General Rings, Ideals, and Morphisms

Release 10.4

The Sage Development Team

Jul 23, 2024
CONTENTS

1  Base Classes for Rings, Algebras and Fields  1
2  Ideals  37
3  Ring Morphisms  69
4  Quotient Rings  107
5  Fraction Fields  137
6  Localization  157
7  Ring Extensions  173
8  Generic Data Structures and Algorithms for Rings  235
9  Utilities  241
10  Derivation  259
11  Indices and Tables  283
   Python Module Index  285
   Index  287
CHAPTER ONE

BASE CLASSES FOR RINGS, ALGEBRAS AND FIELDS

1.1 Rings

This module provides the abstract base class Ring from which all rings in Sage (used to) derive, as well as a selection of more specific base classes.

Warning: Those classes, except maybe for the lowest ones like CommutativeRing and Field, are being progressively deprecated in favor of the corresponding categories, which are more flexible, in particular with respect to multiple inheritance.

The class inheritance hierarchy is:

- Ring (to be deprecated)
  - Algebra (to be deprecated)
  - CommutativeRing
    - NoetherianRing (deprecated)
    - CommutativeAlgebra (deprecated and essentially removed)
    - IntegralDomain (deprecated)
      - DedekindDomain (deprecated and essentially removed)
      - PrincipalIdealDomain (deprecated and essentially removed)

Subclasses of CommutativeRing are

- Field
  - FiniteField

Some aspects of this structure may seem strange, but this is an unfortunate consequence of the fact that Cython classes do not support multiple inheritance.

(A distinct but equally awkward issue is that sometimes we may not know in advance whether or not a ring belongs in one of these classes; e.g. some orders in number fields are Dedekind domains, but others are not, and we still want to offer a unified interface, so orders are never instances of the deprecated DedekindDomain class.)

AUTHORS:

- David Harvey (2006-10-16): changed CommutativeAlgebra to derive from CommutativeRing instead of from Algebra.
class sage.rings.ring.Algebra

Bases: Ring

Generic algebra

class sage.rings.ring.CommutativeAlgebra

Bases: CommutativeRing

class sage.rings.ring.CommutativeRing

Bases: Ring

Generic commutative ring.

def extension(poly, name=None, names=None, **kwds)

Algebraically extends self by taking the quotient self[x] / (f(x)).

INPUT:

- poly – A polynomial whose coefficients are coercible into self
- name – (optional) name for the root of f

Note: Using this method on an algebraically complete field does not return this field; the construction self[x] / (f(x)) is done anyway.

EXAMPLES:

sage: R = QQ['x']
sage: y = polygen(R)
sage: R.extension(y^2 - 5, 'a')
Univariate Quotient Polynomial Ring in a over Univariate Polynomial Ring in x over Rational Field with modulus a^2 - 5

>>> from sage.all import *
>>> R = QQ['x']
>>> y = polygen(R)
>>> R.extension(y**Integer(2) - Integer(5), 'a')
Univariate Quotient Polynomial Ring in a over Univariate Polynomial Ring in x over Rational Field with modulus a^2 - 5

sage: # needs sage.rings.finite_rings
sage: P.<x> = PolynomialRing(GF(5))
sage: F.<a> = GF(5).extension(x^2 - 2)
sage: P.<t> = F[]
sage: R.<b> = F.extension(t^2 - a); R
Univariate Quotient Polynomial Ring in b over Finite Field in a of size 5^2 with modulus b^2 + 4*a

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> P = PolynomialRing(GF(Integer(5)), names=('x',)); (x,) = P._first_ngens(1)

(continues on next page)
fraction_field()  
Return the fraction field of self.

EXAMPLES:

```python
sage: R = Integers(389)[x,y]
sage: Frac(R)
Fraction Field of Multivariate Polynomial Ring in x, y over Ring of integers\n   modulo 389
sage: R.fraction_field()  
Fraction Field of Multivariate Polynomial Ring in x, y over Ring of integers\n   modulo 389
```

ideal_monoid()  
Return the monoid of ideals of this ring.

EXAMPLES:

```python
sage: ZZ.ideal_monoid()
Monoid of ideals of Integer Ring
sage: R.<x>=QQ[]; R.ideal_monoid()
Monoid of ideals of Univariate Polynomial Ring in x over Rational Field
```

is_commutative()  
Return True, since this ring is commutative.

EXAMPLES:

```python
sage: QQ.is_commutative()  
True
```

(continues on next page)
sage: A = QuaternionAlgebra(QQ, -1, -3, names=('i','j','k')); A 
Quaternion Algebra (-1, -3) with base ring Rational Field 
sage: A.is_commutative() 
False

>>> from sage.all import *
>>> QQ.is_commutative()
True
>>> ZpCA(Integer(7)).is_commutative()
True

>>> from sage.all import *
>>> QQ.is_commutative()
False

krull_dimension()

Return the Krull dimension of this commutative ring.

The Krull dimension is the length of the longest ascending chain of prime ideals.

localization(additional_units, names=None, normalize=True, category=None)

Return the localization of self at the given additional units.

EXAMPLES:

sage: R.<x, y> = GF(3)[]
sage: R.localization((x*y, x^2 + y^2))
Multivariate Polynomial Ring in x, y over Finite Field of size 3 localized at (y, x, x^2 + y^2)
sage: -y in _
True

class sage.rings.ring.DedekindDomain

Bases: CommutativeRing

class sage.rings.ring.Field

Bases: CommutativeRing

Generic field
algebraic_closure()  
Return the algebraic closure of self.

**Note:** This is only implemented for certain classes of field.

**EXAMPLES:**

```python
sage: K = PolynomialRing(QQ, 'x').fraction_field(); K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: K.algebraic_closure()
Traceback (most recent call last):
  ...  
NotImplementedError: Algebraic closures of general fields not implemented.
```

```python
>>> from sage.all import *

>>> K = PolynomialRing(QQ, 'x').fraction_field(); K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> K.algebraic_closure()
Traceback (most recent call last):
  ...  
NotImplementedError: Algebraic closures of general fields not implemented.
```

divides(x, y, coerce=True)  
Return True if x divides y in this field (usually True in a field!). If coerce is True (the default), first coerce x and y into self.

**EXAMPLES:**

```python
sage: QQ.divides(2, 3/4)
True
sage: QQ.divides(0, 5)
False
```

```python
>>> from sage.all import *

>>> QQ.divides(Integer(2), Integer(3)/Integer(4))
True
>>> QQ.divides(Integer(0), Integer(5))
False
```

fraction_field()  
Return the fraction field of self.

**EXAMPLES:**

Since fields are their own field of fractions, we simply get the original field in return:

```python
sage: QQ.fraction_field()
Rational Field
sage: RR.fraction_field()  
  # needs sage.rings.real_mpfr
Real Field with 53 bits of precision
sage: CC.fraction_field()  
  # needs sage.rings.real_mpfr
Complex Field with 53 bits of precision
```

(continues on next page)
General Rings, Ideals, and Morphisms, Release 10.4

(continued from previous page)

sage: F = NumberField(x^2 + 1, 'i')  # needs sage.rings.number_field
sage: F.fraction_field()              # needs sage.rings.number_field
Number Field in i with defining polynomial x^2 + 1

>>> from sage.all import *
>>> QQ.fraction_field()
Rational Field
>>> RR.fraction_field()             # needs sage.rings.real_mpfr
Real Field with 53 bits of precision
>>> CC.fraction_field()             # needs sage.rings.real_mpfr
Complex Field with 53 bits of precision
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) + Integer(1), 'i')  # needs sage.rings.number_field
>>> F.fraction_field()              # needs sage.rings.number_field
Number Field in i with defining polynomial x^2 + 1

ideal(*gens, **kwds)

Return the ideal generated by gens.

EXAMPLES:

sage: QQ.ideal(2)
Principal ideal (1) of Rational Field
sage: QQ.ideal(0)
Principal ideal (0) of Rational Field

>>> from sage.all import *
>>> QQ.ideal(Integer(2))
Principal ideal (1) of Rational Field
>>> QQ.ideal(Integer(0))
Principal ideal (0) of Rational Field

integral_closure()

Return this field, since fields are integrally closed in their fraction field.

EXAMPLES:

sage: QQ.integral_closure()
Rational Field
sage: Frac(ZZ['x,y']).integral_closure()
Fraction Field of Multivariate Polynomial Ring in x, y over Integer Ring

>>> from sage.all import *
>>> QQ.integral_closure()
Rational Field
>>> Frac(ZZ['x,y']).integral_closure()
Fraction Field of Multivariate Polynomial Ring in x, y over Integer Ring

Chapter 1. Base Classes for Rings, Algebras and Fields
is_field(proof=True)
Return True since this is a field.

EXAMPLES:

```
sage: Frac(ZZ['x,y']).is_field()
True
```

```
>>> from sage.all import *
>>> Frac(ZZ['x,y']).is_field()
True
```

is_integrally_closed()
Return True since fields are trivially integrally closed in their fraction field (since they are their own fraction field).

EXAMPLES:

```
sage: Frac(ZZ['x,y']).is_integrally_closed()
True
```

```
>>> from sage.all import *
>>> Frac(ZZ['x,y']).is_integrally_closed()
True
```

is_noetherian()
Return True since fields are Noetherian rings.

EXAMPLES:

```
sage: QQ.is_noetherian()
True
```

```
>>> from sage.all import *
>>> QQ.is_noetherian()
True
```

krull_dimension()
Return the Krull dimension of this field, which is 0.

EXAMPLES:

```
sage: QQ.krull_dimension()
0
sage: Frac(QQ['x,y']).krull_dimension()
0
```

```
>>> from sage.all import *
>>> QQ.krull_dimension()
0
>>> Frac(QQ['x,y']).krull_dimension()
0
```

prime_subfield()
Return the prime subfield of self.

EXAMPLES:
```python
# SageMath
sage: k = GF(9, 'a')
# needs sage.rings.finite_rings
sage: k.prime_subfield()
# needs sage.rings.finite_rings
Finite Field of size 3

>>> from sage.all import *
>>> k = GF(Integer(9), 'a')
# needs sage.rings.finite_rings
>>> k.prime_subfield()
# needs sage.rings.finite_rings
Finite Field of size 3
```

class sage.rings.ring.IntegralDomain

Bases: CommutativeRing

Generic integral domain class.

This class is deprecated. Please use the `sage.categories.integral_domains.IntegralDomains` category instead.

**is_field** *(proof=True)*

Return `True` if this ring is a field.

**EXAMPLES:**

```python
sage: GF(7).is_field()
True

>>> from sage.all import *
>>> GF(Integer(7)).is_field()
True
```

The following examples have their own `is_field` implementations:

```python
sage: ZZ.is_field(); QQ.is_field()
False
True
sage: R.<x> = PolynomialRing(QQ); R.is_field()
False
```

```python
>>> from sage.all import *
>>> ZZ.is_field(); QQ.is_field()
False
True
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1); R.is_field()
False
```

**is_integrally_closed()**

Return `True` if this ring is integrally closed in its field of fractions; otherwise return `False`.

When no algorithm is implemented for this, then this function raises a `NotImplementedError`.

Note that `is_integrally_closed` has a naive implementation in fields. For every field $F$, $F$ is its own field of fractions, hence every element of $F$ is integral over $F$.

**EXAMPLES:**
sage: ZZ.is_integrally_closed()
True
sage: QQ.is_integrally_closed()
True
sage: QQbar.is_integrally_closed() # needs sage.rings.number_field
True
sage: GF(5).is_integrally_closed()
True
sage: Z5 = Integers(5); Z5
Ring of integers modulo 5
sage: Z5.is_integrally_closed()
Traceback (most recent call last):
... AttributeError: 'IntegerModRing_generic_with_category' object has no attribute 'is_integrally_closed'

>>> from sage.all import *
>>> ZZ.is_integrally_closed()
True
>>> QQ.is_integrally_closed()
True
>>> QQbar.is_integrally_closed() # needs sage.rings.number_field
True
>>> GF(Integer(5)).is_integrally_closed()
True
>>> Z5 = Integers(Integer(5)); Z5
Ring of integers modulo 5
>>> Z5.is_integrally_closed()
Traceback (most recent call last):
... AttributeError: 'IntegerModRing_generic_with_category' object has no attribute 'is_integrally_closed'

class sage.rings.ring.NoetherianRing
Bases: CommutativeRing
class sage.rings.ring.PrincipalIdealDomain
Bases: CommutativeRing
class sage.rings.ring.Ring
Bases: ParentWithGens

1.1. Rings
category()

Return the category to which this ring belongs.

Note: This method exists because sometimes a ring is its own base ring. During initialisation of a ring $R$, it may be checked whether the base ring (hence, the ring itself) is a ring. Hence, it is necessary that $R.category()$ tells that $R$ is a ring, even before its category is properly initialised.

EXAMPLES:

```python
sage: FreeAlgebra(QQ, 3, 'x').category()  # todo: use a ring which is not an algebra! # needs sage.combinat sage.modules
Category of algebras with basis over Rational Field
```

Since a quotient of the integers is its own base ring, and during initialisation of a ring it is tested whether the base ring belongs to the category of rings, the following is an indirect test that the category() method of rings returns the category of rings even before the initialisation was successful:

```python
sage: I = Integers(15)
sage: I.base_ring() is I
True
sage: I.category()
Join of Category of finite commutative rings
   and Category of subquotients of monoids
   and Category of quotients of semigroups
   and Category of finite enumerated sets
```

epsilon()

Return the precision error of elements in this ring.

EXAMPLES:
For exact rings, zero is returned:

```
sage: ZZ.epsilon()
0
```

This also works over derived rings:

```
sage: RR['x'].epsilon()
0
```

For the symbolic ring, there is no reasonable answer:

```
sage: SR.epsilon()
Traceback (most recent call last):
  ... Not Implemented Error
```

(continues on next page)
ideal(*args, **kwds)

Return the ideal defined by \( x \), i.e., generated by \( x \).

INPUT:

- \( *x \) – list or tuple of generators (or several input arguments)
- \( \text{coerce} \) – bool (default: \( \text{True} \)); this must be a keyword argument. Only set it to \( \text{False} \) if you are certain that each generator is already in the ring.
- \( \text{ideal_class} \) – callable (default: \( \text{self._ideal_class_}() \)); this must be a keyword argument. A constructor for ideals, taking the ring as the first argument and then the generators. Usually a subclass of \( \text{Ideal_generic} \) or \( \text{Ideal_nc} \).
- Further named arguments (such as \( \text{side} \) in the case of non-commutative rings) are forwarded to the ideal class.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: R.ideal(x,y)
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ideal(x+y^2)
Ideal (y^2 + x) of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ideal([x^3,y^3+x^3])
Ideal (x^3, x^3 + y^3) of Multivariate Polynomial Ring in x, y over Rational Field
```

Here is an example over a non-commutative ring:

```python
sage: A = SteenrodAlgebra(2)  # needs sage.combinat sage.modules
sage: A.ideal(A.1, A.2^2)  # needs sage.combinat sage.modules
Twosided Ideal (Sq(2), Sq(2,2)) of mod 2 Steenrod algebra, milnor basis
sage: A.ideal(A.1, A.2^2, side='left')  # needs sage.combinat sage.modules
Left Ideal (Sq(2), Sq(2,2)) of mod 2 Steenrod algebra, milnor basis
```

```python
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> R.ideal(x,y)
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field
>>> R.ideal(x+y**Integer(2))
Ideal (y^2 + x) of Multivariate Polynomial Ring in x, y over Rational Field
>>> R.ideal( [x**Integer(3),y**Integer(3)+x**Integer(3)] )
Ideal (x^3, x^3 + y^3) of Multivariate Polynomial Ring in x, y over Rational Field
```
Twosided Ideal \((\text{Sq}(2), \text{Sq}(2,2))\) of mod 2 Steenrod algebra, milnor basis

```python
>>> A.ideal(A.gen(1), A.gen(2)**Integer(2), side='left')
˓→ # needs sage.combinat sage.modules
```

Left Ideal \((\text{Sq}(2), \text{Sq}(2,2))\) of mod 2 Steenrod algebra, milnor basis

```python
ideal_monoid()
```

Return the monoid of ideals of this ring.

**EXAMPLES:**

```python
sage: # needs sage.combinat sage.modules
sage: F.<x,y,z> = FreeAlgebra(ZZ, 3)
sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2] * F
sage: Q = F.quotient(I)
sage: Q.ideal_monoid()
```

Monoid of ideals of Quotient of Free Algebra on 3 generators \((x, y, z)\) over Integer Ring by the ideal \((x*y + y*z, x^2 + x*y - y*x - y^2)\)

```python
sage: F.<x,y,z> = FreeAlgebra(ZZ, implementation='letterplace')
sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2] * F
sage: Q = F.quotient(I)
```

```python
Q.ideal_monoid()
```

Monoid of ideals of Quotient of Free Associative Unital Algebra on 3 generators \((x, y, z)\) over Integer Ring by the ideal \((x*y + y*z, x*x + x*y - y*x - y*y)\)

```python
from sage.all import *
```

```python
F = FreeAlgebra(ZZ, Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = F._first_ngens(3)
I = F * [x*y + y*z, x**Integer(2) + x*y - y*x - y**Integer(2)] * F
Q = F.quotient(I)
```

```python
Q.ideal_monoid()
```

Monoid of ideals of Quotient of Free Algebra on 3 generators \((x, y, z)\) over Integer Ring by the ideal \((x*y + y*z, x^2 + x*y - y*x - y^2)\)

```python
F = FreeAlgebra(ZZ, implementation='letterplace', names=('x', 'y', 'z',));
(x, y, z,) = F._first_ngens(3)
I = F * [x*y + y*z, x**Integer(2) + x*y - y*x - y**Integer(2)] * F
Q = F.quo(I)
```

```python
Q.ideal_monoid()
```

Monoid of ideals of Quotient of Free Associative Unital Algebra on 3 generators \((x, y, z)\) over Integer Ring by the ideal \((x*y + y*z, x*x + x*y - y*x - y*y)\)

```python
is_exact()
```

Return \(\text{True}\) if elements of this ring are represented exactly, i.e., there is no precision loss when doing arithmetic.

**Note:** This defaults to \(\text{True}\), so even if it does return \(\text{True}\) you have no guarantee (unless the ring has properly overloaded this).

**EXAMPLES:**

```python
sage: QQ.is_exact() # indirect doctest
True
```

(continues on next page)
sage: ZZ.is_exact()
True
sage: Qp(7).is_exact() # needs sage.rings.padics
False
sage: Zp(7, type='capped-abs').is_exact() # needs sage.rings.padics
False

is_field (proof=True)

Return True if this ring is a field.

INPUT:

• proof — (default: True) Determines what to do in unknown cases

ALGORITHM:

If the parameter proof is set to True, the returned value is correct but the method might throw an error. Otherwise, if it is set to False, the method returns True if it can establish that self is a field and False otherwise.

EXAMPLES:

sage: QQ.is_field()
True
sage: GF(9, 'a').is_field() # needs sage.rings.finite_rings
True
sage: ZZ.is_field()
False
sage: QQ['x'].is_field()
False
sage: Frac(QQ['x']).is_field()
True
False

```python
>>> Frac(QQ['x']).is_field()
True
```

This illustrates the use of the `proof` parameter:

```python
sage: R.<a,b> = QQ[]
sage: S.<x,y> = R.quo((b^3))  # needs sage.libs.singular
sage: S.is_field(proof=True)  # needs sage.libs.singular
Traceback (most recent call last):
  ...
NotImplementedError
sage: S.is_field(proof=False)  # needs sage.libs.singular
False
```

`is_prime_field()`

Return `True` if this ring is one of the prime fields $\mathbb{Q}$ or $\mathbb{F}_p$.

**EXAMPLES:**

```python
sage: QQ.is_prime_field()
True
sage: GF(3).is_prime_field()
True
sage: GF(9, 'a').is_prime_field()  # needs sage.rings.finite_rings
False
sage: ZZ.is_prime_field()
False
sage: QQ['x'].is_prime_field()
False
sage: Qp(19).is_prime_field()  # needs sage.rings.padics
False
```

```python
>>> from sage.all import *
>>> QQ.is_prime_field()
True
>>> GF(Integer(3)).is_prime_field()
True
```

(continues on next page)
is_subring(other)
Return True if the canonical map from self to other is injective.
Raises a NotImplementedError if not known.

EXAMPLES:

```
sage: ZZ.is_subring(QQ)
True
sage: ZZ.is_subring(GF(19))
False
```

one()
Return the one element of this ring (cached), if it exists.

EXAMPLES:

```
sage: ZZ.one()
1
sage: QQ.one()
1
sage: QQ['x'].one()
1
```

The result is cached:

```
sage: ZZ.one() is ZZ.one()
True
```
order()

The number of elements of self.

EXAMPLES:

```python
sage: GF(19).order()
19
sage: QQ.order()
+Infinity
```

principal_ideal(gen, coerce=True)

Return the principal ideal generated by gen.

EXAMPLES:

```python
sage: R.<x,y> = ZZ[]
sage: R.principal_ideal(x+2*y)
Ideal (x + 2*y) of Multivariate Polynomial Ring in x, y over Integer Ring
```

random_element(bound=2)

Return a random integer coerced into this ring, where the integer is chosen uniformly from the interval \([-\text{bound}, \text{bound}]\).

INPUT:

• bound – integer (default: 2)

ALGORITHM:

Uses Python’s randint.

unit_ideal()

Return the unit ideal of this ring.

EXAMPLES:

```python
sage: Zp(7).unit_ideal()
# needs sage.rings.padics
Principal ideal (1 + O(7^20)) of 7-adic Ring with capped relative precision 20
```

(continues on next page)
 Principal ideal (1 + O(7^20)) of 7-adic Ring with capped relative precision 20

### zero()

Return the zero element of this ring (cached).

**EXAMPLES:**

```python
sage: ZZ.zero()
0
sage: QQ.zero()
0
sage: QQ['x'].zero()
0
```

The result is cached:

```python
sage: ZZ.zero() is ZZ.zero()
True
```

### zero_ideal()

Return the zero ideal of this ring (cached).

**EXAMPLES:**

```python
>>> from sage.all import *
>>> ZZ.zero_ideal()
Principal ideal (0) of Integer Ring
>>> QQ.zero_ideal()
Principal ideal (0) of Rational Field
>>> QQ['x'].zero_ideal()
Principal ideal (0) of Univariate Polynomial Ring in x over Rational Field
```

The result is cached:

```python
sage: ZZ.zero_ideal() is ZZ.zero_ideal()
True
```
>>> from sage.all import *
>>> ZZ.zero_ideal() is ZZ.zero_ideal()
True

\textbf{zeta} \ (n=2, \ all=False)

Return a primitive \(n\)-th root of unity in \texttt{self} if there is one, or raise a \texttt{ValueError} otherwise.

**INPUT:**

- \(n\) – positive integer
- \(all\) – bool (default: \texttt{False}); whether to return a list of all primitive \(n\)-th roots of unity. If True, raise a \texttt{ValueError} if \texttt{self} is not an integral domain.

**OUTPUT:**

Element of \texttt{self} of finite order

**EXAMPLES:**

\begin{verbatim}
sage: QQ.zeta() -1
sage: QQ.zeta(1) 1
sage: CyclotomicField(6).zeta(6) # needs sage.rings.number_field zeta6
sage: CyclotomicField(3).zeta(3) # needs sage.rings.number_field zeta3
sage: CyclotomicField(3).zeta(3).multiplicative_order() # needs sage.rings.number_field 3
sage: # needs sage.rings.finite_rings
sage: a = GF(7).zeta(); a 3
sage: a.multiplicative_order() 6
sage: a = GF(49,'z').zeta(); a z
sage: a.multiplicative_order() 48
sage: a = GF(49,'z').zeta(2); a 6
sage: a.multiplicative_order() 2
sage: QQ.zeta(3)
Traceback (most recent call last):
  ... ValueError: no n-th root of unity in rational field
sage: Zp(7, prec=8).zeta() 3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + 6*7^6 + 2*7^7 + O(7^8)
\end{verbatim}
zeta_order()

Return the order of the distinguished root of unity in self.

EXAMPLES:

 sage: CyclotomicField(19).zeta_order() 38
 sage: GF(19).zeta_order() 18
 sage: GF(5^3,'a').zeta_order() 124
 sage: Zp(7, prec=8).zeta_order() 6

>>> from sage.all import *

>>> CyclotomicField(Integer(19)).zeta_order() 38
 >>> # needs sage.rings.number_field

>>> GF(Integer(19)).zeta_order()
sage.rings.ring.is_Ring(x)

Return True if x is a ring.

EXAMPLES:

```
sage: from sage.rings.ring import is_Ring
sage: is_Ring(ZZ)
True
sage: MS = MatrixSpace(QQ, 2)
# needs sage.modules
sage: is_Ring(MS)
# needs sage.modules
True
```

```
>>> from sage.all import *
>>> from sage.rings.ring import is_Ring

>>> is_Ring(ZZ)
True
```

```
>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2))
# needs sage.modules
>>> is_Ring(MS)
# needs sage.modules
True
```

## 1.2 Abstract base classes for rings

### class sage.rings.abc.AlgebraicField

_bases: AlgebraicField_common

Abstract base class for AlgebraicField.

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

**EXAMPLES:**

```
sage: import sage.rings.abc
sage: isinstance(QQbar, sage.rings.abc.AlgebraicField)
# needs sage.rings.number_field
True
sage: isinstance(AA, sage.rings.abc.AlgebraicField)
# needs sage.rings.number_field
False
```

```
>>> from sage.all import *
>>> import sage.rings.abc

>>> isinstance(QQbar, sage.rings.abc.AlgebraicField)
# needs sage.rings.number_field
True
```

(continues on next page)
By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.AlgebraicField.__subclasses__()  #...

[<class sage.rings.qqbar.AlgebraicField>]

sage: len(sage.rings.abc.AlgebraicField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> sage.rings.abc.AlgebraicField.__subclasses__()  #...

[<class sage.rings.qqbar.AlgebraicField>]

>>> len(sage.rings.abc.AlgebraicField.__subclasses__()) <= Integer(1)
True
```

```python
class sage.rings.abc.AlgebraicField_common

Bases: Field

Abstract base class for AlgebraicField_common.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(QQbar, sage.rings.abc.AlgebraicField_common)  #...
True

sage: isinstance(AA, sage.rings.abc.AlgebraicField_common)  #...
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(QQbar, sage.rings.abc.AlgebraicField_common)  #...
True

>>> isinstance(AA, sage.rings.abc.AlgebraicField_common)  #...
True
```

By design, other than the abstract subclasses AlgebraicField and AlgebraicRealField, there is only one direct implementation subclass:

```python
sage: sage.rings.abc.AlgebraicField_common.__subclasses__()  #...

[<class 'sage.rings.abc.AlgebraicField'>,
 <class 'sage.rings.abc.AlgebraicRealField'>,
 ...]  
```
```python
>>> from sage.all import *
>>> sage.rings.abc.AlgebraicField_common.__subclasses__()

<sage.rings.abc.AlgebraicField_common>

True
```

```python
class sage.rings.abc.AlgebraicRealField

Bases: AlgebraicField_common

Abstract base class for AlgebraicRealField.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(QQbar, sage.rings.abc.AlgebraicRealField)  # needs sage.rings.number_field
False
sage: isinstance(AA, sage.rings.abc.AlgebraicRealField)  # needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> import sage.rings.abc

```

```python
sage: len(sage.rings.abc.AlgebraicRealField.__subclasses__()) <= 1
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.AlgebraicRealField.__subclasses__()

<list>

True
```

```python
>>> from sage.all import *
>>> sage.rings.abc.AlgebraicRealField.__subclasses__()

<list>

True
```

1.2. Abstract base classes for rings
General Rings, Ideals, and Morphisms, Release 10.4

class sage.rings.abc.CallableSymbolicExpressionRing
Bases: SymbolicRing

Abstract base class for CallableSymbolicExpressionRing_class.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: f = x.function(x).parent()  # needs sage.symbolic
sage: isinstance(f, sage.rings.abc.CallableSymbolicExpressionRing)  # needs sage.symbolic
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.CallableSymbolicExpressionRing.__subclasses__()  # needs sage.symbolic
[<class 'sage.symbolic.callable.CallableSymbolicExpressionRing_class'>]
sage: len(sage.rings.abc.CallableSymbolicExpressionRing.__subclasses__()) <= 1
True
```

class sage.rings.abc.ComplexBallField
Bases: Field

Abstract base class for ComplexBallField.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: CBF = sage.rings.abc.ComplexBallField()  # needs sage.libs.flint
sage: isinstance(CBF, sage.rings.abc.ComplexBallField)  # needs sage.libs.flint
True
```

```python
>>> from sage.all import *
>>> import sage.rings.abc
>>> sage.rings.abc.ComplexBallField.__subclasses__()  # needs sage.symbolic
[<class 'sage.symbolic.callable.CallableSymbolicExpressionRing_class'>]
>>> len(sage.rings.abc.ComplexBallField.__subclasses__()) <= 1
<Integer(1)>
```

(continues on next page)
By design, there is a unique direct subclass:

```python
>>> isinstance(CBF, sage.rings.abc.ComplexBallField)  
# needs sage.libs.flint
True
```

```python
sage: sage.rings.abc.ComplexBallField.__subclasses__()  
# needs sage.libs.flint
[<class 'sage.rings.complex_arb.ComplexBallField'>]

sage: len(sage.rings.abc.ComplexBallField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(CDF, sage.rings.abc.ComplexDoubleField)  
# needs sage.rings.complex_double
True
```

```python
sage: sage.rings.abc.ComplexDoubleField.__subclasses__()  
# needs sage.rings.complex_double
[<class 'sage.rings.complex_double.ComplexDoubleField_class'>]

sage: len(sage.rings.abc.ComplexDoubleField.__subclasses__()) <= Integer(1)
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(CDF, sage.rings.abc.ComplexDoubleField)  
# needs sage.rings.complex_double
True
```

```python
class sage.rings.abc.ComplexDoubleField
Bases: Field

Abstract base class for ComplexDoubleField_class.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(CDF, sage.rings.abc.ComplexDoubleField)  
# needs sage.rings.complex_double
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(CDF, sage.rings.abc.ComplexDoubleField)  
# needs sage.rings.complex_double
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.ComplexDoubleField.__subclasses__()  
# needs sage.rings.complex_double
[<class 'sage.rings.complex_double.ComplexDoubleField_class'>]

sage: len(sage.rings.abc.ComplexDoubleField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(CDF, sage.rings.abc.ComplexDoubleField)  
# needs sage.rings.complex_double
True
```

1.2. Abstract base classes for rings
```python
class sage.rings.abc.ComplexField
    Bases: Field

    Abstract base class for ComplexField_class.

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

    EXAMPLES:

    >>> import sage.rings.abc
    >>> isinstance(CC, sage.rings.abc.ComplexField)  # needs sage.rings.real_mpfr
    True

    By design, there is a unique direct subclass:

    >>> from sage.all import *
    >>> import sage.rings.abc
    >>> isinstance(CC, sage.rings.abc.ComplexField)  # needs sage.rings.real_mpfr
    True

    class sage.rings.abc.ComplexIntervalField
        Bases: Field

        Abstract base class for ComplexIntervalField_class.

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

        EXAMPLES:

        >>> import sage.rings.abc
        >>> isinstance(CIF, sage.rings.abc.ComplexIntervalField)  # needs sage.rings.complex_interval_field
        True

        By design, there is a unique direct subclass:

        >>> from sage.all import *
        >>> import sage.rings.abc
        >>> isinstance(CIF, sage.rings.abc.ComplexIntervalField)  # needs sage.rings.complex_interval_field
        True
```
### sage.rings.abc.ComplexIntervalField

```python
sage: sage.rings.abc.ComplexIntervalField.__subclasses__()
[<class 'sage.rings.complex_interval_field.ComplexIntervalField_class'>]
```

```python
sage: len(sage.rings.abc.ComplexIntervalField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> sage.rings.abc.ComplexIntervalField.__subclasses__()
[<class 'sage.rings.complex_interval_field.ComplexIntervalField_class'>]

>>> len(sage.rings.abc.ComplexIntervalField.__subclasses__()) <= Integer(1)
True
```

### sage.rings.abc.IntegerModRing

```python
class sage.rings.abc.IntegerModRing

Bases: object

Abstract base class for IntegerModRing_generic.

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

EXAMPLES:
```
```python
sage: import sage.rings.abc

sage: isinstance(Integers(7), sage.rings.abc.IntegerModRing)
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> isinstance(Integers(Integer(7)), sage.rings.abc.IntegerModRing)
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.IntegerModRing.__subclasses__()
[<class 'sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic'>]
```

```python
sage: len(sage.rings.abc.IntegerModRing.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> sage.rings.abc.IntegerModRing.__subclasses__()
[<class 'sage.rings.finite_rings.integer_mod_ring.IntegerModRing_generic'>]

>>> len(sage.rings.abc.IntegerModRing.__subclasses__()) <= Integer(1)
True
```

### sage.rings.abc.NumberField_cyclotomic

```python
class sage.rings.abc.NumberField_cyclotomic

Bases: Field

Abstract base class for NumberField_cyclotomic.

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

EXAMPLES:
```
```
sage: import sage.rings.abc
sage: K.<zeta> = CyclotomicField(15)  # needs sage.rings.number_field
sage: isinstance(K, sage.rings.abc.NumberField_cyclotomic)  # needs sage.rings.number_field
True

By design, there is a unique direct subclass:

sage: sage.rings.abc.NumberField_cyclotomic.__subclasses__()  # needs sage.rings.number_field
[<class 'sage.rings.number_field.number_field.NumberField_cyclotomic'>]
sage: len(sage.rings.abc.NumberField_cyclotomic.__subclasses__()) <= 1
True

>>> from sage.all import *
>>> import sage.rings.abc

>>> K = CyclotomicField(Integer(15), names=('zeta',)); (zeta,) = K._first_ngens(1)
# needs sage.rings.number_field
>>> isinstance(K, sage.rings.abc.NumberField_cyclotomic)
# needs sage.rings.number_field
True

class sage.rings.abc.NumberField_quadratic

Bases: Field

Abstract base class for NumberField_quadratic.

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

EXAMPLES:

sage: import sage.rings.abc
sage: K.<sqrt2> = QuadraticField(2)  # needs sage.rings.number_field
sage: isinstance(K, sage.rings.abc.NumberField_quadratic)  # needs sage.rings.number_field
True

>>> from sage.all import *
>>> import sage.rings.abc

>>> K = QuadraticField(Integer(2), names=('sqrt2',)); (sqrt2,) = K._first_ngens(1)
# needs sage.rings.number_field
>>> isinstance(K, sage.rings.abc.NumberField_quadratic)
# needs sage.rings.number_field
True

By design, there is a unique direct subclass:
### Abstract base classes for rings

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

**EXAMPLES:**

```python
sage: import sage.rings.abc
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 1); O = K.order(2*a)  # needs sage.rings.number_field
sage: isinstance(O, sage.rings.abc.Order)  # needs sage.rings.number_field
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.Order.__subclasses__()  # needs sage.rings.number_field
[<class 'sage.rings.number_field.order.Order'>]
```

(continues on next page)
```python
>>> len(sage.rings.abc.Order.__subclasses__()) <= Integer(1)
True

class sage.rings.abc.RealBallField
Bases: Field

Abstract base class for RealBallField.

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

EXAMPLES:
```
By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.RealDoubleField.__subclasses__()
[<class 'sage.rings.real_double.RealDoubleField_class'>]
sage: len(sage.rings.abc.RealDoubleField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *
>>> sage.rings.abc.RealDoubleField.__subclasses__()
[<class 'sage.rings.real_double.RealDoubleField_class'>]
>>> len(sage.rings.abc.RealDoubleField.__subclasses__()) <= Integer(1)
True
```

```python
class sage.rings.abc.RealField
    Bases: Field
    Abstract base class for RealField_class.

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

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(RR, sage.rings.abc.RealField)
# needs sage.rings.real_mpfr
True
```

```python
>>> from sage.all import *
>>> import sage.rings.abc

>>> isinstance(RR, sage.rings.abc.RealField)
# needs sage.rings.real_mpfr
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.RealField.__subclasses__()
[<class 'sage.rings.real_mpfr.RealField_class'>]
sage: len(sage.rings.abc.RealField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *
>>> sage.rings.abc.RealField.__subclasses__()
[<class 'sage.rings.real_mpfr.RealField_class'>]
>>> len(sage.rings.abc.RealField.__subclasses__()) <= Integer(1)
True
```

```python
class sage.rings.abc.RealIntervalField
    Bases: Field
    Abstract base class for RealIntervalField_class.

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

1.2. Abstract base classes for rings
31
EXAMPLES:

```
sage: import sage.rings.abc
sage: isinstance(RIF, sage.rings.abc.RealIntervalField)  # needs sage.rings.real_interval_field
True
```

```
>>> from sage.all import *
>>> import sage.rings.abc

>>> isinstance(RIF, sage.rings.abc.RealIntervalField)  # needs sage.rings.real_interval_field
True
```

By design, there is a unique direct subclass:

```
sage: sage.rings.abc.RealIntervalField.__subclasses__()  # needs sage.rings.real_interval_field
[<class 'sage.rings.real_mpfi.RealIntervalField_class'>]

sage: len(sage.rings.abc.RealIntervalField.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> import sage.rings.abc

>>> sage.rings.abc.RealIntervalField.__subclasses__()  # needs sage.rings.real_interval_field
[<class 'sage.rings.real_mpfi.RealIntervalField_class'>]

>>> len(sage.rings.abc.RealIntervalField.__subclasses__()) <= Integer(1)
True
```

class sage.rings.abc.SymbolicRing

Bases: CommutativeRing

Abstract base class for SymbolicRing.

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

EXAMPLES:

```
sage: import sage.rings.abc
sage: isinstance(SR, sage.rings.abc.SymbolicRing)  # needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> import sage.rings.abc

>>> isinstance(SR, sage.rings.abc.SymbolicRing)  # needs sage.symbolic
True
```

By design, other than the abstract subclass `CallableSymbolicExpressionRing`, there is only one direct implementation subclass:

```
sage: sage.rings.abc.SymbolicRing.__subclasses__()  # needs sage.symbolic
[<class 'sage.rings.abc.CallableSymbolicExpressionRing'>, <class 'sage.symbolic.ring.SymbolicRing'>]
```

(continues on next page)
sage: len(sage.rings.abc.SymbolicRing.__subclasses__()) <= 2
True

```python
>>> from sage.all import *

>>> sage.rings.abc.SymbolicRing.__subclasses__()
[<class 'sage.rings.abc.CallableSymbolicExpressionRing'>,
 <class 'sage.symbolic.ring.SymbolicRing'>]
```

```python
>>> len(sage.rings.abc.SymbolicRing.__subclasses__()) <= Integer(2)
True
```

class sage.rings.abc.UniversalCyclotomicField
Bases: Field

Abstract base class for UniversalCyclotomicField.

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

EXAMPLES:

```python
sage: import sage.rings.abc

sage: K = UniversalCyclotomicField()  # needs sage.libs.gap sage.rings.number_field

sage: isinstance(K, sage.rings.abc.UniversalCyclotomicField)  # needs sage.libs.gap sage.rings.number_field
True
```

```python
>>> from sage.all import *

>>> import sage.rings.abc

>>> K = UniversalCyclotomicField()  # needs sage.libs.gap sage.rings.number_field

>>> isinstance(K, sage.rings.abc.UniversalCyclotomicField)  # needs sage.libs.gap sage.rings.number_field
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.UniversalCyclotomicField.__subclasses__()
[<class 'sage.rings.universal_cyclotomic_field.UniversalCyclotomicField'>]

sage: len(sage.rings.abc.UniversalCyclotomicField.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *

>>> sage.rings.abc.UniversalCyclotomicField.__subclasses__()
[<class 'sage.rings.universal_cyclotomic_field.UniversalCyclotomicField'>]

>>> len(sage.rings.abc.UniversalCyclotomicField.__subclasses__()) <= Integer(1)
True
```

class sage.rings.abc.pAdicField
Bases: Field
Abstract base class for \texttt{pAdicFieldGeneric}.

This class is defined for the purpose of \texttt{isinstance} tests. It should not be instantiated.

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(Zp(5), sage.rings.abc.pAdicField)  # needs sage.rings.padics
False
sage: isinstance(Qp(5), sage.rings.abc.pAdicField)  # needs sage.rings.padics
True
```

By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.pAdicField.__subclasses__()  # needs sage.rings.padics
[<class 'sage.rings.padics.generic_nodes.pAdicFieldGeneric'>]
sage: len(sage.rings.abc.pAdicField.__subclasses__()) <= 1
True
```

class \texttt{sage.rings.abc.pAdicRing}

Bases: \texttt{IntegralDomain}

Abstract base class for \texttt{pAdicRingGeneric}.

This class is defined for the purpose of \texttt{isinstance} tests. It should not be instantiated.

EXAMPLES:

```python
sage: import sage.rings.abc
sage: isinstance(Zp(5), sage.rings.abc.pAdicRing)  # needs sage.rings.padics
True
sage: isinstance(Qp(5), sage.rings.abc.pAdicRing)  # needs sage.rings.padics
False
```
By design, there is a unique direct subclass:

```python
sage: sage.rings.abc.pAdicRing.__subclasses__()  # needs sage.rings.padics
[<class 'sage.rings.padics.generic_nodes.pAdicRingGeneric'>]
sage: len(sage.rings.abc.pAdicRing.__subclasses__()) <= 1
True
```

```python
>>> from sage.all import *
>>> import sage.rings.abc
>>>
>>> isinstance(Zp(Integer(5)), sage.rings.abc.pAdicRing)  # needs sage.rings.padics
True
>>> isinstance(Qp(Integer(5)), sage.rings.abc.pAdicRing)  # needs sage.rings.padics
False
```
IDEALS

2.1 Ideals of commutative rings

Sage provides functionality for computing with ideals. One can create an ideal in any commutative or non-commutative ring \( R \) by giving a list of generators, using the notation \( R.\text{ideal}([a, b, \ldots ]) \). The case of non-commutative rings is implemented in \texttt{noncommutative_ideals}.

A more convenient notation may be \( R^*[a, b, \ldots] \) or \([a, b, \ldots]*R\). If \( R \) is non-commutative, the former creates a left and the latter a right ideal, and \( R^*[a, b, \ldots]*R \) creates a two-sided ideal.

\[
\text{sage.rings.ideal.Cyclic}(R, n=None, homog=False, singular=None)
\]

Ideal of cyclic \( n \)-roots from 1-st \( n \) variables of \( R \) if \( R \) is coercible to \texttt{Singular}.

**INPUT:**

- \( R \) – base ring to construct ideal for
- \( n \) – number of cyclic roots (default: None). If None, then \( n \) is set to \( R.\text{ngens}() \).
- \( \text{homog} \) – (default: False) if True a homogeneous ideal is returned using the last variable in the ideal
- \( \text{singular} \) – singular instance to use

**Note:** \( R \) will be set as the active ring in \texttt{Singular}

**EXAMPLES:**

An example from a multivariate polynomial ring over the rationals:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: I = sage.rings.ideal.Cyclic(P); I
# needs sage.libs.singular
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1)
```

```
sage: I.groebner_basis()
# needs sage.libs.singular
[x + y + z, y^2 + y*z + z^2, z^3 - 1]
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ, Integer(3), order='lex', names=('x', 'y', 'z')); (x, ...
    y, z) = P._first_ngens(3)
>>> I = sage.rings.ideal.Cyclic(P); I
# needs sage.libs.singular
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1)
```

(continues on next page)
of Multivariate Polynomial Ring in x, y, z over Rational Field

```python
>>> I.groebner_basis()

Needs sage.libs.singular
[x + y + z, y^2 + y*z + z^2, z^3 - 1]
```

We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal:

```python
sage: R.<x,y,z,t,u,v> = QQ[\'x,y,z,t,u,v\']
sage: I = sage.rings.ideal.Cyclic(R, 6)

Needs sage.libs.singular

```
```python
sage: B = I.groebner_basis()

Needs sage.libs.singular
```

```python
sage: len(B)

45
```

```python
from sage.all import *

P.<x1,x2,x3,x4> = PolynomialRing(GF(2^4, name='alpha'), 4)

J = sage.rings.ideal.FieldIdeal(Q); J

Ideal (x1^16 + x1, x2^16 + x2, x3^16 + x3, x4^16 + x4) of
```

Let \( q = R.\text{base\_ring\()\text{.order\() \) and \( (x_0, ..., x_n) = R.\text{gens\() \) then if \( q \) is finite this constructor returns

\[
\langle x_0^q - x_0, ..., x_n^q - x_n \rangle.
\]

We call this ideal the field ideal and the generators the field equations.

