Parents, Conversion and Coercion

This section may seem more technical than the previous, but we believe that it is important to understand the meaning of parents and coercion in order to use rings and other algebraic structures in Sage effectively and efficiently.

Note that we try to explain notions, but we do not show here how to implement them. An implementation-oriented tutorial is available as a Sage thematic tutorial.

Elements

If one wants to implement a ring in Python, a first approximation is to create a class for the elements `X` of that ring and provide it with the required double underscore methods such as `__add__`, `__sub__`, `__mul__`, of course making sure that the ring axioms hold.

As Python is a strongly typed (yet dynamically typed) language, one might, at least at first, expect that one implements one Python class for each ring. After all, Python contains one type `<int>` for the integers, one type `<float>` for the reals, and so on. But that approach must soon fail: There are infinitely many rings, and one can not implement infinitely many classes.

Instead, one may create a hierarchy of classes designed to implement elements of ubiquitous algebraic structures, such as groups, rings, skew fields, commutative rings, fields, algebras, and so on.

But that means that elements of fairly different rings can have the same type.

```
sage: P.<x,y> = GF(3)[]
sage: Q.<a,b> = GF(4,'z')[]
sage: type(x)==type(a)
True
```

On the other hand, one could also have different Python classes providing different implementations of the same mathematical structure (e.g., dense matrices versus sparse matrices)

```
sage: P.<a> = PolynomialRing(ZZ)
sage: Q.<b> = PolynomialRing(ZZ, sparse=True)
sage: R.<c> = PolynomialRing(ZZ, implementation='NTL')
sage: type(a); type(b); type(c)
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense
<class 'sage.rings.polynomial.polynomial_element_generic.PolynomialRing_integral_do
<type 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_
```

That poses two problems: On the one hand, if one has elements that are two instances of the same class, then one may expect that their `__add__` method will allow to add them; but one does not want that, if the elements belong to very different rings. On the other hand, if one has elements belonging to different implementations of the same ring, then one wants to add them, but that is not straight forward if they belong to different Python classes.

The solution to these problems is called "coercion" and will be explained below.

However, it is essential that each element knows what it is element of. That is available by the method `parent()`:

```
sage: a.parent(); b.parent(); c.parent()
```

```
Univariate Polynomial Ring in a over Integer Ring
Sparse Univariate Polynomial Ring in b over Integer Ring
Univariate Polynomial Ring in c over Integer Ring (using NTL)
```

Parents and categories

Similar to the hierarchy of Python classes addressed to elements of algebraic structures, Sage also provides classes for the algebraic structures that contain these elements. Structures containing elements are called "parent structures" in Sage, and there is a base class for them. Roughly parallel to the hierarchy of mathematical notions, one has a hierarchy of classes, namely for sets, rings, fields, and so on:

```
sage: isinstance(QQ,Field)
True
sage: isinstance(QQ, Ring)
True
sage: isinstance(ZZ,Field)
False
sage: isinstance(ZZ, Ring)
True
```

In algebra, objects sharing the same kind of algebraic structures are collected in so-called "categories". So, there is a rough analogy between the class hierarchy in Sage and the hierarchy of categories. However, this analogy of Python classes and categories shouldn't be stressed too much. After all, mathematical categories are implemented in Sage as well:

```
sage: Rings()
Category of rings
sage: ZZ.category()
Join of Category of euclidean domains
    and Category of infinite enumerated sets
    and Category of metric spaces
sage: ZZ.category().is_subcategory(Rings())
True
sage: ZZ in Rings()
True
sage: ZZ in Fields()
False
sage: QQ in Fields()
True
```

While Sage's class hierarchy is centered at implementation details, Sage's category framework is more centered on mathematical structure. It is possible to implement generic methods and tests independent of a specific implementation in the categories.

Parent structures in Sage are supposed to be unique Python objects. For example, once a polynomial ring over a certain base ring and with a certain list of generators is created, the result is cached:

```
sage: RR['x','y'] is RR['x','y']
True
```

Types versus parents

The type `RingElement` does not correspond perfectly to the mathematical notion of a ring element. For example, although square matrices belong to a ring, they are not instances of `RingElement`:

```
sage: M = Matrix(ZZ,2,2); M
[0 0]
[0 0]
sage: isinstance(M, RingElement)
False
```

While *parents* are unique, equal *elements* of a parent in Sage are not necessarily identical. This is in contrast to the behaviour of Python for some (albeit not all) integers:

```
sage: int(1) is int(1) # Python int
True
sage: int(-15) is int(-15)
False
sage: 1 is 1           # Sage Integer
False
```

It is important to observe that elements of different rings are in general not distinguished by their type, but by their parent:

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: type(a) is type(b)
True
sage: parent(a)
Finite Field of size 2
sage: parent(b)
Finite Field of size 5
```

Hence, from an algebraic point of view, **the parent of an element is more important than its type.**

Conversion versus Coercion

In some cases it is possible to convert an element of one parent structure into an element of a different parent structure. Such conversion can either be explicit or implicit (this is called *coercion*).

The reader may know the notions *type conversion* and *type coercion* from, e.g., the C programming language. There are notions of *conversion* and *coercion* in Sage as well. But the notions in Sage are centered on *parents*, not on types. So, please don't confuse type conversion in C with conversion in Sage!

We give here a rather brief account. For a detailed description and for information on the implementation, we refer to the section on coercion in the reference manual and to the thematic tutorial.

There are two extremal positions concerning the possibility of doing arithmetic with elements of *different* rings:

- Different rings are different worlds, and it makes no sense whatsoever to add or multiply elements of different rings; even `1 + 1/2` makes no sense, since the first summand is an integer and the second a rational.

Or

- If an element `r1` of one ring `R1` can somehow be interpreted in another ring `R2`, then all arithmetic operations involving `r1` and any element of `R2` are allowed. The multiplicative unit exists in all fields and many rings, and they should all be equal.

Sage favours a compromise. If `P1` and `P2` are parent structures and `p1` is an element of `P1`, then the user may explicitly ask for an interpretation of `p1` in `P2`. This may not be meaningful in all cases or not be defined for all elements of `P1`, and it is up to the user to ensure that it makes sense. We refer to this as **conversion**:

