b, c over Rational Field
sage: pushout(GP_ZZ(ZZ['a,b']), GP_ZZ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
```

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

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

```
sage: pushout(GP_ZZ(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZ(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Rational Field) over Rational Field
sage: pushout(GP_ZZt(QQ), GP_QQ(ZZ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
sage: pushout(GP_ZZt(ZZ), GP_QQ(QQ))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t
  over Rational Field) over Rational Field
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['c,d']))
Traceback (most recent call last):
...
CoercionException: ('Ambiguous Base Extension', ...)
sage: pushout(GP_ZZt(ZZ['a,b']), GP_QQ(ZZ['b,c']))
Generalized Polynomial Ring in X^(Univariate Polynomial Ring in t over Rational␣
↪Field)
  over Multivariate Polynomial Ring in a, b, c over Integer Ring
```

Some tests with Cartesian products:

```
sage: from sage.sets.cartesian_product import CartesianProduct
sage: A = CartesianProduct((ZZ['x'], QQ['y'], QQ['z']), Sets().
↪CartesianProducts())
sage: B = CartesianProduct((ZZ['x'], ZZ['y'], ZZ['t']['z']), Sets().
↪CartesianProducts())
sage: A.construction()
(The cartesian_product functorial construction,
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over Rational Field))
sage: pushout(A, B)
The Cartesian product of
 (Univariate Polynomial Ring in x over Integer Ring,
  Univariate Polynomial Ring in y over Rational Field,
  Univariate Polynomial Ring in z over Univariate Polynomial Ring in t over␣
↪Rational Field)
sage: pushout(ZZ, cartesian_product([ZZ, QQ]))
Traceback (most recent call last):
...
```

```
CoercionException: 'NoneType' object is not iterable
```

```
sage: from sage.categories.pushout import PolynomialFunctor
sage: from sage.sets.cartesian_product import CartesianProduct
sage: class CartesianProductPoly(CartesianProduct):
....:     def __init__(self, polynomial_rings):
....:         sort = sorted(polynomial_rings, key=lambda P: P.variable_name())
....:         super(CartesianProductPoly, self).__init__(sort, Sets().
↪CartesianProducts())
....:     def vars(self):
....:         return tuple(P.variable_name() for P in self.cartesian_factors())
....:     def _pushout_(self, other):
....:         if isinstance(other, CartesianProductPoly):
....:             s_vars = self.vars()
....:             o_vars = other.vars()
....:             if s_vars == o_vars:
....:                 return
....:             return pushout(CartesianProductPoly(
....:                     self.cartesian_factors() +
....:                     tuple(f for f in other.cartesian_factors()
....:                           if f.variable_name() not in s_vars)),
....:                 CartesianProductPoly(
....:                     other.cartesian_factors() +
....:                     tuple(f for f in self.cartesian_factors()
....:                           if f.variable_name() not in o_vars)))
....:         C = other.construction()
....:         if C is None:
....:             return
....:         elif isinstance(C[0], PolynomialFunctor):
....:             return pushout(self, CartesianProductPoly((other,)))
```

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

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

```
    Multivariate Polynomial Ring in a, b over Integer Ring,
  Univariate Polynomial Ring in y over
    Multivariate Polynomial Ring in b, c over Rational Field)
```

```
sage: pushout(CartesianProductPoly((ZZ['x'],)), ZZ)
Traceback (most recent call last):
...
CoercionException: No common base ("join") found for
The cartesian_product functorial construction(...) and None(Integer Ring):
(Multivariate) functors are incompatible.
```

AUTHORS:

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

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

Given a pair of objects R and S, try to construct a reasonable object Y and return maps such that canonically $R \leftarrow Y \rightarrow S$.

ALGORITHM:

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

See the code for a specific worked-out example.

EXAMPLES:

```
sage: from sage.categories.pushout import pushout_lattice
sage: A, B = pushout_lattice(Qp(7), Frac(ZZ['x']))
sage: A.codomain()
Fraction Field of Univariate Polynomial Ring in x over 7-adic Field with capped␣
↪relative precision 20
sage: A.codomain() is B.codomain()
True
sage: A, B = pushout_lattice(ZZ, MatrixSpace(ZZ[['x']], 3, 3))
sage: B
Identity endomorphism of Full MatrixSpace of 3 by 3 dense matrices over Power␣
↪Series Ring in x over Integer Ring
```

AUTHOR:

- Robert Bradshaw

sage.categories.pushout. **type_to_parent** (P)

An auxiliary function that is used in *pushout()* .

INPUT:

A type

OUTPUT:

A Sage parent structure corresponding to the given type

TEST:

```
sage: from sage.categories.pushout import type_to_parent
sage: type_to_parent(int)
Integer Ring
sage: type_to_parent(float)
Real Double Field
sage: type_to_parent(complex)
Complex Double Field
sage: type_to_parent(list)
Traceback (most recent call last):
...
TypeError: Not a scalar type.
```

5.5 Group, ring, etc. actions on objects.

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

A group action $G \times S \to S$ is a functor from G to Sets.

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

```
sage: from sage.categories.action import Action
sage: class P: pass
sage: A = Action(P(),P())
sage: import gc
sage: _ = gc.collect()
sage: A
<repr(<sage.categories.action.Action at 0x...>) failed: RuntimeError: This action␣
↪acted on a set that became garbage collected>
```

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

```
sage: _ = gc.collect()
sage: from sage.categories.action import Action
sage: class P: pass
sage: q = P()
sage: A = Action(P(),q)
sage: gc.collect()
0
sage: A
Left action by <__main__.P instance at ...> on <__main__.P instance at ...>
```

AUTHOR:

- Robert Bradshaw: initial version

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

Bases: sage.categories.functor.Functor

act (*g*, *a*)
This is a consistent interface for acting on a by g, regardless of whether it's a left or right action.

actor ( )

codomain ( )

domain ( )

is_left ( )

left_domain ( )

operation ( )

right_domain ( )

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

The endomorphism defined by the action of one element.

EXAMPLES:

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

class sage.categories.action. **InverseAction**
Bases: *sage.categories.action.Action*

An action that acts as the inverse of the given action.

TESTS:

This illustrates a shortcoming in the current coercion model. See the comments in _call_ below:

```
sage: x = polygen(QQ,'x')
sage: a = 2*x^2+2; a
2*x^2 + 2
sage: a / 2
x^2 + 1
sage: a /= 2
sage: a
x^2 + 1
```

codomain ( )

class sage.categories.action. **PrecomposedAction**
Bases: *sage.categories.action.Action*

A precomposed action first applies given maps, and then applying an action to the return values of the maps.

EXAMPLES:

We demonstrate that an example discussed on trac ticket #14711 did not become a problem:

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

codomain ( )

domain ( )

5.6 Containers for storing coercion data

This module provides *TripleDict* and *MonoDict*. These are structures similar to WeakKeyDictionary in Python's weakref module, and are optimized for lookup speed. The keys for *TripleDict* consist of triples (k1,k2,k3) and are looked up by identity rather than equality. The keys are stored by weakrefs if possible. If any one of the components k1, k2, k3 gets garbage collected, then the entry is removed from the *TripleDict*.

Key components that do not allow for weakrefs are stored via a normal refcounted reference. That means that any entry stored using a triple (k1,k2,k3) so that none of the k1,k2,k3 allows a weak reference behaves as an entry in a normal dictionary: Its existence in *TripleDict* prevents it from being garbage collected.

That container currently is used to store coercion and conversion maps between two parents (trac ticket #715) and to store homsets of pairs of objects of a category (trac ticket #11521). In both cases, it is essential that the parent structures remain garbage collectable, it is essential that the data access is faster than with a usual WeakKeyDictionary, and we enforce the "unique parent condition" in Sage (parent structures should be identical if they are equal).

MonoDict behaves similarly, but it takes a single item as a key. It is used for caching the parents which allow a coercion map into a fixed other parent (trac ticket #12313).

By trac ticket #14159, *MonoDict* and *TripleDict* can be optionally used with weak references on the values.

class sage.structure.coerce_dict. **MonoDict**

Bases: object

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python WeakKeyDictionary in the following important ways:

- Comparison is done using the 'is' rather than '==' operator.
- Only weak references to the keys are stored if at all possible. Keys that do not allow for weak references are stored with a normal refcounted reference.
- The callback of the weak references is safe against recursion, see below.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

IMPLEMENTATION:

It is implemented as a hash table with open addressing, similar to python's dict.

If ki supports weak references then ri is a weak reference to ki with a callback to remove the entry from the dictionary if ki gets garbage collected. If ki is does not support weak references then ri is identical to ki. In the latter case the presence of the key in the dictionary prevents it from being garbage collected.

INPUT:

- `size` – unused parameter, present for backward compatibility.
- `data` – optional iterable defining initial data.
- `threshold` – unused parameter, present for backward compatibility.
- `weak_values` – optional bool (default False). If it is true, weak references to the values in this dictionary will be used, when possible.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict()
sage: a = 'a'; b = 'ab'; c = '-15'
sage: L[a] = 1
sage: L[b] = 2
sage: L[c] = 3
```