EXAMPLES:

The field ideal generated from the polynomial ring over two variables in the finite field of size 2:

```python
sage: P.<x,y> = PolynomialRing(GF(2), 2)
sage: I = sage.rings.ideal.FieldIdeal(P); I
Ideal (x^2 + x, y^2 + y) of
```

(continues on next page)
sage.rings.ideal.Ideal(*args, **kwds)

Create the ideal in ring with given generators.

There are some shorthand notations for creating an ideal, in addition to using the Ideal() function:

- R.ideal(gens, coerce=True)
- gens*R
- R*gens

INPUT:
- R – A ring (optional; if not given, will try to infer it from gens)
- gens – list of elements generating the ideal
- coerce – bool (default: True); whether gens need to be coerced into the ring.

OUTPUT: The ideal of ring generated by gens.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: I
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer
R
```

```

```python
>>> from sage.all import *

>>> R = ZZ['x']; (x,) = R._first_ngens(1)

>>> I = R.ideal([Integer(4) + Integer(3)*x + x**Integer(2), Integer(1) + x**Integer(2)])

>>> I
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer
R
```

(continues on next page)
General Rings, Ideals, and Morphisms, Release 10.4

(continued from previous page)

```python
>>>
Ideal((Integer(4) + Integer(3)*x + x**Integer(2), Integer(1) + x**Integer(2)))
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer...

sage: ideal(x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1))
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer...

sage: ideal([x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1)])
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer...

sage: l = [x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1)]

sage: ideal(f**Integer(2) for f in l)
Ideal (x^4 - 4*x^3 + 6*x^2 - 4*x + 1, x^4 - 2*x^2 + 1) of Univariate Polynomial Ring in x over Integer Ring

>>>
from sage.all import *

>>> ideal(x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1))
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer...

>>> ideal([x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1)])
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer...

>>> i = [x**Integer(2)-Integer(2)*x+Integer(1), x**Integer(2)-Integer(1)]

>>> ideal(f**Integer(2) for f in l)
Ideal (x^4 - 4*x^3 + 6*x^2 - 4*x + 1, x^4 - 2*x^2 + 1) of Univariate Polynomial Ring in x over Integer Ring
```

This example illustrates how Sage finds a common ambient ring for the ideal, even though 1 is in the integers (in this case).

```
sage: R.<t> = ZZ['t']
sage: i = ideal(1,t,t^2)
sage: i
Ideal (1, t, t^2) of Univariate Polynomial Ring in t over Integer Ring

sage: ideal(1/2,t,t**Integer(2))
Principal ideal (1) of Univariate Polynomial Ring in t over Integer Ring

>>> from sage.all import *

>>> R = ZZ['t']; (t,) = R._first_ngens(1)

>>> i = ideal(Integer(1),t,t**Integer(2))

>>> i
Ideal (1, t, t^2) of Univariate Polynomial Ring in t over Integer Ring

>>> ideal(Integer(1)/Integer(2),t,t**Integer(2))
Principal ideal (1) of Univariate Polynomial Ring in t over Rational Field
```

This shows that the issues at Issue #1104 are resolved:

```
sage: Ideal(3, 5)
Principal ideal (1) of Integer Ring

sage: Ideal(ZZ, 3, 5)
Principal ideal (1) of Integer Ring

sage: Ideal(2, 4, 6)
Principal ideal (2) of Integer Ring
```
You have to provide enough information that Sage can figure out which ring to put the ideal in.

```sage
sage: I = Ideal([])
Traceback (most recent call last):
... ValueError: unable to determine which ring to embed the ideal in
```

```sage
sage: I = Ideal()
Traceback (most recent call last):
... ValueError: need at least one argument
```

Note that some rings use different ideal implementations than the standard, even if they are PIDs:

```sage
sage: R.<x> = GF(5)[]
sage: I = R *(x^2 + 3)
sage: type(I)
<class 'sage.rings.polynomial.ideal.Ideal_1poly_field'>
```

```sage
sage: from sage.all import *
>>> from sage.rings.ideal import Ideal_pid

sage: R = GF(Integer(5))[x]; (x,) = R._first_ngens(1)
sage: I = R *(x**Integer(2) + Integer(3))
sage: type(I)
<class 'sage.rings.polynomial.ideal.Ideal_1poly_field'>
```

You can also pass in a specific ideal type:

```sage
sage: from sage.rings.ideal import Ideal_pid
sage: I = Ideal(x^2+3,ideal_class=Ideal_pid)
sage: type(I)
<class 'sage.rings.ideal.Ideal_pid'>
```

```sage
>>> from sage.all import *
>>> from sage.rings.ideal import Ideal_pid

>>> I = Ideal(x**Integer(2)+Integer(3),ideal_class=Ideal_pid)
>>> type(I)
<class 'sage.rings.ideal.Ideal_pid'>
```


```python
class sage.rings.ideal.Ideal_fractional(ring, gens, coerce=True, **kwds)
    Bases: sage.rings.ideal.Ideal_generic
    Fractional ideal of a ring.
    See SageDoc(Ideal()).

class sage.rings.ideal.Ideal_generic(ring, gens, coerce=True, **kwds)
    Bases: sage.rings.monoid.MonoidElement
    An ideal.
    See SageDoc(Ideal()).

    absolute_norm()
    Returns the absolute norm of this ideal.
    In the general case, this is just the ideal itself, since the ring it lies in can't be implicitly assumed to be an extension of anything.
    We include this function for compatibility with cases such as ideals in number fields.

    Todo: Implement this method.

    EXAMPLES:
    >>> from sage.all import *
    >>> R.<t> = GF(9, names='a')[1]
    >>> I = R.ideal(t^4 + t + 1)
    >>> I.absolute_norm()
    Traceback (most recent call last):
      ... Not ImplementedError
```

```python
apply_morphism(phi)
Apply the morphism phi to every element of this ideal. Returns an ideal in the domain of phi.

EXAMPLES:
```
```
Principal ideal \((x - 1.00000000000000)\) of Univariate Polynomial Ring in \(x\) over Complex Field with 53 bits of precision

\[
\text{sage}: \ J.\text{apply}\_\text{morphism}(\psi)
\]

Principal ideal \((x - 1.00000000000000)\) of Univariate Polynomial Ring in \(x\) over Complex Field with 53 bits of precision

\[
\text{sage}: \ \text{psi} = \ZZ['x'].\text{hom}([-\ZZ['x'].0])
\]

\[
\text{sage}: \ J = \text{ideal}(\[-\ZZ['x'].0, 2\]); \ J
\]

Ideal \((x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{sage}: \psi(J)
\]

Ideal \((-x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{sage}: \ J.\text{apply}\_\text{morphism}(\psi)
\]

Ideal \((-x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{sage}: \ \text{psi} = \ZZ['x'].\text{hom}([-\ZZ['x'].0])
\]

\[
\text{sage}: \ J = \text{ideal}(\[-\ZZ['x'].0, \text{Integer}(2)\]); \ J
\]

Ideal \((x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{sage}: \psi(J)
\]

Ideal \((-x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{sage}: \ J.\text{apply}\_\text{morphism}(\psi)
\]

Ideal \((-x, 2)\) of Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\text{associated\_primes}()
\]

Return the list of associated prime ideals of this ideal.

EXAMPLES:

\[
\text{sage}: \ R = \ZZ['x']
\]

\[
\text{sage}: \ I = \text{R.ideal}(\text{Integer}(7))
\]

\[
\text{sage}: \ I.\text{associated}\_\text{primes}()
\]

Traceback (most recent call last):
...
NotImplementedError

\[
\text{from sage.all import } *
\]

\[
\text{from sage.all import } *
\]

\[
\text{from sage.all import } *
\]
```python
base_ring()

Returns the base ring of this ideal.

EXAMPLES:

sage: R = ZZ
sage: I = 3*R; I
Principal ideal (3) of Integer Ring
sage: J = 2*I; J
Principal ideal (6) of Integer Ring
sage: I.base_ring(); J.base_ring()
Integer Ring
Integer Ring

>>> from sage.all import *
>>> R = ZZ
>>> I = Integer(3)*R; I
Principal ideal (3) of Integer Ring
>>> J = Integer(2)*I; J
Principal ideal (6) of Integer Ring
>>> I.base_ring(); J.base_ring()
Integer Ring
Integer Ring

We construct an example of an ideal of a quotient ring:

sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
Rational Field
sage: I = R.ideal(x^2 - 2)
Rational Field

>>> from sage.all import *
>>> R = PolynomialRing(QQ, 'x'); x = R.gen()
>>> I = R.ideal(x**Integer(2) - Integer(2))
>>> I.base_ring()
Rational Field

And $p$-adic numbers:

sage: R = Zp(7, prec=10); R
7-adic Ring with capped relative precision 10
sage: I = 7*R; I
Principal ideal (7 + O(7^11)) of 7-adic Ring with capped relative precision 10
sage: I.base_ring()
7-adic Ring with capped relative precision 10

>>> from sage.all import *
>>> R = Zp(Integer(7), prec=Integer(10)); R
7-adic Ring with capped relative precision 10
>>> I = Integer(7)*R; I
Principal ideal (7 + O(7^11)) of 7-adic Ring with capped relative precision 10
>>> I.base_ring()
7-adic Ring with capped relative precision 10
```

(continues on next page)
category()

Return the category of this ideal.

Note: category is dependent on the ring of the ideal.

EXAMPLES:

```python
sage: P.<x> = ZZ[]
sage: I = ZZ.ideal(7)
sage: J = P.ideal(7, x)
sage: K = P.ideal(7)
sage: I.category()
Category of ring ideals in Integer Ring
sage: J.category()
Category of ring ideals in Univariate Polynomial Ring in x
over Integer Ring
sage: K.category()
Category of ring ideals in Univariate Polynomial Ring in x
over Integer Ring
```

```python
>>> from sage.all import *

>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> I = ZZ.ideal(Integer(7))
>>> J = P.ideal(Integer(7), x)
>>> K = P.ideal(Integer(7))
```

embedded_primes()

Return the list of embedded primes of this ideal.

EXAMPLES:

```python
sage: R.<x, y> = QQ[]
sage: I = R.ideal(x^2, x*y)
sage: I.embedded_primes()
[\text{Ideal} \ (y, x) \ of \ Multivariate \ Polynomial \ Ring \ in \ x, \ y \ over \ Rational \ Field]
```

```python
>>> from sage.all import *

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal(x**Integer(2), x*y)
```

2.1. Ideals of commutative rings
free_resolution(*args, **kwds)

Return a free resolution of self.

For input options, see FreeResolution.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: I = R.ideal([x^4 + 3*x^2 + 2])
sage: I.free_resolution()  # needs sage.modules
S^1 <-- S^1 <-- 0
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> I = R.ideal([x**Integer(4) + Integer(3)*x**Integer(2) + Integer(2)])
>>> I.free_resolution()  # needs sage.modules
S^1 <-- S^1 <-- 0
```

gen(i)

Return the i-th generator in the current basis of this ideal.

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gen(1)
y + 1
```

```
>>> from sage.all import *
>>> P = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = P._first_ngens(2)
>>> I = Ideal([x,y+Integer(1)]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
>>> I.gen(Integer(1))
y + 1
``` 

gens()

Return a set of generators / a basis of self.

This is the set of generators provided during creation of this ideal.

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gens()
[x, y + 1]
```

>>> from sage.all import *

>>> P = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = P._first_ngens(2)
>>> I = Ideal([x, y+Integer(1)]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field

>>> I.gens()
[x, y + 1]

sage: ZZ.ideal(5,10).gens()
(5,)

>>> from sage.all import *

>>> ZZ.ideal(Integer(5),Integer(10)).gens()
(5,)

gens_reduced()

Same as gens() for this ideal, since there is currently no special gens_reduced algorithm implemented for this ring.

This method is provided so that ideals in \( \mathbb{Z} \) have the method gens_reduced(), just like ideals of number fields.

EXAMPLES:

sage: ZZ.ideal(5).gens_reduced()
(5,)

>>> from sage.all import *

>>> ZZ.ideal(Integer(5)).gens_reduced()
(5,)

graded_free_resolution(*args, **kwds)

Return a graded free resolution of self.

For input options, see GradedFiniteFreeResolution.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: I = R.ideal([x^3])
sage: I.graded_free_resolution()  #...
needs sage.modules
S(0) <-> S(-3) <-> 0

>>> from sage.all import *

>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> I = R.ideal([x**Integer(3)])
>>> I.graded_free_resolution()  #...
needs sage.modules
S(0) <-> S(-3) <-> 0

is_maximal()

Return True if the ideal is maximal in the ring containing the ideal.

Todo: This is not implemented for many rings. Implement it!
EXAMPLES:

```python
sage: R = ZZ
sage: I = R.ideal(7)
sage: I.is_maximal()
True
sage: R.ideal(16).is_maximal()
False
sage: S = Integers(8)
sage: S.ideal(0).is_maximal()
False
sage: S.ideal(2).is_maximal()
True
sage: S.ideal(4).is_maximal()
False
```

```python
>>> from sage.all import *

>>> R = ZZ

>>> I = R.ideal(Integer(7))

>>> I.is_maximal()
True

>>> R.ideal(Integer(16)).is_maximal()
False

>>> S = Integers(Integer(8))

>>> S.ideal(Integer(0)).is_maximal()
False

>>> S.ideal(Integer(2)).is_maximal()
True

>>> S.ideal(Integer(4)).is_maximal()
False
```

**is_primary** \((P=None)\)

Returns True if this ideal is primary (or \(P\)-primary, if a prime ideal \(P\) is specified).

Recall that an ideal \(I\) is primary if and only if \(I\) has a unique associated prime (see page 52 in [AM1969]). If this prime is \(P\), then \(I\) is said to be \(P\)-primary.

**INPUT:**

- \(P \) – (default: None) a prime ideal in the same ring

**EXAMPLES:**

```python
sage: R.<x, y> = QQ[]
sage: I = R.ideal([x^2, x*y])
sage: I.is_primary() # needs sage.libs.singular
False

sage: J = I.primary_decomposition()[1]; J # needs sage.libs.singular
Ideal (y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field

sage: J.is_primary() # needs sage.libs.singular
True

sage: J.is_prime() # needs sage.libs.singular
False
```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([x**Integer(2), x*y])
>>> I.is_primary()  # __
→ needs sage.libs.singular
False
>>> J = I.primary_decomposition()[Integer(1)]; J
→ # needs sage.libs.singular
Ideal (y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field  
>>> J.is_primary()  # __
→ needs sage.libs.singular
True
>>> J.is_prime()  # __
→ needs sage.libs.singular
False

Some examples from the Macaulay2 documentation:

sage: # needs sage.rings.finite_rings
sage: R.<x, y, z> = GF(101)[]
sage: I = R.ideal([y^6])
sage: I.is_primary()  # __
→ needs sage.libs.singular
True
sage: I.is_primary(R.ideal([y]))
→ # needs sage.libs.singular
True
sage: I = R.ideal([x^4, y^7])

sage: I.is_primary()  # __
→ needs sage.libs.singular
True
sage: I = R.ideal([x*y, y^2])

sage: I.is_primary()  # __
→ needs sage.libs.singular
False

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> R = GF(Integer(101))[x, y, z]; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal([y^6])

sage: I.is_primary()  # __
→ needs sage.libs.singular
True
sage: I.is_primary(R.ideal([y]))
→ # needs sage.libs.singular
True
sage: I = R.ideal([x^4, y^7])

sage: I.is_primary()  # __
→ needs sage.libs.singular
True
sage: I = R.ideal([x*y, y^2])

sage: I.is_primary()  # __
→ needs sage.libs.singular
False

Note: This uses the list of associated primes.
is_prime()  
Return True if this ideal is prime.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = R.ideal([x, y])
sage: I.is_prime()  # a maximal ideal
    # needs sage.libs.singular
True
sage: I = R.ideal([x^2 - y])
sage: I.is_prime()  # a non-maximal prime ideal
    # needs sage.libs.singular
True
sage: I = R.ideal([x^2, y])
sage: I.is_prime()  # a non-prime primary ideal
False
sage: I = R.ideal([x^2, x*y])
sage: I.is_prime()  # a non-prime non-primary ideal
False
sage: S = Integers(8)
sage: S.ideal(0).is_prime()
False
sage: S.ideal(2).is_prime()
True
sage: S.ideal(4).is_prime()
False
```

```
>>> from sage.all import *
>>> R = QQ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([x, y])
>>> I.is_prime()  # a maximal ideal
    # needs sage.libs.singular
True
>>> I = R.ideal([x**Integer(2) - y])
>>> I.is_prime()  # a non-maximal prime ideal
    # needs sage.libs.singular
True
>>> I = R.ideal([x**Integer(2), y])
>>> I.is_prime()  # a non-prime primary ideal
    # needs sage.libs.singular
False
>>> I = R.ideal([x**Integer(2), x*y])
>>> I.is_prime()  # a non-prime non-primary ideal
    # needs sage.libs.singular
False
>>> S = Integers(Integer(8))
>>> S.ideal(Integer(0)).is_prime()
False
>>> S.ideal(Integer(2)).is_prime()
True
>>> S.ideal(Integer(4)).is_prime()
False
```
Note that this method is not implemented for all rings where it could be:

```python
sage: R.<x> = ZZ[]
sage: I = R.ideal(7)
sage: I.is_prime() # when implemented, should be True
Traceback (most recent call last):
  ... 
NotImplementedError
```

```python
>>> from sage.all import *

>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> I = R.ideal(Integer(7))
>>> I.is_prime() # when implemented, should be True
Traceback (most recent call last):
  ... 
NotImplementedError
```

**Note:** For general rings, uses the list of associated primes.

### `is_principal()`

Returns `True` if the ideal is principal in the ring containing the ideal.

**Todo:** Code is naive. Only keeps track of ideal generators as set during initialization of the ideal. (Can the base ring change? See example below.)

#### EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: I = R.ideal(2, x)
sage: I.is_principal()
Traceback (most recent call last):
  ... 
NotImplementedError
sage: J = R.base_extend(QQ).ideal(2, x)
sage: J.is_principal()
True
```

```python
>>> from sage.all import *

>>> R = ZZ['x']; (x,) = R._first_ngens(1)

>>> I = R.ideal(Integer(2), x)
>>> I.is_principal()
Traceback (most recent call last):
  ... 
NotImplementedError

>>> J = R.base_extend(QQ).ideal(Integer(2), x)

>>> J.is_principal()
True
```

### `is_trivial()`

Returns `True` if this ideal is $(0)$ or $(1)$.

### `minimal_associated_primes()`

Return the list of minimal associated prime ideals of this ideal.

#### EXAMPLES:
sage: R = ZZ['x']
sage: I = R.ideal(7)
sage: I.minimal_associated_primes()
Traceback (most recent call last):
...  
NotImplementedError

```python
>>> from sage.all import *

>>> R = ZZ['x']

>>> I = R.ideal(Integer(7))

>>> I.minimal_associated_primes()
Traceback (most recent call last):
...  
NotImplementedError
```

```
ngens()

Return the number of generators in the basis.

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.ngens()
2

sage: ZZ.ideal(5,10).ngens()
1
```

```
>>> from sage.all import *

>>> P = PolynomialRing(QQ,Integer(2), names=(x, y,)); (x, y,)
˓→ ngens(2)

>>> I = Ideal([x,y+Integer(1)]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field

>>> I.ngens()
2

>>> ZZ.ideal(Integer(5),Integer(10)).ngens()
1
```

```
norm()

Returns the norm of this ideal.

In the general case, this is just the ideal itself, since the ring it lies in can’t be implicitly assumed to be an
extension of anything.

We include this function for compatibility with cases such as ideals in number fields.

EXAMPLES:

sage: R.<t> = GF(8, names='a')[]
˓→ needs sage.rings.finite_rings

sage: I = R.ideal(t^4 + t + 1)
˓→ needs sage.rings.finite_rings

sage: I.norm()
˓→ needs sage.rings.finite_rings

Principal ideal (t^4 + t + 1) of Univariate Polynomial Ring in t
over Finite Field in a of size 2^3
```
>>> from sage.all import *
>>> R = GF(Integer(8), names='a')[t]; (t,) = R._first_ngens(1) # needs sage.rings.finite_rings
>>> I = R.ideal(t**Integer(4) + t + Integer(1)) # needs sage.rings.finite_rings
>>> I.norm()
Principal ideal (t^4 + t + 1) of Univariate Polynomial Ring in t
over Finite Field in a of size 2^3

primary_decomposition()
Return a decomposition of this ideal into primary ideals.

EXAMPLES:

sage: R = ZZ['x']
sage: I = R.ideal(7)
sage: I.primary_decomposition()
Traceback (most recent call last):
... NotImplementedError

random_element(*args, **kwds)
Return a random element in this ideal.

EXAMPLES:

sage: P.<a,b,c> = GF(5)[[[]]

random_element() # random
2*a^5*c + a^2*b*c^4 + ... + O(a, b, c)^13

reduce(f)
Return the reduction of the element of f modulo self.
This is an element of R that is equivalent modulo I to f where I is self.

EXAMPLES:

sage: ZZ.ideal(5).reduce(17)
2
sage: parent(ZZ.ideal(5).reduce(17))
Integer Ring

2.1. Ideals of commutative rings
```python
>>> from sage.all import *
>>> ZZ.ideal(Integer(5)).reduce(Integer(17))
2
>>> parent(ZZ.ideal(Integer(5)).reduce(Integer(17)))
Integer Ring
```

```
ring()
Return the ring containing this ideal.

EXAMPLES:
```
```
sage: R = ZZ
sage: I = 3*R; I
Principal ideal (3) of Integer Ring
sage: J = 2*I; J
Principal ideal (6) of Integer Ring
sage: I.ring(); J.ring()
Integer Ring
Integer Ring
```
```
Note that `self.ring()` is different from `self.base_ring()`
```
```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 - 2)
Rational Field
sage: I.ring()
Univariate Polynomial Ring in x over Rational Field
```
```
Another example using polynomial rings:
```
```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 - 3)
Rational Field
sage: I.ring()
Univariate Polynomial Ring in x over Rational Field
```
```
sage: Rbar = R.quotient(I, names='a')
#→ needs sage.libs.pari
```
```
sage: S = PolynomialRing(Rbar, 'y'); y = Rbar.gen(); S
```
```
(continues on next page)
Univariate Polynomial Ring in y over
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^2 - 3

```
sage: J = S.ideal(y^2 + 1)  # needs sage.libs.pari
sage: J.ring()  # needs sage.libs.pari
```

Univariate Polynomial Ring in y over
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^2 - 3

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, 'x'); x = R.gen()
>>> I = R.ideal(x**Integer(2) - Integer(3))
>>> I.ring()
Univariate Polynomial Ring in x over Rational Field
>>> Rbar = R.quotient(I, names='a')  # needs sage.libs.pari
>>> S = PolynomialRing(Rbar, 'y'); y = Rbar.gen(); S  # needs sage.libs.pari
Univariate Polynomial Ring in y over
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^2 - 3
```

```
>>> J = S.ideal(y**Integer(2) + Integer(1))  # needs sage.libs.pari
>>> J.ring()  # needs sage.libs.pari
```

```
class sage.rings.ideal.Ideal_pid(ring, gens, coerce=True, **kwds)
    An ideal of a principal ideal domain.
    See :meth:`Ideal`.
    EXAMPLES:

    sage: I = 8*ZZ
    sage: I
    Principal ideal (8) of Integer Ring

    >>> from sage.all import *
    >>> I = Integer(8)*ZZ
    >>> I
    Principal ideal (8) of Integer Ring

    .. attribute:: gcd
        Returns the greatest common divisor of the principal ideal with the ideal other; that is, the largest principal ideal contained in both the ideal and other

        Todo: This is not implemented in the case when other is neither principal nor when the generator of self
```

2.1. Ideals of commutative rings
is contained in other. Also, it seems that this class is used only in PIDs–is this redundant?

Note: The second example is broken.

EXAMPLES:

An example in the principal ideal domain \( \mathbb{Z} \):

```python
sage: R = ZZ
sage: I = R.ideal(42)
sage: J = R.ideal(70)
```

```python
sage: I.gcd(J)
Principal ideal (14) of Integer Ring

sage: J.gcd(I)
Principal ideal (14) of Integer Ring
```

```python
>>> from sage.all import *
>>> R = ZZ
>>> I = R.ideal(Integer(42))
>>> J = R.ideal(Integer(70))
```

```python
>>> I.gcd(J)
Principal ideal (14) of Integer Ring
>>> J.gcd(I)
Principal ideal (14) of Integer Ring
```

**is_maximal()**

Returns whether this ideal is maximal.

Principal ideal domains have Krull dimension 1 (or 0), so an ideal is maximal if and only if it’s prime (and nonzero if the ring is not a field).

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: R.<t> = GF(5)[]
```

```python
sage: p = R.ideal(t**2 + 2)
sage: p.is_maximal()
True
sage: p = R.ideal(t**2 + 1)
sage: p.is_maximal()
False
sage: p = R.ideal(0)
sage: p.is_maximal()
False
sage: p = R.ideal(1)
sage: p.is_maximal()
False
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
```

```python
>>> R = GF(Integer(5))[t]; (t,) = R._first_ngens(1)
>>> p = R.ideal(t**Integer(2) + Integer(2))
```

```python
>>> p.is_maximal()
True
```

```python
>>> p = R.ideal(t**Integer(2) + Integer(1))
```

(continues on next page)
is_prime()

Return True if the ideal is prime.

This relies on the ring elements having a method is_irreducible() implemented, since an ideal $(a)$ is prime iff $a$ is irreducible (or 0).

EXAMPLES:

```python
sage: ZZ.ideal(2).is_prime()
True
sage: ZZ.ideal(-2).is_prime()
True
sage: ZZ.ideal(4).is_prime()
False
sage: ZZ.ideal(0).is_prime()
True
sage: R.<x> = QQ[
R.ideal(x**2 + 1); P
Principal ideal (x^2 + 1) of Univariate Polynomial Ring in x over Rational˓→Field
sage: P.is_prime()
# needs sage.libs.pari
True
```

```python
>>> from sage.all import *
>>> ZZ.ideal(Integer(2)).is_prime()
True
>>> ZZ.ideal(-Integer(2)).is_prime()
True
>>> ZZ.ideal(Integer(4)).is_prime()
False
>>> ZZ.ideal(Integer(0)).is_prime()
True
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = R.ideal(x**Integer(2) + Integer(1)); P
Principal ideal (x^2 + 1) of Univariate Polynomial Ring in x over Rational˓→Field
>>> P.is_prime()
# needs sage.libs.pari
True
```

In fields, only the zero ideal is prime:

```python
sage: RR.ideal(0).is_prime()
True
sage: RR.ideal(7).is_prime()
False
```

2.1. Ideals of commutative rings
reduce \( f \)

Return the reduction of \( f \) modulo \( \text{self} \).

**EXAMPLES:**

```
sage: I = 8*ZZ
sage: I.reduce(10)
2
sage: n = 10; n.mod(I)
2
```

residue_field()

Return the residue class field of this ideal, which must be prime.

**Todo:** Implement this for more general rings. Currently only defined for \( \mathbb{Z} \) and for number field orders.

**EXAMPLES:**

```
sage: # needs sage.libs.pari
sage: P = ZZ.ideal(61); P
Principal ideal (61) of Integer Ring
sage: F = P.residue_field(); F
Residue field of Integers modulo 61
sage: pi = F.reduction_map(); pi
Partially defined reduction map:
  From: Rational Field
  To: Residue field of Integers modulo 61
sage: pi(123/234)
6
sage: pi(1/61)
Traceback (most recent call last):
  ... ZeroDivisionError: Cannot reduce rational 1/61 modulo 61: it has negative
  \rightarrow \text{valuation}
```

```
sage: lift = F.lift_map(); lift
Lifting map:
  From: Residue field of Integers modulo 61
  To: Integer Ring
sage: lift(F(12345/67890))
33
sage: (12345/67890) % 61
33
```
>>> from sage.all import *
>>> # needs sage.libs pari
>>> P = ZZ.ideal(Integer(61)); P
Principal ideal (61) of Integer Ring
>>> F = P.residue_field(); F
Residue field of Integers modulo 61
>>> pi = F.reduction_map(); pi
Partially defined reduction map:
    From: Rational Field
    To:   Residue field of Integers modulo 61
>>> pi(Integer(123)/Integer(234))
6
>>> pi(Integer(1)/Integer(61))
Traceback (most recent call last):
... ZeroDivisionError: Cannot reduce rational 1/61 modulo 61: it has negative
˓→valuation
>>> lift = F.lift_map(); lift
Lifting map:
    From: Residue field of Integers modulo 61
    To:   Integer Ring
>>> lift(F(Integer(12345)/Integer(67890)))
33
>>> (Integer(12345)/Integer(67890)) % Integer(61)
33

class sage.rings.ideal.Ideal_principal(ring, gens, coerce=True, **kwds)

Bases: Ideal_generic

A principal ideal.

See Ideal().

divides(other)

    Return True if self divides other.

EXAMPLES:

sage: P.<x> = PolynomialRing(QQ)
sage: I = P.ideal(x)
sage: J = P.ideal(x^2)
sage: I.divides(J)
True
sage: J.divides(I)
False

>>> from sage.all import *
>>> P = PolynomialRing(QQ, names=('x',)); (x,) = P._first_ngens(1)
>>> I = P.ideal(x)
>>> J = P.ideal(x**Integer(2))
>>> I.divides(J)
True
>>> J.divides(I)
False

gen(i=0)

    Return the generator of the principal ideal.

    The generator is an element of the ring containing the ideal.
EXAMPLES:

A simple example in the integers:

```
sage: R = ZZ
sage: I = R.ideal(7)
sage: J = R.ideal(7, 14)
sage: I.gen(); J.gen()
7
7
```

Note that the generator belongs to the ring from which the ideal was initialized:

```
sage: R.<x> = ZZ[

sage: I = R.ideal(x)
sage: J = R.base_extend(QQ).ideal(2,x)
sage: a = I.gen(); a
x
sage: b = J.gen(); b
1
sage: a.base_ring()
Integer Ring
sage: b.base_ring()
Rational Field
```

```
>>> from sage.all import *

>>> R = ZZ

>>> I = R.ideal(Integer(7))

>>> J = R.ideal(Integer(7), Integer(14))

>>> I.gen(); J.gen()
7
7
```

```
>>> from sage.all import *

>>> R = ZZ[x]; (x,) = R._first_ngens(1)

>>> I = R.ideal(x)

>>> J = R.base_extend(QQ).ideal(Integer(2),x)

>>> a = I.gen(); a
x

>>> b = J.gen(); b
1

>>> a.base_ring()
Integer Ring

>>> b.base_ring()
Rational Field
```

```
is_principal()
```

Returns True if the ideal is principal in the ring containing the ideal. When the ideal construction is explicitly principal (i.e. when we define an ideal with one element) this is always the case.

EXAMPLES:

Note that Sage automatically coerces ideals into principal ideals during initialization:

```
sage: R.<x> = ZZ[

sage: I = R.ideal(x)

sage: J = R.ideal(2,x)

sage: K = R.base_extend(QQ).ideal(2,x)
```

(continues on next page)
sage: I
Principal ideal (x) of Univariate Polynomial Ring in x
over Integer Ring
sage: J
Ideal (2, x) of Univariate Polynomial Ring in x over Integer Ring
sage: K
Principal ideal (1) of Univariate Polynomial Ring in x
over Rational Field
sage: I.is_principal()
True
sage: K.is_principal()
True

sage.rings.ideal.Katsura(R, n=None, homog=False, singular=None)
n-th katsura ideal of R if R is coercible to Singular.

INPUT:

- R – base ring to construct ideal for
- n – (default: None) which katsura ideal of R. If None, then n is set to R.ngens().
- homog – if True a homogeneous ideal is returned using the last variable in the ideal (default: False)
- singular – singular instance to use

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = sage.rings.ideal.Katsura(P, 3); I
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y)
of Multivariate Polynomial Ring in x, y, z over Rational Field

sage: P = PolynomialRing(QQ, Integer(3), names=('x', 'y', 'z',)); (x, y, z) = P._first_ngens(3)
sage: I = sage.rings.ideal.Katsura(P, Integer(3)); I
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y)
of Multivariate Polynomial Ring in x, y, z over Rational Field

(continues on next page)
Ideal \((x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y)\)
of Multivariate Polynomial Ring in \(x, y, z\) over Rational Field

\[
\text{sage: } Q.<x> = PolynomialRing(QQ, implementation="singular") # needs sage.libs.singular
\text{sage: } J = sage.rings.ideal.Katsura(Q,1); J # needs sage.libs.singular
\]
Ideal \((x - 1)\) of Multivariate Polynomial Ring in \(x\) over Rational Field

sage.rings.ideal.is_Ideal(x)
Return True if object is an ideal of a ring.

EXAMPLES:
A simple example involving the ring of integers. Note that Sage does not interpret rings objects themselves as ideals. However, one can still explicitly construct these ideals:

\[
\text{sage: from sage.rings.ideal import is_Ideal}
\text{sage: R = ZZ}
\text{sage: is_Ideal(R)}
False
\text{sage: 1*R; is_Ideal(1*R)}
Principal ideal \((1)\) of Integer Ring
True
\text{sage: 0*R; is_Ideal(0*R)}
Principal ideal \((0)\) of Integer Ring
True
\]

Sage recognizes ideals of polynomial rings as well:

\[
\text{sage: } R = PolynomialRing(QQ, 'x'); x = R.gen()
\text{sage: } I = R.ideal(x^2 + 1); I
\text{sage: } is_Ideal(I)
True
\text{sage: } is_Ideal((x^2 + 1)*R)
True
\]
2.2 Monoid of ideals in a commutative ring

WARNING: This is used by some rings that are not commutative!

```python
sage: MS = MatrixSpace(QQ, 3, 3)  # needs sage.modules
sage: type(MS.ideal(MS.one()).parent())  # needs sage.modules
<class 'sage.rings.ideal_monoid.IdealMonoid_c_with_category'>
```

```python
sage: MS = MatrixSpace(QQ, Integer(3), Integer(3))  # needs sage.modules
sage: type(MS.ideal(MS.one()).parent())  # needs sage.modules
<class 'sage.rings.ideal_monoid.IdealMonoid_c_with_category'>
```

sage.rings.ideal_monoid.IdealMonoid(R)
Return the monoid of ideals in the ring $R$.

EXAMPLES:

```python
sage: R = QQ['x']
sage: from sage.rings.ideal_monoid import IdealMonoid
sage: IdealMonoid(R)
Monoid of ideals of Univariate Polynomial Ring in x over Rational Field
```

class sage.rings.ideal_monoid.IdealMonoid_c(R)
Bases: Parent
The monoid of ideals in a commutative ring.

Element
alias of Ideal_generic

ring()
Return the ring of which this is the ideal monoid.

EXAMPLES:
2.3 Ideals of non-commutative rings

Generic implementation of one- and two-sided ideals of non-commutative rings.

**AUTHOR:**

- Simon King (2011-03-21), <simon.king@uni-jena.de>, Issue #7797.

**EXAMPLES:**

```python
sage: MS = MatrixSpace(ZZ,2,2)
sage: MS*MS([[0,1,-2,3]])
Left Ideal
{
    [ 0 1]
    [-2 3]
}
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: MS([0,1,-2,3])*MS
Right Ideal
{
    [ 0 1]
    [-2 3]
}
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: MS*MS([0,1,-2,3])*MS
Twosided Ideal
{
    [ 0 1]
    [-2 3]
}
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

```python
>>> MS([[Integer(0),Integer(1),-Integer(2),Integer(3)]])*MS
Right Ideal
(
    [ 0 1 ]
    [-2 3 ]
)
```
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

```python
>>> MS*MS([[Integer(0),Integer(1),-Integer(2),Integer(3)]])*MS
Twosided Ideal
(
    [ 0 1 ]
    [-2 3 ]
)
```
of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

See `letterplace_ideal` for a more elaborate implementation in the special case of ideals in free algebras.

```python
class sage.rings.noncommutative_ideals.IdealMonoid_nc(R)
Bases: IdealMonoid_c

Base class for the monoid of ideals over a non-commutative ring.

Note: This class is essentially the same as `IdealMonoid_c`, but does not complain about non-commutative rings.

EXAMPLES:
```n
```python
sage: MS = MatrixSpace(ZZ,2,2)
sage: MS.ideal_monoid()
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```
```python
>>> from sage.all import *

```python
sage: MS = MatrixSpace(ZZ,Integer(2),Integer(2))
```python
sage: MS.ideal_monoid()
Monoid of ideals of Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

```python
class sage.rings.noncommutative_ideals.Ideal_nc(ring, gens, coerce=True, side='twosided')
Bases: Ideal_generic

Generic non-commutative ideal.

All fancy stuff such as the computation of Groebner bases must be implemented in sub-classes. See `LetterplaceIdeal` for an example.

EXAMPLES:
```n
```python
sage: MS = MatrixSpace(QQ,2,2)
sage: I = MS*[MS.1,MS.2]; I
Left Ideal
(
    [0 1]
    [0 0],
    [0 0]
)
```
(continues on next page)
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

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

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

>>> from sage.all import *

>>> MS = MatrixSpace(QQ, Integer(2), Integer(2))

>>> I = MS*[MS.gen(1), MS.gen(2)]; I
Left Ideal
(
  [0 1]
  [0 0],
  <BLANKLINE>
  [0 0]
  [1 0])
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

>>> [MS.gen(1), MS.gen(2)]*MS
Right Ideal
(
  [0 1]
  [0 0],
  <BLANKLINE>
  [0 0]
  [1 0])
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

>>> MS*[MS.gen(1), MS.gen(2)]*MS
Twosided Ideal
(
  [0 1]
  [0 0],
  <BLANKLINE>
  [0 0]
  [1 0])
of Full MatrixSpace of 2 by 2 dense matrices over Rational Field

side()
Return a string that describes the sidedness of this ideal.

