Sage Reference Manual: Coercion

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

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 5
```

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(1.50000000000000000000000000000000000)
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
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 - 0.00703
0.997997869801216 + 0.00140712894524925*z - 0.000498127610960097*z^2 + 0.000118835596665956*z^3 - 0.0
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
```

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
```

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 Fielc
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 R
```

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
```

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: steps, res = cm.analyse(GF(7), ZZ)
sage: print steps
['Coercion on right operand via', Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 7, 'Arithmetic performed after coercions.']
sage: print 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))
```

coercion_maps (*R*, *S*)

Give two parents R and S, return a pair of coercion maps $f : R \rightarrow Z$ and $g : S \rightarrow Z$, 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: f, g = cm.coercion_maps(ZZ['x'], 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: cm.coercion_maps(QQ, GF(7)) 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)
Call morphism:
  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)
Call morphism:
  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
```

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: cm.common_parent(ZZ[['T']], QQ['T'], 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: cm.common_parent(*[RealField(prec) for prec in [10,20..100]])
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: cm.common_parent(QQ['x,y'], QQ['y,z']) == cm.common_parent(QQ['y,z'], QQ['x,y'])
True
sage: cm.common_parent(QQ['x,y'], QQ['y,z'], QQ['z,t'])
Multivariate Polynomial Ring in x, y, z, t over Rational Field
sage: cm.common_parent(QQ['x,y'], QQ['z,t'], QQ['y,z'])
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Multivariate Polynomial Rin
```

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
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 R
```

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
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
```

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
```

Bug trac ticket #17740:

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

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: cm.discover_coercion(ZZ['x,y'], 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: cm.division_parent(ZZ['x'])
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: cm.division_parent(GF(41))
Finite Field of size 41
sage: cm.division_parent(Integers(100))
Ring of integers modulo 100
sage: cm.division_parent(SymmetricGroup(5))
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
```

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
```

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
```

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: cm.explain(ZZ['x'], QQ)
Action discovered.
    Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over In
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: cm.explain(RealField(100), float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(RealField(100)(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: cm.explain(ZZ['x'], QQ['x'], 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(ZZ['x'], ZZ, operator.div)
Action discovered.
    Right inverse action by Rational Field on Univariate Polynomial Ring in x over Integer R
    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: cm.get_action(ZZ['x'], ZZ, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
sage: cm.get_action(ZZ['x'], ZZ, operator.imul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
sage: cm.get_action(ZZ['x'], QQ, operator.mul)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Intege
sage: cm.get_action(QQ['x'], int, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Rational
with precomposition on right by Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring

sage: R.<x> = QQ['x']
sage: A = cm.get_action(R, ZZ, operator.div); A
Right inverse action by Rational Field on Univariate Polynomial Ring in x over Rational Fiel
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
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: 1 + 1/2
3/2
sage: cm = sage.structure.element.get_coercion_model()
sage: maps, actions = cm.get_cache()
```

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

```
sage: left_morphism, right_morphism = maps[ZZ, QQ]
sage: print copy(left_morphism)
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print right_morphism
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
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:
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()
({}, {})
```

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 Intege
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
```

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.**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.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()
```

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:

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

sage: sage.structure.coerce_actions.GenericAction(QQ, Zmod(6), 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, Zmod(6), 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:

```
sage: A = sage.structure.coerce_actions.ActedUponAction(MatrixSpace(ZZ, 2), Cusps, True)
sage: A.codomain()
Set P^1(QQ) of all cusps
sage: A = sage.structure.coerce_actions.ActOnAction(SymmetricGroup(3), QQ['x,y,z'], 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.

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`

class `sage.structure.coerce_actions.`**`LeftModuleAction`**

Bases: `sage.structure.coerce_actions.ModuleAction`

This creates an action of an element of a module by an element of its base ring. The simplest example to keep in mind is R acting on the polynomial ring R[x].

EXAMPLES:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: LeftModuleAction(ZZ, ZZ['x'])
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer Ring
sage: LeftModuleAction(ZZ, QQ['x'])
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Rational Fiel
sage: LeftModuleAction(QQ, ZZ['x'])
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Rin
sage: LeftModuleAction(QQ, ZZ['x']['y'])
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in y over Univariate
```

The following tests against a problem that was relevant during work on trac ticket #9944:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(ZZ, sparse=True)
sage: 1/R.0
1/x
sage: 1/S.0
1/x
```

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:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: A = LeftModuleAction(QQ, ZZ['x,y,z'])
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:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: A = LeftModuleAction(QQ, ZZ['x,y,z'])
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

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

Bases: `sage.structure.coerce_actions.ModuleAction`

This creates an action of an element of a module by an element of its base ring. The simplest example to keep in mind is R acting on the polynomial ring R[x].

EXAMPLES:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: LeftModuleAction(ZZ, ZZ['x'])
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer Ring
sage: LeftModuleAction(ZZ, QQ['x'])
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Rational Fiel
```

```
sage: LeftModuleAction(QQ, ZZ['x'])
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Rin
sage: LeftModuleAction(QQ, ZZ['x']['y'])
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in y over Univariate
```

The following tests against a problem that was relevant during work on trac ticket #9944:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(ZZ, sparse=True)
sage: 1/R.0
1/x
sage: 1/S.0
1/x
```

is_inplace

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:

```
sage: from sage.structure.coerce_actions import detect_element_action
sage: detect_element_action(ZZ['x'], ZZ, False)
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer Ring
sage: detect_element_action(ZZ['x'], QQ, True)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer Ri
sage: detect_element_action(Cusps, MatrixSpace(ZZ, 2), False)
Left action by Full MatrixSpace of 2 by 2 dense matrices over Integer Ring on Set P^1(QQ) of all
sage: detect_element_action(Cusps, MatrixSpace(ZZ, 2), 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
```

CHAPTER

SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

S

A

B

C

D

E

G

I

L

M

N

P

R

S

T

V