The key is expected to be a unique object. Hence, the item stored for `c` can not be obtained by providing another equal string:

```
sage: L[a]
1
sage: L[b]
2
sage: L[c]
3
sage: L['-15']
Traceback (most recent call last):
...
KeyError: '-15'
```

Not all features of Python dictionaries are available, but iteration over the dictionary items is possible:

```
sage: # for some reason the following failed in "make ptest"
sage: # on some installations, see #12313 for details
sage: sorted(L.iteritems()) # random layout
[('-15', 3), ('a', 1), ('ab', 2)]
sage: # the following seems to be more consistent
sage: set(L.iteritems())
{('-15', 3), ('a', 1), ('ab', 2)}
sage: del L[c]
sage: sorted(L.iteritems())
[('a', 1), ('ab', 2)]
sage: len(L)
2
sage: for i in range(1000):
...       L[i] = i
sage: len(L)
1002
```

```
sage: L['a']
1
sage: L['c']
Traceback (most recent call last):
...
KeyError: 'c'
```

Note that this kind of dictionary is also used for caching actions and coerce maps. In previous versions of Sage, the cache was by strong references and resulted in a memory leak in the following example. However, this leak was fixed by trac ticket #715, using weak references:

```
sage: K = GF(1<<55,'t')
sage: for i in range(50):
...       a = K.random_element()
...       E = EllipticCurve(j=a)
...       P = E.random_point()
...       Q = 2*P
sage: import gc
sage: n = gc.collect()
sage: from sage.schemes.elliptic_curves.ell_finite_field import␣
↪EllipticCurve_finite_field
sage: LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_
↪finite_field)]
sage: len(LE)     # indirect doctest
1
```

TESTS:

Here, we demonstrate the use of weak values.

```
sage: M = MonoDict(13)
sage: MW = MonoDict(13, weak_values=True)
sage: class Foo: pass
sage: a = Foo()
sage: b = Foo()
sage: k = 1
sage: M[k] = a
sage: MW[k] = b
sage: M[k] is a
True
sage: MW[k] is b
True
sage: k in M
True
sage: k in MW
True
```

While `M` uses a strong reference to `a`, `MW` uses a *weak* reference to `b`, and after deleting `b`, the corresponding item of `MW` will be removed during the next garbage collection:

```
sage: import gc
sage: del a,b
sage: _ = gc.collect()
sage: k in M
True
sage: k in MW
False
sage: len(MW)
```

```
0
sage: len(M)
1
```