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = SteenrodAlgebra(2)
sage: IL = A*[A.1+A.2,A.1^2]
sage: IR = [A.1+A.2,A.1^2]*A
sage: IT = A*[A.1+A.2,A.1^2]*A
sage: IL.side()
'left'
sage: IR.side()
'right'
sage: IT.side()
'twosided'
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = SteenrodAlgebra(Integer(2))
>>> IL = A*[A.gen(1)+A.gen(2),A.gen(1)**Integer(2)]
>>> IR = [A.gen(1)+A.gen(2),A.gen(1)**Integer(2)]*A
>>> IT = A*[A.gen(1)+A.gen(2),A.gen(1)**Integer(2)]*A
>>> IL.side()
'left'
>>> IR.side()
'right'
>>> IT.side()
'twosided'
```
3.1 Homomorphisms of rings

We give a large number of examples of ring homomorphisms.

EXAMPLES:

Natural inclusion $\mathbb{Z} \hookrightarrow \mathbb{Q}$:

```sage
sage: H = Hom(ZZ, QQ)
sage: phi = H([[1]])
sage: phi(10)
10
sage: phi(3/1)
3
sage: phi(2/3)
Traceback (most recent call last):
  ...TypeError: 2/3 fails to convert into the map's domain Integer Ring,
  but a `pushforward' method is not properly implemented
```

There is no homomorphism in the other direction:

```sage
sage: H = Hom(QQ, ZZ)
sage: H([[1]])
Traceback (most recent call last):
  ...ValueError: relations do not all (canonically) map to 0
  under map determined by images of generators
```
>>> from sage.all import *
>>> H = Hom(QQ, ZZ)
>>> H([Integer(1)])
Traceback (most recent call last):
  ...
ValueError: relations do not all (canonically) map to 0 under map determined by images of generators

EXAMPLES:

Reduction to finite field:

```python
sage: # needs sage.rings.finite_rings
sage: H = Hom(ZZ, GF(9, 'a'))
sage: phi = H([1])
sage: phi(5)
2
sage: psi = H([4])
sage: psi(5)
2
```

Map from single variable polynomial ring:

```python
sage: R.<x> = ZZ[]
sage: phi = R.hom([2], GF(5)); phi
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Finite Field of size 5
  Defn: x |--> 2
sage: phi(x + 12)
4
```

Identity map on the real numbers:

```python
sage: # needs sage.rings.real_mpfr
sage: f = RR.hom([RR(1)]); f
Ring endomorphism of Real Field with 53 bits of precision
```
3.1. Homomorphisms of rings

Homomorphism from one precision of field to another.

From smaller to bigger doesn’t make sense:

```
sage: R200 = RealField(200)  # needs sage.rings.real_mpfr
sage: f = RR.hom( R200 )  # needs sage.rings.real_mpfr
Traceback (most recent call last):
  ... TypeError: natural coercion morphism from Real Field with 53 bits of precision to Real Field with 200 bits of precision not defined
```

From bigger to small does:

```
sage: f = RR.hom(RealField(15))  # needs sage.rings.real_mpfr
sage: f(2.5)  # needs sage.rings.real_mpfr
2.500
sage: f(RR.pi())  # needs sage.rings.real_mpfr
3.142
```
Inclusion map from the reals to the complexes:

```python
sage: i = RR.hom(CC([1])); i
Ring morphism:
  From: Real Field with 53 bits of precision
  To:   Complex Field with 53 bits of precision
  Defn: 1.00000000000000 |--> 1.00000000000000
sage: i(RR(3.1))
3.10000000000000
```

A map from a multivariate polynomial ring to itself:

```python
sage: phi = R.hom([y, z, x^2]); phi
Ring endomorphism of Multivariate Polynomial Ring in x, y, z over Rational Field
  Defn: x |--> y
        y |--> z
        z |--> x^2
sage: phi(x + y + z)
x^2 + y + z
```

An endomorphism of a quotient of a multivariate polynomial ring:

```python
sage: phi = R.hom([y, z, x**Integer(2)]); phi
Ring endomorphism of Multivariate Polynomial Ring in x, y, z over Rational Field
  Defn: x |--> y
        y |--> z
        z |--> x^2
sage: phi(x + y + z)
x^2 + y + z
```
sage: S.<a,b> = quo(R, ideal(1 + y^2))
sage: phi = S.hom([a^2, -b]); phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y
  over Rational Field by the ideal (y^2 + 1)
  Defn: a |--> a^2
     b |--> -b
sage: phi(b)
-b
sage: phi(a^2 + b^2)
a^4 - 1

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> R = PolynomialRing(QQ, names=('x', 'y'),); (x, y,) = R._first_ngens(2)
>>> S = quo(R, ideal(Integer(1) + y**Integer(2)), names=(a, b,)); (a, b,) = S._
  → first_ngens(2)
>>> phi = S.hom([a**Integer(2), -b]); phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y
  over Rational Field by the ideal (y^2 + 1)
  Defn: a |--> a^2
     b |--> -b
>>> phi(b)
-b
>>> phi(a**Integer(2) + b**Integer(2))
a^4 - 1

The reduction map from the integers to the integers modulo 8, viewed as a quotient ring:

sage: R = ZZ.quo(8*ZZ)
sage: pi = R.cover(); pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 8
  Defn: Natural quotient map
sage: pi.domain()
Integer Ring
sage: pi.codomain()
Ring of integers modulo 8
sage: pi(10)
2
sage: pi.lift()
Set-theoretic ring morphism:
  From: Ring of integers modulo 8
  To:   Integer Ring
  Defn: Choice of lifting map
sage: pi.lift(13)
5

>>> from sage.all import *
>>> R = ZZ.quo(Integer(8)*ZZ)
>>> pi = R.cover(); pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 8
  Defn: Natural quotient map
>>> pi.domain()
Integer Ring

```python
>>> pi.codomain()
Ring of integers modulo 8
```

```python
>>> pi(Integer(10))
2
```

```python
>>> pi.lift()
Set-theoretic ring morphism:
  From: Ring of integers modulo 8
  To:   Integer Ring
  Defn: Choice of lifting map
```

```python
>>> pi.lift(Integer(13))
5
```

Inclusion of \( GF(2) \) into \( GF(4, \ 'a' ) \):

```python
sage: # needs sage.rings.finite_rings
sage: k = GF(2)
sage: i = k.hom(GF(4, 'a'))
sage: i
Ring morphism:
  From: Finite Field of size 2
  To:   Finite Field in a of size 2^2
  Defn: 1 |--> 1

sage: i(0)
0
sage: a = i(1); a.parent()
Finite Field in a of size 2^2
```

```python
>>> from sage.all import *
```

```python
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(2))
```

```python
>>> i = k.hom(GF(Integer(4), 'a'))
```

```python
>>> i
Ring morphism:
  From: Finite Field of size 2
  To:   Finite Field in a of size 2^2
  Defn: 1 |--> 1

>>> i(Integer(0))
0
>>> a = i(Integer(1)); a.parent()
Finite Field in a of size 2^2
```

We next compose the inclusion with reduction from the integers to \( GF(2) \):

```python
sage: # needs sage.rings.finite_rings
sage: pi = ZZ.hom(k); pi
Natural morphism:
  From: Integer Ring
  To:   Finite Field

sage: f = i * pi; f
Composite map:
  From: Integer Ring
  To:   Finite Field in a of size 2^2
  Defn:  Natural morphism:
      From: Integer Ring
      To:   Finite Field
```

(continues on next page)
then

```
Ring morphism:
  From: Finite Field of size 2
  To:  Finite Field in a of size 2^2
  Defn: 1 |---> 1
```

```
sage: a = f(5); a
1
sage: a.parent()
Finite Field in a of size 2^2
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> pi = ZZ.hom(k); pi
Natural morphism:
  From: Integer Ring
  To:  Finite Field of size 2
>>> f = i * pi; f
Composite map:
  From: Integer Ring
  To:  Finite Field in a of size 2^2
  Defn:  Natural morphism:
        From: Integer Ring
        To:  Finite Field of size 2
        then
        Ring morphism:
        From: Finite Field of size 2
        To:  Finite Field in a of size 2^2
        Defn: 1 |---> 1
>>> a = f(Integer(5)); a
1
>>> a.parent()
Finite Field in a of size 2^2
```

Inclusion from \(\mathbb{Q}\) to the 3-adic field:

```
sage: # needs sage.rings.padics
sage: phi = QQ.hom(Qp(3, print_mode='series'))
sage: phi
Ring morphism:
  From: Rational Field
  To:  3-adic Field with capped relative precision 20
sage: phi.codomain()
3-adic Field with capped relative precision 20
sage: phi(394)
1 + 2*3 + 3^2 + 2*3^3 + 3^4 + 3^5 + O(3^20)
```

```
>>> from sage.all import *
>>> # needs sage.rings.padics
>>> phi = QQ.hom(Qp(Integer(3), print_mode='series'))
>>> phi
Ring morphism:
  From: Rational Field
  To:  3-adic Field with capped relative precision 20
>>> phi.codomain()
3-adic Field with capped relative precision 20
>>> phi(Integer(394))
(continues on next page)
```
An automorphism of a quotient of a univariate polynomial ring:

```sage
# needs sage.libs.pari
R.<x> = PolynomialRing(QQ)
s.<sqrt2> = R.quo(x^2 - 2)
sqrt2^2
2
sage: (3+sqrt2)^10
993054*sqrt2 + 1404491
sage: c = S.hom([-sqrt2])
c(1+sqrt2)
-sqrt2 + 1
```

Note that Sage verifies that the morphism is valid:

```sage
(1 - sqrt2)^2
-2*sqrt2 + 3
sage: c = S.hom([1 - sqrt2])  # this is not valid
Traceback (most recent call last):
... ValueError: relations do not all (canonically) map to 0
```

Endomorphism of power series ring:

```sage
R.<t> = PowerSeriesRing(QQ, default_prec=10); R
Power Series Ring in t over Rational Field
sage: f = R.hom([t^2]); f
```
Ring endomorphism of Power Series Ring in \( t \) over Rational Field

**Defn:** \( t \mapsto t^2 \)

```
sage: s = 1/(1 + t); s
1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)
sage: f(s)
1 - t^2 + t^4 - t^6 + t^8 - t^10 + t^12 - t^14 + t^16 - t^18 + O(t^20)
```
```python
sage: R.<t> = LaurentSeriesRing(QQ, 10)
sage: f = R.hom([t^3 + t]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t + t^3
sage: s = 2/t^2 + 1/(1 + t); s
2*t^-2 + 1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)
sage: f(s)
2*t^-2 - 3 - t + 7*t^2 - 2*t^3 - 5*t^4 - 4*t^5 + 16*t^6 - 9*t^7 + O(t^8)
sage: f = R.hom([t^3]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t^3
sage: f(s)
2*t^-6 + 1 - t^3 + t^6 - t^9 + t^12 - t^15 + t^18 - t^21 + t^24 - t^27 + O(t^30)
```

Note that the homomorphism must result in a converging Laurent series, so the valuation of the image of the generator must be positive:

```python
sage: R.hom([1/t])
Traceback (most recent call last):
... ValueError: relations do not all (canonically) map to 0 under map determined by images of generators
```

Complex conjugation on cyclotomic fields:

```python
>>> from sage.all import *

>>> R = LaurentSeriesRing(QQ, Integer(10), names=('t',)); (t,) = R._first_ngens(1)

>>> f = R.hom([t**Integer(3) + t]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t + t^3

>>> s = Integer(2)/t**Integer(2) + Integer(1)/(Integer(1) + t); s
2*t^-2 + 1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)

>>> f(s)
2*t^-2 - 3 - t + 7*t^2 - 2*t^3 - 5*t^4 - 4*t^5 + 16*t^6 - 9*t^7 + O(t^8)

>>> f = R.hom([t**Integer(3)]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t^3

>>> f(s)
2*t^-6 + 1 - t^3 + t^6 - t^9 + t^12 - t^15 + t^18 - t^21 + t^24 - t^27 + O(t^30)
```
sage: # needs sage.rings.number_field
sage: K.<zeta7> = CyclotomicField(7)
sage: c = K.hom([1/zeta7]); c
Ring endomorphism of Cyclotomic Field of order 7 and degree 6
  Defn: zeta7 |--> -zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - zeta7 - 1
sage: a = (1+zeta7)^5; a
zeta7^5 + 5*zeta7^4 + 10*zeta7^3 + 10*zeta7^2 + 5*zeta7 + 1
sage: c(a)
5*zeta7^5 + 5*zeta7^4 - 4*zeta7^2 - 5*zeta7 - 4
sage: c(zeta7 + 1/zeta7)
# this element is obviously fixed by inversion
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1
sage: zeta7 + 1/zeta7
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(7), names=('zeta7',)); (zeta7,) = K._first_ngens(1)
>>> c = K.hom([Integer(1)/zeta7]); c
Ring endomorphism of Cyclotomic Field of order 7 and degree 6
  Defn: zeta7 |--> -zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - zeta7 - 1
>>> a = (Integer(1)+zeta7)**Integer(5); a
zeta7^5 + 5*zeta7^4 + 10*zeta7^3 + 10*zeta7^2 + 5*zeta7 + 1
>>> c(a)
5*zeta7^5 + 5*zeta7^4 - 4*zeta7^2 - 5*zeta7 - 4
>>> c(zeta7 + Integer(1)/zeta7)
# this element is obviously fixed by inversion
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1
>>> zeta7 + Integer(1)/zeta7
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1

Embedding a number field into the reals:

sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<beta> = NumberField(x^3 - 2)
sage: alpha = RR(2)^(1/3); alpha
1.25992104989487
sage: i = K.hom([alpha],check=False); i
Ring morphism:
  From: Number Field in beta with defining polynomial x^3 - 2
  To:   Real Field with 53 bits of precision
  Defn: beta |--> 1.25992104989487
sage: i(beta)
1.25992104989487
sage: i(beta^3)
2.00000000000000
sage: i(beta^2 + 1)
2.58740105196820

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(3) - Integer(2), names=('beta',)); (beta,) = K._first_ngens(1)
>>> alpha = RR(Integer(2))**(Integer(1)/Integer(3)); alpha
1.25992104989487
>>> i = K.hom([alpha],check=False); i
Ring morphism:
  From: Number Field in beta with defining polynomial x**3 - 2
  To:   Real Field with 53 bits of precision
  Defn: beta |--> 1.25992104989487
sage: i(beta)
1.25992104989487
sage: i(beta^3)
2.00000000000000
sage: i(beta^2 + 1)
2.58740105196820

(continues on next page)
From: Number Field in beta with defining polynomial x^3 - 2
To: Real Field with 53 bits of precision
Defn: beta |--> 1.25992104989487

```python
>>> i(beta)
1.25992104989487
>>> i(beta**Integer(3))
2.00000000000000
>>> i(beta**Integer(2) + Integer(1))
2.58740105196820
```

An example from Jim Carlson:

```python
sage: K = QQ  # by the way :-)
sage: R.<a,b,c,d> = K[]; R
Multivariate Polynomial Ring in a, b, c, d over Rational Field
sage: S.<u> = K[]; S
Univariate Polynomial Ring in u over Rational Field
sage: f = R.hom([0,0,0,u], S); f
Ring morphism:
    From: Multivariate Polynomial Ring in a, b, c, d over Rational Field
    To:   Univariate Polynomial Ring in u over Rational Field
    Defn: a |--> 0
          b |--> 0
          c |--> 0
          d |--> u
sage: f(a + b + c + d)
u
sage: f((a+b+c+d)**Integer(2))
u^2
```

```python
from sage.all import *
```

```python
K = QQ  # by the way :-)
R = K[['a, b, c, d']]; (a, b, c, d,) = R._first_ngens(4); R
S = K['u']; (u,) = S._first_ngens(1); S
f = R.hom([Integer(0),Integer(0),Integer(0),u], S); f
Ring morphism:
    From: Multivariate Polynomial Ring in a, b, c, d over Rational Field
    To:   Univariate Polynomial Ring in u over Rational Field
    Defn: a |--> 0
          b |--> 0
          c |--> 0
          d |--> u
>>> f(a + b + c + d)
u
>>> f((a+b+c+d)**Integer(2))
u^2
```

```python
class sage.rings.morphism.FrobeniusEndomorphism_generic
    Bases: RingHomomorphism

    A class implementing Frobenius endomorphisms on rings of prime characteristic.

def power(self, n):
    """Return an integer n such that this endomorphism is the n-th power of the absolute (arithmetic) Frobenius."""
    
    EXAMPLES:
```

Chapter 3. Ring Morphisms
class sage.rings.morphism.RingHomomorphism
Bases: RingMap

Homomorphism of rings.

inverse()

Return the inverse of this ring homomorphism if it exists.

Raises a ZeroDivisionError if the inverse does not exist.

ALGORITHM:
By default, this computes a Gröbner basis of the ideal corresponding to the graph of the ring homomorphism.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = R.hom([2*t - 1], R)
sage: f.inverse()  # needs sage.libs.singular
Ring endomorphism of Univariate Polynomial Ring in t over Rational Field
Defn: t |--> 1/2*t + 1/2
```

The following non-linear homomorphism is not invertible, but it induces an isomorphism on a quotient ring:

```
sage: # needs sage.libs.singular
sage: R.<x,y,z> = QQ[]
sage: f = R.hom([y*z, x*z, x*y], R)
sage: f.inverse()  # needs sage.libs.singular
Traceback (most recent call last):
...
ZeroDivisionError: ring homomorphism not surjective
```
sage: f.is_injective()
True
sage: Q.<x,y,z> = R.quotient(x*y*z - 1)
sage: g = Q.hom([y*z, x*z, x*y], Q)
sage: g.inverse()
Ring endomorphism of Quotient of Multivariate Polynomial Ring
in x, y, z over Rational Field by the ideal (x*y*z - 1)
Defn: x |--> y*z
     y |--> x*z
     z |--> x*y

>>> from sage.all import *
>>>
# needs sage.libs.singular
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = R.hom([y*z, x*z, x*y], R)
>>> f.inverse()
Traceback (most recent call last):
  ... ZeroDivisionError: ring homomorphism not surjective
>>> f.is_injective()
True
>>> Q = R.quotient(x*y*z - Integer(1), names=(x, y, z,)); (x, y, z,) =
˓→Q._first_ngens(3)
>>> g = Q.hom([y*z, x*z, x*y], Q)
>>> g.inverse()
Ring endomorphism of Quotient of Multivariate Polynomial Ring
in x, y, z over Rational Field by the ideal (x*y*z - 1)
Defn: x |--> y*z
     y |--> x*z
     z |--> x*y

Homomorphisms over the integers are supported:

sage: S.<x,y> = ZZ[]
sage: f = S.hom([x + 2*y, x + 3*y], S)
sage: f.inverse()  # needs sage.libs.singular
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> 3*x - 2*y
     y |--> -x + y
sage: (f.inverse() * f).is_identity()  # needs sage.libs.singular
True

>>> from sage.all import *
>>>
# needs sage.libs.singular
>>> S = ZZ['x, y']; (x, y,) = S._first_ngens(2)
>>> f = S.hom([x + Integer(2)*y, x + Integer(3)*y], S)
>>> f.inverse()  # needs sage.libs.singular
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> 3*x - 2*y
     y |--> -x + y
>>> (f.inverse() * f).is_identity()  # needs sage.libs.singular
True

The following homomorphism is invertible over the rationals, but not over the integers:
This example by M. Nagata is a wild automorphism:

```python
sage: R.<x,y,z> = QQ[]
sage: sigma = R.hom([x - 2*y*(z*x+y^2) - z*(z*x+y^2)^2, y + z*(z*x+y^2), z], R)

sage: tau = sigma.inverse(); tau  # needs sage.libs.singular
Ring endomorphism of Multivariate Polynomial Ring in x, y, z over Rational Field
  Defn: x |--> -y^4*z - 2*x*y^2*z^2 - x^2*z^3 + 2*y^3 + 2*x*y*z + x
       y |--> -y^2*z - x*z^2 + y
       z |--> z

sage: (tau * sigma).is_identity()  # needs sage.libs.singular
True
```

```python
>>> from sage.all import *

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)

>>> sigma = R.hom([x - Integer(2)*y*(z*x+y^Integer(2)) - z*(z*x+y^Integer(2))^Integer(2),
...                     y + z*(z*x+y^Integer(2)), z], R)

>>> tau = sigma.inverse(); tau  # needs sage.libs.singular

>>> from sage.all import *

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)

>>> sigma = R.hom([x - Integer(2)*y*(z*x+y^Integer(2)) - z*(z*x+y^Integer(2))^Integer(2),
...                     y + z*(z*x+y^Integer(2)), z], R)

>>> tau = sigma.inverse(); tau  # needs sage.libs.singular

>>> from sage.all import *

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)

>>> sigma = R.hom([x - Integer(2)*y*(z*x+y^Integer(2)) - z*(z*x+y^Integer(2))^Integer(2),
...                     y + z*(z*x+y^Integer(2)), z], R)

>>> tau = sigma.inverse(); tau  # needs sage.libs.singular

>>> from sage.all import *

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)

>>> sigma = R.hom([x - Integer(2)*y*(z*x+y^Integer(2)) - z*(z*x+y^Integer(2))^Integer(2),
...                     y + z*(z*x+y^Integer(2)), z], R)

>>> tau = sigma.inverse(); tau  # needs sage.libs.singular
```

3.1. Homomorphisms of rings
We compute the triangular automorphism that converts moments to cumulants, as well as its inverse, using the moment generating function. The choice of a term ordering can have a great impact on the computation time of a Gröbner basis, so here we choose a weighted ordering such that the images of the generators are homogeneous polynomials.

```sage
d = 12
T = TermOrder('wdegrevlex', [1..d])
R = PolynomialRing(QQ, ["x%s" % j for j in (1..d)], order=T)
S.<t> = PowerSeriesRing(R)
egf = S([0] + list(R.gens())).ogf_to_egf().exp(prec=d+1)
phi = R.hom(egf.egf_to_ogf().list()[1:], R)
sage: all(p.is_homogeneous() for p in phi.im_gens())
# needs sage.libs.singular
True
sage: phi.inverse().im_gens()[5]
[1, -x1^2 + x2, 2*x1^3 - 3*x1*x2 + x3, -6*x1^4 + 12*x1^2*x2 - 3*x2^2 - 4*x1*x3 + x4, 24*x1^5 - 60*x1^3*x2 + 30*x1*x2^2 + 20*x1^2*x3 - 10*x2*x3 - 5*x1*x4 + x5]
sage: (phi.inverse() * phi).is_identity()
# needs sage.libs.singular
True
```
Automorphisms of number fields as well as Galois fields are supported:

```
sage: K.<zeta7> = CyclotomicField(7)  # needs sage.rings.number_field
sage: c = K.hom([1/zeta7])  # needs sage.rings.number_field
sage: (c.inverse() * c).is_identity()  # needs sage.libs.singular.is_identity
True

sage: F.<t> = GF(7^3)  # needs sage.rings.finite_rings
sage: f = F.hom(t^7, F)  # needs sage.rings.finite_rings
sage: (f.inverse() * f).is_identity()  # needs sage.libs.singular.is_identity
True

>>> from sage.all import *

>>> K = CyclotomicField(Integer(7), names=(zeta7,)); (zeta7,) = K._first_ngens(1)  # needs sage.rings.number_field
>>> c = K.hom([Integer(1)/zeta7])  # needs sage.rings.number_field
>>> (c.inverse() * c).is_identity()  # needs sage.libs.singular.is_identity
True

>>> F = GF(Integer(7)**Integer(3), names=(t,)); (t,) = F._first_ngens(1)  # needs sage.rings.finite_rings
>>> f = F.hom(t**Integer(7), F)  # needs sage.rings.finite_rings
>>> (f.inverse() * f).is_identity()  # needs sage.libs.singular.is_identity
True
```

An isomorphism between the algebraic torus and the circle over a number field:

```
sage: K.<i> = QuadraticField(-1)  # needs sage.libs.singular.is_identity
sage: A.<z,w> = K[z,w].quotient(z*w - 1)
sage: B.<x,y> = K[x,y].quotient(x^2 + y^2 - 1)
sage: f = A.hom([x + i*y, x - i*y], B)
sage: g = f.inverse()
sage: g.morphism_from_cover().im_gens()
[1/2*z + 1/2*w, (-1/2*i)*z + (1/2*i)*w]
sage: all(g(f(z)) == z for z in A.gens())
True

>>> from sage.all import *

>>> K = QuadraticField(-Integer(1), names=(i,)); (i,) = K._first_ngens(1)  # needs sage.rings.finite_rings
>>> A = K[z,w].quotient(z*w - 1, names=(z, w,)); (z, w,) = A._first_ngens(2)
>>> B = K[x,y].quotient(x^2 + y^2 - 1, names=(x, y,)); (x, y,) = B._first_ngens(2)
>>> f = A.hom([x + i*y, x - i*y], B)
>>> g = f.inverse()
```

(continues on next page)
**inverse_image(I)**

Return the inverse image of an ideal or an element in the codomain of this ring homomorphism.

**INPUT:**
- **I** – an ideal or element in the codomain

**OUTPUT:**
For an ideal \( I \) in the codomain, this returns the largest ideal in the domain whose image is contained in \( I \).

Given an element \( b \) in the codomain, this returns an arbitrary element \( a \) in the domain such that \( \text{self}(a) = b \) if one such exists. The element \( a \) is unique if this ring homomorphism is injective.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: S.<u,v> = QQ[]
sage: f = R.hom([u^2, u*v, v^2], S)
sage: I = S.ideal([u^6, u^5*v, u^4*v^2, u^3*v^3])
sage: J = f.inverse_image(I); J
Ideal (y^2 - x*z, x*y*z, x^2*z, x^2*y, x^3)
of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: f(J) == I
True
```

Under the above homomorphism, there exists an inverse image for every element that only involves monomials of even degree:

```python
sage: f.inverse_image(p) for p in [u^2, u^4, u*v + u^3*v^3])
Ideal (x, x^2, x*y*z + y)
sage: f.inverse_image(u*v^2)
Traceback (most recent call last):
  ... ValueError: element u*v^2 does not have preimage
```
The image of the inverse image ideal can be strictly smaller than the original ideal:

```python
>>> from sage.all import *
>>> # needs sage.libs.singular sage.rings.number_field
>>> S.<u,v> = QQ['u,v'].quotient('v^2 - 2')
>>> f = QuadraticField(2).hom([v], S)
>>> I = S.ideal(u + v)
>>> J = f.inverse_image(I)
>>> J.is_zero()
True
>>> f(J) < I
True
```

Fractional ideals are not yet fully supported:

```python
>>> from sage.all import *
>>> # needs sage.libs.singular sage.rings.number_field
>>> K.<a> = NumberField(QQ['x'](x^2+2))
>>> f = K.hom([-a], K)
>>> I = K.ideal([a + Integer(1)])
>>> f.inverse_image(I)  # needs sage.libs.singular
Traceback (most recent call last):
...  
NotImplementedError: inverse image not implemented...
```

### 3.1. Homomorphisms of rings

(continues on next page)
is_inverse()  
Return whether this ring homomorphism is bijective.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: R.hom([y*z, x*z, x*y], R).is_inverse()  # needs sage.libs.singular
False
sage: Q.<x,y,z> = R.quotient(x*y*z - Integer(1))
```

ALGORITHM:
By default, this computes a Gröbner basis of an ideal related to the graph of the ring homomorphism.

REFERENCES:
• Proposition 2.5.12 [DS2009]

is_surjective()  
Return whether this ring homomorphism is surjective.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: R.hom([y*z, x*z, x*y], R).is_surjective()  # needs sage.libs.singular
False
```

ALGORITHM:
By default, this requires the computation of a Gröbner basis.
False

```
sage: Q.<x,y,z> = R.quotient(x*y*z - 1)
                # needs sage.libs.singular
sage: R.hom([y*z, x*z, x*y], Q).is_surjective()       # needs sage.libs.singular
True
```

>>> from sage.all import *

```python
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> R.hom([y*z, x*z, x*y], R).is_surjective()        # needs sage.libs.singular
False
```  

```
>>> Q = R.quotient(x*y*z - Integer(1), names=('x', 'y', 'z')); (x, y, z,) = Q._first_ngens(3) # needs sage.libs.singular
>>> R.hom([y*z, x*z, x*y], Q).is_surjective()       # needs sage.libs.singular
True
```

**ALGORITHM:**

By default, this requires the computation of a Gröbner basis.

**kernel()**

Return the kernel ideal of this ring homomorphism.

**EXAMPLES:**

```
sage: A.<x,y> = QQ[]
sage: B.<t> = QQ[]
sage: f = A.hom([t^4, t^3 - t^2], B)
sage: f.kernel()  # needs sage.libs.singular
Ideal (y^4 - x^3 + 4*x^2*y - 2*x*y^2 + x^2)
```

```
>>> from sage.all import *

```python
>>> A = QQ['x, y']; (x, y,) = A._first_ngens(2)
>>> B = QQ['t']; (t,) = B._first_ngens(1)
>>> f = A.hom([t**Integer(4), t**Integer(3) - t**Integer(2)], B)
>>> f.kernel() # needs sage.libs.singular
Ideal (y^4 - x^3 + 4*x^2*y - 2*x*y^2 + x^2)
```

We express a Veronese subring of a polynomial ring as a quotient ring:

```
sage: A.<a,b,c,d> = QQ[]
sage: B.<u,v> = QQ[]
sage: f = A.hom([u^3, u^2*v, u*v^2, v^3], B)
sage: f.kernel() == A.ideal(matrix.hankel([a, b, c], [d]).minors(2))  # needs sage.libs.singular
True
```  

```
sage: Q = A.quotient(f.kernel())
                # needs sage.libs.singular
sage: Q.hom(f.im_gens(), B).is_injective()  # needs sage.libs.singular
True
```

3.1. Homomorphisms of rings
>>> from sage.all import *
>>> A = QQ['a, b, c, d']; (a, b, c, d,) = A._first_ngens(4)
>>> B = QQ['u, v']; (u, v,) = B._first_ngens(2)
>>> f = A.hom([u**Integer(3), u**Integer(2)*v, u*v**Integer(2),
→ v**Integer(3)], B)
>>> f.kernel() == A.ideal(matrix.hankel([a, b, c], [d]).minors(Integer(2)))  # needs sage.libs.singular
True

>> Q = A.quotient(f.kernel())  # needs sage.libs.singular
>> Q.hom(f.im_gens(), B).is_injective()  # needs sage.libs.singular
True

The Steiner-Roman surface:

sage: R.<x,y,z> = QQ[

sage: S = R.quotient(x^2 + y^2 + z^2 - 1)

sage: f = R.hom([x*y, x*z, y*z], S)  # needs sage.libs.singular

sage: f.kernel()  # needs sage.libs.singular
Ideal (x^2*y^2 + x^2*z^2 + y^2*z^2 - x*y*z)
of Multivariate Polynomial Ring in x, y, z over Rational Field

lift (x=None)

Return a lifting map associated to this homomorphism, if it has been defined.

If x is not None, return the value of the lift morphism on x.

EXAMPLES:

sage: R.<x,y> = QQ[

sage: f = R.hom([x,x])

sage: f(x+y)
2*x

sage: f.lift()  # Traceback (most recent call last):
...

ValueError: no lift map defined

sage: g = R.hom(R)

sage: f._set_lift(g)

sage: f.lift() == g
True

sage: f.lift(x)

x
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> f = R.hom([x,x])
>>> f(x+y)
2*x
>>> f.lift()
Traceback (most recent call last):
  ... ValueError: no lift map defined
>>> g = R.hom(R)
>>> f._set_lift(g)
>>> f.lift() == g
True
>>> f.lift(x)
x

pushforward(I)

Returns the pushforward of the ideal \( I \) under this ring homomorphism.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]; S.<xx,yy> = R.quo([x^2, y^2]); f = S.cover()  # needs sage.libs.singular
sage: f.pushforward(R.ideal([x, 3*x + x*y + y^2]))  # needs sage.libs.singular
Ideal (xx, xx*yy + 3*xx) of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2, y^2)
```

```python
sage: R = QQ['x, y']; (x, y,) = R._first_ngens(2); S = R.quo([x**Integer(2), y**Integer(2)], names=(xx, yy,)); (xx, yy,) = S._first_ngens(2); f = S.\rightarrow cover()  # needs sage.libs.singular
sage: f.pushforward(R.ideal([x, Integer(3)*x + x*y + y**Integer(2)]))  # needs sage.libs.singular
Ideal (xx, xx*yy + 3*xx) of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2, y^2)
```

class sage.rings.morphism.RingHomomorphism_cover

Bases: RingHomomorphism

A homomorphism induced by quotienting a ring out by an ideal.

EXAMPLES:

```python
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quo(x^2 + y^2)  # needs sage.libs.singular
sage: phi = S.cover(); phi  # needs sage.libs.singular
Ring morphism:
  From: Multivariate Polynomial Ring in x, y over Rational Field
  To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
  Defn: Natural quotient map
sage: phi(x + y)  # needs sage.libs.singular
a + b
```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y')); (x, y,) = R._first_ngens(2)
>>> S = R.quo(x**Integer(2) + y**Integer(2), names=('a', 'b')); (a, b,) = S._first_ngens(2)  # needs sage.libs.singular
>>> phi = S.cover(); phi

Ring morphism:
    From: Multivariate Polynomial Ring in x, y over Rational Field
    To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field
          by the ideal (x^2 + y^2)
    Defn: Natural quotient map

Return the kernel of this covering morphism, which is the ideal that was quotiented out by.

```sage
sage: f = Zmod(6).cover()
sage: f.kernel()
Principal ideal (6) of Integer Ring
```

```sage
>>> from sage.all import *
>>> R = QQ[x, y]; (x, y,) = R._first_ngens(2)
>>> S = QQ[z]; (z,) = S._first_ngens(1)
>>> f = R.hom([Integer(2)*z,Integer(3)*z],S)

Now we construct polynomial rings based on R and S, and let f act on the coefficients:
```sage
sage: PR.<t> = R[]
sage: PS = S['t']
sage: Pf = PR.hom(f,PS)
sage: Pf
```
Ring morphism:
From: Univariate Polynomial Ring in $t$
   over Multivariate Polynomial Ring in $x, y$ over Rational Field
To:   Univariate Polynomial Ring in $t$
   over Univariate Polynomial Ring in $z$ over Rational Field
Defn: Induced from base ring by
      
      Ring morphism:
      From: Multivariate Polynomial Ring in $x, y$ over Rational Field
      To:   Univariate Polynomial Ring in $z$ over Rational Field
      Defn: $x \mapsto 2z$
      $y \mapsto 3z$

```
sage: p = (x - 4*y + 1/13)*t^2 + (1/2*x^2 - 1/3*y^2)*t + 2*y^2 + x
sage: Pf(p)
(-10*z + 1/13)*t^2 - z^2*t + 18*z^2 + 2*z
```

Similarly, we can construct the induced homomorphism on a matrix ring over our polynomial rings:

```
sage: # needs sage.modules
sage: MR = MatrixSpace(R, 2, 2)
sage: MS = MatrixSpace(S, 2, 2)
sage: M = MR([x^2 + 1/7*x*y - y^2, -1/2*y^2 + 2*y + 1/6,
             4*x^2 - 14*x, 1/2*y^2 + 13/4*x - 2/11*y])
sage: Mf = MR.hom(f, MS)
sage: Mf
```

```
Ring morphism:
From: Full MatrixSpace of 2 by 2 dense matrices
   over Multivariate Polynomial Ring in $x, y$ over Rational Field
To:   Full MatrixSpace of 2 by 2 dense matrices
   over Univariate Polynomial Ring in $z$ over Rational Field
Defn: Induced from base ring by
      
      Ring morphism:
      From: Multivariate Polynomial Ring in $x, y$ over Rational Field
      To:   Univariate Polynomial Ring in $z$ over Rational Field
      Defn: $x \mapsto 2z$
```
\[
y \mapsto 3z
\]
\[
\text{sage: } Mf(M)
\]
\[
\begin{bmatrix}
-29/7z^2 - 9/2z^2 + 6z + 1/6 \\
16z^2 - 28z + 9/2z^2 + 131/22z
\end{bmatrix}
\]

\[\text{The construction of induced homomorphisms is recursive, and so we have:}\]
\[
\text{sage: } \# \text{ needs sage.modules}
\]
\[
\text{sage: } \text{MPR = MatrixSpace(PR, 2)}
\]
\[
\text{sage: } \text{MPS = MatrixSpace(PS, 2)}
\]
\[
\text{sage: } M = \text{MPR}\left(\begin{array}{c}
-1/7x^2 + y^2 + 5x^2 + 3x^2 + x^2 \\
-1/3y^2 + 1/3x^2 + y^2 + 5/2y + 1/4
\end{array}\right)
\]
\[
\text{sage: } \text{MPf = MPR.hom(f, MPS)}; \text{MPf}
\]
\[
\text{Ring morphism:}
\]
\[
\text{From: Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in t over Multivariate Polynomial Ring in x, y over Rational Field}
\]
\[
\text{To: Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in t over Univariate Polynomial Ring in z over Rational Field}
\]
\[
\text{Defn: Induced from base ring by}
\]
\[
\text{Ring morphism:}
\]
\[
\text{From: Univariate Polynomial Ring in t over Multivariate Polynomial Ring in x, y over Rational Field}
\]
\[
\text{To: Univariate Polynomial Ring in t over Univariate Polynomial Ring in z over Rational Field}
\]
\[
\text{Defn: Induced from base ring by}
\]
\[
\text{Ring morphism:}
\]
\[
\text{From: Multivariate Polynomial Ring in x, y over Rational Field}
\]
\[
\text{To: Univariate Polynomial Ring in z over Rational Field}
\]
\[
\text{Defn: x |--> 2z}
\]
```sage
y |--> 3*z

sage: MPf(M)
[ z*t^2 + 58*t - 6*z^2 (-6/7*z^2 - 1/20*z)*t^2 + 29*z^2*t +...
→6*z]
[ (-z + 1)*t^2 + 11*z^2 + 15/2*z + 1/4 (20*z + 1)*t^...
→2]
```

```sage
>>> from sage.all import *
>>> # needs sage.modules
>>> MPR = MatrixSpace(PR, Integer(2))
>>> MPS = MatrixSpace(PS, Integer(2))
>>> M = MPR([(-x + y)*t**Integer(2) + Integer(58)*t - Integer(3)*x**Integer(2) +...
→x*y,
... (- Integer(1)/Integer(7)*x*y - Integer(1)/
→Integer(40)*x)*t**Integer(2) + (Integer(5)*x**Integer(2) + y**Integer(2))*t +...
→Integer(2)*y,
... (- Integer(1)/Integer(3)*y + Integer(1))/t**Integer(2) + Integer(1)/
→Integer(3)*x*y + y**Integer(2) + Integer(5)/Integer(2)*y + Integer(1)/
→Integer(4),
... (x + Integer(6)*y + Integer(1))*t**Integer(2))])
>>> MPf = MPR.hom(f, MPS); MPf
Ring morphism:
   From: Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial
   Ring in t over Multivariate Polynomial Ring in x, y over Rational Field
   To:   Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial
   Ring in t over Univariate Polynomial Ring in z over Rational Field
   Defn: Induced from base ring by
   Ring morphism:
      From: Univariate Polynomial Ring in t
      over Multivariate Polynomial Ring in x, y over Rational Field
      To:   Univariate Polynomial Ring in t
          over Univariate Polynomial Ring in z over Rational Field
      Defn: Induced from base ring by
      Ring morphism:
         From: Multivariate Polynomial Ring in x, y over Rational Field
         To:   Univariate Polynomial Ring in z over Rational Field
         Defn: x |--> 2*z
                      y |--> 3*z
```

```sage
inverse()
```

Return the inverse of this ring homomorphism if the underlying homomorphism of the base ring is invertible.

**EXAMPLES:**

```sage
sage: R.<x,y> = QQ[]
sage: S.<a,b> = QQ[]
sage: f = R.hom([a + b, a - b], S)
sage: PR.<t> = R[]
sage: PS = S['t']
sage: Pf = PR.hom(f, PS)
sage: Pf.inverse()
```

(continues on next page)
Ring morphism:
From: Univariate Polynomial Ring in \( t \) over Multivariate Polynomial Ring in \( a, b \) over \( \mathbb{Q} \)
To: Univariate Polynomial Ring in \( t \) over Multivariate Polynomial Ring in \( x, y \) over \( \mathbb{Q} \)
Defn: Induced from base ring by

Ring morphism:
From: Multivariate Polynomial Ring in \( a, b \) over \( \mathbb{Q} \)
To: Multivariate Polynomial Ring in \( x, y \) over \( \mathbb{Q} \)
Defn: \( a \mapsto \frac{1}{2}x + \frac{1}{2}y \)
\( b \mapsto \frac{1}{2}x - \frac{1}{2}y \)

```
sage: Pf.inverse()(Pf(x*t^2 + y*t))
```

```
x*t^2 + y*t
```

```
from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> S = QQ['a, b']; (a, b,) = S._first_ngens(2)
>>> f = R.hom([a + b, a - b], S)
>>> PR = R['t']; (t,) = PR._first_ngens(1)
>>> PS = S['t']
>>> Pf = PR.hom(f, PS)
>>> Pf.inverse()    # needs sage.libs.singular
```

```
x*t^2 + y*t
```

```
underlying_map()
```

Return the underlying homomorphism of the base ring.

**EXAMPLES:**

```
sage: # needs sage.modules
sage: R.<x,y> = QQ[]

sage: S.<z> = QQ[]

sage: f = R.hom([2*z, 3*z], S)

sage: MR = MatrixSpace(R, 2)

sage: MS = MatrixSpace(S, 2)

sage: g = MR.hom(f, MS)

sage: g.underlying_map() == f
True
```
```python
>>> from sage.all import *
>>> # needs sage.modules
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> S = QQ['z']; (z,) = S._first_ngens(1)
>>> f = R.hom([Integer(2)*z, Integer(3)*z], S)
>>> MR = MatrixSpace(R, Integer(2))
>>> MS = MatrixSpace(S, Integer(2))
>>> g = MR.hom(f, MS)
>>> g.underlying_map() == f
True
```

```python
class sage.rings.morphism.RingHomomorphism_from_fraction_field
Bases: RingHomomorphism
Morphisms between fraction fields.

inverse()
Return the inverse of this ring homomorphism if it exists.

EXAMPLES:
```
sage: S.<x> = QQ[]
sage: f = S.hom([2*x - 1])
sage: g = f.extend_to_fraction_field()
#-- needs sage.libs.singular
sage: g.inverse()
#-- needs sage.libs.singular
Ring endomorphism of Fraction Field of Univariate Polynomial Ring
in x over Rational Field
Defn: x |--> 1/2*x + 1/2
```
```
```python
class sage.rings.morphism.RingHomomorphism_from_quotient
Bases: RingHomomorphism
A ring homomorphism with domain a generic quotient ring.

INPUT:
- parent -- a ring homset Hom(R, S)
- phi -- a ring homomorphism C --> S, where C is the domain of R.cover()

OUTPUT: a ring homomorphism

The domain R is a quotient object C -> R, and R.cover() is the ring homomorphism \( \varphi : C \to R \). The condition on the elements im_gens of S is that they define a homomorphism C -> S such that each generator of the kernel of \( \varphi \) maps to 0.

EXAMPLES:
```
sage: # needs sage.libs.singular
sage: R.<x, y, z> = PolynomialRing(QQ, 3)
sage: S.<a, b, c> = R.quo(x^3 + y^3 + z^3)
sage: phi = S.hom([b, c, a]); phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y, z
over Rational Field by the ideal (x^3 + y^3 + z^3)
  Defn: a |--> b
  b |--> c
  c |--> a
sage: phi(a + b + c)
a + b + c
sage: loads(dumps(phi)) == phi
True

Validity of the homomorphism is determined, when possible, and a \texttt{TypeError} is raised if there is no homomorphism sending the generators to the given images:

sage: S.hom([b^2, c^2, a^2])  # needs sage.libs.singular
Traceback (most recent call last):
...
ValueError: relations do not all (canonically) map to 0
under map determined by images of generators

\texttt{morphism_from_cover}()

Underlying morphism used to define this quotient map, i.e., the morphism from the cover of the domain.