```
sage: a = GF(2)(1)
sage: b = GF(5)(1)
sage: GF(5)(a) == b
```

```
True
sage: GF(2)(b) == a
True
```

However, an *implicit* (or automatic) conversion will only happen if this can be done *thoroughly* and *consistently*. Mathematical rigour is essential at that point.

Such an implicit conversion is called **coercion**. If coercion is defined, then it must coincide with conversion. Two conditions must be satisfied for a coercion to be defined:

1. A coercion from `P1` to `P2` must be given by a structure preserving map (e.g., a ring homomorphism). It does not suffice that *some* elements of `P1` can be mapped to `P2`, and the map must respect the algebraic structure of `P1`.
2. The choice of these coercion maps must be consistent: If `P3` is a third parent structure, then the composition of the chosen coercion from `P1` to `P2` with the coercion from `P2` to `P3` must coincide with the chosen coercion from `P1` to `P3`. In particular, if there is a coercion from `P1` to `P2` and `P2` to `P1`, the composition must be the identity map of `P1`.

So, although it is possible to convert each element of `GF(2)` into `GF(5)`, there is no coercion, since there is no ring homomorphism between `GF(2)` and `GF(5)`.

The second aspect - consistency - is a bit more difficult to explain. We illustrate it with multivariate polynomial rings. In applications, it certainly makes most sense to have name preserving coercions. So, we have:

```
sage: R1.<x,y> = ZZ[]
sage: R2 = ZZ['y','x']
sage: R2.has_coerce_map_from(R1)
True
sage: R2(x)
x
sage: R2(y)
y
```

If there is no name preserving ring homomorphism, coercion is not defined. However, conversion may still be possible, namely by mapping ring generators according to their position in the list of generators:

```
sage: R3 = ZZ['z','x']
sage: R3.has_coerce_map_from(R1)
False
sage: R3(x)
z
sage: R3(y)
x
```

But such position preserving conversions do not qualify as coercion: By composing a name preserving map from `ZZ['x','y']` to `ZZ['y','x']` with a position preserving map from `ZZ['y','x']` to `ZZ['a','b']`, a map would result that is neither name preserving nor position preserving, in violation to consistency.

If there is a coercion, it will be used to compare elements of different rings or to do arithmetic. This is often convenient, but the user should be aware that extending the ==-relation across the borders of different parents may easily result in overdoing it. For example, while == is supposed to be an equivalence relation on the elements of *one* ring, this is not necessarily the case if *different* rings are involved. For example, `1` in `ZZ` and in a finite field are considered equal, since there is a canonical coercion from the integers to any finite field. However, in general there is no coercion between two different finite fields. Therefore we have

```
sage: GF(5)(1) == 1
True
sage: 1 == GF(2)(1)
True
sage: GF(5)(1) == GF(2)(1)
False
```

```
sage: GF(5)(1) != GF(2)(1)
True
```

Similarly, we have

```
sage: R3(R1.1) == R3.1
True
sage: R1.1 == R3.1
False
sage: R1.1 != R3.1
True
```

Another consequence of the consistency condition is that coercions can only go from exact rings (e.g., the rationals `QQ`) to inexact rings (e.g., real numbers with a fixed precision `RR`), but not the other way around. The reason is that the composition of the coercion from `QQ` to `RR` with a conversion from `RR` to `QQ` is supposed to be the identity on `QQ`. But this is impossible, since some distinct rational numbers may very well be treated equal in `RR`, as in the following example:

```
sage: RR(1/10^200+1/10^100) == RR(1/10^100)
True
sage: 1/10^200+1/10^100 == 1/10^100
False
```

When comparing elements of two parents `P1` and `P2`, it is possible that there is no coercion between the two rings, but there is a canonical choice of a parent `P3` so that both `P1` and `P2` coerce into `P3`. In this case, coercion will take place as well. A typical use case is the sum of a rational number and a polynomial with integer coefficients, yielding a polynomial with rational coefficients:

```
sage: P1.<x> = ZZ[]
sage: p = 2*x+3
sage: q = 1/2
sage: parent(p)
Univariate Polynomial Ring in x over Integer Ring
sage: parent(p+q)
Univariate Polynomial Ring in x over Rational Field
```

Note that in principle the result would also make sense in the fraction field of `ZZ['x']`. However, Sage tries to choose a *canonical* common parent that seems to be most natural (`QQ['x']` in our example). If several potential common parents seem equally natural, Sage will *not* pick one of them at random, in order to have a reliable result. The mechanisms which that choice is based upon is explained in the thematic tutorial.

No coercion into a common parent will take place in the following example:

```
sage: R.<x> = QQ[]
sage: S.<y> = QQ[]
sage: x+y
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: 'Univariate Polynomial Ring in x ove
```

The reason is that Sage would not choose one of the potential candidates `QQ['x']['y']`, `QQ['y']['x']`, `QQ['x','y']` or `QQ['y','x']`, because all of these four pairwise different structures seem natural common parents, and there is no apparent canonical choice.