Note that `MW` also accepts values that do not allow for weak references:

```
    sage: MW[k] = int(5)
    sage: MW[k]
    5

The following demonstrates that :class:`MonoDict` is safer than
:class:`~weakref.WeakKeyDictionary` against recursions created by nested
callbacks; compare :trac:`15069` (the mechanism used now is different, though)::

    sage: M = MonoDict(11)
    sage: class A: pass
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000
    sage: del a
    sage: len(M)
    0

The corresponding example with a Python :class:`weakref.WeakKeyDictionary`
would result in a too deep recursion during deletion of the dictionary
items::

    sage: import weakref
    sage: M = weakref.WeakKeyDictionary()
    sage: a = A()
    sage: prev = a
    sage: for i in range(1000):
    ....:     newA = A()
    ....:     M[prev] = newA
    ....:     prev = newA
    sage: len(M)
    1000
    sage: del a
    Exception RuntimeError: 'maximum recursion depth exceeded while calling a␣
 ↪Python object' in <function remove at ...> ignored
    sage: len(M)>0
    True

Check that also in the presence of circular references, :class:`MonoDict`
gets properly collected::

    sage: import gc
    sage: def count_type(T):
    ....:     return len([c for c in gc.get_objects() if isinstance(c,T)])
    sage: _=gc.collect()
    sage: N=count_type(MonoDict)
    sage: for i in range(100):
    ....:     V = [ MonoDict(11,{"id":j+100*i}) for j in range(100)]
```

```
    ....:     n= len(V)
    ....:     for i in range(n): V[i][V[(i+1)%n]]=(i+1)%n
    ....:     del V
    ....:     _=gc.collect()
    ....:     assert count_type(MonoDict) == N
    sage: count_type(MonoDict) == N
    True

AUTHORS:

- Simon King (2012-01)
- Nils Bruin (2012-08)
- Simon King (2013-02)
- Nils Bruin (2013-11)
```

iteritems ( )

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: L = MonoDict(31)
sage: L[1] = None
sage: L[2] = True
sage: list(sorted(L.iteritems()))
[(1, None), (2, True)]
```

class sage.structure.coerce_dict. **MonoDictEraser**

Bases: object

Erase items from a *MonoDict* when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import MonoDict
sage: class A: pass
sage: a = A()
sage: M = MonoDict()
sage: M[a] = 1
sage: len(M)
1
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(M)     # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

class sage.structure.coerce_dict. **TripleDict**

Bases: object

This is a hashtable specifically designed for (read) speed in the coercion model.

It differs from a python dict in the following important ways:

- All keys must be sequence of exactly three elements. All sequence types (tuple, list, etc.) map to the same item.
- Comparison is done using the 'is' rather than '==' operator.

There are special cdef set/get methods for faster access. It is bare-bones in the sense that not all dictionary methods are implemented.

It is implemented as a list of lists (hereafter called buckets). The bucket is chosen according to a very simple hash based on the object pointer, and each bucket is of the form [id(k1), id(k2), id(k3), r1, r2, r3, value, id(k1), id(k2), id(k3), r1, r2, r3, value, ...], on which a linear search is performed. If a key component ki supports weak references then ri is a weak reference to ki; otherwise ri is identical to ki.

INPUT:

- `size` – an integer, the initial number of buckets. To spread objects evenly, the size should ideally be a prime, and certainly not divisible by 2.
- `data` – optional iterable defining initial data.
- `threshold` – optional number, default 0.7. It determines how frequently the dictionary will be resized (large threshold implies rare resizing).
- `weak_values` – optional bool (default False). If it is true, weak references to the values in this dictionary will be used, when possible.

If any of the key components k1,k2,k3 (this can happen for a key component that supports weak references) gets garbage collected then the entire entry disappears. In that sense this structure behaves like a nested `WeakKeyDictionary`.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict()
sage: a = 'a'; b = 'b'; c = 'c'
sage: L[a,b,c] = 1
sage: L[a,b,c]
1
sage: L[c,b,a] = -1
sage: list(L.iteritems())      # random order of output.
[(('c', 'b', 'a'), -1), (('a', 'b', 'c'), 1)]
sage: del L[a,b,c]
sage: list(L.iteritems())
[(('c', 'b', 'a'), -1)]
sage: len(L)
1
sage: for i in range(1000):
...       L[i,i,i] = i
sage: len(L)
1001
sage: L = TripleDict(L)
sage: L[c,b,a]
-1
sage: L[a,b,c]
Traceback (most recent call last):
...
KeyError: ('a', 'b', 'c')
sage: L[a]
Traceback (most recent call last):
...
KeyError: 'a'
```

```
sage: L[a] = 1
Traceback (most recent call last):
...
KeyError: 'a'
```