\textbf{EXAMPLES:}

sage: R.<x,y> = QQ[]; S.<xx,yy> = R.quo([x^2, y^2])  # needs sage.libs.singular

(continues on next page)
```python
sage: S.hom([yy,xx]).morphism_from_cover()  # needs sage.libs.singular
Ring morphism:
  From: Multivariate Polynomial Ring in x, y over Rational Field
  To:   Quotient of Multivariate Polynomial Ring in x, y
         over Rational Field by the ideal (x^2, y^2)
  Defn: x |--> yy
           y |--> xx
```

```python
>>> from sage.all import *

>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2); S = R.quo([x**Integer(2), y**Integer(2)], names=('xx', 'yy',)); (xx, yy,) = S._first_ngens(2)  # needs sage.libs.singular

>>> S.hom([yy,xx]).morphism_from_cover()  # needs sage.libs.singular
```

```python
class sage.rings.morphism.RingHomomorphism_im_gens

Bases: RingHomomorphism

A ring homomorphism determined by the images of generators.

base_map()

Return the map on the base ring that is part of the defining data for this morphism. May return None if a coercion is used.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: R.<x> = ZZ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: cc = K.hom([-i])
sage: S.<y> = K[]
sage: phi = S.hom([y^2], base_map=cc)

sage: phi
Ring endomorphism of Univariate Polynomial Ring in y over Number Field in i with defining polynomial x^2 + 1
  Defn: y |--> y^2
          with map of base ring
sage: phi(y)
y^2
sage: phi(i*y)
-i*y^2
sage: phi.base_map()
Composite map:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Univariate Polynomial Ring in y over Number Field in i
        with defining polynomial x^2 + 1
  Defn: Ring endomorphism of Number Field in i with defining polynomial x^2 + 1
        ↦ i
           Defn: i |--> -i
```

(continues on next page)
Polynomial base injection morphism:
From: Number Field in i with defining polynomial \( x^2 + 1 \)
To: Univariate Polynomial Ring in y over Number Field in i
with defining polynomial \( x^2 + 1 \)

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(2) + Integer(1), names=('i',)); (i,) = K._
˓→first_ngens(1)
>>> cc = K.hom([-i])
>>> S = K['y']; (y,) = S._first_ngens(1)
>>> phi = S.hom([y**Integer(2)], base_map=cc)
>>> phi
Ring endomorphism of Univariate Polynomial Ring in y
over Number Field in i with defining polynomial \( x^2 + 1 \)
Defn: y |--> y^2
with map of base ring
```  

```python
>>> phi(y)
y^2
>>> phi(i*y)
-i*y^2
```  

```python
>>> phi.base_map()
Composite map:
From: Number Field in i with defining polynomial \( x^2 + 1 \)
To: Univariate Polynomial Ring in y over Number Field in i
with defining polynomial \( x^2 + 1 \)
Defn: Ring endomorphism of Number Field in i with defining polynomial \( x^2 \)
\( \rightarrow \) 1
  Defn: i |--> -i
then
Polynomial base injection morphism:
From: Number Field in i with defining polynomial \( x^2 + 1 \)
To: Univariate Polynomial Ring in y over Number Field in i
with defining polynomial \( x^2 + 1 \)
```  

```python
im_gens()
```

Return the images of the generators of the domain.

OUTPUT:

- list -- a copy of the list of gens (it is safe to change this)

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: f = R.hom([x, x + y])
sage: f.im_gens()
[x, x + y]
```  

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R = QQ('x', 'y'); (x, y) = R._first_ngens(2)
>>> f = R.hom([x, x + y])
>>> f.im_gens()
[x, x + y]
```  

We verify that the returned list of images of gens is a copy, so changing it doesn’t change \( f \):
sage: f.im_gens()[0] = 5
sage: f.im_gens()
[x, x + y]

>>> from sage.all import *
>>> f.im_gens()[Integer(0)] = Integer(5)
>>> f.im_gens()
[x, x + y]

class sage.rings.morphism.RingMap
Bases: Morphism

Set-theoretic map between rings.

class sage.rings.morphism.RingMap_lift
Bases: RingMap

Given rings \( R \) and \( S \) such that for any \( x \in R \) the function \( x.\text{lift}() \) is an element that naturally coerces to \( S \), this returns the set-theoretic ring map \( R \to S \) sending \( x \) to \( x.\text{lift}() \).

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: S.<xbar,ybar> = R.quo( (x^2 + y^2, y) )
# needs sage.libs.singular
sage: S.lift()
# needs sage.libs.singular
Set-theoretic ring morphism:
From: Quotient of Multivariate Polynomial Ring in x, y
       over Rational Field by the ideal (x^2 + y^2, y)
To:   Multivariate Polynomial Ring in x, y over Rational Field
Defn: Choice of lifting map
sage: S.lift() == 0
False

>>> from sage.all import *
>>> R = QQ[\'x, y\']; (x, y,) = R._first_ngens(2)
>>> S = R.quo( (x**Integer(2) + y**Integer(2), y) , names=('xbar', 'ybar'),);
\# needs sage.libs.singular
>>> S.lift()
\# needs sage.libs.singular
Set-theoretic ring morphism:
From: Quotient of Multivariate Polynomial Ring in x, y
       over Rational Field by the ideal (x^2 + y^2, y)
To:   Multivariate Polynomial Ring in x, y over Rational Field
Defn: Choice of lifting map
>>> S.lift() == Integer(0)
\# needs sage.libs.singular
False

Since Issue #11068, it is possible to create quotient rings of non-commutative rings by two-sided ideals. It was needed to modify \texttt{RingMap\_lift} so that rings can be accepted that are no instances of \texttt{sage.rings.ring\_RING}, as in the following example:

sage: # needs sage.modules sage.rings.finite_rings
sage: MS = MatrixSpace(GF(5), 2, 2)

(continues on next page)
3.2 Space of homomorphisms between two rings

sage.rings.homset.RingHomset \((R, S, \text{category}={None})\)

Construct a space of homomorphisms between the rings \(R\) and \(S\).

For more on homsets, see \(\text{Hom}()\).

EXAMPLES:

```python
sage: Hom(ZZ, QQ) # indirect doctest
Set of Homomorphisms from Integer Ring to Rational Field
```

class sage.rings.homset.RingHomset_generic\((R, S, \text{category}={None})\)

Bases: HomsetWithBase

A generic space of homomorphisms between two rings.

EXAMPLES:

```python
sage: Hom(ZZ, QQ) # indirect doctest
Set of Homomorphisms from Integer Ring to Rational Field
```

Element

alias of RingHomomorphism
has_coerce_map_from(x)

The default for coercion maps between ring homomorphism spaces is very restrictive (until more implementation work is done).

Currently this checks if the domains and the codomains are equal.

EXAMPLES:

```sage
H = Hom(ZZ, QQ)
sage: H = Hom(QQ, ZZ)
sage: H.has_coerce_map_from(H2)
False
```

natural_map()

Returns the natural map from the domain to the codomain.

The natural map is the coercion map from the domain ring to the codomain ring.

EXAMPLES:

```sage
H = Hom(ZZ, QQ)
sage: H.natural_map()
Natural morphism:
  From: Integer Ring
  To: Rational Field
```

zero()

Return the zero element of this homset.

EXAMPLES:

Since a ring homomorphism maps 1 to 1, there can only be a zero morphism when mapping to the trivial ring:

```sage
Hom(ZZ, Zmod(1)).zero()
```

(continues on next page)
Ring morphism:
From: Integer Ring
To: Ring of integers modulo 1
Defn: 1 |--> 0
>>> Hom(ZZ, Zmod(Integer(2))).zero()
Traceback (most recent call last):
... ValueError: homset has no zero element

class sage.rings.homset.RingHomset_quo_ring(R, S, category=None)
Bases: RingHomset_generic

Space of ring homomorphisms where the domain is a (formal) quotient ring.

EXAMPLES:

sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quotient(x^2 + y^2)
# needs sage.libs.singular
sage: phi = S.hom([b,a]); phi
# needs sage.libs.singular
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y
over Rational Field by the ideal (x^2 + y^2)
Defn: a |--> b
       b |--> a
sage: phi(a)
# needs sage.libs.singular
b
sage: phi(b)
# needs sage.libs.singular
a

>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = R._first_
   ngens(2)
>>> S = R.quotient(x**Integer(2) + y**Integer(2), names=('a', 'b',)); (a, b,) = S.
   _first_ngens(2) # needs sage.libs.singular
>>> phi = S.hom([b,a]); phi
# needs sage.libs.singular
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y
over Rational Field by the ideal (x^2 + y^2)
Defn: a |--> b
       b |--> a
>>> phi(a)
# needs sage.libs.singular
b
>>> phi(b)
# needs sage.libs.singular
a

Element
alias of RingHomomorphism_from_quotient

sage.rings.homset.is_RingHomset(H)
Return True if H is a space of homomorphisms between two rings.

EXAMPLES:
sage: from sage.rings.homset import is_RingHomset as is_RH
sage: is_RH(Hom(ZZ, QQ))
doctest:warning...
DeprecationWarning: the function is_RingHomset is deprecated;
use 'isinstance(..., RingHomset_generic)' instead
See https://github.com/sagemath/sage/issues/37922 for details.
True
sage: is_RH(ZZ)
False
sage: is_RH(Hom(RR, CC))
# needs sage.rings.real_mpfr
True
sage: is_RH(Hom(FreeModule(ZZ,1), FreeModule(QQ,1)))
# needs sage.modules
False

>>> from sage.all import *
>>> from sage.rings.homset import is_RingHomset as is_RH
>>> is_RH(Hom(ZZ, QQ))
doctest:warning...
DeprecationWarning: the function is_RingHomset is deprecated;
use 'isinstance(..., RingHomset_generic)' instead
See https://github.com/sagemath/sage/issues/37922 for details.
True
>>> is_RH(ZZ)
False
>>> is_RH(Hom(RR, CC))
# needs sage.rings.real_mpfr
True
>>> is_RH(Hom(FreeModule(ZZ,Integer(1)), FreeModule(QQ,Integer(1))))
# needs sage.modules
False

3.2. Space of homomorphisms between two rings
4.1 Quotient Rings

AUTHORS:

- William Stein
- Simon King (2011-04): Put it into the category framework, use the new coercion model.

Todo: The following skipped tests should be removed once Issue #13999 is fixed:

```python
sage: TestSuite(S).run(skip=['_test_nonzero_equal', '_test_elements', '_test_zero'])
```

In Issue #11068, non-commutative quotient rings \( R/I \) were implemented. The only requirement is that the two-sided ideal \( I \) provides a `reduce` method so that \( I.\text{reduce}(x) \) is the normal form of an element \( x \) with respect to \( I \) (i.e., we have \( I.\text{reduce}(x) == I.\text{reduce}(y) \) if \( x - y \in I \), and \( x - I.\text{reduce}(x) \in I \)). Here is a toy example:

```python
sage: from sage.rings.noncommutative_ideals import Ideal_nc
sage: from itertools import product
sage: class PowerIdeal(Ideal_nc):
  ....:  def __init__(self, R, n):
  ....:      self._power = n
  ....:      Ideal_nc.__init__(self, R, [R.prod(m) for m in product(R.gens(), repeat=n)])
  ....:  def reduce(self,x):
  ....:      R = self.ring()
  ....:      return add([c*R(m) for m,c in x if len(m)<self._power],R(0))
sage: F.<x,y,z> = FreeAlgebra(QQ, 3)
sage: I3 = PowerIdeal(F,3); I3
# needs sage.combinat sage.modules
Twosided Ideal (x^3, x^2*y, x^2*z, x*y*x, x*y*z, x*z*x, x*z*y, x*z^2, y*x^2, y*x*y, y*x*z, y^2*x, y^2*z, y*z*x, y*z*y, y*z^2, z*x^2, z*x*y, z*x*z, z*y*x, z*y^2, z*y*z, z^2*x, z^2*y, z^2*z) of Free Algebra on 3 generators (x, y, z) over Rational Field
```
Free algebras have a custom quotient method that serves at creating finite dimensional quotients defined by multiplication matrices. We are bypassing it, so that we obtain the default quotient:

```python
sage: # needs sage.combinat sage.modules
sage: Q3.<a,b,c> = F.quotient(I3)
sage: Q3
Quotient of Free Algebra on 3 generators (x, y, z) over Rational Field by
the ideal (x^3, x^2*y, x^2*z, x*y*x, x*y*z, x*z*x, x*z*y, x*z^2, y*x^2, y*x*y, y*x*z, y^2*x, y^2*z, y*z*x, y*z*y, y*z^2, z*x^2, z*x*y, z*x*z, z*y*x, z*y*z, z^2*x, z^2*y, z^2) of
Free Algebra on 3 generators (x, y, z) over Rational Field
sage: (a+b+Integer(2))**Integer(4)
16 + 32*a + 32*b + 24*a^2 + 24*a*b + 24*b*a + 24*b^2
sage: Q3.is_commutative()
False
```

Even though $Q_3$ is not commutative, there is commutativity for products of degree three:

```python
sage: a*(b*c)-(b*c)*a==F.zero()  # needs sage.combinat sage.modules
True
```

108 Chapter 4. Quotient Rings
If we quotient out all terms of degree two then of course the resulting quotient ring is commutative:

```
sage: # needs sage.combinat sage.modules
sage: I2 = PowerIdeal(F,2); I2
Twosided Ideal (x^2, x*y, x*z, y*x, y^2, y*z, z*x, z*y, z^2) of Free Algebra
on 3 generators (x, y, z) over Rational Field
sage: Q2.<a,b,c> = F.quotient(I2)
sage: Q2.is_commutative()
True
sage: (a+b+2)^4
16 + 32*a + 32*b
```

Since Issue #7797, there is an implementation of free algebras based on Singular’s implementation of the Letterplace Algebra. Our letterplace wrapper allows to provide the above toy example more easily:

```
sage: # needs sage.combinat sage.libs.singular sage.modules
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: Q3 = F.quo(F*[F.prod(m) for m in product(F.gens(), repeat=3)]*F)
sage: Q3
Quotient of Free Associative Unital Algebra on 3 generators (x, y, z)
over Rational Field by the ideal (x*x*x, x*x*y, x*x*z, x*y*x, x*y*y, x*y*z, x*z*x, x*z*y, x*z*z, y*x*x, y*x*y, y*x*z, y*y*x, y*y*y, y*y*z, y*z*x, y*z*y, y*z*z, z*x*x, z*x*y, z*x*z, z*y*x, z*y*y, z*y*z, z*z*x, z*z*y, z*z*z)
sage: Q3.0*Q3.1 - Q3.1*Q3.0
xbar*ybar - ybar*xbar
sage: Q3.0*(Q3.1*Q3.2) - (Q3.1*Q3.2)*Q3.0
0
sage: Q2 = F.quo(F*[F.prod(m) for m in product(F.gens(), repeat=Integer(2))]*F)
sage: Q2.is_commutative()
True
```

(continues on next page)
sage.rings.quotient_ring.**QuotientRing**(R, I, names=\textit{None}, \*\*\textit{kwds})

Creates a quotient ring of the ring \( R \) by the twosided ideal \( I \).

Variables are labeled by \textit{names} (if the quotient ring is a quotient of a polynomial ring). If \textit{names} isn’t given, ‘bar’ will be appended to the variable names in \( R \).

**INPUT:**

• \( R \) – a ring.

• \( I \) – a twosided ideal of \( R \).

• \textit{names} – (optional) a list of strings to be used as names for the variables in the quotient ring \( R/I \).

• further named arguments that will be passed to the constructor of the quotient ring instance.

**OUTPUT:** \( R/I \) - the quotient ring \( R \) mod the ideal \( I \)

**ASSUMPTION:**

\( I \) has a method \( I.\text{reduce}(x) \) returning the normal form of elements \( x \in R \). In other words, it is required that 
\( I.\text{reduce}(x)==I.\text{reduce}(y) \iff x-y \in I, \text{and } x-I.\text{reduce}(x) \in I, \text{for all } x, y \in R. \)

**EXAMPLES:**

Some simple quotient rings with the integers:

```python
sage: R = QuotientRing(ZZ, 7*ZZ); R
Quotient of Integer Ring by the ideal (7)

sage: R.gens()
(1,)

sage: 1*R(3); 6*R(3); 7*R(3)
3
4
0
```

```python
>>> from sage.all import *

>>> R = QuotientRing(ZZ, Integer(7)*ZZ); R
Quotient of Integer Ring by the ideal (7)

>>> R.gens()
(1,)

>>> Integer(1)*R(Integer(3)); Integer(6)*R(Integer(3)); Integer(7)*R(Integer(3))
3
4
0
```

```python
sage: S = QuotientRing(ZZ, ZZ.ideal(8)); S
Quotient of Integer Ring by the ideal (8)

sage: 2*S(4)
0
```
With polynomial rings (note that the variable name of the quotient ring can be specified as shown below):

```
sage: # needs sage.libs.pari
sage: P.<x> = QQ[]
sage: R.<xx> = QuotientRing(P, P.ideal(x^2 + 1))
sage: R
Univariate Quotient Polynomial Ring in xx over Rational Field
with modulus x^2 + 1
sage: R.gens(); R.gen()
(xx,)
xx
sage: for n in range(4): xx^n
 1
xx
-1
-xx
```
Sage coerces objects into ideals when possible:

```python
sage: P.<x> = QQ[]
sage: R = QuotientRing(P, x^2 + 1); R #...
   → needs sage.libs.pari
Univariate Quotient Polynomial Ring in xbar over Rational Field
   with modulus x^2 + 1
```

By Noether's homomorphism theorems, the quotient of a quotient ring of $R$ is just the quotient of $R$ by the sum of the ideals. In this example, we end up modding out the ideal $(x)$ from the ring $\mathbb{Q}[x, y]$:

```python
sage: # needs sage.libs.pari sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = QuotientRing(R, R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S, S.ideal(a))
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
   by the ideal (x, y^2 + 1)
sage: R.gens(); S.gens(); T.gens()
(x, y)
(a, b)
(0, d)
sage: for n in range(4): d^n
1
d
-d
```

```python
>>> from sage.all import *
... # needs sage.libs.pari sage.libs.singular
... R = PolynomialRing(QQ, Integer(2), names=(x, y,)); (x, y,) = R._first_ngens(2)
... S = QuotientRing(R, R.ideal(Integer(1) + y**Integer(2)), names=(a, b,)); (a, b,) = S._first_ngens(2)
... T = QuotientRing(S, S.ideal(a), names=('c', 'd',)); (c, d,) = T._first_ngens(2)
... T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
```

(continues on next page)
by the ideal \((x, y^2 + 1)\)

```python
>>> R.gens(); S.gens(); T.gens()
(x, y)
(a, b)
(0, d)
```

```python
>>> for n in range(Integer(4)): d**n
1
d
-1
-d
```

```python
class sage.rings.quotient_ring.QuotientRingIdeal_generic(ring, gens, coerce=True, **kwds)

Bases: Ideal_generic

Specialized class for quotient-ring ideals.

EXAMPLES:

```python
sage: Zmod(9).ideal([-6,9])
Ideal (3, 0) of Ring of integers modulo 9
```

```python
>>> from sage.all import *
```
I = R.ideal([Integer(4) + Integer(3)*x + x**Integer(2), Integer(1) + x**Integer(2)])
S = R.quotient_ring(I); S

Quotient of Univariate Polynomial Ring in x over Integer Ring
by the ideal (x^2 + 3*x + 4, x^2 + 1)

class sage.rings.quotient_ring.QuotientRing_nc(R, I, names, category=None)

Bases: Ring, ParentWithGens

The quotient ring of $R$ by a twosided ideal $I$.

This class is for rings that do not inherit from CommutativeRing.

EXAMPLES:

Here is a quotient of a free algebra by a twosided homogeneous ideal:

sage: # needs sage.combinat sage.libs.singular sage.modules
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')

sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2]*F
sage: Q.<a,b,c> = F.quo(I); Q

Quotient of Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
by the ideal (x*y + y*z, x*x + x*y - y*x - y*y)

sage: a*b
-b*c
sage: a^3
-b*c*a - b*c*b - b*c*c

>>> from sage.all import *

A quotient of a quotient is just the quotient of the original top ring by the sum of two ideals:

sage: # needs sage.combinat sage.libs.singular sage.modules
sage: J = Q * [a^3 - b^3] * Q
sage: R.<i,j,k> = Q.quo(J); R

Quotient of Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
by the ideal (-y*y*z - y*z*x - 2*y*z*z, x*y + y*z, x*x + x*y - y*x - y*y)

sage: i^3
-j*k*i - j*k*j - j*k*k
sage: j^3
-j*k*i - j*k*j - j*k*k

A quotient of a quotient is just the quotient of the original top ring by the sum of two ideals:
For rings that do inherit from `CommutativeRing`, we provide a subclass `QuotientRing_generic`, for backwards compatibility.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ,'x')
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I); S
Quotient of Univariate Polynomial Ring in x over Integer Ring
by the ideal (x^2 + 3*x + 4, x^2 + 1)
```

Again, a quotient of a quotient is just the quotient of the original top ring by the sum of two ideals.

```python
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
```
sage: S.<a,b> = R.quo(1 + y^2)
sage: T.<c,d> = S.quo(a)
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x, y^2 + 1)
sage: T.gens()
(0, d)

>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y')); (x, y) = R._first_ngens(2)
>>> S = R.quo(Integer(1) + y**Integer(2), names=('a', 'b')); (a, b) = S._first_ngens(2)
>>> T = S.quo(a, names=('c', 'd')); (c, d) = T._first_ngens(2)

Element

alias of QuotientRingElement

ambient()

Returns the cover ring of the quotient ring: that is, the original ring \( R \) from which we modded out an ideal, \( I \).

EXAMPLES:

sage: Q = QuotientRing(ZZ, 7 * ZZ)
sage: Q.cover_ring()
Integer Ring

>>> from sage.all import *
>>> Q = QuotientRing(ZZ, Integer(7) * ZZ)

Univariate Polynomial Ring in x over Rational Field

Univariate Polynomial Ring in x over Rational Field

characteristic()

Return the characteristic of the quotient ring.
Todo: Not yet implemented!

EXAMPLES:

```sage
sage: Q = QuotientRing(ZZ, 7*ZZ)
sage: Q.characteristic()
Traceback (most recent call last):
  ...  
NotImplementedError

>>> from sage.all import *
>>> Q = QuotientRing(ZZ, Integer(7)*ZZ)
>>> Q.characteristic()
Traceback (most recent call last):
  ...  
NotImplementedError
```

casection()  
Returns the functorial construction of self.

EXAMPLES:

```sage
sage: R.<x> = PolynomialRing(ZZ, 'x')
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: R.quotient_ring(I).construction()
(QuotientFunctor, Univariate Polynomial Ring in x over Integer Ring)

sage: # needs sage.combinat sage.libs.singular sage.modules
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2] * F
sage: Q = F.quo(I)
sage: Q.construction()
(QuotientFunctor,
  Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field)
```

cover()  
The covering ring homomorphism \( R \rightarrow R/I \), equipped with a section.
EXAMPLES:

```sage
sage: R = ZZ.quo(3 * ZZ)
sage: pi = R.cover()
sage: pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 3
  Defn: Natural quotient map
sage: pi(5)
2
sage: l = pi.lift()
```

```sage
>>> from sage.all import *
>>> R = ZZ.quo(Integer(3) * ZZ)
>>> pi = R.cover()
>>> pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 3
  Defn: Natural quotient map
>>> pi(Integer(5))
2
>>> l = pi.lift()
```

```sage
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ)
sage: Q = R.quo((x^2, y^2))
sage: pi = Q.cover()
sage: pi(x^3 + y)
ybar
sage: l = pi.lift(x + y^3)
sage: l
x
sage: l = pi.lift(); l
Set-theoretic ring morphism:
  From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field
  by the ideal (x^2, y^2)
  To:   Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Choice of lifting map
sage: l(x + y^3)
x
```

```sage
>>> from sage.all import *
>>> # needs sage.libs.singular
>>> R = PolynomialRing(QQ, names=('x', 'y')); (x, y,)=R._first_ngens(2)
>>> Q = R.quo((x**Integer(2), y**Integer(2)))
>>> pi = Q.cover()
>>> pi(x**Integer(3) + y)
ybar
>>> l = pi.lift(x + y**Integer(3))
>>> l
x
>>> l = pi.lift(); l
Set-theoretic ring morphism:
  From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field
  by the ideal (x^2, y^2)
  To:   Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Choice of lifting map
```
To: Multivariate Polynomial Ring in x, y over Rational Field  
Defn: Choice of lifting map

```python
>>> l(x + y**Integer(3))
x
```

**cover_ring()**

Returns the cover ring of the quotient ring: that is, the original ring \( R \) from which we modded out an ideal, \( I \).

**EXAMPLES:**

```python
sage: Q = QuotientRing(ZZ, 7 * ZZ)
sage: Q.cover_ring()
Integer Ring
```

```python
>>> from sage.all import *

Q = QuotientRing(ZZ, Integer(7) * ZZ)
Q.cover_ring()
Integer Ring
```

```python
sage: P.<x> = QQ[]
sage: Q = QuotientRing(P, x^2 + 1)  # needs sage.libs.pari

Q.cover_ring()
```

```python
Q = QuotientRing(P, x**Integer(2) + Integer(1))  # needs sage.libs.pari
Q.cover_ring()
```

**defining_ideal()**

Returns the ideal generating this quotient ring.

**EXAMPLES:**

In the integers:

```python
sage: Q = QuotientRing(ZZ, 7*ZZ)
sage: Q.defining_ideal()
Principal ideal (7) of Integer Ring
```

```python
>>> from sage.all import *

Q = QuotientRing(ZZ, Integer(7)*ZZ)
Q.defining_ideal()
Principal ideal (7) of Integer Ring
```

An example involving a quotient of a quotient. By Noether’s homomorphism theorems, this is actually a quotient by a sum of two ideals:
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = QuotientRing(R, R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S, S.ideal(a))
sage: S.defining_ideal()
Ideal (y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: T.defining_ideal()
Ideal (x, y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field

>>> from sage.all import *

>>> # needs sage.libs.singular

>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = R._first_ngens(2)

>>> S = QuotientRing(R, R.ideal(Integer(1) + y**Integer(2)), names=('a', 'b',)); (a, b,) = S._first_ngens(2)

>>> T = QuotientRing(S, S.ideal(a), names=('c', 'd',)); (c, d,) = T._first_ngens(2)

>>> S.defining_ideal()
Ideal (y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field

>>> T.defining_ideal()
Ideal (x, y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field

\textbf{gen}(i=0)

Returns the \(i\)-th generator for this quotient ring.

\textbf{EXAMPLES:}

sage: R = QuotientRing(ZZ, 7*ZZ)
sage: R.gen(0)
1

>>> from sage.all import *

>>> R = QuotientRing(ZZ, Integer(7)*ZZ)

>>> R.gen(Integer(0))
1

sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R, R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S, S.ideal(a))

Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x, y^2 + 1)

sage: T.gen(0); T.gen(1)
0
d

>>> from sage.all import *

>>> # needs sage.libs.singular

>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
ideal (*gens, **kwds)

Return the ideal of self with the given generators.

EXAMPLES:

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = R.quotient_ring(x^2 + y^2)
sage: S.ideal()
Ideal (0) of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
sage: S.ideal(x + y + 1)
Ideal (xbar + ybar + 1) of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
```

is_commutative()

Tell whether this quotient ring is commutative.

**Note:** This is certainly the case if the cover ring is commutative. Otherwise, if this ring has a finite number of generators, it is tested whether they commute. If the number of generators is infinite, a NotImplementedError is raised.

**AUTHOR:**

EXAMPLES:

Any quotient of a commutative ring is commutative:

```python
sage: P.<a,b,c> = QQ[]
sage: P.quo(P.random_element()).is_commutative()  # True
```

The non-commutative case is more interesting:

```python
sage: # needs sage.combinat sage.libs.singular sage.modules
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2] * F
sage: Q = F.quo(I)
sage: Q.is_commutative()  # False
```

In the next example, the generators apparently commute:

```python
sage: # needs sage.combinat sage.libs.singular sage.modules
sage: J = F * [x*y - y*x, x*z - z*x, y*z - z*y, x^3 - y^3] * F
sage: R = F.quo(J)
sage: R.is_commutative()  # True
```

`is_field (proof=True)`

Returns True if the quotient ring is a field. Checks to see if the defining ideal is maximal.

`is_integral_domain (proof=True)`

With proof equal to True (the default), this function may raise a Not Implemented Error.
When `proof` is `False`, if `True` is returned, then `self` is definitely an integral domain. If the function returns `False`, then either `self` is not an integral domain or it was unable to determine whether or not `self` is an integral domain.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: R.quo(x^2 - y).is_integral_domain()  # needs sage.libs.singular
True
sage: R.quo(x^2 - y^2).is_integral_domain()  # needs sage.libs.singular
False
sage: R.quo(x^2 - y^2).is_integral_domain(proof=False)  # needs sage.libs.singular
False
sage: R.<a,b,c> = ZZ[]
sage: Q = R.quotient_ring([a, b])
sage: Q.is_integral_domain()  # needs sage.libs.singular
False
```

```python
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> R.quo(x**Integer(2) - y).is_integral_domain()  # needs sage.libs.singular
True
>>> R.quo(x**Integer(2) - y**Integer(2)).is_integral_domain()  # needs sage.libs.singular
False
>>> R.quo(x**Integer(2) - y**Integer(2)).is_integral_domain(proof=False)  # needs sage.libs.singular
False
```

```python
is_noetherian()

Return `True` if this ring is Noetherian.

**EXAMPLES:**

```python
sage: R = QuotientRing(ZZ, 102 * ZZ)
sage: R.is_noetherian()
True
sage: P.<x> = QQ[]
sage: R = QuotientRing(P, x^2 + 1)  # needs sage.libs.pari
sage: R.is_noetherian()
```

(continues on next page)
True

```python
>>> from sage.all import *

>>> R = QuotientRing(ZZ, Integer(102) * ZZ)
>>> R.is_noetherian()
True

>>> P = QQ['x']; (x,) = P._first_ngens(1)
>>> R = QuotientRing(P, x**Integer(2) + Integer(1))
-> # needs sage.libs.pari
>>> R.is_noetherian()
True
```

If the cover ring of `self` is not Noetherian, we currently have no way of testing whether `self` is Noetherian, so we raise an error:

```
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: R.is_noetherian()
False

sage: I = R.ideal([x[1]^2, x[2]])
sage: S = R.quotient(I)
sage: S.is_noetherian()
Traceback (most recent call last):
... NotImplementedError
```

```python
>>> from sage.all import *

>>> R = InfinitePolynomialRing(QQ, names=(x,)); (x,) = R._first_ngens(1)
>>> R.is_noetherian()
False

>>> I = R.ideal([x[Integer(1)]**Integer(2), x[Integer(2)]])
>>> S = R.quotient(I)
>>> S.is_noetherian()
Traceback (most recent call last):
... NotImplementedError
```

**lift** (`x=None`)

Return the lifting map to the cover, or the image of an element under the lifting map.

**Note:** The category framework imposes that `Q.lift(x)` returns the image of an element `x` under the lifting map. For backwards compatibility, we let `Q.lift()` return the lifting map.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quotient(x^2 + y^2)
sage: S.lift()
# needs sage.libs.singular
```

Set-theoretic ring morphism:

- From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
- To: Multivariate Polynomial Ring in x, y over Rational Field
- Defn: Choice of lifting map
lifting_map()
Return the lifting map to the cover.

EXAMPLES:

sage: from sage.all import *

sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quotient(x^2 + y^2)
sage: pi = S.cover(); pi
Ring morphism:
From: Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2 + y^2)
To: Quotient of Multivariate Polynomial Ring in x, y over Rational Field
Defn: Natural quotient map
sage: L = S.lifting_map(); L
Set-theoretic ring morphism:
From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2 + y^2)
To: Multivariate Polynomial Ring in x, y over Rational Field
Defn: Choice of lifting map
sage: L(S.0)
x
sage: L(S.1)
y
sage: L(S.gen(0)) == x   # needs sage.libs.singular
True

sage: S.lift(S.0) == x   # needs sage.libs.singular
True
Set-theoretic ring morphism:
From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
To: Multivariate Polynomial Ring in x, y over Rational Field
Defn: Choice of lifting map

>>> L(S.gen(0))
x
>>> L(S.gen(1))
y
Note that some reduction may be applied so that the lift of a reduction need not equal the original element:

```python
sage: z = pi(x^3 + 2*y^2); z
-\bar{x}\bar{y}^2 + 2\bar{y}^2
sage: L(z)
-x*y^2 + 2*y^2
```

Test that there also is a lift for rings that are no instances of `Ring` (see Issue #11068):

```python
sage: # needs sage.modules
sage: MS = MatrixSpace(GF(5), 2, 2)
sage: I = MS * [MS.0*MS.1, MS.2 + MS.3] * MS
sage: Q = MS.quo(I)
sage: Q.lift()
Set-theoretic ring morphism:
From: Quotient of Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5 by the ideal
\[
\begin{pmatrix}
0 & 1 \\
0 & 0
\end{pmatrix},
\begin{pmatrix}
0 & 0 \\
1 & 1
\end{pmatrix}
\]
To: Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
Defn: Choice of lifting map
```

(continues on next page)
>> MS = MatrixSpace(GF(Integer(5)), Integer(2), Integer(2))
>> I = MS * [MS.gen(0)*MS.gen(1), MS.gen(2) + MS.gen(3)] * MS
>> Q = MS.quo(I)
>> Q.lift()
Set-theoretic ring morphism:
   From: Quotient of Full MatrixSpace of 2 by 2 dense matrices
         over Finite Field of size 5 by the ideal
   (   [ 0 1 ]
       [ 0 0 ] ,
      <BLANKLINE>
   [ 0 0 ]
   [ 1 1 ]
   )
   <BLANKLINE>
   To:   Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
   Defn: Choice of lifting map

ngens()

Returns the number of generators for this quotient ring.

Todo: Note that ngens counts 0 as a generator. Does this make sense? That is, since 0 only generates itself and the fact that this is true for all rings, is there a way to “knock it off” of the generators list if a generator of some original ring is modded out?

EXAMPLES:

sage: R = QuotientRing(ZZ, 7*ZZ)
sage: R.gens(); R.ngens()
(1,) 1

>>> from sage.all import *

>>> R = QuotientRing(ZZ, Integer(7)*ZZ)

>>> R.gens(); R.ngens()
(1,) 1

sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R, R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S, S.ideal(a))
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
      by the ideal (x, y^2 + 1)
sage: R.gens(); S.gens(); T.gens()
(x, y) (a, b) (0, d)
sage: R.ngens(); S.ngens(); T.ngens()
2 2 2
random_element()

Return a random element of this quotient ring obtained by sampling a random element of the cover ring and reducing it modulo the defining ideal.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: S = R.quotient([x^3, y^2])
sage: S.random_element()  # random
-8/5*xbar^2 + 3/2*xbar*ybar + 2*xbar - 4/23
```

```
>>> from sage.all import *
>>> R = QQ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> S = R.quotient([x^3, y^2])
>>> S.random_element()  # random
-8/5*xbar^2 + 3/2*xbar*ybar + 2*xbar - 4/23
```

retract(x)

The image of an element of the cover ring under the quotient map.

INPUT:

- x – An element of the cover ring

OUTPUT:

The image of the given element in self.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quotient(x^2 + y^2)
sage: S.retract((x+y)^2)
# needs sage.libs.singular
2*xbar*ybar
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(2), names=('x', 'y',)); (x, y,) = R._first_ngens(2)
>>> S = QuotientRing(R, R.ideal(Integer(1) + y**Integer(2)), names=('a', 'b', '→'))
>>> (a, b, c) = S._first_ngens(3)
>>> T = QuotientRing(S, S.ideal(a), names=('c', 'd',))
>>> (c, d,) = T._first_ngens(2)
>>> T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x, y^2 + 1)
>>> R.gens(); S.gens(); T.gens()
(x, y)
(a, b)
(0, d)
>>> R.ngens(); S.ngens(); T.ngens()
2
2
2
```
term_order()

Return the term order of this ring.

EXAMPLES:

```python
sage: P.<a,b,c> = PolynomialRing(QQ)
sage: I = Ideal([a^2 - a, b^2 - b, c^2 - c])
sage: Q = P.quotient(I)
sage: Q.term_order()
Degree reverse lexicographic term order
```

sage.rings.quotient_ring.is_QuotientRing(x)

Tests whether or not x inherits from QuotientRing_nc.

EXAMPLES:

```python
sage: from sage.rings.quotient_ring import is_QuotientRing
sage: R.<x> = PolynomialRing(ZZ,x)
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I)
sage: is_QuotientRing(S)
True
sage: is_QuotientRing(R)
False
```

```python
>>> from sage.all import *
>>> from sage.rings.quotient_ring import is_QuotientRing

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F * [x*y + y*z, x^2 + x*y - y*x - y^2] * F
sage: Q = F.quo(I)
sage: is_QuotientRing(Q)
```

4.1. Quotient Rings 129
4.2 Quotient Ring Elements

AUTHORS:

• William Stein

class sage.rings.quotient_ring_element.QuotientRingElement (parent, rep, reduce=True)

Bases: RingElement

An element of a quotient ring $R/I$.

INPUT:

• parent – the ring $R/I$

• rep – a representative of the element in $R$; this is used as the internal representation of the element

• reduce – bool (default: True) – if True, then the internal representation of the element is rep reduced modulo the ideal $I$

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<xbar> = R.quo((4 + 3*x + x^2, 1 + x^2)); S
Quotient of Univariate Polynomial Ring in x over Integer Ring by the ideal (x^2 + 3*x + 4, x^2 + 1)
sage: v = S.gens(); v
(xbar,)
```

```python
sage: loads(v[0].dumps()) == v[0]
True
```
```python
>>> from sage.all import *
>>> loads(v[Integer(0)].dumps()) == v[Integer(0)]
True
```

```python
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quo(x^2 + y^2); S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2 + y^2)
sage: S.gens()
# needs sage.libs.singular
(xbar, ybar)
```

```python
>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(2), names=(x, y,)); (x, y,) = R._first_ngens(2)
>>> S = R.quo(x**Integer(2) + y**Integer(2)); S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
by the ideal (x^2 + y^2)
>>> S.gens()
# needs sage.libs.singular
(xbar, ybar)
```

We name each of the generators.

```python
sage: # needs sage.libs.singular
sage: S.<a,b> = R.quotient(x^2 + y^2)
sage: a
a
sage: b
b
sage: a^2 + b^2 == 0
True
sage: b.lift()
y
sage: (a^3 + b^2).lift() -x*y^2 + y^2
```

```python
>>> from sage.all import *
>>> # needs sage.libs.singular
>>> S = R.quotient(x**Integer(2) + y**Integer(2), names=('a', 'b',)); (a, b,) = S._first_ngens(2)
>>> a
a
>>> b
b
>>> a**Integer(2) + b**Integer(2) == Integer(0)
True
>>> b.lift()
y
>>> (a**Integer(3) + b**Integer(2)).lift() -x*y^2 + y^2
```

`is_unit()`

Return True if self is a unit in the quotient ring.

EXAMPLES:
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(1 - x*y); type(a)  # needs sage.libs.singular
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
sage: a*b  # needs sage.libs.singular
1
sage: S(2).is_unit()  # needs sage.libs.singular
True

Check that Issue #29469 is fixed:
sage: a.is_unit()  # needs sage.libs.singular
True
sage: (a+b).is_unit()  # needs sage.libs.singular
False

lc()
Return the leading coefficient of this quotient ring element.

EXAMPLES:
sage: # needs sage.libs.singular
sage: R.<x,y,z> = PolynomialRing(GF(7), 3, order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo(I)
sage: f = Q(z*y + 2*x)
sage: f.lc() 2

(continues on next page)
General Rings, Ideals, and Morphisms, Release 10.4

(continued from previous page)

```python
>>> R = PolynomialRing(GF(Integer(7)), Integer(3), order='lex', names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> I = sage.rings.ideal.FieldIdeal(R)
>>> Q = R.quo(I)
>>> f = Q(z*y + Integer(2)*x)
>>> f.lc()
2
```

**lift()**

If self is an element of $R/I$, then return self as an element of $R$.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)  # needs sage.libs.singular
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
sage: a.lift()  # needs sage.libs.singular
x
sage: (3/5*(a + a^2 + b^2)).lift()  # needs sage.libs.singular
3/5*x
```

```python
>>> from sage.all import *
```  

```python
>>> R = QQ['x', 'y']; (x, y,) = R._first_ngens(2); S = R.quo(x**Integer(2) + y**Integer(2), names=(a, b,)); (a, b,) = S._first_ngens(2); type(a)  # needs sage.libs.singular
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
sage: a.lift()  # needs sage.libs.singular
x
sage: (Integer(3)/Integer(5)*(a + a**Integer(2) + b**Integer(2))).lift()  # needs sage.libs.singular
3/5*x
```

**lm()**

Return the leading monomial of this quotient ring element.

**EXAMPLES:**

```python
sage: # needs sage.libs.singular
sage: R.<x,y,z> = PolynomialRing(GF(7), 3, order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo(I)
sage: f = Q(z*y + Integer(2)*x)
sage: f.lm()
xbar
```

```python
>>> from sage.all import *
```  

```python
>>> R = PolynomialRing(GF(Integer(7)), Integer(3), order='lex', names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> I = sage.rings.ideal.FieldIdeal(R)
>>> Q = R.quo(I)
```

(continues on next page)
lt()  
Return the leading term of this quotient ring element.

EXAMPLES:

```python
sage: # needs sage.libs.singular
sage: R.<x,y,z> = PolynomialRing(GF(7), 3, order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo(I)
sage: f = Q(z*y + 2*x)
sage: f.lt()
xbar
```

monomials()  
Return the monomials in self.

OUTPUT:
A list of monomials.

EXAMPLES:

```python
sage: # needs sage.libs.singular
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
sage: a.monomials()
[a]
sage: (a + a*b).monomials()
[a*b, a]
sage: R.zero().monomials()
[]
```

```python
>>> from sage.all import *
```
reduce ($G$)
Reduce this quotient ring element by a set of quotient ring elements $G$.

**INPUT:**
- $G$ – a list of quotient ring elements

**Warning:** This method is not guaranteed to return unique minimal results. For quotients of polynomial rings, use `reduce()` on the ideal generated by $G$, instead.

**EXAMPLES:**

```python
sage: P.<a,b,c,d,e> = PolynomialRing(GF(2), 5, order='lex')
sage: I1 = ideal([a*b + c*d + 1, a*c*e + d*e,
                ...: a*b*e + c*e, b*c + c*d*e + 1])
sage: Q = P.quotient(sage.rings.ideal.FieldIdeal(P))
sage: I2 = ideal([Q(f) for f in I1.gens()])
sage: f = Q((a*b + c*d + 1)^2 + e)
sage: f.reduce(I2.gens())
```

```
0
```

Notice that the result above is not minimal:

```python
sage: I2.reduce(f)              # needs sage.libs.singular
0
```

```python
>>> from sage.all import *
>>> P = PolynomialRing(GF(Integer(2)), Integer(5), order='lex', names=('a', 'b', 'c', 'd', 'e'),); (a, b, c, d, e,) = P._first_ngens(5)
>>> I1 = ideal([a*b + c*d + Integer(1), a*c*e + d*e,
               ... a*b*e + c*e, b*c + c*d*e + Integer(1)])
>>> Q = P.quotient(sage.rings.ideal.FieldIdeal(P))
>>> I2 = ideal([Q(f) for f in I1.gens()])
>>> f = Q((a*b + c*d + Integer(1))^2 + e)
>>> f.reduce(I2.gens())
```

```
0
```

**variables()**
Return all variables occurring in self.

**OUTPUT:**
A tuple of linear monomials, one for each variable occurring in self.

**EXAMPLES:**
sage: # needs sage.libs.singular
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
sage: a.variables()
(a,)
sage: b.variables()
(b,)
sage: s = a^2 + b^2 + 1; s
1
sage: s.variables()
()
sage: (a + b).variables()
(a, b)

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2); S = R.quo(x^Integer(2) +
˓→y^Integer(2), names=('a', 'b',)); (a, b,) = S._first_ngens(2); type(a)
<class 'sage.rings.quotient_ring.QuotientRing_generic_with_category.element_class'>
>>> a.variables()
(a,)
>>> b.variables()
(b,)
>>> s = a^Integer(2) + b^Integer(2) + Integer(1); s
1
>>> s.variables()
()
>>> (a + b).variables()
(a, b)
5.1 Fraction Field of Integral Domains

AUTHORS:
- William Stein (with input from David Joyner, David Kohel, and Joe Wetherell)
- Burcin Erocal
- Julian Rüth (2017-06-27): embedding into the field of fractions and its section

EXAMPLES:
Quotienting is a constructor for an element of the fraction field:

```python
sage: R.<x> = QQ[]
sage: (x^2-1)/(x+1)
x - 1
sage: parent((x^2-1)/(x+1))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

```python
>>> from sage.all import *

>>> R = QQ[x]; (x,) = R._first_ngens(1)

>>> (x**Integer(2)-Integer(1))/(x+Integer(1))
x - 1

>>> parent((x**Integer(2)-Integer(1))/(x+Integer(1)))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

The GCD is not taken (since it doesn't converge sometimes) in the inexact case:

```python
sage: # needs sage.rings.real_mpfr
sage: Z.<z> = CC[]
sage: I = CC.gen()
sage: (1+I+z)/(z+0.1*I)
(0.000000000000000 + 1.000000000000000*I)/(0.000000000000000 + 1.000000000000000*I)

sage: (1+I*z)/(z+1.1)
(1.000000000000000 + 0.000000000000000*I)/(0.000000000000000 + 1.000000000000000*I)
```

(continues on next page)
sage.rings.fraction_field.FractionField(R, names=None)

Create the fraction field of the integral domain $R$.

**INPUT:**
- $R$ – an integral domain
- names – ignored

**EXAMPLES:**

We create some example fraction fields:

```python
sage: FractionField(IntegerRing())
Rational Field
sage: FractionField(PolynomialRing(RationalField(), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: FractionField(PolynomialRing(IntegerRing(), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: FractionField(PolynomialRing(RationalField(), Integer(2), 'x'))
Fraction Field of Multivariate Polynomial Ring in x0, x1 over Rational Field
```

Dividing elements often implicitly creates elements of the fraction field:

```python
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = x / (x+1)
sage: g = x**3 / (x+1)
sage: f/g
1/x^2
sage: g/f
x^2
```

```python
>>> from sage.all import *
>>> FractionField(IntegerRing())
Rational Field
>>> FractionField(PolynomialRing(RationalField(), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> FractionField(PolynomialRing(IntegerRing(), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
>>> FractionField(PolynomialRing(RationalField(), Integer(2), 'x'))
Fraction Field of Multivariate Polynomial Ring in x0, x1 over Rational Field
```

The input must be an integral domain:
```python
class sage.rings.fraction_field.FractionFieldEmbedding

Bases: DefaultConvertMap_unique

The embedding of an integral domain into its field of fractions.