Note that this kind of dictionary is also used for caching actions and coerce maps. In previous versions of Sage, the cache was by strong references and resulted in a memory leak in the following example. However, this leak was fixed by trac ticket #715, using weak references:

```
sage: K = GF(1<<55,'t')
sage: for i in range(50):
...       a = K.random_element()
...       E = EllipticCurve(j=a)
...       P = E.random_point()
...       Q = 2*P
sage: import gc
sage: n = gc.collect()
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_
↪finite_field
sage: LE = [x for x in gc.get_objects() if isinstance(x, EllipticCurve_finite_
↪field)]
sage: len(LE)     # indirect doctest
1
```

TESTS:

Here, we demonstrate the use of weak values.

```
sage: class Foo: pass
sage: T = TripleDict(13)
sage: TW = TripleDict(13, weak_values=True)
sage: a = Foo()
sage: b = Foo()
sage: k = 1
sage: T[a,k,k]=1
sage: T[k,a,k]=2
sage: T[k,k,a]=3
sage: T[k,k,k]=a
sage: TW[b,k,k]=1
sage: TW[k,b,k]=2
sage: TW[k,k,b]=3
sage: TW[k,k,k]=b
sage: len(T)
4
sage: len(TW)
4
sage: (k,k,k) in T
True
sage: (k,k,k) in TW
True
sage: T[k,k,k] is a
True
sage: TW[k,k,k] is b
True
```

Now, `T` holds a strong reference to `a`, namely in `T[k,k,k]`. Hence, when we delete `a`, *all* items of `T` survive:

```
sage: del a
sage: _ = gc.collect()
sage: len(T)
4
```

Only when we remove the strong reference, the items become collectable:

```
sage: del T[k,k,k]
sage: _ = gc.collect()
sage: len(T)
0
```

The situation is different for `TW` , since it only holds *weak* references to `a` . Therefore, all items become collectable after deleting `a` :

```
sage: del b
sage: _ = gc.collect()
sage: len(TW)
0
```

Note: The index h corresponding to the key [k1, k2, k3] is computed as a value of unsigned type size_t as follows:

$$h = id(k1) + 13 * id(k2) xor 503 id(k3)$$

The natural type for this quantity is Py_ssize_t, which is a signed quantity with the same length as size_t. Storing it in a signed way gives the most efficient storage into PyInt, while preserving sign information.

In previous situations there were some problems with ending up with negative indices, which required casting to an unsigned type, i.e., (<size_t> h)% N since C has a sign-preserving % operation This caused problems on 32 bits systems, see trac ticket #715 for details. This is irrelevant for the current implementation.

AUTHORS:

- Robert Bradshaw, 2007-08
- Simon King, 2012-01
- Nils Bruin, 2012-08
- Simon King, 2013-02
- Nils Bruin, 2013-11

iteritems ( )

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: L = TripleDict(31)
sage: L[1,2,3] = None
sage: list(L.iteritems())
[((1, 2, 3), None)]
```

class sage.structure.coerce_dict. **TripleDictEraser**

Bases: object

Erases items from a *TripleDict* when a weak reference becomes invalid.

This is of internal use only. Instances of this class will be passed as a callback function when creating a weak reference.

EXAMPLES:

```
sage: from sage.structure.coerce_dict import TripleDict
sage: class A: pass
sage: a = A()
sage: T = TripleDict()
sage: T[a,ZZ,None] = 1
sage: T[ZZ,a,1] = 2
sage: T[a,a,ZZ] = 3
sage: len(T)
3
sage: del a
sage: import gc
sage: n = gc.collect()
sage: len(T) # indirect doctest
0
```

AUTHOR:

- Simon King (2012-01)
- Nils Bruin (2013-11)

5.7 Exceptions raised by the coercion model

exception sage.structure.coerce_exceptions. **CoercionException**

Bases: exceptions.TypeError

This is the baseclass of exceptions that the coercion model raises when trying to discover coercions. We don't use standard Python exceptions to avoid inadvertently catching and suppressing real errors.

Usually one raises this to indicate the attempted action isn't implemented/appropriate, but if there are other things to try not to immediately abort to the user.

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

C

S

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

V