EXAMPLES:

```
section()

Return a section of this map.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: R.fraction_field().coerce_map_from(R).section()
Section map:
    From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
    To:    Univariate Polynomial Ring in x over Rational Field
```

class sage.rings.fraction_field.FractionFieldEmbeddingSection

Bases: Section

The section of the embedding of an integral domain into its field of fractions.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = R.fraction_field().coerce_map_from(R).section(); f
Section map:
    From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
    To:    Univariate Polynomial Ring in x over Rational Field
```

class sage.rings.fraction_field.FractionField_1poly_field

Bases: FractionField_generic

The fraction field of a univariate polynomial ring over a field.

Many of the functions here are included for coherence with number fields.

class_number()

Here for compatibility with number fields and function fields.

EXAMPLES:
General Rings, Ideals, and Morphisms, Release 10.4

```python
sage: R.<t> = GF(5)[]; K = R.fraction_field()
sage: K.class_number()
1
```

```python
>>> from sage.all import *
>>> R = GF(Integer(5))['t']; (t,) = R._first_ngens(1); K = R.fraction_field()
>>> K.class_number()
1
```

**function_field()**

Return the isomorphic function field.

**EXAMPLES:**

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: K.function_field()
Rational function field in t over Finite Field of size 5
```

```python
>>> from sage.all import *
>>> R = GF(Integer(5))['t']; (t,) = R._first_ngens(1)
>>> K = R.fraction_field()
>>> K.function_field()
Rational function field in t over Finite Field of size 5
```

**See also:**

`sage.rings.function_field.RationalFunctionField.field()`

**maximal_order()**

Return the maximal order in this fraction field.

**EXAMPLES:**

```python
sage: K = FractionField(GF(5)['t'])
sage: K.maximal_order()
Univariate Polynomial Ring in t over Finite Field of size 5
```

```python
>>> from sage.all import *
>>> K = FractionField(GF(Integer(5))['t'])
>>> K.maximal_order()
Univariate Polynomial Ring in t over Finite Field of size 5
```

**ring_of_integers()**

Return the ring of integers in this fraction field.

**EXAMPLES:**

```python
sage: K = FractionField(GF(5)['t'])
sage: K.ring_of_integers()
Univariate Polynomial Ring in t over Finite Field of size 5
```

```python
>>> from sage.all import *
>>> K = FractionField(GF(Integer(5))['t'])
>>> K.ring_of_integers()
Univariate Polynomial Ring in t over Finite Field of size 5
```
class sage.rings.fraction_field.FractionField_generic(R, element_class=<class 'sage.rings.fraction_field_element.FractionFieldElement'>, category=Category of quotient fields)

Bases: Field

The fraction field of an integral domain.

base_ring()

Return the base ring of self.

This is the base ring of the ring which this fraction field is the fraction field of.

EXAMPLES:

sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring

>>> from sage.all import *
>>> R = Frac(ZZ['t'])
>>> R.base_ring()
Integer Ring

characteristic()

Return the characteristic of this fraction field.

EXAMPLES:

sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
sage: R = Frac(ZZ['t']); R.characteristic()
0
sage: R = Frac(GF(5)['w']); R.characteristic()
5

>>> from sage.all import *
>>> R = Frac(ZZ['t'])
>>> R.base_ring()
Integer Ring
>>> R = Frac(ZZ['t']); R.characteristic()
0
>>> R = Frac(GF(Integer(5))['w']); R.characteristic()
5

construction()

EXAMPLES:

sage: Frac(ZZ['x']).construction()
(FractionField, Univariate Polynomial Ring in x over Integer Ring)
sage: K = Frac(GF(3)['t'])
sage: f, R = K.construction()
sage: f(R)
Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 3
(continues on next page)
sage: f(R) == K
True

>>> from sage.all import *

>>> K = Frac(GF(Integer(3))[t])

>>> f, R = K.construction()

>>> f(R)
Fraction Field of Univariate Polynomial Ring in t
    over Finite Field of size 3

>>> f(R) == K
True

\textbf{gen} (i=0)

Return the i-th generator of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R = Frac(PolynomialRing(QQ,z,Integer(10))); R
Fraction Field of Multivariate Polynomial Ring
    in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
sage: R.gen(0)
z0
sage: R.gen(Integer(3))
z3
sage: R.gen(3)
z3
\end{verbatim}

\textbf{is\_exact} ()

Return if \texttt{self} is exact which is if the underlying ring is exact.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: Frac(ZZ['x']).is_exact()
True
sage: Frac(CDF['x']).is_exact()  # needs sage.rings.complex_double
False
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> Frac(ZZ['x']).is_exact()
True

>>> Frac(CDF['x']).is_exact()  # needs sage.rings.complex_double
False
\end{verbatim}
needs sage.rings.complex_double
False

**is_field** *(proof=True)*

Return `True`, since the fraction field is a field.

**EXAMPLES:**

```
sage: Frac(ZZ).is_field()
True
```

```
>>> from sage.all import *

>>> Frac(ZZ).is_field()
True
```

**is_finite()**

Tells whether this fraction field is finite.

**Note:** A fraction field is finite if and only if the associated integral domain is finite.

**EXAMPLES:**

```
sage: Frac(QQ['a','b','c']).is_finite()
False
```

```
>>> from sage.all import *

>>> Frac(QQ['a','b','c']).is_finite()
False
```

**ngens()**

This is the same as for the parent object.

**EXAMPLES:**

```
sage: R = Frac(PolynomialRing(QQ,'z',10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
sage: R.ngens()
10
```

```
>>> from sage.all import *

>>> R = Frac(PolynomialRing(QQ,'z',Integer(10))); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
>>> R.ngens()
10
```

**random_element** *(**args**, **kwargs)**

Return a random element in this fraction field.

The arguments are passed to the random generator of the underlying ring.

**EXAMPLES:**
sage: F = ZZ['x'].fraction_field()
sage: F.random_element()  # random
(2*x - 8)/(-x^2 + x)

>>> from sage.all import *
>>> F = ZZ['x'].fraction_field()
>>> F.random_element()  # random
(2*x - 8)/(-x^2 + x)

sage: f = F.random_element(degree=5)
sage: f.numerator().degree() == f.denominator().degree()
True
sage: f.denominator().degree() <= 5
True
sage: while f.numerator().degree() != 5:
    ...:     f = F.random_element(degree=5)

>>> from sage.all import *
>>> f = F.random_element(degree=Integer(5))
>>> f.numerator().degree() == f.denominator().degree()
True
>>> f.denominator().degree() <= Integer(5)
True
>>> while f.numerator().degree() != Integer(5):
    ...     f = F.random_element(degree=Integer(5))

ring()

Return the ring that this is the fraction field of.

EXAMPLES:

sage: R = Frac(QQ['x,y'])
sage: R
Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ring()
Multivariate Polynomial Ring in x, y over Rational Field

>>> from sage.all import *
>>> R = Frac(QQ['x,y'])
>>> R
Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
>>> R.ring()
Multivariate Polynomial Ring in x, y over Rational Field

some_elements()

Return some elements in this field.

EXAMPLES:

sage: R.<x> = QQ[]
sage: R.fraction_field().some_elements()
[0, 1, x, 2*x, x/(x^2 + 2*x + 1),
(continues on next page)
sage.rings.fraction_field.is_FractionField(x)

Test whether or not x inherits from FractionField_generic.

EXAMPLES:

```python
sage: from sage.rings.fraction_field import is_FractionField
sage: is_FractionField(Frac(ZZ['x']))
True
sage: is_FractionField(QQ)
False
```
5.2 Fraction Field Elements

AUTHORS:

- William Stein (input from David Joyner, David Kohel, and Joe Wetherell)
- Sebastian Pancratz (2010-01-06): Rewrite of addition, multiplication and derivative to use Henrici’s algorithms [Hor1972]

```python
class sage.rings.fraction_field_element.FractionFieldElement
    Bases: FieldElement

EXAMPLES:

sage: K = FractionField(PolynomialRing(QQ, 'x'))
sage: K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: loads(K.dumps()) == K
True
sage: x = K.gen()
sage: f = (x^3 + x)/(17 - x^19); f
(-x^3 - x)/(x^19 - 17)
sage: loads(f.dumps()) == f
True
```

```python
>>> from sage.all import *
>>> K = FractionField(PolynomialRing(QQ, 'x'))
>>> K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> loads(K.dumps()) == K
True
>>> x = K.gen()
>>> f = (x**Integer(3) + x)/(Integer(17) - x**Integer(19)); f
(-x^3 - x)/(x^19 - 17)
>>> loads(f.dumps()) == f
True
```

denominator()

Return the denominator of self.

EXAMPLES:

```python
sage: R.<x,y> = ZZ[]
sage: f = x/y + 1; f
(x + y)/y
sage: f.denominator()
y
```

```python
>>> from sage.all import *
>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> f = x/y + Integer(1); f
(x + y)/y
>>> f.denominator()
y
```

is_one()

Return True if this element is equal to one.
EXAMPLES:

```python
sage: F = ZZ['x,y'].fraction_field()
sage: x, y = F.gens()
sage: (x/x).is_one()
True
sage: (x/y).is_one()
False

>>> from sage.all import *
>>> F = ZZ['x,y'].fraction_field()
>>> x, y = F.gens()
>>> (x/x).is_one()
True
>>> (x/y).is_one()
False
```

`is_square(root=False)`

Return whether or not `self` is a perfect square.

If the optional argument `root` is `True`, then also returns a square root (or `None`, if the fraction field element is not square).

INPUT:

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

OUTPUT:

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

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: (1/t).is_square()
False
sage: (1/t^6).is_square()
True
sage: ((1+t)^4/t^6).is_square()
True
sage: (4*(1+t)^4/t^6).is_square()
True
sage: (2*(1+t)^4/t^6).is_square()
False
sage: ((1+t)/t^6).is_square()
False

sage: (4*(1+t)^4/t^6).is_square(root=True)
(True, (2*t^2 + 4*t + 2)/t^3)

sage: (2*(1+t)^4/t^6).is_square(root=True)
(False, None)

sage: R.<x> = QQ[]
sage: a = 2*(x+1)^2 / (2*(x-1)^2); a
(x^2 + 2*x + 1)/(2*x^2 - 2*x + 1)
sage: a.is_square()
True
```

(continues on next page)
sage: (0/x).is_square()
True

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> (Integer(1)/t).is_square()
False
>>> (Integer(1)/t**Integer(6)).is_square()
True
>>> ((Integer(1)+t)**Integer(4)/t**Integer(6)).is_square()
True
>>> (Integer(4)*(Integer(1)+t)**Integer(4)/t**Integer(6)).is_square()
True
>>> (Integer(2)*(Integer(1)+t)**Integer(4)/t**Integer(6)).is_square()
False
>>> ((Integer(1)+t)/t**Integer(6)).is_square()
False

>>> (Integer(4)*(Integer(1)+t)**Integer(4)/t**Integer(6)).is_square(root=True)
(True, (2*t^2 + 4*t + 2)/t^3)
>>> (Integer(2)*(Integer(1)+t)**Integer(4)/t**Integer(6)).is_square(root=True)
(False, None)

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> a = Integer(2)*(x+Integer(1))**Integer(2) / (Integer(2)*(x-
˓
Integer(1))**Integer(2)); a
(x^2 + 2*x + 1)/(x^2 - 2*x + 1)
>>> a.is_square()
True

is_zero()

Return True if this element is equal to zero.

EXAMPLES:

sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: t = F(0)/x
sage: t.is_zero()
True
sage: u = 1/x - 1/x
sage: u.is_zero()
True
sage: u.parent() == F
True

>>> from sage.all import *
>>> F = ZZ['x,y'].fraction_field()
>>> x,y = F.gens()
>>> t = F(Integer(0))/x
>>> t.is_zero()
True
>>> u = Integer(1)/x - Integer(1)/x
>>> u.is_zero()
True

(continues on next page)
True
>>> u.parent() \texttt{is F}
True

\texttt{nth\_root}(n)

Return a \textit{n}-th root of this element.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R = QQ['t'].fraction_field()
sage: t = R.gen()
sage: p = (t+1)^3 / (t^2+t-1)^3
sage: p.nth_root(3)
(t + 1)/(t^2 + t - 1)
sage: p = (t+1) / (t-1)
sage: p.nth_root(2)
Traceback (most recent call last):
  ... ValueError: not a 2nd power
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> R = QQ['t'].fraction_field()

>>> p = (t+Integer(1))^3 / (t^2+t-1)^3
>>> p.nth_root(Integer(3))
(t + 1)/(t^2 + t - 1)

>>> p = (t+Integer(1)) / (t-Integer(1))
>>> p.nth_root(Integer(2))
Traceback (most recent call last):
  ... ValueError: not a 2nd power
\end{verbatim}

\texttt{numerator()}

Return the numerator of \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x,y> = ZZ[]
sage: f = x/y + 1; f
(x + y)/y
sage: f.numerator()
x + y
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *

>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)

>>> f = x/y + Integer(1); f
(x + y)/y
>>> f.numerator()
x + y
\end{verbatim}

\texttt{reduce()}

Reduce this fraction.

Divides out the gcd of the numerator and denominator. If the denominator becomes a unit, it becomes 1.
Additionally, depending on the base ring, the leading coefficients of the numerator and the denominator may be normalized to 1.

Automatically called for exact rings, but because it may be numerically unstable for inexact rings it must be called manually in that case.

**EXAMPLES:**

```python
sage: R.<x> = RealField(10)[]
# needs sage.rings.real_mpfr
sage: f = (x^2+2*x+1)/(x+1); f
# needs sage.rings.real_mpfr
(x^2 + 2.0*x + 1.0)/(x + 1.0)
```

```python
sage: f.reduce(); f
# needs sage.rings.real_mpfr
x + 1.0
```

**specialization** (*D=None, phi=None*)

Returns the specialization of a fraction element of a polynomial ring

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

Substitute variables in the numerator and denominator of `self`. If a dictionary is passed, the keys are mapped to generators of the parent ring. Otherwise, the arguments are transmitted unchanged to the method `subs` of the numerator and the denominator.

**EXAMPLES:**

```python
sage: x, y = PolynomialRing(ZZ, 2, 'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: (1/f).subs(x=5)  
1/(25*y^2 + y + 30)
```

```python
>>> from sage.all import *

>>> R = RealField(Integer(10))['x']; (x,) = R._first_ngens(1)# needs sage.
->rings.real_mpfr
>>> f = (x**Integer(2)+Integer(2)*x+Integer(1))/(x+Integer(1)); f
# needs sage.rings.real_mpfr
(x^2 + 2.0*x + 1.0)/(x + 1.0)
```

```python
>>> f.reduce(); f
# needs sage.rings.real_mpfr
x + 1.0
```

**valuation** (*v=None*)

Return the valuation of `self`, assuming that the numerator and denominator have valuation functions defined on them.

**EXAMPLES:**

```python
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = (x^3 + x)/(x^2 - 2*x^3)
sage: f
(continues on next page)
```
class sage.rings.fraction_field_element.FractionFieldElement_univariate_poly_field

A fraction field element where the parent is the fraction field of a univariate polynomial ring over a field.

Many of the functions here are included for coherence with number fields.

is_integral()

Returns whether this element is actually a polynomial.

EXAMPLES:

sage: R.<t> = QQ[]
sage: elt = (t^2 + t - 2) / (t + 2); elt
(t^2 + t - 2)/(t + 2)
sage: elt.is_integral()
True
sage: elt = (t^2 - t) / (t+2); elt
(t^2 - t)/(t + 2)
sage: elt.is_integral()
False

reduce()

Pick a normalized representation of self.

In particular, for any \( a == b \), after normalization they will have the same numerator and denominator.

EXAMPLES:

For univariate rational functions over a field, we have:
sage: R.<x> = QQ[]
sage: (2 + 2*x) / (4*x)  # indirect doctest
(1/2*x + 1/2)/x

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> (Integer(2) + Integer(2)*x) / (Integer(4)*x)  # indirect doctest
(1/2*x + 1/2)/x

Compare with:

sage: R.<x> = ZZ[]
sage: (2 + 2*x) / (4*x)
(x + 1)/(2*x)

>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> (Integer(2) + Integer(2)*x) / (Integer(4)*x)
(x + 1)/(2*x)

support()

Returns a sorted list of primes dividing either the numerator or denominator of this element.

EXAMPLES:

sage: R.<t> = ZZ[]
sage: h = (t^14 + 2*t^12 - 4*t^11 + 8*t^9 + 12*t^8 + 6*t^7 - 4*t^6
....: - 8*t^5 + t^3 + 2)/(t^6 + 6*t^5 + 9*t^4 - 2*t^2 - 12*t - 18)
sage: h.support()  # needs sage.libs.pari
[t - 1, t + 3, t^2 + 2, t^2 + t + 1, t^4 - 2]

sage: is_FractionFieldElement(x)
Return whether or not x is a FractionFieldElement.

EXAMPLES:

sage: from sage.rings.fraction_field_element import is_FractionFieldElement
sage: R.<x> = ZZ[]
sage: is_FractionFieldElement(x/2)
doctest:warning...  
DeprecationWarning: The function is_FractionFieldElement is deprecated; use 'isinstance(..., FractionFieldElement)' instead.  

General Rings, Ideals, and Morphisms, Release 10.4

See https://github.com/sagemath/sage/issues/38128 for details.
False
sage: is_FractionFieldElement(2/x)
True
sage: is_FractionFieldElement(1/3)
False

```python
>>> from sage.all import *
>>> from sage.rings.fraction_field_element import is_FractionFieldElement
>>>
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> is_FractionFieldElement(x/Integer(2))
doctest:warning...
DeprecationWarning: The function is_FractionFieldElement is deprecated;
use 'isinstance(..., FractionFieldElement)' instead.
See https://github.com/sagemath/sage/issues/38128 for details.
False
>>> is_FractionFieldElement(Integer(2)/x)
True
>>> is_FractionFieldElement(Integer(1)/Integer(3))
False
```

`sage.rings.fraction_field_element.make_element` *(parent, numerator, denominator)*

Used for unpickling `FractionFieldElement` objects (and subclasses).

**EXAMPLES:**

```python
sage: from sage.rings.fraction_field_element import make_element
sage: R = ZZ['x,y']
sage: x,y = R.gens()
sage: F = R.fraction_field()
sage: make_element(F, 1 + x, 1 + y)
(x + 1)/(y + 1)
```

```python
>>> from sage.all import *
>>> from sage.rings.fraction_field_element import make_element
>>>
>>> R = ZZ['x,y']
>>>
>>> x,y = R.gens()
>>>
>>> F = R.fraction_field()
>>>
>>> make_element(F, Integer(1) + x, Integer(1) + y)
(x + 1)/(y + 1)
```

`sage.rings.fraction_field_element.make_element_old` *(parent, cdict)*

Used for unpickling old `FractionFieldElement` pickles.

**EXAMPLES:**

```python
sage: from sage.rings.fraction_field_element import make_element_old
sage: R.<x,y> = ZZ[]
sage: F = R.fraction_field()
sage: make_element_old(F, {_FractionFieldElement__numerator: x + y, 
...: _FractionFieldElement__denominator: x - y})
(x + y)/(x - y)
```

```python
>>> from sage.all import *
>>> from sage.rings.fraction_field_element import make_element_old

(continues on next page)```
R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
F = R.fraction_field()
make_element_old(F, {_FractionFieldElement__numerator: x + y,
...    '_FractionFieldElement__denominator': x - y})
(x + y)/(x - y)
6.1 Localization

Localization is an important ring construction tool. Whenever you have to extend a given integral domain such that it contains the inverses of a finite set of elements but should allow non injective homomorphic images this construction will be needed. See the example on Ariki-Koike algebras below for such an application.

EXAMPLES:

```python
sage: # needs sage.modules
sage: LZ = Localization(ZZ, (5, 11))
sage: m = matrix(LZ, [[5, 7], [0, 11]])
sage: m.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring localized at (5, 11)
sage: ~m  # parent of inverse is different: see documentation of m.__invert__
[ 1/5 -7/55]
[ 0  1/11]
sage: _.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: mi = matrix(LZ, ~m)
sage: mi.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring localized at (5, 11)
sage: mi == ~m
True
```

```python
>>> from sage.all import *
>>> # needs sage.modules
>>> LZ = Localization(ZZ, (Integer(5), Integer(11)))
>>> m = matrix(LZ, [[Integer(5), Integer(7)], [Integer(0), Integer(11)]])
>>> m.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring localized at (5, 11)
>>> ~m  # parent of inverse is different: see documentation of m.__invert__
[ 1/5 -7/55]
[ 0  1/11]
>>> _.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
>>> mi = matrix(LZ, ~m)
>>> mi.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring localized at (5, 11)
>>> mi == ~m
True
```

The next example defines the most general ring containing the coefficients of the irreducible representations of the Ariki-Koike algebra corresponding to the three colored permutations on three elements:
Define the representation matrices (of one of the three dimensional irreducible representations):

```python
sage: # needs sage.libs.pari sage.modules
sage: m1 = matrix(L, [[u1, 0, 0], [0, u0, 0], [0, 0, u0]])
```

```python
sage: m2 = matrix(L, [[(u0*q - u0)/(u0 - u1), (u0*q - u1)/(u0 - u1), 0],
                  [(-u1*q + u0)/(u0 - u1), (-u1*q + u1)/(u0 - u1), 0],
                  [0, 0, -1]])
```

```python
sage: m3 = matrix(L, [[-1, 0, 0],
                  [0, u0*(1 - q)/(u1*q - u0), q*(u1 - u0)/(u1*q - u0)],
                  [0, (u1*q**2 - u0)/(u1*q - u0), (u1*q**2 - u1*q)/(u1*q - u0)]])
```

```python
sage: m1.base_ring() == L
True
```

Check relations of the Ariki-Koike algebra:

```python
>>> from sage.all import *

sage: # needs sage.libs.pari sage.modules
>>> m1 = matrix(L, [[u1, Integer(0), Integer(0)], [Integer(0), u0, Integer(0)],
                 [Integer(0), Integer(0), -u0]])
```
Obtain specializations in positive characteristic:

```python
sage: # needs sage.libs.pari sage.modules
sage: Fp = GF(17)
sage: f = L.hom((3,5,7,11), codomain=Fp); f
Ring morphism:
    From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring localized at
    (q, q + 1, u2, u1, u1 - u2, u0, u0 - u2, u0 - u1, u2*q - u1, u2*q - u0, u1*q - u2, u1*q - u0, u0*q - u2, u0*q - u1)
    To:   Finite Field of size 17
    Defn: u0 |---> 3
          u1 |---> 5
          u2 |---> 7
          q |---> 11
```

```python
sage: mFp1 = matrix({k: f(v) for k, v in m1.dict().items()}); mFp1
```

(continues on next page)
General Rings, Ideals, and Morphisms, Release 10.4

(continued from previous page)

```
[0 3 0]
[0 0 3]
sage: mFp1.base_ring()
Finite Field of size 17
sage: mFp2 = matrix({k: f(v) for k, v in m2.dict().items()}); mFp2
[ 2 3 0]
[ 9 8 0]
[ 0 0 16]
sage: mFp3 = matrix({k: f(v) for k, v in m3.dict().items()}); mFp3
[16 0 0]
[ 0 4 5]
[ 0 7 6]
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari sage.modules
>>> Fp = GF(Integer(17))
>>> f = L.hom((Integer(3),Integer(5),Integer(7),Integer(11)), codomain=Fp); f
Ring morphism:
  From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring localized at
       (q, q + 1, u2, u1 - u2, u0 - u2, u0 - u1, u2*q - u1, u2*q - u0,
        u1*q - u0, u0*q - u2, u0*q - u1)
  To:   Finite Field of size 17
  Defn: u0 |--> 3
         u1 |--> 5
         u2 |--> 7
         q |--> 11
>>> mFp1 = matrix({k: f(v) for k, v in m1.dict().items()}); mFp1
[5 0 0]
[0 3 0]
[0 0 3]
>>> mFp1.base_ring()
Finite Field of size 17
>>> mFp2 = matrix({k: f(v) for k, v in m2.dict().items()}); mFp2
[ 2 3 0]
[ 9 8 0]
[ 0 0 16]
>>> mFp3 = matrix({k: f(v) for k, v in m3.dict().items()}); mFp3
[16 0 0]
[ 0 4 5]
[ 0 7 6]
```

Obtain specializations in characteristic 0:

```
sage: # needs sage.libs.pari
sage: fQ = L.hom((3,5,7,11), codomain=QQ); fQ
Ring morphism:
  From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring localized at
       (q, q + 1, u2, u1 - u2, u0 - u2, u0 - u1, u2*q - u1, u2*q - u0,
        u1*q - u0, u0*q - u2, u0*q - u1)
  To:   Rational Field
  Defn: u0 |--> 3
         u1 |--> 5
         u2 |--> 7
         q |--> 11
sage: # needs sage.libs.pari sage.modules sage.rings.finite_rings
(continues on next page)
sage: mQ1 = matrix({k: fQ(v) for k, v in m1.dict().items()}); mQ1
\[
\begin{bmatrix}
5 & 0 & 0 \\
0 & 3 & 0 \\
0 & 0 & 3
\end{bmatrix}
\]
sage: mQ1.base_ring()
Rational Field
sage: mQ2 = matrix({k: fQ(v) for k, v in m2.dict().items()}); mQ2
\[
\begin{bmatrix}
-15 & -14 & 0 \\
26 & 25 & 0 \\
0 & 0 & -1
\end{bmatrix}
\]
sage: mQ3 = matrix({k: fQ(v) for k, v in m3.dict().items()}); mQ3
\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & -15/26 & 11/26 \\
0 & 301/26 & 275/26
\end{bmatrix}
\]

sage: # needs sage.libs.pari sage.libs.singular
sage: S.<x, y, z, t> = QQ[]
sage: T = S.quo(x + y + z)
sage: F = T.fraction_field()
sage: fF = L.hom((x, y, z, t), codomain=F); fF
Ring morphism:
    From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring
          localized at (q, q + 1, u2, u1 - u2, u0 - u2, u0 - u1, u2*q - u1, u2^2*q - u0, u1*q - u2, u1*u1*q - u0, u0*q - u2, u0*u1*q - u1)
    To:   Fraction Field of Quotient of Multivariate Polynomial Ring in x, y, z, t
          over Rational Field by the ideal (x + y + z)
    Defn: u0 |--> -ybar - zbar
           u1 |--> ybar
           u2 |--> zbar
           q |--> tbar
sage: mF1 = matrix({k: fF(v) for k, v in m1.dict().items()}); mF1
\[
\begin{bmatrix}
ybar & 0 & 0 \\
0 & -ybar - zbar & 0 \\
0 & 0 & -ybar - zbar
\end{bmatrix}
\]
sage: mF1.base_ring() == F
True

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> fQ = L.hom((Integer(3),Integer(5),Integer(7),Integer(11)), codomain=QQ); fQ
Ring morphism:
    From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring
          localized at (q, q + 1, u2, u1 - u2, u0 - u2, u0 - u1, u2*q - u1, u2^2*q - u0, u1*q - u2, u1*u1*q - u0, u0*q - u2, u0*u1*q - u1)
    To:   Rational Field
    Defn: u0 |--> 3
           u1 |--> 5
           u2 |--> 7
           q |--> 11

>>> # needs sage.libs.pari sage.modules sage.rings.finite_rings
>>> mQ1 = matrix({k: fQ(v) for k, v in m1.dict().items()}); mQ1
\[
\begin{bmatrix}
5 & 0 & 0 \\
0 & 3 & 0 \\
0 & 0 & 3
\end{bmatrix}
\]
>>> mQ1.base_ring()
Rational Field
>>> mQ2 = matrix({k: fQ(v) for k, v in m2.dict().items()}); mQ2
[-15 -14 0]
[ 26 25 0]
[ 0 0 -1]
>>> mQ3 = matrix({k: fQ(v) for k, v in m3.dict().items()}); mQ3
[-1 0 0]
[ 0 -15/26 11/26]
[ 0 301/26 275/26]

# needs sage.libs.pari sage.libs.singular
>>> S = QQ['x, y, z, t']; (x, y, z, t,) = S._first_ngens(4)
>>> T = S.quo(x + y + z)
>>> FF = L.hom((x, y, z, t), codomain=F); FF
Ring morphism:
From: Multivariate Polynomial Ring in u0, u1, u2, q over Integer Ring
   localized at (q, q + 1, u2, u1 - u2, u0 - u2, u0 - u1,
   u2*q - u1, u2*q - u0, u1*q - u2, u1*q - u0, u0*q - u2, u0*q - u1)
   over Rational Field by the ideal (x + y + z)
Defn: u0 |--> -ybar - zbar
u1 |--> ybar
u2 |--> zbar
q |--> tbar
>>> mF1 = matrix({k: ff(v) for k, v in m1.dict().items()}); mF1
# needs sage.modules
[ ybar 0 0]
[ 0 -ybar - zbar 0]
[ 0 0 -ybar - zbar]
>>> mF1.base_ring() == F
True

AUTHORS:
- Sebastian Oehms 2019-12-09: initial version.
- Sebastian Oehms 2022-03-05: fix some corner cases and add factor() (Issue #33463)

class sage.rings.localization.Localization(base_ring, extra_units, names=None, normalize=True, category=None, warning=True)

Bases: IntegralDomain, UniqueRepresentation

The localization generalizes the construction of the field of fractions of an integral domain to an arbitrary ring. Given a (not necessarily commutative) ring $R$ and a subset $S$ of $R$, there exists a ring $R[S^{-1}]$ together with the ring homomorphism $R \rightarrow R[S^{-1}]$ that “inverts” $S$; that is, the homomorphism maps elements in $S$ to unit elements in $R[S^{-1}]$ and, moreover, any ring homomorphism from $R$ that “inverts” $S$ uniquely factors through $R[S^{-1}]$.

The ring $R[S^{-1}]$ is called the localization of $R$ with respect to $S$. For example, if $R$ is a commutative ring and $f$ an element in $R$, then the localization consists of elements of the form $r/f, r \in R, n \geq 0$ (to be precise, $R[f^{-1}] = R[t]/(ft - 1)$).

The above text is taken from Wikipedia. The construction here used for this class relies on the construction of the field of fraction and is therefore restricted to integral domains.
Accordingly, this class is inherited from `IntegralDomain` and can only be used in that context. Furthermore, the base ring should support `sage.structure.element.CommutativeRingElement.divides()` and the exact division operator `//` (`sage.structure.element.Element.__floordiv__()`) in order to guarantee a successful application.

**INPUT:**

- `base_ring` – an instance of `Ring` allowing the construction of `fraction_field()` (that is an integral domain)
- `extra_units` – tuple of elements of `base_ring` which should be turned into units
- `names` – passed to `IntegralDomain`
- `normalize` – (default: `True`) passed to `IntegralDomain`
- `category` – (default: None) passed to `IntegralDomain`
- `warning` – (default: `True`) to suppress a warning which is thrown if self cannot be represented uniquely

**REFERENCES:**

- [Wikipedia article Ring (mathematics)#Localization](https://en.wikipedia.org/wiki/Ring_(mathematics)#Localization)

**EXAMPLES:**

```python
sage: L = Localization(ZZ, (3,5))
sage: 1/45 in L
True
sage: 1/43 in L
False

sage: Localization(L, (7,11))
Integer Ring localized at (3, 5, 7, 11)
sage: _.is_subring(QQ)
True

sage: L(-7)
Traceback (most recent call last):
  ... ValueError: factor 7 of denominator is not a unit

sage: Localization(Zp(7), (3, 5)) # needs sage.rings.padics
Traceback (most recent call last):
  ... ValueError: all given elements are invertible in 7-adic Ring with capped relative precision 20

sage: # needs sage.libs.pari
sage: R.<x> = ZZ[]
sage: L = R.localization(x**2 + 1)
sage: s = (x+5)/(x**2+1)
sage: s in L
True
sage: t = (x+5)/(x**2+2)
sage: t in L
False
sage: L(t)
Traceback (most recent call last):
  ...
```

(continues on next page)
TypeError: fraction must have unit denominator

```
sage: L(s) in R
False
sage: y = L(x)
sage: g = L(s)
sage: g.parent()
Univariate Polynomial Ring in x over Integer Ring localized at (x^2 + 1,)
sage: f = (y+5)/(y**2+1); f
(x + 5)/(x^2 + 1)
sage: f == g
True
sage: (y+5)/(y**2+2)
Traceback (most recent call last):
...  
ValueError: factor x^2 + 2 of denominator is not a unit
```

```
sage: Lau.<u, v> = LaurentPolynomialRing(ZZ)  # needs sage.modules
sage: LauL = Lau.localization(u + 1)  # needs sage.modules
sage: LauL(~u).parent()  # needs sage.modules
Multivariate Polynomial Ring in u, v over Integer Ring localized at (v, u, u + 1)
```

```
>>> from sage.all import *
>>> L = Localization(ZZ, (Integer(3),Integer(5)))
>>> Integer(1)/Integer(45) in L
True
>>> Integer(1)/Integer(43) in L
False

>>> Localization(L, (Integer(7),Integer(11)))
Integer Ring localized at (3, 5, 7, 11)
>>> _.is_subring(QQ)
True

>>> L(~Integer(7))
Traceback (most recent call last):
...  
ValueError: factor 7 of denominator is not a unit
```

```
>>> Localization(Zp(Integer(7)), (Integer(3), Integer(5)))  
...  
Traceback (most recent call last):
...  
ValueError: all given elements are invertible in
7-adic Ring with capped relative precision 20
```

```
>>> # needs sage.libs.pari
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> L = R.localization(x**Integer(2) + Integer(1))
>>> s = (x+Integer(5))/(x**Integer(2)+Integer(1))
>>> s in L
True
>>> t = (x+Integer(5))/(x**Integer(2)+Integer(2))
```
L(t)
False

```python
>>> L(s) in R
False
```

L(x)
```python
y = L(x)
n = L(s)
```

```python
g = g.parent()
Univariate Polynomial Ring in x over Integer Ring localized at (x^2 + 1,)
```

```python
f = (y+Integer(5))/(y**Integer(2)+Integer(1)); f
(x + 5)/(x^2 + 1)
```

```python
f == g
True
```

```python
(y+Integer(5))/(y**Integer(2)+Integer(2))
Traceback (most recent call last):
    ...TypeError: fraction must have unit denominator
```

```
Lau = LaurentPolynomialRing(ZZ, names=('u', 'v',)); (u, v,) = Lau._first_ngens(2) # needs sage.modules
```

```python
LauL = Lau.localization(u + Integer(1)) # needs sage.modules
```

```python
LauL(-u).parent() # needs sage.modules
Multivariate Polynomial Ring in u, v over Integer Ring localized at (v, u, u + 1)
```

More examples will be shown typing `sage.rings.localization?`

**Element**

alias of `LocalizationElement`

**characteristic()**

Return the characteristic of `self`.

**EXCEPTIONS:**

```python
sage: # needs sage.libs.pari
sage: R.<a> = GF(5)[]
sage: L = R.localization((a**2 - 3, a))
sage: L.characteristic()
5
```

```python
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> R = GF(Integer(5))['a']; (a,) = R._first_ngens(1)
>>> L = R.localization((a**Integer(2) - Integer(3), a))
>>> L.characteristic()
5
```

**fraction_field()**

Return the fraction field of `self`.

**EXCEPTIONS:**

6.1. Localization
sage: # needs sage.libs.pari
sage: R.<a> = GF(5)[]

sage: L = Localization(R, (a**2 - 3, a))

type: Fraction Field of Univariate Polynomial Ring in a over Finite Field of size 5

sage: L.is_subring(_)
True

>>> from sage.all import *

>>> R = GF(Integer(5))['a']; (a,) = R._first_ngens(1)

>>> L = Localization(R, (a**Integer(2) - Integer(3), a))

>>> L.fraction_field()
Fraction Field of Univariate Polynomial Ring in a over Finite Field of size 5

>>> L.is_subring(_)
True

\textbf{gen}(i)

\noindent Return the \textit{i}-th generator of \texttt{self} which is the \textit{i}-th generator of the base ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x, y> = ZZ[]

sage: R.localization((x**2 + 1, y - 1)).gen(0)  # needs sage.libs.pari
x

sage: ZZ.localization(2).gen(0)
1

>>> from sage.all import *

>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)

>>> R.localization((x**Integer(2) + Integer(1), y - Integer(1))).gen(Integer(0))  # needs sage.libs.pari
x

>>> ZZ.localization(Integer(2)).gen(Integer(0))
1
\end{verbatim}

\textbf{gens}()

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

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x, y> = ZZ[]

sage: Localization(R, (x**2 + 1, y - 1)).gens()  # needs sage.libs.pari
(x, y)

sage: Localization(ZZ, 2).gens()
(1,)

>>> from sage.all import *

>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)

>>> Localization(R, (x**Integer(2) + Integer(1), y - Integer(1))).gens()  # needs sage.libs.pari
(x, y)

(continues on next page)
General Rings, Ideals, and Morphisms, Release 10.4

6.1. Localization 167

(x, y)

>>> Localization(ZZ, Integer(2)).gens()
(1,)

is_field (proof=True)

Return True if this ring is a field.

INPUT:

• proof – (default: True) Determines what to do in unknown cases

ALGORITHM:

If the parameter proof is set to True, the returned value is correct but the method might throw an error. Otherwise, if it is set to False, the method returns True if it can establish that self is a field and False otherwise.

EXAMPLES:

sage: R = ZZ.localization((2, 3))
sage: R.is_field()
False

>>> from sage.all import *
>>>
R = ZZ.localization((Integer(2), Integer(3)))
>>>
R.is_field()
False

krull_dimension()

Return the Krull dimension of this localization.

Since the current implementation just allows integral domains as base ring and localization at a finite set of elements the spectrum of self is open in the irreducible spectrum of its base ring. Therefore, by density we may take the dimension from there.

EXAMPLES:

sage: R = ZZ.localization((2, 3))
sage: R.krull_dimension()
1

>>> from sage.all import *
>>> R = ZZ.localization((Integer(2), Integer(3)))
>>> R.krull_dimension()
1

ngens()

Return the number of generators of self according to the same method for the base ring.

EXAMPLES:

sage: R.<x, y> = ZZ[]
sage: Localization(R, (x**2 + 1, y - 1)).gens() # needs sage.libs.pari
2
sage: Localization(ZZ, 2).ngens()
1

numerical

>>> from sage.all import *

>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)

>>> Localization(R, (x**Integer(2) + Integer(1), y - Integer(1))).ngens() # needs sage.libs.pari
2

>>> Localization(ZZ, Integer(2)).ngens()
1

class sage.rings.localization.LocalizationElement (parent, x)

Bases: IntegralDomainElement

Element class for localizations of integral domains

INPUT:

• parent – instance of Localization

• x – instance of FractionFieldElement whose parent is the fraction
field of the parent’s base ring

EXAMPLES:

sage: # needs sage.libs.pari
sage: from sage.rings.localization import LocalizationElement
sage: P.<x,y,z> = GF(5)[]

sage: L = P.localization((x, y*z - x))

sage: LocalizationElement(L, 4/(y*z-x)**2)
(-1)/(y^2*z^2 - 2*x*y*z + x^2)

sage: _.parent()
Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
localized at (x, y*z - x)

>>> from sage.all import *

>>> # needs sage.libs.pari

>>> from sage.rings.localization import LocalizationElement

>>> P = GF(Integer(5))[x, y, z]; (x, y, z,) = P._first_ngens(3)

>>> L = P.localization((x, y*z - x))

>>> LocalizationElement(L, Integer(4)/(y*z-x)**Integer(2))
(-1)/(y^2*z^2 - 2*x*y*z + x^2)

>>> _.parent()
Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
localized at (x, y*z - x)

denominator ()

Return the denominator of self.

EXAMPLES:

sage: L = Localization(ZZ, (3,5))

sage: L(7/15).denominator()
15

168 Chapter 6. Localization
factor (proof=None)

Return the factorization of this polynomial.

INPUT:

- proof – (optional) if given it is passed to the corresponding method of the numerator of self

EXAMPLES:

```python
sage: p = (x^2 - y^2)/(x-y)^2
# needs sage.libs.singular
sage: p.factor()                      # needs sage.libs.singular
(1/(x - y)) * (x + y)
```

inverse_of_unit()

Return the inverse of self.

EXAMPLES:

```python
sage: L(x*y*z).inverse_of_unit()  # needs sage.libs.singular
1/(x*y*z)
```
is_unit()
Return True if self is a unit.

EXAMPLES:

```
sage: # needs sage.libs.pari sage.singular
sage: P.<x,y,z> = QQ[]
sage: L = P.localization((x, y*z))
sage: L(y*z).is_unit()
True
sage: L(z).is_unit()
True
sage: L(x*y*z).is_unit()
True
```

numerator()
Return the numerator of self.

EXAMPLES:

```
sage: L = ZZ.localization((3,5))
sage: L(7/15).numerator()
7
```

sage.rings.localization.normalize_extra_units(base_ring, add_units, warning=True)
Function to normalize input data.

The given list will be replaced by a list of the involved prime factors (if possible).

INPUT:

• base_ring – an instance of IntegralDomain
• add_units – list of elements from base ring
• warning – (default: True) to suppress a warning which is thrown if no normalization was possible

OUTPUT:
List of all prime factors of the elements of the given list.

EXAMPLES:
```python
sage: from sage.rings.localization import normalize_extra_units
sage: normalize_extra_units(ZZ, [3, -15, 45, 9, 2, 50])
[2, 3, 5]
sage: P.<x,y,z> = ZZ[]
sage: normalize_extra_units(P, #...
  → needs sage.libs.pari
...: [3*x, z*y**2, 2*z, 18*(x*y*z)**2, x*z, 6*x*z, 5])
[2, 3, 5, z, y, x]
sage: normalize_extra_units(P, #...
  → needs sage.libs.pari
...: [3*x, z*y**2, 2*z, 18*(x*y*z)**2, x*z, 6*x*z, 5])
[2, 3, 5, z, y, x]
sage: # needs sage.libs.singular
sage: R.<x, y> = ZZ[]
sage: Q.<a, b> = R.quo(x**2 - 5)
sage: p = b**2 - 5
sage: p == (b-a)*(b+a)
True
sage: normalize_extra_units(Q, [p]) #...
  → needs sage.libs.pari
doctest:...: UserWarning: Localization may not be represented uniquely
[b^2 - 5]
sage: normalize_extra_units(Q, [p], warning=False) #...
  → needs sage.libs.pari
[b^2 - 5]
```

---

6.1. Localization 171

```
from sage.all import *
from sage.rings.localization import normalize_extra_units
>>> normalize_extra_units(ZZ, [Integer(3), -Integer(15), Integer(45), Integer(9),
  → Integer(2), Integer(50)])
[2, 3, 5]
>>> P = ZZ['x, y, z']; (x, y, z,) = P._first_ngens(3)
>>> normalize_extra_units(P, #...
  → needs sage.libs.pari
... [Integer(3)*x, z*y**Integer(2), Integer(2)*z, #...
  → Integer(18)*(x*y*z)**Integer(2), x*z, Integer(6)*x*z, Integer(5)])
>>> P = QQ['x, y, z']; (x, y, z,) = P._first_ngens(3)
>>> normalize_extra_units(P, #...
  → needs sage.libs.pari
... [Integer(3)*x, z*y**Integer(2), Integer(2)*z, #...
  → Integer(18)*(x*y*z)**Integer(2), x*z, Integer(6)*x*z, Integer(5)])
>>> # needs sage.libs.singular
>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> Q = R.quo(x**Integer(2) - Integer(5), names=('a', 'b',)); (a, b,) = Q._first_ #...
  → ngens(2)
>>> p = b**Integer(2) - Integer(5)
>>> p == (b-a)*(b+a)
True
>>> normalize_extra_units(Q, [p]) #...
  → needs sage.libs.pari
doctest:...: UserWarning: Localization may not be represented uniquely
[b^2 - 5]
```

(continues on next page)
>>> normalize_extra_units(Q, [p], warning=False)
   -- needs sage.libs.pari
[b^2 - 5]
7.1 Extension of rings

Sage offers the possibility to work with ring extensions $L/K$ as actual parents and perform meaningful operations on them and their elements.

The simplest way to build an extension is to use the method `sage.categories.commutative_rings.CommutativeRings.ParentMethods.over()` on the top ring, that is $L$. For example, the following line constructs the extension of finite fields $F_5^4/F_5^2$:

```python
sage: GF(5^4).over(GF(5^2))
Field in z4 with defining polynomial x^2 + (4*z2 + 3)*x + z2 over its base
```

By default, Sage reuses the canonical generator of the top ring (here $z_4 \in F_5^4$), together with its name. However, the user can customize them by passing in appropriate arguments:

```python
sage: F = GF(5^2)
sage: k = GF(5^4)
sage: z4 = k.gen()
sage: K.<a> = k.over(F, gen=1-z4); K
Field in a with defining polynomial x^2 + z2*x + 4 over its base
```

The base of the extension is available via the method `base()` (or equivalently `base_ring()`):

```python
sage: K.base()
Finite Field in z2 of size 5^2
```
General Rings, Ideals, and Morphisms, Release 10.4

>>> from sage.all import *

>>> K.base()  # needs sage.rings.finite_rings
Finite Field in z2 of size 5^2

It is also possible to build an extension on top of another extension, obtaining this way a tower of extensions:

sage: L.<b> = GF(5^8).over(K); L  # needs sage.rings.finite_rings
Field in b with defining polynomial x^2 + (4*z2 + 3*a)*x + 1 - a over its base
sage: L.base()  # needs sage.rings.finite_rings
Field in a with defining polynomial x^2 + z2*x + 4 over its base
sage: L.base().base()  # needs sage.rings.finite_rings
Finite Field in z2 of size 5^2

>>> from sage.all import *

>>> L = GF(Integer(5)**Integer(8)).over(K, names=(b,)); (b,) = L._first_ngens(1); L  # needs sage.rings.finite_rings
Field in b with defining polynomial x^2 + (4*z2 + 3*a)*x + 1 - a over its base
>>> L.base()  # needs sage.rings.finite_rings
Field in a with defining polynomial x^2 + z2*x + 4 over its base
>>> L.base().base()  # needs sage.rings.finite_rings
Finite Field in z2 of size 5^2

The method bases() gives access to the complete list of rings in a tower:

sage: L.bases()  # needs sage.rings.finite_rings
[Field in b with defining polynomial x^2 + (4*z2 + 3*a)*x + 1 - a over its base, Field in a with defining polynomial x^2 + z2*x + 4 over its base, Finite Field in z2 of size 5^2]

>>> from sage.all import *

>>> L.bases()  # needs sage.rings.finite_rings
[Field in b with defining polynomial x^2 + (4*z2 + 3*a)*x + 1 - a over its base, Field in a with defining polynomial x^2 + z2*x + 4 over its base, Finite Field in z2 of size 5^2]

Once we have constructed an extension (or a tower of extensions), we have interesting methods attached to it. As a basic example, one can compute a basis of the top ring over any base in the tower:

sage: L.basis_over(K)  # needs sage.rings.finite_rings
[1, b]
sage: L.basis_over(F)  # needs sage.rings.finite_rings
[1, a, b, a*b]

(continues on next page)
When the base is omitted, the default is the natural base of the extension:

```python
sage: L.basis_over()
# needs sage.rings.finite_rings
[1, b]
```

The method `sage.rings.ring_extension_element.RingExtensionWithBasis.vector()` computes the coordinates of an element according to the above basis:

```python
sage: u = a + 2*b + 3*a*b
# needs sage.rings.finite_rings
sage: u.vector() # over K
(a, 2 + 3*a)
sage: u.vector(F) # over K, then over F
(0, 1, 2, 3)

>>> from sage.all import *
>>>
```

One can also compute traces and norms with respect to any base of the tower:

```python
sage: # needs sage.rings.finite_rings
sage: u.trace() # over K
(2*z2 + 1) + (2*z2 + 1)*a
sage: u.trace(F)
z2 + 1
sage: u.trace().trace() # over K, then over F
z2 + 1
sage: u.norm() # over K
(z2 + 1) + (4*z2 + 2)*a
sage: u.norm(F)
2*z2 + 2
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> u.trace() # over K
```

(continues on next page)
\[(2z^2 + 1) + (2z^2 + 1)a\]
\[
>>> u.trace(F)
z2 + 1
\]
\[
>>> u.trace().trace()  \quad \# \, \text{over } K, \, \text{then over } F
z2 + 1
\]
\[
>>> u.norm()  \quad \# \, \text{over } K
(z2 + 1) + (4z2 + 2)a
\]
\[
>>> u.norm(F)
2z2 + 2
\]

And minimal polynomials:

\[
\text{sage: } u\text{.minpoly()}  \quad \# \ldots
\]
\[
\longrightarrow \text{needs } \text{sage}\text{.rings}\text{.finite}\_\text{rings}
\]
\[
x^2 + ((3z2 + 4) + (3z2 + 4)a)x + (z2 + 1) + (4z2 + 2)a
\]
\[
\text{sage: } u\text{.minpoly}(F)  \quad \# \ldots
\]
\[
\longrightarrow \text{needs } \text{sage}\text{.rings}\text{.finite}\_\text{rings}
\]
\[
x^4 + (4z2 + 4)x^3 + x^2 + (z2 + 1)x + 2z2 + 2
\]

\[
>>> \text{from } \text{sage}\text{.all import } *
\]
\[
>>> u\text{.minpoly()}  \quad \# \ldots
\]
\[
\longrightarrow \text{needs } \text{sage}\text{.rings}\text{.finite}\_\text{rings}
\]
\[
x^2 + ((3z2 + 4) + (3z2 + 4)a)x + (z2 + 1) + (4z2 + 2)a
\]
\[
>>> u\text{.minpoly}(F)  \quad \# \ldots
\]
\[
\longrightarrow \text{needs } \text{sage}\text{.rings}\text{.finite}\_\text{rings}
\]
\[
x^4 + (4z2 + 4)x^3 + x^2 + (z2 + 1)x + 2z2 + 2
\]

AUTHOR:

• Xavier Caruso (2019)

\textbf{class} \text{sage.rings.ring_extension.RingExtensionFactory}

\begin{itemize}
  \item \text{create_key_and_extra_args}(\text{ring, defining_morphism=\text{None}, gens=\text{None}, names=\text{None}, constructors=\text{None}})
\end{itemize}

Create a key and return it together with a list of constructors of the object.

\begin{itemize}
  \item \text{INPUT:}
  \begin{itemize}
    \item \text{ring} – a commutative ring
    \item \text{defining_morphism} – a ring homomorphism or a commutative ring or \text{None} (default: None); the defining morphism of this extension or its base (if it coerces to \text{ring})
    \item \text{gens} – a list of generators of this extension (over its base) or \text{None} (default: None);
    \item \text{names} – a list or a tuple of variable names or \text{None} (default: None)
    \item \text{constructors} – a list of constructors; each constructor is a pair \text{(class, arguments)} where \text{class} is the class implementing the extension and \text{arguments} is the dictionary of arguments to pass in to \text{init} function
  \end{itemize}
\end{itemize}

\textbf{create_object}(\text{version, key, **extra_args})

Return the object associated to a given key.
class sage.rings.ring_extension.RingExtensionFractionField

Bases: RingExtension_generic

A class for ring extensions of the form \`extrm{Frac}(A)/A`. 

Element

alias of RingExtensionFractionFieldElement

ring()

Return the ring whose fraction field is this extension.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: A.<a> = ZZ.extension(x**2 - 2)
sage: OK = A.over()
sage: K = OK.fraction_field(); K
Fraction Field of
  Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
sage: K.ring()
Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
sage: K.ring() is OK
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> A = ZZ.extension(x**Integer(2) - Integer(2), names=('a',)); (a,) = A._first_ngens(1)
>>> OK = A.over()
>>> K = OK.fraction_field(); K
Fraction Field of
  Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
>>> K.ring()
Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
>>> K.ring() is OK
True
```

class sage.rings.ring_extension.RingExtensionWithBasis

Bases: RingExtension_generic

A class for finite free ring extensions equipped with a basis.

Element

alias of RingExtensionWithBasisElement

basis_over (base=None)

Return a basis of this extension over base.

INPUT:

- base – a commutative ring (which might be itself an extension)

EXAMPLES:
If `base` is omitted, it is set to its default which is the base of the extension:

```python
sage: L.basis_over()  # needs sage.rings.finite_rings
[1, c, c^2]
sage: K.basis_over()  # needs sage.rings.finite_rings
[1, b]
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```python
sage: L.degree_over(GF(5^6))  # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z6 of size 5^6
```

```python
>>> from sage.all import *
```

```python
```
fraction_field(extend_base=False)

Return the fraction field of this extension.

INPUT:

- extend_base - a boolean (default: False);

If extend_base is False, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/L/K \), except if \( L \) is already a field in which base the fraction field of \( L/K \) is \( L/K \) itself.

If extend_base is True, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/\text{Frac}(K) \) (provided that the defining morphism extends to the fraction fields, i.e. is injective).

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: A.<a> = ZZ.extension(x^2 - 5)
sage: OK = A.over()  # over ZZ
sage: OK
Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
sage: K1 = OK.fraction_field(); K1
Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
sage: K1.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base, Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base, Integer Ring]
sage: K2 = OK.fraction_field(extend_base=True); K2
Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
sage: K2.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base, Rational Field]
```

```python
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> A = ZZ.extension(x**Integer(2) - Integer(5), names=('a',)); (a,) = A._first_ngens(1)
>>> OK = A.over()  # over ZZ
>>> OK
Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
>>> K1 = OK.fraction_field(); K1
Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
>>> K1.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base, Rational Field]
```
with defining polynomial \( x^2 - 5 \) over its base,
Order of conductor 2 generated by \( a \) in Number Field in \( a \\)
with defining polynomial \( x^2 - 5 \) over its base,
Integer Ring]

\[ K2 = OK.fraction_field(extend_base=True); K2 \]
Fraction Field of Order of conductor 2 generated by \( a \\)
in Number Field in \( a \\) with defining polynomial \( x^2 - 5 \) over its base

\[ K2.bases() \]
[Fraction Field of Order of conductor 2 generated by \( a \\)
in Number Field in \( a \\) with defining polynomial \( x^2 - 5 \) over its base,
Rational Field]

Note that there is no coercion map between \( K_1 \) and \( K_2 \):

\[
\text{sage: } K1.has_coerce_map_from(K2) \quad \text{\# needs sage.rings.number_field}
\]
False
\[
\text{sage: } K2.has_coerce_map_from(K1) \quad \text{\# needs sage.rings.number_field}
\]
False

\[
\text{sage: } \textbf{from sage.all import \*}
\]
\[
\text{sage: } K1.has_coerce_map_from(K2) \quad \text{\# needs sage.rings.number_field}
\]
False
\[
\text{sage: } K2.has_coerce_map_from(K1) \quad \text{\# needs sage.rings.number_field}
\]
False

We check that when the extension is a field, its fraction field does not change:

\[
\text{sage: } K1.fraction_field() \text{ is K1} \quad \text{\# needs sage.rings.number_field}
\]
True
\[
\text{sage: } K2.fraction_field() \text{ is K2} \quad \text{\# needs sage.rings.number_field}
\]
True

\[
\text{sage: } \textbf{from sage.all import \*}
\]
\[
\text{sage: } K1.fraction_field() \text{ is K1} \quad \text{\# needs sage.rings.number_field}
\]
True
\[
\text{sage: } K2.fraction_field() \text{ is K2} \quad \text{\# needs sage.rings.number_field}
\]
True

\textbf{free_module}(base=\text{None}, map=\text{True})

Return a free module \( \mathbf{V} \) over \( \text{base} \) which is isomorphic to this ring

INPUT:

- \( \text{base} \) – a commutative ring (which might be itself an extension) or \( \text{None} \) (default: \( \text{None} \))
- \( \text{map} \) – boolean (default \( \text{True} \)); whether to return isomorphisms between this ring and \( \mathbf{V} \)

OUTPUT:

- A finite-rank free module \( \mathbf{V} \) over \( \text{base} \)
• The isomorphism from $V$ to this ring corresponding to the basis output by the method $\texttt{basis\_over()}$ (only included if $\texttt{map}$ is True)

• The reverse isomorphism of the isomorphism above (only included if $\texttt{map}$ is True)

EXAMPLES:

```sage
sage: F = GF(11)
sage: K.<a> = GF(11^2).over()
# needs sage.rings.finite_rings
sage: L.<b> = GF(11^6).over(K)
# needs sage.rings.finite_rings
>>> from sage.all import *
>>> F = GF(Integer(11))
>>> K = GF(Integer(11)**Integer(2)).over(names=(a,)); (a,) = K._first_
# needs sage.rings.finite_rings
>>> L = GF(Integer(11)**Integer(6)).over(K, names=(b,)); (b,) = L._first_
# needs sage.rings.finite_rings
```

Forgetting a part of the multiplicative structure, the field $L$ can be viewed as a vector space of dimension 3 over $K$, equipped with a distinguished basis, namely $(1, b, b^2)$:

```sage
sage: # needs sage.rings.finite_rings
sage: V, i, j = L.free_module(K)
sage: V
Vector space of dimension 3 over Field in a with defining polynomial $x^2 + 7*x + 2$ over its base
sage: i
Generic map:
From: Vector space of dimension 3 over Field in a with defining polynomial $x^2 + 7*x + 2$ over its base
To: Field in b with defining polynomial $x^3 + (7 + 2*a)*x^2 + (2 - a)*x - a$ over its base
sage: j
Generic map:
From: Field in b with defining polynomial $x^3 + (7 + 2*a)*x^2 + (2 - a)*x - a$ over its base
To: Vector space of dimension 3 over Field in a with defining polynomial $x^2 + 7*x + 2$ over its base
sage: j(b)
(0, 1, 0)
sage: i((1, a, a+1))
1 + a*b + (1 + a)*b^2
```

```sage
>>> from sage.all import *
>>> V, i, j = L.free_module(K)
>>> V
Vector space of dimension 3 over Field in a with defining polynomial $x^2 + 7*x + 2$ over its base
>>> i
Generic map:
From: Vector space of dimension 3 over Field in a with defining polynomial $x^2 + 7*x + 2$ over its base
To: Field in b with defining polynomial $x^3 + (7 + 2*a)*x^2 + (2 - a)*x - a$ over its base
>>> j
```

(continues on next page)
Generic map:
  From: Field in b with defining polynomial
         \( x^3 + (7 + 2a)\times x^2 + (2 - a)\times x - a \) over its base
  To: Vector space of dimension 3 over
       Field in a with defining polynomial \( x^2 + 7x + 2 \) over its base
>>> j(b)
(0, 1, 0)
>>> i((Integer(1), a, a+Integer(1)))
1 + a*b + (1 + a)*b^2

Similarly, one can view \( L \) as a \( F \)-vector space of dimension 6:

```python
sage: V, i, j, = L.free_module(F)  # needs sage.rings.finite_rings
```
```python
sage: V  # needs sage.rings.finite_rings
Vector space of dimension 6 over Finite Field of size 11
```
```python
>>> from sage.all import *
```
```python
>>> V, i, j, = L.free_module(F)  # needs sage.rings.finite_rings
>>> V  # needs sage.rings.finite_rings
Vector space of dimension 6 over Finite Field of size 11
```

In this case, the isomorphisms between \( V \) and \( L \) are given by the basis \((1, a, b, ab, b^2, ab^2)\):

```python
sage: j(a*b)  # needs sage.rings.finite_rings (0, 0, 0, 1, 0, 0)
sage: i((1,2,3,4,5,6))  # needs sage.rings.finite_rings (1+2*a)+(3+4*a)*b+(5+6*a)*b^2
```

When `base` is omitted, the default is the base of this extension:

```python
sage: L.free_module(map=False)  # needs sage.rings.finite_rings
Vector space of dimension 3 over
   Field in a with defining polynomial \( x^2 + 7x + 2 \) over its base
```
```python
>>> from sage.all import *
```
```python
>>> L.free_module(map=False)  # needs sage.rings.finite_rings
Vector space of dimension 3 over
   Field in a with defining polynomial \( x^2 + 7x + 2 \) over its base
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```python
sage: L.degree(GF(11^3))  # needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: not (explicitly) defined over Finite Field in z3 of size 11^3
```
```python
>>> from sage.all import *
```
```python
>>> L.degree(GF(Integer(11)**Integer(3)))  # needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: not (explicitly) defined over Finite Field in z3 of size 11^3

class sage.rings.ring_extension.RingExtensionWithGen
  Bases: RingExtensionWithBasis
  A class for finite free ring extensions generated by a single element

  fraction_field(extend_base=False)
  Return the fraction field of this extension.
  INPUT:
  • extend_base -- a boolean (default: False);
  If extend_base is False, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/L/K \), except is
  \( L \) is already a field in which base the fraction field of \( L/K \) is \( L/K \) itself.
  If extend_base is True, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/\text{Frac}(K) \) (provided
  that the defining morphism extends to the fraction fields, i.e. is injective).

  EXAMPLES:

  sage: # needs sage.rings.number_field
  sage: x = polygen(ZZ, 'x')
  sage: A.<a> = ZZ.extension(x^2 - 5)
  sage: OK = A.over()  # over ZZ
  sage: OK
  Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base
  sage: K1 = OK.fraction_field(); K1
  Fraction Field of Order of conductor 2 generated by a
  in Number Field in a with defining polynomial x^2 - 5 over its base
  sage: K1.bases()
  [Fraction Field of Order of conductor 2 generated by a
   in Number Field in a with defining polynomial x^2 - 5 over its base,
   Order of conductor 2 generated by a in Number Field in a
   with defining polynomial x^2 - 5 over its base,
   Integer Ring]
  sage: K2 = OK.fraction_field(extend_base=True); K2
  Fraction Field of Order of conductor 2 generated by a
  in Number Field in a with defining polynomial x^2 - 5 over its base
  sage: K2.bases()
  [Fraction Field of Order of conductor 2 generated by a
   in Number Field in a with defining polynomial x^2 - 5 over its base,
   Rational Field]

  >>> from sage.all import *
  >>> # needs sage.rings.number_field
  >>> x = polygen(ZZ, 'x')
  >>> A = ZZ.extension(x**Integer(2) - Integer(5), names=('a',)); (a,) = A._
  >>> first_ngens(1)
  >>> OK = A.over()  # over ZZ
  >>> OK
  Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base
  >>> K1 = OK.fraction_field(); K1
  Fraction Field of Order of conductor 2 generated by a

...
in Number Field in a with defining polynomial x^2 - 5 over its base

```python
>>> K1.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base,
Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base,
Integer Ring]
```

```python
>>> K2 = OK.fraction_field(extend_base=True); K2
Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base
``` 

```python
>>> K2.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial x^2 - 5 over its base,
Rational Field]
```

Note that there is no coercion map between $K_1$ and $K_2$:

```python
sage: K1.has_coerce_map_from(K2)   # needs sage.rings.number_field
False
sage: K2.has_coerce_map_from(K1)   # needs sage.rings.number_field
False
```

```python
>>> from sage.all import *
``` 

```python
>>> K1.has_coerce_map_from(K2)   # needs sage.rings.number_field
False
>>> K2.has_coerce_map_from(K1)   # needs sage.rings.number_field
False
```

We check that when the extension is a field, its fraction field does not change:

```python
sage: K1.fraction_field() is K1  # needs sage.rings.number_field
True
sage: K2.fraction_field() is K2  # needs sage.rings.number_field
True
``` 

```python
>>> from sage.all import *
``` 

```python
>>> K1.fraction_field() is K1  # needs sage.rings.number_field
True
>>> K2.fraction_field() is K2  # needs sage.rings.number_field
True
```

**gens** (*base=None*)

Return the generators of this extension over *base*.

**INPUT:**

- *base* – a commutative ring (which might be itself an extension) or *None* (default: *None*)

**EXAMPLES:**
modulus (var='x')

Return the defining polynomial of this extension, that is the minimal polynomial of the given generator of this extension.

INPUT:

• var – a variable name (default: x)

EXAMPLES:

sage: from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(7)**Integer(4)).over(K, names=('a',)); (a,) = K._first_ngens(1)
>>> K.gens()  # over GF(5)
(a,)
>>> L = GF(Integer(7)**Integer(10)).over(K, names=('b',)); (b,) = L._first_ngens(1)
>>> L.gens()  # over GF(5)
(b,)
>>> L.gens(GF(Integer(5)))
(b, a)

We can use a different variable name:

sage: K.modulus('y')  # needs sage.rings.finite_rings
y^5 + (6*z2 + 4)*y^4 + (3*z2 + 5)*y^3 + (2*z2 + 2)*y^2 + 4*y + 6*z2
>>> from sage.all import *
>>> K.modulus('y')
# needs sage.rings.finite_rings
y^5 + (6*z2 + 4)*y^4 + (3*z2 + 5)*y^3 + (2*z2 + 2)*y^2 + 4*y + 6*z2

class sage.rings.ring_extension.RingExtension_generic

Bases: CommutativeRing

A generic class for all ring extensions.

Element

alias of RingExtensionElement

absolute_base()

Return the absolute base of this extension.

By definition, the absolute base of an iterated extension $K_n/\cdots K_2/K_1$ is the ring $K_1$.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: F = GF(5^2).over()  # over GF(5)
sage: K = GF(5^4).over(F)
sage: L = GF(5^12).over(K)
sage: F.absolute_base()
Finite Field of size 5
sage: K.absolute_base()
Finite Field of size 5
sage: L.absolute_base()
Finite Field of size 5

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(5)**Integer(2)).over()  # over GF(5)
>>> K = GF(Integer(5)**Integer(4)).over(F)
>>> L = GF(Integer(5)**Integer(12)).over(K)
>>> F.absolute_base()
Finite Field of size 5
>>> K.absolute_base()
Finite Field of size 5
>>> L.absolute_base()
Finite Field of size 5

See also:

base(), bases(), is_defined_over()

absolute_degree()

Return the degree of this extension over its absolute base

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: A = GF(5^4).over(GF(5^2))
sage: B = GF(5^12).over(A)
sage: A.absolute_degree()
2
sage: B.absolute_degree()
6

Chapter 7. Ring Extensions
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> A = GF(Integer(5)**Integer(4)).over(GF(Integer(5)**Integer(2)))
>>> B = GF(Integer(5)**Integer(12)).over(A)
>>> A.absolute_degree()
2
>>> B.absolute_degree()
6

See also:

degree(), relative_degree()

backend (force=False)
Return the backend of this extension.

INPUT:

- force – a boolean (default: False); if False, raise an error if the backend is not exposed

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: K = GF(5^3)
sage: E = K.over()

Field in z3 with defining polynomial x^3 + 3*x + 3 over its base
sage: E.backend()
Finite Field in z3 of size 5^3
sage: E.backend() is K
True

sage: from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(5)**Integer(3))
>>> E = K.over()

Field in z3 with defining polynomial x^3 + 3*x + 3 over its base
>>> E.backend()
Finite Field in z3 of size 5^3
>>> E.backend() is K
True

base()
Return the base of this extension.

EXAMPLES:

sage: F = GF(5^2)

Field in z2 with defining polynomial x^2 + 2*x + 1 over its base
sage: K = GF(5^4).over(F)

Field in z2 of size 5^2

sage: from sage.all import *
>>> F = GF(Integer(5)**Integer(2))

(continues on next page)
In case of iterated extensions, the base is itself an extension:

```
sage: L = GF(5^8).over(K)  # needs sage.rings.finite_rings
```
```
sage: L.base()  # needs sage.rings.finite_rings
```
```
Field in z4 with defining polynomial $x^2 + (4z2 + 3)x + z2$ over its base
```
```
sage: L.base() is K  # needs sage.rings.finite_rings
```
```
True
```

See also:

-bases(), absolute_base(), is_defined_over()

**bases()**

Return the list of successive bases of this extension (including itself).

**EXAMPLES:**

```
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2).over()  # over GF(5)
```
```
sage: K = GF(5^4).over(F)
```
```
sage: L = GF(5^12).over(K)
```
```
sage: F.bases()  # needs sage.rings.finite_rings
```
```
[Field in z2 with defining polynomial $x^2 + 4x + 2$ over its base,
  Finite Field of size 5]
```
```
sage: K.bases()  # needs sage.rings.finite_rings
```
```
[Field in z4 with defining polynomial $x^2 + (3 - z2)x + z2$ over its base,
  Field in z2 with defining polynomial $x^2 + 4x + 2$ over its base,
  Finite Field of size 5]
```
```
sage: L.bases()  # needs sage.rings.finite_rings
```
```
[Field in z12 with defining polynomial $x^3 + (1 + (2 - z2)z4)x^2 + (2 + 2z4)x - z4$ over its base,
  Field in z4 with defining polynomial $x^2 + (3 - z2)x + z2$ over its base,
  Field in z2 with defining polynomial $x^2 + 4x + 2$ over its base,
  Finite Field of size 5]
```

```python
>>> from sage.all import *
```
```python
>>> L = GF(Integer(5)**Integer(8)).over(K)  # needs sage.rings.finite_rings
```
```python
>>> L.base()  # needs sage.rings.finite_rings
```
```python
Field in z4 with defining polynomial $x^2 + (4z2 + 3)x + z2$ over its base
```
```python
>>> L.base() is K  # needs sage.rings.finite_rings
```
```python
True
```
F = GF(Integer(5)**Integer(2)).over()  # over GF(5)
K = GF(Integer(5)**Integer(4)).over(F)
L = GF(Integer(5)**Integer(12)).over(K)

F.bases()
[Field in z2 with defining polynomial x^2 + 4*x + 2 over its base,
 Finite Field of size 5]

K.bases()
[Field in z4 with defining polynomial x^2 + (3 - z2)*x + z2 over its base,
 Field in z2 with defining polynomial x^2 + 4*x + 2 over its base,
 Finite Field of size 5]

L.bases()
[Field in z12 with defining polynomial
 x^3 + (1 + (2 - z2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base,
 Field in z4 with defining polynomial x^2 + (3 - z2)*x + z2 over its base,
 Field in z2 with defining polynomial x^2 + 4*x + 2 over its base,
 Finite Field of size 5]

See also:
base(), absolute_base(), is_defined_over()
characteristic()

Return the characteristic of the extension as a ring.

OUTPUT:
A prime number or zero.

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: F = GF(5^2).over()  # over GF(5)
sage: K = GF(5^4).over(F)
sage: L = GF(5^12).over(K)
sage: F.characteristic()
5
sage: K.characteristic()
5
sage: L.characteristic()
5

from sage.all import *  # needs sage.rings.finite_rings

sage: F = GF(Integer(5)**Integer(2)).over()  # over GF(5)
sage: K = GF(Integer(5)**Integer(4)).over(F)
sage: L = GF(Integer(5)**Integer(12)).over(K)
sage: F.characteristic()
5
sage: K.characteristic()
5
sage: L.characteristic()
5

sage: F = RR.over(ZZ)
sage: F.characteristic()
0
```python
>>> from sage.all import *
>>> F = RR.over(ZZ)
>>> F.characteristic()
0

sage: F = GF(11)
sage: A.<x> = F[]
sage: K = Frac(F).over(F)
sage: K.characteristic()
11

>>> from sage.all import *
>>> F = GF(Integer(11))
>>> A = F['x']; (x,) = A._first_ngens(1)
>>> K = Frac(F).over(F)
>>> K.characteristic()
11

sage: E = GF(7).over(ZZ)
sage: E.characteristic()
7

>>> from sage.all import *
>>> E = GF(Integer(7)).over(ZZ)
>>> E.characteristic()
7
```

**construction()**

Return the functorial construction of this extension, if defined.

**EXAMPLES:**

```python
sage: E = GF(5^3).over() # needs sage.rings.finite_rings
sage: E.construction() # needs sage.rings.finite_rings

>>> from sage.all import *
>>> E = GF(Integer(5)**Integer(3)).over() # needs sage.rings.finite_rings
>>> E.construction() # needs sage.rings.finite_rings
```

**defining_morphism**(base=None)

Return the defining morphism of this extension over base.

**INPUT:**

- base – a commutative ring (which might be itself an extension) or None (default: None)

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2)
sage: K = GF(5^4).over(F)
sage: L = GF(5^12).over(K)
```

(continues on next page)
One can also pass in a base over which the extension is explicitly defined (see also `is_defined_over()`):

```python
sage: L.defining_morphism(F)  # needs sage.rings.finite_rings
Ring morphism:
    From: Finite Field in z2 of size 5^2
    To: Field in z12 with defining polynomial x^3 + (1 + (4*z2 + 2)*z4)*x^2 + (2 + 2*z4)*x - z4 over its base
    Defn: z2 --> z2

sage: L.defining_morphism(GF(5))  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: not (explicitly) defined over Finite Field of size 5
```

(continues on next page)
degree(base)

Return the degree of this extension over base.

INPUT:

* base — a commutative ring (which might be itself an extension)

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: A = GF(5^4).over(GF(5^2))
sage: B = GF(5^12).over(A)
sage: A.degree(GF(5^2))
2
sage: B.degree(A)
3
sage: B.degree(GF(5^2))
6
```

Note that base must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```python
sage: A.degrees(GF(5))  #... needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: not (explicitly) defined over Finite Field of size 5
```

See also:

* relative_degree(), absolute_degree()

degree_over(base=None)

Return the degree of this extension over base.
INPUT:

- `base` - a commutative ring (which might be itself an extension) or `None` (default: `None`)

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2)
sage: K = GF(5^4).over(F)
sage: L = GF(5^12).over(K)
sage: K.degree_over(F)
2
sage: L.degree_over(K)
3
sage: L.degree_over(F)
6
```

If `base` is omitted, the degree is computed over the base of the extension:

```python
sage: K.degree_over()  # needs sage.rings.finite_rings
2
sage: L.degree_over()  # needs sage.rings.finite_rings
3
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```python
sage: K.degree_over(GF(5))  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: not (explicitly) defined over Finite Field of size 5
```

(continues on next page)
fraction_field(extend_base=False)

Return the fraction field of this extension.

INPUT:

- extend_base – a boolean (default: False);

If extend_base is False, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/L/K \), except if \( L \) is already a field in which base the fraction field of \( L/K \) is \( L/K \) itself.

If extend_base is True, the fraction field of the extension \( L/K \) is defined as \( \text{Frac}(L)/\text{Frac}(K) \) (provided that the defining morphism extends to the fraction fields, i.e. is injective).

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: A.<a> = ZZ.extension(x^2 - 5)
sage: OK = A.over()  # over ZZ
sage: OK
Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base
sage: K1 = OK.fraction_field(); K1
Fraction Field of Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base
sage: K1.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base, Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base, Integer Ring]
sage: K2 = OK.fraction_field(extend_base=True); K2
Fraction Field of Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base
sage: K2.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a
  with defining polynomial x^2 - 5 over its base, Rational Field]
```
in Number Field in a with defining polynomial \(x^2 - 5\) over its base, Order of conductor 2 generated by a in Number Field in a with defining polynomial \(x^2 - 5\) over its base, Integer Ring]

```python
>>> K2 = OK.fraction_field(extend_base=True); K2
Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial \(x^2 - 5\) over its base
```

```python
>>> K2.bases()
[Fraction Field of Order of conductor 2 generated by a in Number Field in a with defining polynomial \(x^2 - 5\) over its base, Rational Field]
```

Note that there is no coercion between \(K_1\) and \(K_2\):

```python
sage: K1.has_coerce_map_from(K2)  # needs sage.rings.number_field
False
sage: K2.has_coerce_map_from(K1)  # needs sage.rings.number_field
False
```

```python
from sage.all import *

>>> K1.has_coerce_map_from(K2)  # needs sage.rings.number_field
False
>>> K2.has_coerce_map_from(K1)  # needs sage.rings.number_field
False
```

We check that when the extension is a field, its fraction field does not change:

```python
sage: K1.fraction_field() is K1  # needs sage.rings.number_field
True
sage: K2.fraction_field() is K2  # needs sage.rings.number_field
True
```

```python
from sage.all import *

>>> K1.fraction_field() is K1  # needs sage.rings.number_field
True
>>> K2.fraction_field() is K2  # needs sage.rings.number_field
True
```

**from_base_ring\(r\)**

Return the canonical embedding of \(r\) into this extension.

**INPUT:**

- \(r\) – an element of the base of the ring of this extension

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: k = GF(5)
```
sage: K.<u> = GF(5^2).over(k)
sage: L.<v> = GF(5^4).over(K)
sage: x = L.from_base_ring(k(2)); x
2
sage: x.parent()
Field in v with defining polynomial x^2 + (3 - u)*x + u over its base
sage: x = L.from_base_ring(u); x
u
sage: x.parent()
Field in v with defining polynomial x^2 + (3 - u)*x + u over its base

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = GF(Integer(5))
>>> K = GF(Integer(5)**Integer(2)).over(k, names=('u',)); (u,) = K._first_
˓→ngens(1)
>>> L = GF(Integer(5)**Integer(4)).over(K, names=('v',)); (v,) = L._first_
˓→ngens(1)
>>> x = L.from_base_ring(k(Integer(2))); x
2
>>> x.parent()
Field in v with defining polynomial x^2 + (3 - u)*x + u over its base
>>> x = L.from_base_ring(u); x
u
>>> x.parent()
Field in v with defining polynomial x^2 + (3 - u)*x + u over its base

gen()

Return the first generator of this extension.

EXAMPLES:

sage: K = GF(5^2).over()  # over GF(5)
˓→needs sage.rings.finite_rings
sage: x = K.gen(); x       #...
˓→needs sage.rings.finite_rings
z2

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(5)**Integer(2)).over()  # over GF(5)
˓→needs sage.rings.finite_rings
>>> x = K.gen(); x        #...
˓→needs sage.rings.finite_rings
z2

Observe that the generator lives in the extension:

sage: x.parent()          #...
˓→needs sage.rings.finite_rings
Field in z2 with defining polynomial x^2 + 4*x + 2 over its base
sage: x.parent() is K     #...
˓→needs sage.rings.finite_rings
True

>>> from sage.all import *
>>> #...
(continues on next page)
Field in z2 with defining polynomial $x^2 + 4x + 2$ over its base

```python
>>> x.parent() is K # ...
needs sage.rings.finite_rings
True
```

**`gens(base=None)`**

Return the generators of this extension over `base`.

**INPUT:**

- `base` – a commutative ring (which might be itself an extension) or `None` (default: `None`); if omitted, use the base of this extension

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: K.<a> = GF(5^2).over()   # over GF(5)
sage: K.gens()
(a,)
sage: L.<b> = GF(5^4).over(K)
sage: L.gens()
(b,)
sage: L.gens(GF(5))
(b, a)
sage: S.<x> = QQ[]
sage: T.<y> = S[

```
• `im_gens` – the images of the generators of this extension
• `codomain` – the codomain of the homomorphism; if omitted, it is set to the smallest parent containing all the entries of `im_gens`
• `base_map` – a map from one of the bases of this extension into something that coerces into the codomain; if omitted, coercion maps are used
• `category` – the category of the resulting morphism
• `check` – a boolean (default: True); whether to verify that the images of generators extend to define a map (using only canonical coercions)

**EXAMPLES:**

```sage
sage: K.<a> = GF(5^2).over()       # over GF(5)
sage: L.<b> = GF(5^6).over(K)      # needs sage.rings.finite_rings
>>> from sage.all import *
K = GF(Integer(5)**Integer(2)).over(names=('a',)); (a,) = K._first_ngens(1) # over GF(5) # needs sage.rings.finite_rings
L = GF(Integer(5)**Integer(6)).over(K, names=('b',)); (b,) = L._first_ngens(1) # needs sage.rings.finite_rings
```

We define (by hand) the relative Frobenius endomorphism of the extension $L/K$:

```sage
sage: L.hom([b^25])                  # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
Defn: b |--> 2 + 2*a*b + (2 - a)*b^2
```

```sage
>>> from sage.all import *
L.hom([b**Integer(25)])   # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
Defn: b |--> 2 + 2*a*b + (2 - a)*b^2
```

Defining the absolute Frobenius of $L$ is a bit more complicated because it is not a homomorphism of $K$-algebras. For this reason, the construction `L.hom([b^5])` fails:

```sage
sage: L.hom([b^5])                  # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: images do not define a valid homomorphism
```

```sage
>>> from sage.all import *
L.hom([b**Integer(5)])      # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: images do not define a valid homomorphism
```

What we need is to specify a base map:
As a shortcut, we may use the following construction:

```python
sage: phi = L.hom([b^5, a^5]); phi  # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
Defn: b |--> (-1 + a) + (1 + 2*a)*b + a*b^2
   with map on base ring:
   a |--> 1 - a
sage: phi == FrobL  # needs sage.rings.finite_rings
True
```

```python
>>> from sage.all import *
>>> FrobK = K.hom([a**Integer(5)]) # needs sage.rings.finite_rings
>>> FrobL = L.hom([b**Integer(5)], base_map=FrobK); FrobL # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
Defn: b |--> (-1 + a) + (1 + 2*a)*b + a*b^2
   with map on base ring:
   a |--> 1 - a
```

**is_defined_over** *(base)*

Return whether or not *base* is one of the bases of this extension.

**INPUT:**

- *base* – a commutative ring, which might be itself an extension

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: A = GF(5^4).over(GF(5^2))
(... continues on next page)
sage: B = GF(5^12).over(A)
sage: A.is_defined_over(GF(5^2))
True
sage: A.is_defined_over(GF(5))
False
sage: # needs sage.rings.finite_rings
sage: B.is_defined_over(A)
True
sage: B.is_defined_over(GF(5^4))
True
sage: B.is_defined_over(GF(5^2))
True
sage: B.is_defined_over(GF(5))
False

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> A = GF(Integer(5)^Integer(4)).over(GF(Integer(5)^Integer(2)))
>>> B = GF(Integer(5)^Integer(12)).over(A)
>>> A.is_defined_over(GF(Integer(5)^Integer(2)))
True
>>> A.is_defined_over(GF(Integer(5)))
False

Note that an extension is defined over itself:

sage: A.is_defined_over(A)  # needs sage.rings.finite_rings
True
sage: A.is_defined_over(GF(5^4))  # needs sage.rings.finite_rings
True

>>> from sage.all import *
>>> A.is_defined_over(A)  # needs sage.rings.finite_rings
True
>>> A.is_defined_over(GF(Integer(5)^Integer(4)))  # needs sage.rings.finite_rings
True

See also:

base(), bases(), absolute_base()
is_field(proof=True)
Return whether or not this extension is a field.

INPUT:

- **proof** – a boolean (default: False)

EXAMPLES:

```python
sage: K = GF(5^5).over()  # over GF(5)
    # needs sage.rings.finite_rings
sage: K.is_field()        # needs sage.rings.finite_rings
True
sage: S.<x> = QQ[]
sage: A = S.over(QQ)
sage: A.is_field()        False
sage: B = A.fraction_field()
sage: B.is_field()        True
```

### `is_finite_over(base=None)`

Return whether or not this extension is finite over `base` (as a module).

INPUT:

- **base** – a commutative ring (which might be itself an extension) or None (default: None)

EXAMPLES:

```python
>>> from sage.all import *
>>> K = GF(Integer(5)**Integer(5)).over()  # over GF(5)
    # needs sage.rings.finite_rings
>>> K.is_field()        # needs sage.rings.finite_rings
True
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> A = S.over(QQ)
>>> A.is_field()        False
>>> B = A.fraction_field()
>>> B.is_field()        True
```

(continues on next page)
If `base` is omitted, it is set to its default which is the base of the extension:

```
sage: L.is_free_over()  
˓→needs sage.rings.finite_rings  
True
```

```
>>> from sage.all import *  
>>> L.is_free_over()  
˓→needs sage.rings.finite_rings  
True
```

```
is_free_over(\text{base}=\text{None})
```

Return `True` if this extension is free (as a module) over `base`.

**INPUT:**

- `base` -- a commutative ring (which might be itself an extension) or `None` (default: `None`)

**EXAMPLES:**

```
sage: # needs sage.rings.finite_rings  
sage: K = GF(5^2).over()  
˓→needs sage.rings.finite_rings  
sage: L = GF(5^4).over(K)  
sage: L.is_free_over(K)  
True  
sage: L.is_free_over(GF(5))  
True
```

```
>>> from sage.all import *  
>>> K = GF(Integer(5)**Integer(2)).over()  
˓→needs sage.rings.finite_rings  
>>> L = GF(Integer(5)**Integer(4)).over(K)  
>>> L.is_free_over(K)  
True  
>>> L.is_free_over(GF(Integer(5)))  
True
```

If `base` is omitted, it is set to its default which is the base of the extension:

```
sage: L.is_free_over()  
˓→needs sage.rings.finite_rings  
True
```

```
>>> from sage.all import *  
>>> L.is_free_over()  
˓→needs sage.rings.finite_rings  
True
```

```
ngens(\text{base}=\text{None})
```

Return the number of generators of this extension over `base`.

```
INPUT:

- `base` – a commutative ring (which might be itself an extension) or `None` (default: `None`)

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: K = GF(5^2).over()  # over GF(5)
sage: K.gens()
(z2,)
sage: K.ngens()
1
sage: L = GF(5^4).over(K)
sage: L.gens(GF(5))
(z4, z2)
sage: L.ngens(GF(5))
2
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(5)**Integer(2)).over(names=(a,)); (a,) = K._first_.ngens(1)  # over GF(5)
>>> L = GF(Integer(5)**Integer(4)).over(K, names=(b,)); (b,) = L._first_.ngens(1)
>>> C = GF(Integer(5)**Integer(12)).over(B, names=(c,)); (c,) = C._first_.ngens(1)
```

`print_options(**options)`

Update the printing options of this extension.

INPUT:

- `over` – an integer or `Infinity` (default: 0); the maximum number of bases included in the printing of this extension
- `base` – a base over which this extension is finite free; elements in this extension will be printed as a linear combination of a basis of this extension over the given base

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: A.<a> = GF(5^2).over()  # over GF(5)
sage: B.<b> = GF(5^4).over(A)
sage: C.<c> = GF(5^12).over(B)
sage: D.<d> = GF(5^24).over(C)
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> A.<a> = GF(Integer(5)**Integer(2)).over(names=('a',)); (a,) = A._first_.ngens(1)  # over GF(5)
>>> B.<b> = GF(Integer(5)**Integer(4)).over(A, names=('b',)); (b,) = B._first_.ngens(1)
>>> C.<c> = GF(Integer(5)**Integer(12)).over(B, names=('c',)); (c,) = C._first_.ngens(1)
```

(continues on next page)
Observe what happens when we modify the option over:

```
sage: # needs sage.rings.finite_rings
sage: D
Field in d with defining polynomial
x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2 + a) + (1 - a)*b)*c^2)*x + c over...
    its base
sage: D.print_options(over=2)
sage: D
Field in d with defining polynomial x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2
    + a) + (1 - a)*b)*c^2)*x + c over Field in c with defining polynomial x^3 + (1 + (2 - a)*b)*x^2 + (2 + 2*b)*x
    -b over Field in b with defining polynomial x^2 + (3 - a)*x + a over its base
sage: D.print_options(over=Infinity)
sage: D
Field in d with defining polynomial x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2
    + a) + (1 - a)*b)*c^2)*x + c over Field in c with defining polynomial x^3 + (1 + (2 - a)*b)*x^2 + (2 + 2*b)*x
    -b over Field in b with defining polynomial x^2 + (3 - a)*x + a over Field in a with defining polynomial x^2 + 4*x + 2 over Field of size 5
```

Now the option base:

```
sage: D
Field in d with defining polynomial
x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2 + a) + (1 - a)*b)*c^2)*x + c over...
    its base
sage: D.print_options(over=Integer(2))
sage: D
Field in d with defining polynomial x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2
    + a) + (1 - a)*b)*c^2)*x + c over Field in c with defining polynomial x^3 + (1 + (2 - a)*b)*x^2 + (2 + 2*b)*x
    -b over Field in b with defining polynomial x^2 + (3 - a)*x + a over Field in a with defining polynomial x^2 + 4*x + 2 over Finite Field of size 5
```

```
sage: from sage.all import *

>>> # needs sage.rings.finite_rings
>>> D
Field in d with defining polynomial
x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2 + a) + (1 - a)*b)*c^2)*x + c over...
    its base
>>> D.print_options(over=Integer(2))
>>> D
Field in d with defining polynomial x^2 + ((1 - a) + ((1 + 2*a) - b)*c + ((2
    + a) + (1 - a)*b)*c^2)*x + c over Field in c with defining polynomial x^3 + (1 + (2 - a)*b)*x^2 + (2 + 2*b)*x
    -b over Field in b with defining polynomial x^2 + (3 - a)*x + a over Field in a with defining polynomial x^2 + 4*x + 2 over Finite Field of size 5
```

Now the option base:

```
sage: # needs sage.rings.finite_rings
sage: d^2
-c + ((-1 + a) + ((-1 + 3*a) + b)*c + ((3 - a) + (-1 + a)*b)*c^2)*d
sage: D.basis_over(B)
```
random_element()

Return a random element in this extension.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: K = GF(5^2).over()  # over GF(5)
sage: x = K.random_element(); x  # random
3 + z2
sage: x.parent()
Field in z2 with defining polynomial x^2 + 4*x + 2 over its base
sage: x.parent() is K
True
```

relative_degree()

Return the degree of this extension over its base

EXAMPLES:

```python
sage: A = GF(5^4).over(GF(5^2))  # ...
```

(continues on next page)
sage: A.relative_degree()  # needs sage.rings.finite_rings
2

>>> from sage.all import *

>>> A = GF(Integer(5)**Integer(4)).over(GF(Integer(5)**Integer(2)))  # needs sage.rings.finite_rings

See also:

degree(), absolute_degree()

sage.rings.ring_extension.common_base(K, L, degree)

Return a common base on which K and L are defined.

INPUT:

• K – a commutative ring
• L – a commutative ring
• degree – a boolean; if true, return the degree of K and L over their common base

EXAMPLES:

>>> from sage.rings.ring_extension import common_base

sage: common_base(GF(5^3), GF(5^7), False)  # needs sage.rings.finite_rings
(Finite Field of size 5)

sage: common_base(GF(5^3), GF(5^7), True)  # needs sage.rings.finite_rings
((Finite Field of size 5, 3, 7)

sage: common_base(GF(5^3), GF(7^5), False)  # needs sage.rings.finite_rings
Traceback (most recent call last):
... Not Implemented: unable to find a common base

>>> from sage.all import *

>>> from sage.rings.ring_extension import common_base

>>> common_base(GF(Integer(5)**Integer(3)), GF(Integer(5)**Integer(7)), False)  # needs sage.rings.finite_rings
(Finite Field of size 5)

>>> common_base(GF(Integer(5)**Integer(3)), GF(Integer(7)**Integer(5)), False)  # needs sage.rings.finite_rings
Traceback (most recent call last):
... Not Implemented: unable to find a common base
When degree is set to True, we only look up for bases on which both $K$ and $L$ are finite:

```python
sage: S.<x> = QQ[]
sage: common_base(S, QQ, False)
Rational Field
sage: common_base(S, QQ, True)
Traceback (most recent call last):
  ... Not ImplementedError: unable to find a common base
```

```python
>>> from sage.all import *
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> common_base(S, QQ, False)
Rational Field
>>> common_base(S, QQ, True)
Traceback (most recent call last):
  ... Not ImplementedError: unable to find a common base
```

```python
sage.rings.ring_extension.generators(ring, base)
```

Return the generators of ring over base.

**INPUT:**
- `ring` – a commutative ring
- `base` – a commutative ring

**EXAMPLES:**

```python
sage: from sage.rings.ring_extension import generators
sage: S.<x> = QQ[]
sage: T.<y> = S[]
sage: generators(T, S)
(y,)
sage: generators(T, QQ)
(y, x)
```

```python
>>> from sage.all import *
>>> from sage.rings.ring_extension import generators
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> T = S['y']; (y,) = T._first_ngens(1)

>>> generators(T, S)
(y,)
>>> generators(T, QQ)
(y, x)
```

```python
sage.rings.ring_extension.tower_bases(ring, degree)
```

Return the list of bases of ring (including itself); if degree is True, restrict to finite extensions and return in addition the degree of ring over each base.

**INPUT:**
- `ring` – a commutative ring
- `degree` – a boolean

**EXAMPLES:**

7.1. Extension of rings
>>> from sage.all import *
>>> from sage.rings.ring_extension import tower_bases
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> T = S['y']; (y,) = T._first_ngens(1)
>>> tower_bases(T, False)
([Univariate Polynomial Ring in y over
  Univariate Polynomial Ring in x over Rational Field,
  Univariate Polynomial Ring in x over Rational Field,
  Rational Field], []
>>> tower_bases(T, True)
([Univariate Polynomial Ring in y over
  Univariate Polynomial Ring in x over Rational Field], [1])

Return the variable names of the generators of ring over base.

INPUT:

- ring – a commutative ring
• base – a commutative ring

EXAMPLES:

```
sage: from sage.rings.ring_extension import variable_names
sage: S.<x> = QQ[]
sage: T.<y> = S[]
```

```
sage: variable_names(T, S)
('y',)
sage: variable_names(T, QQ)
('y', 'x')
```

>>> from sage.all import *
>>> from sage.rings.ring_extension import variable_names

```
>>> S = QQ['x']; (x,) = S._first_ngens(1)
```

```
>>> T = S['y']; (y,) = T._first_ngens(1)
```

```
>>> variable_names(T, S)
('y',)
```

```
>>> variable_names(T, QQ)
('y', 'x')
```

### 7.2 Elements lying in extension of rings

AUTHOR:

• Xavier Caruso (2019)

```python
class sage.rings.ring_extension_element.RingExtensionElement
    Bases: CommutativeAlgebraElement

    Generic class for elements lying in ring extensions.

    additive_order()

        Return the additive order of this element.

        EXAMPLES:

```

```
sage: K.<a> = GF(5^4).over(GF(5^2))

# needs sage.rings.finite_rings
sage: a.additive_order()
# needs sage.rings.finite_rings
5
```

>>> from sage.all import *

```
>>> K = GF(Integer(5)**Integer(4)).over(GF(Integer(5)**Integer(2)), names=('a',)); (a,) = K._first_ngens(1)
```

```
>>> a.additive_order()
# needs sage.rings.finite_rings
5
```

backend(force=False)

        Return the backend of this element.

        INPUT:
```
• force – a boolean (default: False): if False, raise an error if the backend is not exposed

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2)
sage: K.<z> = GF(5^4).over(F)
sage: x = z^10
sage: y = x.backend()
4*z4^3 + 2*z4^2 + 4*z4 + 4
sage: y.parent()
Finite Field in z4 of size 5^4
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
F = GF(Integer(5)**Integer(2))
K = GF(Integer(5)**Integer(4)).over(F, names=('z',)); (z,) = K._first_→ngens(1)
>>> x = z**Integer(10)
>>> y = x.backend()
4*z4^3 + 2*z4^2 + 4*z4 + 4
>>> y.parent()
Finite Field in z4 of size 5^4
```

**in_base()**

Return this element as an element of the base.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: F = GF(5^2)
sage: K.<z> = GF(5^4).over(F)
sage: x = z^3 + z^2 + z + 4
sage: y = x.in_base()
z2 + 1
sage: y.parent()
Finite Field in z2 of size 5^2
```

```python
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(5)**Integer(2))
>>> K = GF(Integer(5)**Integer(4)).over(F, names=('z',)); (z,) = K._first_→ngens(1)
>>> x = z**Integer(3) + z**Integer(2) + z + Integer(4)
>>> y = x.in_base()
z2 + 1
>>> y.parent()
Finite Field in z2 of size 5^2
```

When the element is not in the base, an error is raised:
sage: z.in_base()  
˓→needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: z is not in the base

>>> from sage.all import *
>>> z.in_base()  
˓→needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: z is not in the base

sage: # needs sage.rings.finite_rings
sage: S.<X> = F[]
sage: E = S.over(F)
sage: f = E(1)
sage: g = f.in_base(); g
1
sage: g.parent()
Finite Field in z2 of size 5^2

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> S = F['X']; (X,) = S._first_ngens(1)
>>> E = S.over(F)
>>> f = E(Integer(1))
>>> g = f.in_base(); g
1
>>> g.parent()
Finite Field in z2 of size 5^2

is_nilpotent()  
Return whether if this element is nilpotent in this ring.

EXAMPLES:

sage: A.<x> = PolynomialRing(QQ)
sage: E = A.over(QQ)
sage: E(0).is_nilpotent()  
True
sage: E(x).is_nilpotent()  
False

>>> from sage.all import *
>>> A = PolynomialRing(QQ, names=('x',)); (x,) = A._first_ngens(1)
>>> E = A.over(QQ)
>>> E(Integer(0)).is_nilpotent()  
True
>>> E(x).is_nilpotent()  
False

is_prime()  
Return whether this element is a prime element in this ring.

EXAMPLES:

7.2. Elements lying in extension of rings
sage: A.<x> = PolynomialRing(QQ)
sage: E = A.over(QQ)
sage: E(x^2 + 1).is_prime()                   # needs sage.libs.pari
True
sage: E(x^2 - 1).is_prime()                   # needs sage.libs.pari
False

>>> from sage.all import *
>>> A = PolynomialRing(QQ, names=('x',)); (x,) = A._first_ngens(1)
>>> E = A.over(QQ)
>>> E(x**Integer(2) + Integer(1)).is_prime() # needs sage.libs.pari
True
>>> E(x**Integer(2) - Integer(1)).is_prime() # needs sage.libs.pari
False

is_square (root=False)
Return whether this element is a square in this ring.

INPUT:

• root – a boolean (default: False); if True, return also a square root

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: K.<a> = GF(5^3).over()
sage: a.is_square()
False
sage: a.is_square(root=True)
(False, None)
sage: b = a + 1
sage: b.is_square()
True
sage: b.is_square(root=True)
(True, 2 + 3*a + a^2)

>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(5)**Integer(3)).over(names=(a,)); (a,) = K._first_ngens(1)
>>> a.is_square()
False
>>> a.is_square(root=True)
(False, None)
>>> b = a + Integer(1)
>>> b.is_square()
True
>>> b.is_square(root=True)
(True, 2 + 3*a + a^2)

is_unit ()
Return whether if this element is a unit in this ring.

EXAMPLES:
sage: A.<x> = PolynomialRing(QQ)
sage: E = A.over(QQ)
sage: E(4).is_unit()
  True
sage: E(x).is_unit()
  False

>>> from sage.all import *
>>> A = PolynomialRing(QQ, names=('x',)); (x,) = A._first_ngens(1)
>>> E = A.over(QQ)
>>> E(Integer(4)).is_unit()
  True
>>> E(x).is_unit()
  False

### multiplicative_order()
Return the multiplicative order of this element.

**EXAMPLES:**

```python
sage: K.<a> = GF(5^4).over(GF(5^2))   # needs sage.rings.finite_rings
sage: a.multiplicative_order()
  624
```

```python
>>> from sage.all import *
>>> K = GF(Integer(5)**Integer(4)).over(GF(Integer(5)**Integer(2)), names=('a',)); (a,) = K._first_ngens(1)  # needs sage.rings.finite_rings
>>> a.multiplicative_order()
  624
```

### sqrt (extend=True, all=False, name=None)
Return a square root or all square roots of this element.

**INPUT:**

- `extend` -- a boolean (default: True); if “True”, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the root is not in the ring.
- `all` -- a boolean (default: False); if True, return all square roots of this element, instead of just one.
- `name` -- Required when `extend=True` and `self` is not a square. This will be the name of the generator extension.

**Note:** The option `extend=True` is often not implemented.

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
sage: K.<a> = GF(5^3).over()
sage: b = a + 1
sage: b.sqrt()  # extend=True
  2 + 3*a + a^2
sage: b.sqrt(all=True)  # extend=True
  [2 + 3*a + a^2, 3 + 2*a - a^2]
```

7.2. Elements lying in extension of rings
class sage.rings.ring_extension_element.RingExtensionFractionFieldElement

Bases: RingExtensionElement

A class for elements lying in fraction fields of ring extensions.

denominator()

Return the denominator of this element.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: R.<x> = ZZ[]
sage: A.<a> = ZZ.extension(x^2 - 2)
sage: OK = A.over()  # over ZZ
sage: K = OK.fraction_field(); K
Fraction Field of Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
sage: x = K(1/a); x
a/2
sage: denom = x.denominator(); denom
2
```

The denominator is an element of the ring which was used to construct the fraction field:

```python
sage: denom.parent()  # needs sage.rings.number_field
Maximal Order generated by a in Number Field in a with defining polynomial x^2 - 2 over its base
sage: denom.parent() is OK  # needs sage.rings.number_field
True
```
numererator()

Return the numerator of this element.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: A.<a> = ZZ.extension(x^2 - 2)
sage: OK = A.over()  # over ZZ
sage: K = OK.fraction_field(); K
Fraction Field of Maximal Order generated by a in Number Field in a
with defining polynomial x^2 - 2 over its base
sage: x = K(1/a); x
a/2
sage: num = x.numerator(); num
a
```

The numerator is an element of the ring which was used to construct the fraction field:

```python
sage: num.parent()  # needs sage.rings.number_field
Maximal Order generated by a in Number Field in a
with defining polynomial x^2 - 2 over its base
sage: num.parent() is OK  # needs sage.rings.number_field
True
```

(continues on next page)
needs sage.rings.number_field

True

class sage.rings.ring_extension_element.RingExtensionWithBasisElement

    Bases: RingExtensionElement

A class for elements lying in infinite free extensions.

    charpoly (base=None, var='x')

    Return the characteristic polynomial of this element over base.

    INPUT:

        • base – a commutative ring (which might be itself an extension) or None

    EXAMPLES:

    sage: # needs sage.rings.finite_rings
    sage: F = GF(5)
    sage: K.<a> = GF(5^3).over(F)
    sage: L.<b> = GF(5^6).over(K)
    sage: u = a/(1+b)
    sage: chi = u.charpoly(K); chi
    x^2 + (1 + 2*a + 3*a^2)*x + 3 + 2*a^2

    >>> from sage.all import *
    >>> # needs sage.rings.finite_rings
    >>> F = GF(Integer(5))
    >>> K = GF(Integer(5)**Integer(3)).over(F, names=('a',)); (a,) = K._first_ngens(1)
    >>> L = GF(Integer(5)**Integer(6)).over(K, names=('b',)); (b,) = L._first_ngens(1)
    >>> u = a/(Integer(1)+b)
    >>> chi = u.charpoly(K); chi
    x^2 + (1 + 2*a + 3*a^2)*x + 3 + 2*a^2

    We check that the charpoly has coefficients in the base ring:

    sage: chi.base_ring()  # needs sage.rings.finite_rings
    Field in a with defining polynomial x^3 + 3*x + 3 over its base

    >>> from sage.all import *
    >>> # needs sage.rings.finite_rings
    >>> chi.base_ring()  # needs sage.rings.finite_rings
    True

    and that it annihilates u:
Similarly, one can compute the characteristic polynomial over $F$:

```python
sage: u.charpoly(F)
# needs sage.rings.finite_rings
x^6 + x^4 + 2*x^3 + 3*x + 4
```

A different variable name can be specified:

```python
sage: u.charpoly(F, var='t')
# needs sage.rings.finite_rings
t^6 + t^4 + 2*t^3 + 3*t + 4
```

If `base` is omitted, it is set to its default which is the base of the extension:

```python
sage: u.charpoly()
# needs sage.rings.finite_rings
x^2 + (1 + 2*a + 3*a^2)*x + 3 + 2*a^2
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):
matrix (base=\text{None})

Return the matrix of the multiplication by this element (in the basis output by \texttt{basis\_over()}).

\textbf{INPUT:}

- \texttt{base} – a commutative ring (which might be itself an extension) or \texttt{None}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: # needs sage.rings.finite_rings
sage: K.<a> = GF(5^3).over()   # over GF(5)
sage: L.<b> = GF(5^6).over(K)
sage: u = a/(1+b)
sage: u
(2 + a + 3*a^2) + (3 + 3*a + a^2)*b
sage: b*u
(3 + 2*a^2) + (2 + 2*a - a^2)*b
sage: u.matrix(K)
[2 + a + 3*a^2 3 + 3*a + a^2]
[ 3 + 2*a^2 2 + 2*a - a^2]
sage: u.matrix(GF(5))
[2 1 3 3 3 1]
[1 3 1 2 0 3]
[2 3 3 1 3 0]
[3 0 2 2 2 4]
[4 2 0 3 0 2]
[0 4 2 4 2 0]
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> K = GF(Integer(5)**Integer(3)).over(names=('a',)); (a,) = K._first_
˓→ngens(1)# over GF(5)
>>> L = GF(Integer(5)**Integer(6)).over(K, names=('b',)); (b,) = L._first_
˓→ngens(1)
>>> u = a/(Integer(1)+b)
>>> u
(2 + a + 3*a^2) + (3 + 3*a + a^2)*b
>>> b*u
(3 + 2*a^2) + (2 + 2*a - a^2)*b
>>> u.matrix(K)
[2 + a + 3*a^2 3 + 3*a + a^2]
[ 3 + 2*a^2 2 + 2*a - a^2]
>>> u.matrix(GF(Integer(5)))
[2 1 3 3 3 1]
[1 3 1 2 0 3]
[2 3 3 1 3 0]
[3 0 2 2 2 4]
[4 2 0 3 0 2]
[0 4 2 4 2 0]
\end{verbatim}

If \texttt{base} is omitted, it is set to its default which is the base of the extension:
Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```
sage: u.matrix(GF(5^2))
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```

```
sage: u.matrix(GF(Integer(5)**Integer(2)))
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```

### minpoly

`minpoly(base=None, var='x')`

Return the minimal polynomial of this element over `base`.

**INPUT:**

- `base` – a commutative ring (which might be itself an extension) or `None`

**EXAMPLES:**

```
sage: # needs sage.rings.finite_rings
sage: F = GF(5)
sage: K.<a> = GF(5^3).over(F)
sage: L.<b> = GF(5^6).over(K)
sage: u = 1 / (a+b)
sage: chi = u.minpoly(K); chi
x^2 + (2*a + a^2)*x - 1 + a
```

We check that the minimal polynomial has coefficients in the base ring:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(Integer(5))
 >>> K = GF(Integer(5)**Integer(3)).over(F, names=('a',)); (a,) = K._first_ngens(1)
 >>> L = GF(Integer(5)**Integer(6)).over(K, names=('b',)); (b,) = L._first_ngens(1)
 >>> u = Integer(1) / (a+b)
 >>> chi = u.minpoly(K); chi
x^2 + (2*a + a^2)*x - 1 + a
```
sage: chi.base_ring()  # needs sage.rings.finite_rings
Field in a with defining polynomial x^3 + 3*x + 3 over its base
sage: chi.base_ring() is K  # needs sage.rings.finite_rings
True

and that it annihilates u:

sage: chi(u)  # needs sage.rings.finite_rings
0

Similarly, one can compute the minimal polynomial over F:

sage: u.minpoly(F)  # needs sage.rings.finite_rings
x^6 + 4*x^5 + x^4 + 2*x^2 + 3

A different variable name can be specified:

sage: u.minpoly(F, var='t')  # needs sage.rings.finite_rings
t^6 + 4*t^5 + t^4 + 2*t^2 + 3

If base is omitted, it is set to its default which is the base of the extension:

different base name can be specified:

sage: u.minpoly()  # needs sage.rings.finite_rings
x^2 + (2*a + a^2)*x - 1 + a
>>> from sage.all import *
>>> u.minpoly()
    # needs sage.rings.finite_rings
x^2 + (2*a + a^2)*x - 1 + a

Note that base must be an explicit base over which the extension has been defined (as listed by the method bases()):

```
sage: u.minpoly(GF(5^2))
    # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```

```
>>> from sage.all import *
>>> u.minpoly(GF(Integer(5)**Integer(2)))
    # needs sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```

**norm** (base=None)

Return the norm of this element over base.

**INPUT:**

- base - a commutative ring (which might be itself an extension) or None

**EXAMPLES:**

```
sage: # needs sage.rings.finite_rings
sage: F = GF(5)
sage: K.<a> = GF(5^3).over(F)
sage: L.<b> = GF(5^6).over(K)
sage: u = a/(1+b)
sage: nr = u.norm(K); nr
3 + 2*a^2
```

We check that the norm lives in the base ring:

```
sage: nr.parent()
    # needs sage.rings.finite_rings
Field in a with defining polynomial x^3 + 3*x + 3 over its base
sage: nr.parent() is K
    # needs sage.rings.finite_rings
True
```

7.2. Elements lying in extension of rings 221
Similarly, one can compute the norm over F:

```python
sage: u.norm(F)  # needs sage.rings.finite_rings
4
```

We check the transitivity of the norm:

```python
sage: u.norm(F) == nr.norm(F)  # needs sage.rings.finite_rings
True
```

```python
sage: u.norm(F) == nr.norm(F)  # needs sage.rings.finite_rings
True
```

If `base` is omitted, it is set to its default which is the base of the extension:

```python
sage: u.norm()  # needs sage.rings.finite_rings
3 + 2*a^2
```

```python
sage: u.norm()  # needs sage.rings.finite_rings
3 + 2*a^2
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```python
sage: u.norm(GF(5^2))  # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```

```python
sage: u.norm(GF(Integer(5)**Integer(2)))  # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z2 of size 5^2
```
polynomial\((base=\text{None}, \ var='x')\)

Return a polynomial (in one or more variables) over \(base\) whose evaluation at the generators of the parent equals this element.

INPUT:

- \(base\) – a commutative ring (which might be itself an extension) or \text{None}

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(5**2).over() # over GF(5)
sage: K.<b> = GF(5**4).over(F)
sage: L.<c> = GF(5**12).over(K)
sage: u = 1/(a + b + c); u
(2 + (-1 - a)*b) + ((2 + 3*a) + (1 - a)*b)*c + ((-1 - a) - a*b)*c^2
sage: P = u.polynomial(K); P
((-1 - a) - a*b)*x^2 + ((2 + 3*a) + (1 - a)*b)*x + 2 + (-1 - a)*b
sage: P.base_ring() is K
True
sage: P(c) == u
True
```

When the base is \(F\), we obtain a bivariate polynomial:

```
sage: P = u.polynomial(F); P  # needs sage.rings.finite_rings
(-a)*x0^2*x1 + (-1 - a)*x0^2 + (1 - a)*x0*x1 + (2 + 3*a)*x0 + (-1 - a)*x1 + 2
```

We check that its value at the generators is the element we started with:

```
sage: L.gens(F) # needs sage.rings.finite_rings
(c, b)
sage: P(c, b) == u # needs sage.rings.finite_rings
(continues on next page)```
Similarly, when the base is $\text{GF}(5)$, we get a trivariate polynomial:

```sage
sage: P = u.polynomial(GF(5)); P
-x0^2*x1*x2 - x0^2*x2 - x0*x1*x2 - x0^2 + x0*x1 - 2*x0*x2 - x1*x2 + 2*x0 - x1 + 2
```

Sage: $P(c, b, a)$ == $u$ # needs sage.rings.finite_rings
True

Different variable names can be specified:

```sage
sage: u.polynomial(GF(5), var='y')
-x^2*y*z - x^2*z - x*y*z - x^2 + x*y - 2*x*z - y*z + 2*x - y + 2
```

If `base` is omitted, it is set to its default which is the base of the extension:

```sage
sage: u.polynomial()
((-1 - a) - a*b)*x^2 + ((2 + 3*a) + (1 - a)*b)*x + 2 + (-1 - a)*b
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```sage
sage: u.polynomial(GF(5^3))
Traceback (most recent call last):
  ...
ValueError: not (explicitly) defined over Finite Field in z3 of size 5^3
```
trace\(\text{\texttt{(base=\texttt{None})}}\)

Return the trace of this element over base.

INPUT:

- base – a commutative ring (which might be itself an extension) or None

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(Integer(5))
sage: K.<a> = GF(5^3).over(F)
sage: L.<b> = GF(5^6).over(K)
sage: u = a/(1+b)
sage: tr = u.trace(K); tr
-1 + 3*a + 2*a^2
```

We check that the trace lives in the base ring:

```
sage: tr.parent()  # needs sage.rings.finite_rings
Field in a with defining polynomial x^3 + 3*x + 3 over its base
sage: tr.parent() is K  # needs sage.rings.finite_rings
True
```

Similarly, one can compute the trace over F:

```
sage: u.trace(F)  # needs sage.rings.finite_rings
0
```
>>> from sage.all import *
>>> u.trace(F)                      #...
˓→needs sage.rings.finite_rings
0

We check the transitivity of the trace:

```
sage: u.trace(F) == tr.trace(F)      #...
˓→needs sage.rings.finite_rings
True
```

>>> from sage.all import *
>>> u.trace(F) == tr.trace(F)       #...
˓→needs sage.rings.finite_rings
True

If `base` is omitted, it is set to its default which is the base of the extension:

```
sage: u.trace()                     #...
˓→needs sage.rings.finite_rings
-1 + 3*a + 2*a^2
```

```
>>> from sage.all import *
>>> u.trace()                       #...
˓→needs sage.rings.finite_rings
-1 + 3*a + 2*a^2
```

Note that `base` must be an explicit base over which the extension has been defined (as listed by the method `bases()`):

```
sage: u.trace(GF(5^2))              #...
˓→needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueException: not (explicitly) defined over Finite Field in z2 of size 5^2
```

```
>>> from sage.all import *
>>> u.trace(GF(Integer(5)**Integer(2)))  # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueException: not (explicitly) defined over Finite Field in z2 of size 5^2
```

`vector (base=None)`

Return the vector of coordinates of this element over `base` (in the basis output by the method `basis_over()`).

INPUT:

- `base` – a commutative ring (which might be itself an extension) or `None`

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(5)
sage: K.<a> = GF(5^2).over()  # over F
sage: L.<b> = GF(5^6).over(K)
```

(continues on next page)
sage: x = (a+b)^4; x
(-1 + a) + (3 + a)*b + (1 - a)*b^2
sage: x.vector(K) # basis is (1, b, b^2)
(-1 + a, 3 + a, 1 - a)
sage: x.vector(F) # basis is (1, a, b, a*b, b^2, a*b^2)
(4, 1, 3, 1, 1, 4)

If base is omitted, it is set to its default which is the base of the extension:

sage: x.vector() # needs sage.rings.finite_rings
(-1 + a, 3 + a, 1 - a)

Note that base must be an explicit base over which the extension has been defined (as listed by the method bases()):

sage: x.vector(GF(5^3)) # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z3 of size 5^3

sage: x.vector(GF(Integer(5)**Integer(3))) # needs sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: not (explicitly) defined over Finite Field in z3 of size 5^3

7.2. Elements lying in extension of rings
7.3 Morphisms between extension of rings

AUTHOR:
- Xavier Caruso (2019)

class sage.rings.ring_extension_morphism.MapFreeModuleToRelativeRing
Bases: Map

Base class of the module isomorphism between a ring extension and a free module over one of its bases.

is_injective()
Return whether this morphism is injective.

EXAMPLES:

```
sage: K = GF(11^6).over(GF(11^3))
# needs sage.rings.finite_rings
sage: V, i, j = K.free_module()
# needs sage.rings.finite_rings
sage: i.is_injective()
# needs sage.rings.finite_rings
True
```

is_surjective()
Return whether this morphism is surjective.

EXAMPLES:

```
sage: K = GF(11^6).over(GF(11^3))
# needs sage.rings.finite_rings
sage: V, i, j = K.free_module()
# needs sage.rings.finite_rings
sage: i.is_surjective()
# needs sage.rings.finite_rings
True
```

class sage.rings.ring_extension_morphism.MapRelativeRingToFreeModule
Bases: Map
Base class of the module isomorphism between a ring extension and a free module over one of its bases.

**is_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

```python
sage: K = GF(11^6).over(GF(11^3))  # needs sage.rings.finite_rings
sage: V, i, j = K.free_module()  # needs sage.rings.finite_rings
sage: j.is_injective()  # needs sage.rings.finite_rings
True
```

**is_surjective()**

Return whether this morphism is injective.

**EXAMPLES:**

```python
sage: K = GF(11^6).over(GF(11^3))  # needs sage.rings.finite_rings
sage: V, i, j = K.free_module()  # needs sage.rings.finite_rings
sage: j.is_surjective()  # needs sage.rings.finite_rings
True
```

class sage.rings.ring_extension_morphism.

A class for implementing isomorphisms taking an element of the backend to its ring extension.

class sage.rings.ring_extension_morphism.

A class for implementing isomorphisms from a ring extension to its backend.
A class for ring homomorphisms between extensions.

**base_map()**

Return the base map of this morphism or just None if the base map is a coercion map.

**EXAMPLES:**

```
sage: F = GF(5)
sage: K.<a> = GF(5^2).over(F)  # needs sage.rings.finite_rings
sage: L.<b> = GF(5^6).over(K)  # needs sage.rings.finite_rings

>>> from sage.all import *
>>> F = GF(Integer(5))
>>> K = GF(Integer(5)**Integer(2)).over(F, names=('a',)); (a,) = K._first_ngens(1)  # needs sage.rings.finite_rings
>>> L = GF(Integer(5)**Integer(6)).over(K, names=('b',)); (b,) = L._first_ngens(1)  # needs sage.rings.finite_rings
```

We define the absolute Frobenius of L:

```
sage: FrobL = L.hom([b^5, a^5]); FrobL  # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
  Defn: b |--> (-1 + a) + (1 + 2*a)*b + a*b^2
  with map on base ring:
  a |--> 1 - a
sage: FrobL.base_map()  # needs sage.rings.finite_rings
Ring morphism:
  From: Field in a with defining polynomial x^2 + 4*x + 2 over its base
  To:   Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
    Defn: a |--> 1 - a
```

```
>>> from sage.all import *
>>> FrobL = L.hom([b**Integer(5), a**Integer(5)]); FrobL  # needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
  Defn: b |--> (-1 + a) + (1 + 2*a)*b + a*b^2
  with map on base ring:
  a |--> 1 - a
sage: FrobL.base_map()  # needs sage.rings.finite_rings
Ring morphism:
  From: Field in a with defining polynomial x^2 + 4*x + 2 over its base
  To:   Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
    Defn: a |--> 1 - a
```

The square of FrobL acts trivially on K; in other words, it has a trivial base map: 
sage: phi = FrobL^2; phi
# needs sage.rings.finite_rings
Ring endomorphism of
Field in b with defining polynomial x^3 + (2 + 2*a)*x - a over its base
Defn: b |--> 2 + 2*a*b + (2 - a)*b^2
sage: phi.base_map()
# needs sage.rings.finite_rings

is_identity()
Return whether this morphism is the identity.

EXAMPLES:

```sage
sage: # needs sage.rings.finite_rings
K.<a> = GF(5^2).over()
# over GF(5)
FrobK = K.hom([a^5])
FrobK.is_identity()
False
(FrobK^2).is_identity()
True
```

Coercion maps are not considered as identity morphisms:

```sage
sage: # needs sage.rings.finite_rings
K = GF(Integer(5)**Integer(2)).over(names=('a',)); (a,) = K._first_
FrobK = K.hom([a**Integer(5)])
FrobK.is_identity()
False
(FrobK**Integer(2)).is_identity()
True
```

7.3. Morphisms between extension of rings

is_injective()

Return whether this morphism is injective.

EXAMPLES:

```sage
# needs sage.rings.finite_rings
sage: K = GF(5^10).over(GF(5^5))
sage: iota = K.defining_morphism(); iota
Ring morphism:
  From: Finite Field in z5 of size 5^5
  To:   Finite Field of size 7 over its base
    Defn: z5 |--> z5
sage: iota.is_injective()
True

sage: K = GF(7).over(ZZ)
sage: iota = K.defining_morphism(); iota
Ring morphism:
  From: Integer Ring
  To:   Finite Field of size 7 over its base
    Defn: 1 |--> 1
sage: iota.is_injective()
False
```

is_surjective()

---
Return whether this morphism is surjective.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: K = GF(5^10).over(GF(5^5))
\text{Ring morphism:}
\text{From: Finite Field in } z5 \text{ of size } 5^5
\text{To: Field in } z10 \text{ with defining polynomial}
\quad x^2 + (2*z5^3 + 2*z5^2 + 4*z5 + 4)*x + z5 \text{ over its base}
\text{Defn: } z5 \mapsto z5
sage: iota.is_surjective()
\text{False}

sage: K = GF(7).over(ZZ)
\text{Ring morphism:}
\text{From: Integer Ring}
\text{To: Finite Field of size 7 over its base}
\text{Defn: } 1 \mapsto 1
sage: iota.is_surjective()
\text{True}
```

```python
>>> from sage.all import *

>>> K = GF(Integer(5)**Integer(10)).over(GF(Integer(5)**Integer(5)))

>>> iota = K.defining_morphism(); iota
\text{Ring morphism:}
\text{From: Finite Field in } z5 \text{ of size } 5^5
\text{To: Field in } z10 \text{ with defining polynomial}
\quad x^2 + (2*z5^3 + 2*z5^2 + 4*z5 + 4)*x + z5 \text{ over its base}
\text{Defn: } z5 \mapsto z5

>>> iota.is_surjective()
\text{False}

>>> K = GF(Integer(7)).over(ZZ)

>>> iota = K.defining_morphism(); iota
\text{Ring morphism:}
\text{From: Integer Ring}
\text{To: Finite Field of size 7 over its base}
\text{Defn: } 1 \mapsto 1

>>> iota.is_surjective()
\text{True}
```
8.1 Generic data structures and algorithms for rings

AUTHORS:

• Lorenz Panny (2022): ProductTree, prod_with_derivative()

class sage.rings.generic.ProductTree(leave)

Bases: object

A simple binary product tree, i.e., a tree of ring elements in which every node equals the product of its children. (In particular, the root equals the product of all leaves.)

Product trees are a very useful building block for fast computer algebra. For example, a quasilinear-time Discrete Fourier Transform (the famous Fast Fourier Transform) can be implemented as follows using the remainders() method of this class:

sage: # needs sage.rings.finite_rings
sage: from sage.rings.generic import ProductTree
sage: F = GF(65537)
sage: a = F(1111)
sage: assert a.multiplicative_order() == 1024
sage: R.<x> = F[]
sage: ms = [x - a^i for i in range(1024)]  # roots of unity
sage: ys = [F.random_element() for _ in range(1024)]  # input vector
sage: tree = ProductTree(ms)
sage: zs = tree.remainders(R(ys))  # compute FFT!
sage: zs == [R(ys) % m for m in ms]
True

Similarly, the interpolation() method can be used to implement the inverse Fast Fourier Transform:
This class encodes the tree as *layers*: Layer 0 is just a tuple of the leaves. Layer $i + 1$ is obtained from layer $i$ by replacing each pair of two adjacent elements by their product, starting from the left. (If the length is odd, the unpaired element at the end is simply copied as is.) This iteration stops as soon as it yields a layer containing only a single element (the root).

**Note:** Use this class if you need the `remainders()` method. To compute just the product, `prod()` is likely faster.

**INPUT:**

- *leaves* – an iterable of elements in a common ring

**EXAMPLES:**

```python
sage: from sage.rings.generic import ProductTree
sage: R.<x> = GF(101)[]

sage: vs = [x - i for i in range(1,10)]

sage: tree = ProductTree(vs)

sage: tree.root()
x^9 + 56*x^8 + 62*x^7 + 44*x^6 + 47*x^5 + 42*x^4 + 15*x^3 + 11*x^2 + 12*x + 13

sage: tree.remainders(x^7 + x + 1)
[3, 30, 70, 27, 58, 72, 98, 98, 23]

sage: tree.remainders(x^100)
[1, 1, 1, 1, 1, 1, 1, 1, 1]

>>> from sage.all import *

>>> from sage.rings.generic import ProductTree

>>> R = GF(Integer(101))[x]; (x,) = R._first_ngens(1)

>>> vs = [x - i for i in range(Integer(1),Integer(10))]

>>> tree = ProductTree(vs)

>>> tree.root()
x^9 + 56*x^8 + 62*x^7 + 44*x^6 + 47*x^5 + 42*x^4 + 15*x^3 + 11*x^2 + 12*x + 13

>>> tree.remainders(x^7 + x + Integer(1))
[3, 30, 70, 27, 58, 72, 98, 98, 23]

>>> tree.remainders(x^100)
[1, 1, 1, 1, 1, 1, 1, 1, 1]

sage: # needs sage.libs.pari

sage: vs = prime_range(100)

sage: tree = ProductTree(vs)

sage: tree.root().factor()
2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 * 23 * 29 * 31 * 37 * 41 * 43 * 47 * 53 * 59 * 61 * 67 * 71 * 73 * 79 * 83 * 89 * 97

sage: tree.remainders(1000)
[1, 1, 1, 1, 1, 1, 1, 1, 1]
```

236 Chapter 8. Generic Data Structures and Algorithms for Rings
General Rings, Ideals, and Morphisms, Release 10.4

>>> from sage.all import *
>>> # needs sage.libs.pari
>>> vs = prime_range(Integer(100))
>>> tree = ProductTree(vs)
>>> tree.root().factor()
2 * 3 * 5 * 7 * 11 * 13 * 17 * 19 * 23 * 29 * 31 * 37 * 41 * 43 * 47 * 53 * 59 ...
   → 61 * 67 * 71 * 73 * 79 * 83 * 89 * 97
>>> tree.remainders(Integer(3599))
[1, 2, 4, 1, 2, 11, 12, 8, 11, 3, 3, 10, 32, 30, 27, 48, 0, 0, 48, 49, 22, 44, 30, ...
   → 39, 10]

We can access the individual layers of the tree:

\[
\begin{align*}
\text{sage: } & \text{tree.layers} \\
\text{# needs sage.libs.pari} \\
& [(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, \ldots \\
\text{→ 79, 83, 89, 97)}, \\
& (6, 35, 143, 323, 667, 1147, 1763, 2491, 3599, 4757, 5767, 7387, 97), \\
& (210, 46189, 765049, 4391633, 17120443, 42600829, 97), \\
& (9699690, 3359814435017, 729345064647247, 97), \\
& (32589158477190044730, 70746471270782959), \\
& (2305567963945518424753102147331756070,)]
\end{align*}
\]

\[
\begin{align*}
\text{sage: } & \text{tree.layers} \\
\text{# needs sage.libs.pari} \\
& [(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, \ldots \\
\text{→ 79, 83, 89, 97)}, \\
& (6, 35, 143, 323, 667, 1147, 1763, 2491, 3599, 4757, 5767, 7387, 97), \\
& (210, 46189, 765049, 4391633, 17120443, 42600829, 97), \\
& (9699690, 3359814435017, 729345064647247, 97), \\
& (32589158477190044730, 70746471270782959), \\
& (2305567963945518424753102147331756070,)]
\end{align*}
\]

interpolation(xs)

Given a sequence \(xs\) of values, one per leaf, return a single element \(x\) which is congruent to the \(i\)th value in \(xs\) modulo the \(i\)th leaf, for all \(i\).

This is an explicit version of the Chinese remainder theorem; see also \(\text{CRT()}\). Using this product tree is faster for repeated calls since the required CRT bases are cached after the first run.

The base ring must support the \(\text{xgcd()}\) function for this method to work.

EXAMPLES:

\[
\begin{align*}
\text{sage: from sage.rings.generic import ProductTree} \\
\text{sage: vs = prime_range(100)} \\
\text{sage: tree = ProductTree(vs)} \\
\text{sage: tree.interpolation([1, 1, 2, 1, 9, 1, 7, 15, 8, 20, 15, 6, 27, 11, 2, 6,} \\
\text{   → 0, 25, 49, 5, 51, 4, 19, 74, 13])} \\
1085749272377676749812331719267
\end{align*}
\]

\[
\begin{align*}
\text{>>> from sage.all import *} \\
\text{>>> from sage.rings.generic import ProductTree} \\
\text{>>> vs = prime_range(Integer(100))} \\
\text{>>> tree = ProductTree(vs)} \\
\text{>>> tree.interpolation([Integer(1), Integer(1), Integer(2), Integer(1),} \\
\text{   → Integer(1)])} \\
\text{(continues on next page)}
\end{align*}
\]
This method is faster than CRT() for repeated calls with the same moduli:

```python
sage: vs = prime_range(1000, 2000)
sage: rs = lambda: [randrange(1, 100) for _ in vs]
sage: tree = ProductTree(vs)
sage: %timeit CRT(rs(), vs) # not tested
372 µs ± 3.34 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
sage: %timeit tree.interpolation(rs()) # not tested
146 µs ± 479 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

leaves()

Return a tuple containing the leaves of this product tree.

EXAMPLES:

```python
sage: from sage.rings.generic import ProductTree
sage: R.<x> = GF(101)[]
sage: vs = [x - i for i in range(1, 10)]
sage: tree = ProductTree(vs)
sage: tree.leaves()
(x + 100, x + 99, x + 98, ..., x + 93, x + 92)
sage: tree.leaves() == tuple(vs)
True
```

remainders(x)

Given a value x, return a list of all remainders of x modulo the leaves of this product tree.

The base ring must support the % operator for this method to work.

INPUT:

- x – an element of the base ring of this product tree
EXAMPLES:

```python
sage: # needs sage.libs.pari
sage: from sage.rings.generic import ProductTree
sage: vs = prime_range(100)
\[1, 1, 2, 1, 9, 1, 7, 15, 8, 20, 15, 6, 27, 11, 2, 6, 0, 25, 49, 5, 51, 4, 19, \rightarrow 74, 13]$$
```

```python
\[\text{[n % v for v in vs]}\]
\[[1, 1, 2, 1, 9, 1, 7, 15, 8, 20, 15, 6, 27, 11, 2, 6, 0, 25, 49, 5, 51, 4, 19, \rightarrow 74, 13]\]
```

```python
sage: # needs sage.libs.pari
sage: from sage.rings.generic import ProductTree
sage: vs = prime_range(Integer(100))
\[1, 1, 2, 1, 9, 1, 7, 15, 8, 20, 15, 6, 27, 11, 2, 6, 0, 25, 49, 5, 51, 4, 19, \rightarrow 74, 13]\n```

```python
\[\text{[n % v for v in vs]}\]
\[[1, 1, 2, 1, 9, 1, 7, 15, 8, 20, 15, 6, 27, 11, 2, 6, 0, 25, 49, 5, 51, 4, 19, \rightarrow 74, 13]\]
```

```python
sage: from sage.rings.generic import ProductTree
sage: R.<x> = GF(101)
\[x^9 + 56*x^8 + 62*x^7 + 44*x^6 + 47*x^5 + 42*x^4 + 15*x^3 + 11*x^2 + 12*x + 13\]
```

```python
sage: tree.root() == prod(vs)
True
```

```python
sage: from sage.rings.generic import ProductTree
sage: R = GF(Integer(101))[x]; (x,) = R._first_ngens(1)
\[x^9 + 56*x^8 + 62*x^7 + 44*x^6 + 47*x^5 + 42*x^4 + 15*x^3 + 11*x^2 + 12*x + 13\]
```

```python
sage: tree.root() == prod(vs)
True
```

```python
sage.rings.generic.prod_with_derivative(pairs)
```

Given an iterable of pairs \((f, \partial f)\) of ring elements, return the pair \((\prod f, \partial \prod f)\), assuming \(\partial\) is an operator obeying the standard product rule.

This function is entirely algebraic, hence still works when the elements \(f\) and \(\partial f\) are all passed through some ring homomorphism first. One particularly useful instance of this is evaluating the derivative of a product of polynomials at a point without fully expanding the product; see the second example below.

8.1. Generic data structures and algorithms for rings
INPUT:

• pairs – an iterable of tuples \((f, \partial f)\) of elements of a common ring

ALGORITHM: Repeated application of the product rule.

EXAMPLES:

```python
sage: from sage.rings.generic import prod_with_derivative
sage: R.<x> = ZZ[]
sage: fs = [x^2 + 2*x + 3, 4*x + 5, 6*x^7 + 8*x + 9]

sage: prod(fs)
24*x^10 + 78*x^9 + 132*x^8 + 90*x^7 + 32*x^4 + 140*x^3 + 293*x^2 + 318*x + 135

sage: prod(fs).derivative()
240*x^9 + 702*x^8 + 1056*x^7 + 630*x^6 + 128*x^3 + 420*x^2 + 586*x + 318

sage: F, dF = prod_with_derivative((f, f.derivative()) for f in fs)

sage: F
24*x^10 + 78*x^9 + 132*x^8 + 90*x^7 + 32*x^4 + 140*x^3 + 293*x^2 + 318*x + 135

sage: dF
240*x^9 + 702*x^8 + 1056*x^7 + 630*x^6 + 128*x^3 + 420*x^2 + 586*x + 318
```

The main reason for this function to exist is that it allows us to evaluate the derivative of a product of polynomials at a point \(\alpha\) without ever fully expanding the product as a polynomial:
9.1 Big O for various types (power series, p-adics, etc.)

See also:
- asymptotic expansions
- p-adic numbers
- power series
- polynomials

`sage.rings.big_oh.O(*x, **kwds)`

Big O constructor for various types.

**EXAMPLES:**

This is useful for writing power series elements:

```
sage: R.<t> = ZZ[[t]]
sage: (1+t)^10 + O(t^5)
1 + 10*t + 45*t^2 + 120*t^3 + 210*t^4 + O(t^5)
```

```
>>> from sage.all import *
>>> R = ZZ[['t']]; (t,) = R._first_ngens(1)
>>> (Integer(1)+t)**Integer(10) + O(t**Integer(5))
1 + 10*t + 45*t^2 + 120*t^3 + 210*t^4 + O(t^5)
```

A power series ring is created implicitly if a polynomial element is passed:

```
sage: R.<x> = QQ['x']
sage: O(x^100)
O(x^100)
```

```
>>> from sage.all import *
>>> R = QQ[['x']]; (x,) = R._first_ngens(1)
>>> 1/(1+x+O(x^5))
1 - x + x^2 - x^3 + x^4 + O(x^5)
```

```
sage: R.<u,v> = QQ[[[]]

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> O(x^100)
O(x^100)
```

```
>>> Integer(1)/(Integer(1)+x+O(x**Integer(5)))
```

(continues on next page)
1 - x + x^2 - x^3 + x^4 + O(x^5)

```python
>>> R = QQ['[u, v]']; (u, v,) = R._first_ngens(2)
>>> Integer(1) + u + v**Integer(2) + O(u, v)**Integer(5)
1 + u + v^2 + O(u, v)^5
```

This is also useful to create \( p \)-adic numbers:

```python
sage: O(7^6)  # needs sage.rings.padics
O(7^6)
sage: 1/3 + O(7^6)  # needs sage.rings.padics
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + 4*7^5 + O(7^6)
```

It behaves well with respect to adding negative powers of \( p \):

```python
sage: a = O(11^-32); a  # needs sage.rings.padics
O(11^-32)
sage: a.parent()  # needs sage.rings.padics
11-adic Field with capped relative precision 20
```

There are problems if you add a rational with very negative valuation to an \( O \)-Term:

```python
sage: 11^-12 + O(11^15)  # needs sage.rings.padics
11^-12 + O(11^8)
```

The reason that this fails is that the constructor doesn’t know the right precision cap to use. If you cast explicitly or use other means of element creation, you can get around this issue:

```python
sage: # needs sage.rings.padics
sage: K = Qp(11, 30)
```
sage: \( K(11^{-12}) + O(11^{15}) \)  
\( 11^{-12} + O(11^{15}) \)
sage: \( 11^{-12} + K(11^{15}) \)  
\( 11^{-12} + O(11^{15}) \)
sage: \( K(11^{-12}, \text{absprec}=15) \)  
\( 11^{-12} + O(11^{15}) \)
sage: \( K(11^{-12}, 15) \)  
\( 11^{-12} + O(11^{15}) \)

```python
>>> from sage.all import *

>>> K = Qp(Integer(11), Integer(30))
>>> K(Integer(11)**-Integer(12)) + O(Integer(11)**Integer(15))
\( 11^{-12} + O(11^{15}) \)
>>> Integer(11)**-Integer(12) + K(O(Integer(11)**Integer(15)))
\( 11^{-12} + O(11^{15}) \)
>>> K(Integer(11)**-Integer(12), absprec=15)
\( 11^{-12} + O(11^{15}) \)
>>> K(Integer(11)**-Integer(12), Integer(15))
\( 11^{-12} + O(11^{15}) \)
```

We can also work with asymptotic expansions:

```python
sage: A.<n> = AsymptoticRing(growth_group='QQ^n * n^QQ * log(n)^QQ',  
   \# needs sage.symbolic
   ....:  \text{coefficient} \_\text{ring}=QQ); A
Asymptotic Ring <QQ^n * n^QQ * log(n)^QQ * Signs^n> over Rational Field
sage: O(n)  
\# needs sage.symbolic
O(n)
```

```python
>>> from sage.all import *

>>> A = AsymptoticRing(growth_group='QQ^n * n^QQ * log(n)^QQ',  
   \# needs sage.symbolic
   ....:  \text{coefficient} \_\text{ring}=QQ, \text{name}=('n',)); (n,) = A._first__  
   \text{ngens}(1); A
Asymptotic Ring <QQ^n * n^QQ * log(n)^QQ * Signs^n> over Rational Field
>>> O(n)  
\# needs sage.symbolic
O(n)
```

Application with Puiseux series:

```python
sage: P.<y> = PuiseuxSeriesRing(ZZ)
sage: y**(Integer(1)/Integer(5)) + O(y**(Integer(1)/Integer(3)))
\( y^{1/5} + O(y^{1/3}) \)
sage: y**(Integer(1)/Integer(3)) + O(y**(Integer(1)/Integer(5)))
O(y^{1/5})
```

```python
>>> from sage.all import *

>>> P = PuiseuxSeriesRing(ZZ, \text{names}=('y',)); (y,) = P._first__  
\text{ngens}(1)
>>> y**(Integer(1)/Integer(5)) + O(y**(Integer(1)/Integer(3)))
\( y^{1/5} + O(y^{1/3}) \)
>>> y**(Integer(1)/Integer(3)) + O(y**(Integer(1)/Integer(5)))
O(y^{1/5})
```
9.2 Signed and Unsigned Infinities

The unsigned infinity “ring” is the set of two elements

1. infinity
2. A number less than infinity

The rules for arithmetic are that the unsigned infinity ring does not canonically coerce to any other ring, and all other rings canonically coerce to the unsigned infinity ring, sending all elements to the single element “a number less than infinity” of the unsigned infinity ring. Arithmetic and comparisons then take place in the unsigned infinity ring, where all arithmetic operations that are well-defined are defined.

The infinity “ring” is the set of five elements

1. plus infinity
2. a positive finite element
3. zero
4. a negative finite element
5. negative infinity

The infinity ring coerces to the unsigned infinity ring, sending the infinite elements to infinity and the non-infinite elements to “a number less than infinity.” Any ordered ring coerces to the infinity ring in the obvious way.

Note: The shorthand oo is predefined in Sage to be the same as +Infinity in the infinity ring. It is considered equal to, but not the same as Infinity in the UnsignedInfinityRing.

EXAMPLES:

We fetch the unsigned infinity ring and create some elements:

```
sage: P = UnsignedInfinityRing; P
The Unsigned Infinity Ring
sage: P(5)
A number less than infinity
sage: P.ngens()
1
sage: unsigned_oo = P.0; unsigned_oo
Infinity
```

```
>>> from sage.all import *
>>> P = UnsignedInfinityRing; P
The Unsigned Infinity Ring
>>> P(Integer(5))
A number less than infinity
>>> P.ngens()
1
>>> unsigned_oo = P.gen(0); unsigned_oo
Infinity
```

We compare finite numbers with infinity:

```
sage: 5 < unsigned_oo
True
sage: 5 > unsigned_oo
```

(continues on next page)
Demonstrating the shorthand `oo` versus `Infinity`:

```python
sage: oo +Infinity
sage: oo is InfinityRing.0
True
sage: oo is UnsignedInfinityRing.0
False
sage: oo == UnsignedInfinityRing.0
True
```

We do arithmetic:

```python
sage: unsigned_oo + 5
Infinity
```

We make `1 / unsigned_oo` return the integer 0 so that arithmetic of the following type works:

```python
sage: (1/unsigned_oo) + 2
2
sage: 32/5 - (2.439/unsigned_oo)
32/5
```
Note that many operations are not defined, since the result is not well-defined:

```
sage: unsigned_oo/0
Traceback (most recent call last):
  ... ValueError: quotient of number < oo by number < oo not defined
```

What happened above is that 0 is canonically coerced to “A number less than infinity” in the unsigned infinity ring. Next, Sage tries to divide by multiplying with its inverse. Finally, this inverse is not well-defined.

```
sage: 0/unsigned_oo
0
sage: unsigned_oo * 0
Traceback (most recent call last):
  ... ValueError: unsigned oo times smaller number not defined
sage: unsigned_oo/unsigned_oo
Traceback (most recent call last):
  ... ValueError: unsigned oo times smaller number not defined
```

In the infinity ring, we can negate infinity, multiply positive numbers by infinity, etc.

```
sage: P = InfinityRing; P
The Infinity Ring
sage: P(5)
A positive finite number
```

```
The symbol $\infty$ is predefined as a shorthand for $+\text{Infinity}$:

```python
sage: oo
+\text{Infinity}
```

```python
>>> \text{from sage.all import *}
>>> oo
+\text{Infinity}
```

We compare finite and infinite elements:

```python
sage: 5 < oo
True
sage: \text{P(-5)} < \text{P(5)}
True
sage: \text{P(2)} < \text{P(3)}
False
sage: -oo < oo
True
```

```python
>>> \text{from sage.all import *}
>>> \text{Integer(5)} < oo
True
>>> \text{P(-Integer(5))} < \text{P(Integer(5))}
True
>>> \text{P(Integer(2))} < \text{P(Integer(3))}
False
>>> -oo < oo
True
```

We can do more arithmetic than in the unsigned infinity ring:

```python
sage: 2 * oo
+\text{Infinity}
sage: -2 * oo
-\text{Infinity}
sage: 1 - oo
-\text{Infinity}
sage: 1 / oo
0
sage: -1 / oo
0
```

```python
>>> \text{from sage.all import *}
>>> \text{Integer(2)} * oo
+\text{Infinity}
>>> -\text{Integer(2)} * oo
-\text{Infinity}
>>> \text{Integer(1)} - oo
-\text{Infinity}
>>> \text{Integer(1)} / oo
0
>>> -\text{Integer(1)} / oo
0
```

We make $1 / \infty$ and $1 / -\infty$ return the integer 0 instead of the infinity ring Zero so that arithmetic of the following type works:

9.2. Signed and Unsigned Infinities 247
If we try to subtract infinities or multiply infinity by zero we still get an error:

```sage
sage: oo - oo
Traceback (most recent call last):
... SignError: cannot add infinity to minus infinity
sage: 0 * oo
Traceback (most recent call last):
... SignError: cannot multiply infinity by zero
sage: P(2) + P(-3)
Traceback (most recent call last):
... SignError: cannot add positive finite value to negative finite value

Signed infinity can also be represented by RR / RDF elements. But unsigned infinity cannot:

```sage
sage: set in RR, oo in RDF
(True, True)
sage: unsigned_infinity in RR, unsigned_infinity in RDF
(False, False)
```

**class** `sage.rings.infinity.AnInfinity`  

Bases: `object`
\texttt{lcm}(x)

Return the least common multiple of $\infty$ and $x$, which is by definition $\infty$ unless $x$ is 0.

**EXAMPLES:**

```python
sage: oo.lcm(0)
0
sage: oo.lcm(oo)
+Infinity
sage: oo.lcm(-oo)
+Infinity
sage: oo.lcm(10)
+Infinity
sage: (-oo).lcm(10)
+Infinity
```

```python
>>> from sage.all import *
>>> oo.lcm(Integer(0))
0
>>> oo.lcm(oo)
+Infinity
>>> oo.lcm(-oo)
+Infinity
>>> oo.lcm(Integer(10))
+Infinity
>>> (-oo).lcm(Integer(10))
+Infinity
```

**class sage.rings.infinity.FiniteNumber**(parent, x)

**Bases:** RingElement

Initialize self.

**sign**()

Return the sign of self.

**EXAMPLES:**

```python
sage: sign(InfinityRing(2))
1
sage: sign(InfinityRing(0))
0
sage: sign(InfinityRing(-2))
-1
```

```python
>>> from sage.all import *
>>> sign(InfinityRing(Integer(2)))
1
>>> sign(InfinityRing(Integer(0)))
0
>>> sign(InfinityRing(-Integer(2)))
-1
```

**sqrt**()

**EXAMPLES:**

9.2. Signed and Unsigned Infinities 249
```python
sage: InfinityRing(7).sqrt()
A positive finite number
sage: InfinityRing(0).sqrt()
Zero
sage: InfinityRing(-.001).sqrt()
Traceback (most recent call last):
  ...
SignError: cannot take square root of a negative number
```

```python
>>> from sage.all import *
>>>
>>> InfinityRing(Integer(7)).sqrt()
A positive finite number
>>>
>>> InfinityRing(Integer(0)).sqrt()
Zero
>>>
>>> InfinityRing(-RealNumber('.001')).sqrt()
Traceback (most recent call last):
  ...
SignError: cannot take square root of a negative number
```

class sage.rings.infinity.InfinityRing_class

Based on [Singleton, CommutativeRing]

Initialize self.

fraction_field()

This isn't really a ring, let alone an integral domain.

gen(n=0)

The two generators are plus and minus infinity.

EXAMPLES:

```python
sage: InfinityRing.gen(0)
+Infinity
sage: InfinityRing.gen(1)
-Infinity
sage: InfinityRing.gen(2)
Traceback (most recent call last):
  ...
IndexError: n must be 0 or 1
```

```python
>>> from sage.all import *
>>>
>>> InfinityRing.gen(Integer(0))
+Infinity
>>>
>>> InfinityRing.gen(Integer(1))
-Infinity
>>>
>>> InfinityRing.gen(Integer(2))
Traceback (most recent call last):
  ...
IndexError: n must be 0 or 1
```

gens()

The two generators are plus and minus infinity.

EXAMPLES:
\texttt{sage}: \texttt{InfinityRing.gens()}
\begin{verbatim}
(+Infinity, -Infinity)
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.gens()
(+Infinity, -Infinity)
\end{verbatim}

\textbf{is\_commutative()}

The Infinity Ring is commutative

\textbf{EXAMPLES:}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.is_commutative()
True
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.is_commutative()
True
\end{verbatim}

\textbf{is\_zero()}

The Infinity Ring is not zero

\textbf{EXAMPLES:}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.is_zero()
False
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.is_zero()
False
\end{verbatim}

\textbf{ngens()}

The two generators are plus and minus infinity.

\textbf{EXAMPLES:}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.ngens()
2
>>> len(InfinityRing.gens())
2
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> InfinityRing.ngens()
2
>>> len(InfinityRing.gens())
2
\end{verbatim}

\textbf{class} \texttt{sage.rings.infinity.LessThanInfinity(*args)}

\textbf{Bases:} \texttt{_uniq.RingElement}

Initialize self.

\textbf{EXAMPLES:}

\begin{verbatim}
>>> sage: sage.rings.infinity.LessThanInfinity() \texttt{is} UnsignedInfinityRing(5)
True
\end{verbatim}
```python
>>> from sage.all import *
>>> sage.rings.infinity.LessThanInfinity() is UnsignedInfinityRing(Integer(5))
True

sign()

Raise an error because the sign of self is not well defined.

EXAMPLES:

```python
sage: sign(UnsignedInfinityRing(2))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
sage: sign(UnsignedInfinityRing(0))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
sage: sign(UnsignedInfinityRing(-2))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
```

```python
>>> from sage.all import *
>>> sign(UnsignedInfinityRing(Integer(2)))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
>>> sign(UnsignedInfinityRing(Integer(0)))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
>>> sign(UnsignedInfinityRing(-Integer(2)))
Traceback (most recent call last):
  ... 
NotImplementedError: sign of number < oo is not well defined
```

class sage.rings.infinity.MinusInfinity(*args)

Bases: _uniq, AnInfinity, InfinityElement

Initialize self.

sqrt()

EXAMPLES:

```python
sage: (-oo).sqrt()
Traceback (most recent call last):
  ... 
SignError: cannot take square root of negative infinity
```

```python
>>> from sage.all import *
>>> (-oo).sqrt()
Traceback (most recent call last):
  ... 
SignError: cannot take square root of negative infinity
```

class sage.rings.infinity.PlusInfinity(*args)

Bases: _uniq, AnInfinity, InfinityElement
```
Initialize self.

\texttt{sqrt}()

The square root of \texttt{self}.

The square root of infinity is infinity.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: oo.sqrt()
+Infinity

>>> from sage.all import *
>>> oo.sqrt()
+Infinity
\end{verbatim}

\textbf{exception} \texttt{sage.rings.infinity.SignError}

Bases: \texttt{ArithmeticError}

Sign error exception.

\textbf{class} \texttt{sage.rings.infinity.UnsignedInfinity}(*\texttt{args})

Bases: \_\texttt{uniq}, \texttt{AnInfinity}, \texttt{InfinityElement}

Initialize self.

\textbf{class} \texttt{sage.rings.infinity.UnsignedInfinityRing_class}

Bases: \texttt{Singleton}, \texttt{Parent}

Initialize self.

\texttt{gen}(n=0)

The “generator” of \texttt{self} is the infinity object.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: UnsignedInfinityRing.gen()
Infinity
sage: UnsignedInfinityRing.gen(1)
Traceback (most recent call last):
  ...
IndexError: UnsignedInfinityRing only has one generator

>>> from sage.all import *
>>> UnsignedInfinityRing.gen()
Infinity
>>> UnsignedInfinityRing.gen(Integer(1))
Traceback (most recent call last):
  ...
IndexError: UnsignedInfinityRing only has one generator
\end{verbatim}

\texttt{gens}()

The “generator” of \texttt{self} is the infinity object.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: UnsignedInfinityRing.gens()
(Infinity,)
\end{verbatim}
>>> from sage.all import *
>>> UnsignedInfinityRing.gens()
(Infinity,)

**less_than_infinity()**

This is the element that represents a finite value.

**EXAMPLES:**

```python
sage: UnsignedInfinityRing.less_than_infinity()
A number less than infinity
sage: UnsignedInfinityRing(5) is UnsignedInfinityRing.less_than_infinity()
True
```

```python
>>> UnsignedInfinityRing(Integer(5)) is UnsignedInfinityRing.less_than_infinity()
True
```

**ngens()**

The unsigned infinity ring has one “generator.”

**EXAMPLES:**

```python
sage: UnsignedInfinityRing.ngens()
1
sage: len(UnsignedInfinityRing.gens())
1
```

```python
>>> UnsignedInfinityRing.ngens()
1
>>> len(UnsignedInfinityRing.gens())
1
```

**sage.rings.infinity.is_Infinite(x)**

This is a type check for infinity elements.

**EXAMPLES:**

```python
sage: sage.rings.infinity.is_Infinite(oo)
True
sage: sage.rings.infinity.is_Infinite(-oo)
True
sage: sage.rings.infinity.is_Infinite(unsigned_infinity)
True
sage: sage.rings.infinity.is_Infinite(3)
False
sage: sage.rings.infinity.is_Infinite(RR(infinity))
False
sage: sage.rings.infinity.is_Infinite(ZZ)
False
```
>>> from sage.all import *
>>> sage.rings.infinity.is_Infinite(oo)
True
>>> sage.rings.infinity.is_Infinite(-oo)
True
>>> sage.rings.infinity.is_Infinite(unsigned_infinity)
True
>>> sage.rings.infinity.is_Infinite(Integer(3))
False
>>> sage.rings.infinity.is_Infinite(RR(infinity))
False
>>> sage.rings.infinity.is_Infinite(ZZ)
False

sage.rings.infinity.test_comparison(ring)

Check comparison with infinity

INPUT:

• ring – a sub-ring of the real numbers

OUTPUT:

Various attempts are made to generate elements of ring. An assertion is triggered if one of these elements does not compare correctly with plus/minus infinity.

EXAMPLES:

```python
sage: from sage.rings.infinity import test_comparison
sage: rings = [ZZ, QQ, RDF]
# needs sage.rings.real_mpfr
sage: rings += [RR, RealField(Integer(200))]
# needs sage.rings.real_mpfr
sage: rings += [RLF, RIF]  # needs sage.rings.real_interval_field
sage: for R in rings:
    ....:     print('testing {}\n'.format(R))
    ....:     test_comparison(R)
    testing Integer Ring
    testing Rational Field
    testing Real Double Field...
```

```python
sage: test_comparison(AA)  # needs sage.rings.number_field
```

```python
>>> from sage.all import *
>>> from sage.rings.infinity import test_comparison
>>> rings = [ZZ, QQ, RDF]
>>> rings += [RR, RealField(Integer(200))]  # needs sage.rings.real_mpfr
>>> rings += [RLF, RIF]  # needs sage.rings.real_interval_field
>>> for R in rings:
    ...
    print('testing {}\n'.format(R))
    ...
    test_comparison(R)
    testing Integer Ring
    testing Rational Field
    testing Real Double Field...
>>> test_comparison(AA)  # needs sage.rings.number_field
```
Comparison with number fields does not work:

```
sage: x = polygen(ZZ, 'x')
sage: K.<sqrt3> = NumberField(x^2 - 3)  # needs sage.rings.number_field
sage: (-oo < 1 + sqrt3) and (1 + sqrt3 < oo)  # known bug  # needs sage.rings.number_field
False
```

From Sage all import *

```
from sage.all import *
K = NumberField(x**Integer(2) - Integer(3), names=('sqrt3',)); (sqrt3,) = K._first_ngens(1)  # needs sage.rings.number_field
(-oo < Integer(1) + sqrt3) and (Integer(1) + sqrt3 < oo)  # known bug  # needs sage.rings.number_field
False
```

The symbolic ring handles its own infinities, but answers False (meaning: cannot decide) already for some very elementary comparisons:

```
sage: test_comparison(SR)  # known bug  # needs sage.symbolic
Traceback (most recent call last):
...
AssertionError: testing -1000.0 in Symbolic Ring: id = ...
```

```
from sage.all import *
test_comparison(SR)  # known bug  # needs sage.symbolic
Traceback (most recent call last):
...
AssertionError: testing -1000.0 in Symbolic Ring: id = ...
```

sage.rings.infinity.test_signed_infinity(pos_inf)

Test consistency of infinity representations.

There are different possible representations of infinity in Sage. These are all consistent with the infinity ring, that is, compare with infinity in the expected way. See also Issue #14045

**INPUT:**

- pos_inf – a representation of positive infinity.

**OUTPUT:**

An assertion error is raised if the representation is not consistent with the infinity ring.

Check that Issue #14045 is fixed:

```
sage: InfinityRing(float('+inf'))
+Infinity
sage: InfinityRing(float('-inf'))
-Infinity
sage: oo > float('+inf')
False
sage: oo == float('+inf')
True
```
EXAMPLES:

```python
>>> from sage.all import *
>>> InfinityRing(float('+inf'))
+Infinity
>>> InfinityRing(float('-inf'))
-Infinity
>>> oo > float('+inf')
False
>>> oo == float('+inf')
True
```

9.3 Support Python's numbers abstract base class

See also:

PEP 3141 for more information about numbers.


10.1 Derivations

Let $A$ be a ring and $B$ be a bimodule over $A$. A derivation $d : A \to B$ is an additive map that satisfies the Leibniz rule

$$d(xy) = xd(y) + d(x)y.$$ 

If $B$ is an algebra over $A$ and if we are given in addition a ring homomorphism $\theta : A \to B$, a twisted derivation with respect to $\theta$ (or a $\theta$-derivation) is an additive map $d : A \to B$ such that

$$d(xy) = \theta(x)d(y) + d(x)y.$$ 

When $\theta$ is the morphism defining the structure of $A$-algebra on $B$, a $\theta$-derivation is nothing but a derivation. In general, if $\iota : A \to B$ denotes the defining morphism above, one easily checks that $\theta - \iota$ is a $\theta$-derivation.

This file provides support for derivations and twisted derivations over commutative rings with values in algebras (i.e. we require that $B$ is a commutative $A$-algebra). In this case, the set of derivations (resp. $\theta$-derivations) is a module over $B$.

Given a ring $A$, the module of derivations over $A$ can be created as follows:

```sage
A.<x,y,z> = QQ[]
sage: M = A.derivation_module()
sage: M
Module of derivations over Multivariate Polynomial Ring in x, y, z over Rational Field
```

The method `gens()` returns the generators of this module:

```sage
sage: A.<x,y,z> = QQ[]
sage: M = A.derivation_module()
sage: M.gens()
(d/dx, d/dy, d/dz)
```

```sage
>>> from sage.all import *
>>> A = QQ['x, y, z']; (x, y, z,) = A._first_ngens(3)
>>> M = A.derivation_module()
>>> M
Module of derivations over Multivariate Polynomial Ring in x, y, z over Rational Field
```

```sage
sage: A.<x,y,z> = QQ[]
sage: M = A.derivation_module()
sage: M.gens()
(d/dx, d/dy, d/dz)
```

```sage
>>> from sage.all import *
>>> A = QQ['x, y, z']; (x, y, z,) = A._first_ngens(3)
>>> M = A.derivation_module()
>>> M.gens()
(d/dx, d/dy, d/dz)
```
We can combine them in order to create all derivations:

```plaintext
sage: d = 2*M.gen(0) + z*M.gen(1) + (x^2 + y^2)*M.gen(2)
sage: d
2*d/dx + z*d/dy + (x^2 + y^2)*d/dz
```

```plaintext
>>> from sage.all import *
```

```plaintext
>>> d = Integer(2)*M.gen(Integer(0)) + z*M.gen(Integer(1)) + (x**Integer(2) + y**Integer(2))*M.gen(Integer(2))
>>> d
2*d/dx + z*d/dy + (x^2 + y^2)*d/dz
```

and now play with them:

```plaintext
sage: d(x + y + z)
x^2 + y^2 + z + 2
sage: P = A.random_element()
sage: Q = A.random_element()
sage: d(P*Q) == P*d(Q) + d(P)*Q
True
```

```plaintext
>>> from sage.all import *
```

```plaintext
>>> d(x + y + z)
x^2 + y^2 + z + 2
```

```plaintext
>>> P = A.random_element()
>>> Q = A.random_element()
>>> d(P*Q) == P*d(Q) + d(P)*Q
True
```

Alternatively we can use the method `derivation()` of the ring `A` to create derivations:

```plaintext
sage: Dx = A.derivation(x); Dx
d/dx
sage: Dy = A.derivation(y); Dy
d/dy
sage: Dz = A.derivation(z); Dz
d/dz
sage: A.derivation([2, z, x^2+y^2])
2*d/dx + z*d/dy + (x^2 + y^2)*d/dz
```

```plaintext
>>> from sage.all import *
```

```plaintext
>>> Dx = A.derivation(x); Dx
```

```plaintext
>>> Dy = A.derivation(y); Dy
```

```plaintext
>>> Dz = A.derivation(z); Dz
```

```plaintext
>>> A.derivation([Integer(2), z, x**Integer(2)+y**Integer(2)])
2*d/dx + z*d/dy + (x^2 + y^2)*d/dz
```

Sage knows moreover that `M` is a Lie algebra:

```plaintext
sage: M.category()
Join of
  Category of Lie algebras with basis over Rational Field and
  Category of modules with basis over
  Multivariate Polynomial Ring in x, y, z over Rational Field
```
Computation of Lie brackets are implemented as well:

```python
sage: Dx.bracket(Dy)
0
sage: d.bracket(Dx)
-2*x*d/dz
```

At the creation of a module of derivations, a codomain can be specified:

```python
sage: B = A.fraction_field()
sage: A.derivation_module(B)
Module of derivations from Multivariate Polynomial Ring in x, y, z over Rational Field
    to Fraction Field of Multivariate Polynomial Ring in x, y, z over Rational Field
```

Alternatively, one can specify a morphism $f$ with domain $A$. In this case, the codomain of the derivations is the codomain of $f$ but the latter is viewed as an algebra over $A$ through the homomorphism $f$. This construction is useful, for example, if we want to work with derivations on $A$ at a certain point, e.g. $(0, 1, 2)$. Indeed, in order to achieve this, we first define the evaluation map at this point:

```python
sage: ev = A.hom([QQ(0), QQ(1), QQ(2)])
sage: ev
Ring morphism:
    From: Multivariate Polynomial Ring in x, y, z over Rational Field
    To:    Rational Field
    Defn: x |--> 0
           y |--> 1
           z |--> 2
```

```python
>>> from sage.all import *

>>> M.category()
Join of
    Category of Lie algebras with basis over Rational Field and
    Category of modules with basis over
        Multivariate Polynomial Ring in x, y, z over Rational Field
```

10.1. Derivations
Now we use this ring homomorphism to define a structure of $A$-algebra on $\mathbb{Q}$ and then build the following module of derivations:

```
sage: M = A.derivation_module(ev)
sage: M
Module of derivations
    from Multivariate Polynomial Ring in x, y, z over Rational Field
to Rational Field
sage: M.gens()
(d/dx, d/dy, d/dz)
```

Elements in $M$ then acts as derivations at $(0, 1, 2)$:

```
sage: Dx = M.gen(0)
sage: Dy = M.gen(1)
sage: Dz = M.gen(2)
sage: f = x^2 + y^2 + z^2
sage: Dx(f)  # = 2*x evaluated at (0,1,2)
0
sage: Dy(f)  # = 2*y evaluated at (0,1,2)
2
sage: Dz(f)  # = 2*z evaluated at (0,1,2)
4
```

Twisted derivations are handled similarly:

```
sage: theta = B.hom([B(y),B(z),B(x)])
sage: theta
Ring endomorphism of Fraction Field of
    Multivariate Polynomial Ring in x, y, z over Rational Field
    Defn: x |--> y
        y |--> z
        z |--> x
sage: M = B.derivation_module(twist=theta)
sage: M
```

(continues on next page)
Module of twisted derivations over Fraction Field of Multivariate Polynomial Ring
in x, y, z over Rational Field (twisting morphism: x |--> y, y |--> z, z |--> x)

```python
>>> from sage.all import *
>>> theta = B.hom([B(y),B(z),B(x)])
>>> theta
Ring endomorphism of Fraction Field of
Multivariate Polynomial Ring in x, y, z over Rational Field
  Defn: x |--> y
       y |--> z
       z |--> x

>>> M = B.derivation_module(twist=theta)
>>> M
Module of twisted derivations over Fraction Field of Multivariate Polynomial Ring
in x, y, z over Rational Field (twisting morphism: x |--> y, y |--> z, z |--> x)

Over a field, one proves that every \( \theta \)-derivation is a multiple of \( \theta - id \), so that:

```sage```
d = M.gen(); d
[x |--> y, y |--> z, z |--> x] - id
```

```python```
>>> from sage.all import *
>>> d = M.gen(); d
[x |--> y, y |--> z, z |--> x] - id
```

and then:

```sage```
d(x)
- x + y
d(y)
- y + z
d(z)
x - z
d(x + y + z)
```
```
sage: d(x)
sage: d(y)
sage: d(z)
sage: d(x + y + z)
```

```python```
>>> from sage.all import *
>>> d(x)
>>> d(y)
>>> d(z)
>>> d(x + y + z)
```
```
- x + y
- y + z
x - z
0
```

AUTHOR:
• Xavier Caruso (2018-09)

class sage.rings.derivation.RingDerivation
    Bases: ModuleElement

    An abstract class for twisted and untwisted derivations over commutative rings.
**codomain()**

Return the codomain of this derivation.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: f = R.derivation(); f
d/dx
sage: f.codomain()
Univariate Polynomial Ring in x over Rational Field
sage: f.codomain() is R
True

>>> from sage.all import *

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f = R.derivation(); f
d/dx
>>> f.codomain()
Univariate Polynomial Ring in x over Rational Field
>>> f.codomain() is R
True
```

```python
sage: S.<y> = R[]
sage: M = R.derivation_module(S)
sage: M.random_element().codomain()
Univariate Polynomial Ring in y over
Univariate Polynomial Ring in x over Rational Field
sage: M.random_element().codomain() is S
True
```

**domain()**

Return the domain of this derivation.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: f = R.derivation(y); f
d/dy
sage: f.domain()
Multivariate Polynomial Ring in x, y over Rational Field
sage: f.domain() is R
True

>>> from sage.all import *

>>> S = R['y']; (y,) = S._first_ngens(1)
>>> M = R.derivation_module(S)
>>> M.random_element().codomain()
Univariate Polynomial Ring in y over
Univariate Polynomial Ring in x over Rational Field
>>> M.random_element().codomain() is S
True
```
class sage.rings.derivation.RingDerivationModule(domain, codomain, twist=None)

Bases: Module, UniqueRepresentation

A class for modules of derivations over a commutative ring.

basis()

Return a basis of this module of derivations.

EXAMPLES:

sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)


codomain()

Return the codomain of the derivations in this module.

EXAMPLES:

sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module()
sage: M.codomain()
Multivariate Polynomial Ring in x, y over Integer Ring


defining_morphism()

Return the morphism defining the structure of algebra of the codomain over the domain.

EXAMPLES:

sage: R.<x> = QQ[]
sage: M = R.derivation_module()
sage: M.defining_morphism()
Identity endomorphism of Univariate Polynomial Ring in x over Rational Field
sage: S.<y> = R[]
sage: M = R.derivation_module(S)
sage: M.defining_morphism()
Polynomial base injection morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Univariate Polynomial Ring in y over
         Univariate Polynomial Ring in x over Rational Field

sage: ev = R.hom([QQ(0)])
sage: M = R.derivation_module(ev)
sage: M.defining_morphism()
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 0

\texttt{domain()} 

Return the domain of the derivations in this module.

\textbf{EXAMPLES:}

sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y over Integer Ring
sage: M.domain()
Multivariate Polynomial Ring in x, y over Integer Ring

\texttt{from sage.all import *}

\texttt{R = ZZ['x', 'y']; (x, y) = R._first_ngens(2)}
\texttt{M = R.derivation_module(); M}
Module of derivations over Multivariate Polynomial Ring in x, y over Integer Ring
\texttt{M.domain()}
**dual_basis()**

Return the dual basis of the canonical basis of this module of derivations (which is that returned by the method `basis()`).

**Note:** The dual basis of \((d_1, \ldots, d_n)\) is a family \((x_1, \ldots, x_n)\) of elements in the domain such that \(d_i(x_i) = 1\) and \(d_i(x_j) = 0\) if \(i \neq j\).

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)
sage: M.dual_basis()
Family (x, y)
```

**gen\((n=0)\)**

Return the \(n\)-th generator of this module of derivations.

**INPUT:**

- \(n\) – an integer (default: 0)

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y over Integer
˓→Ring
sage: M.gen()
d/dx
sage: M.gen(1)
d/dy
```

```
>>> from sage.all import *
>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y over Integer
˓→Ring
>>> M.gen()
d/dx
>>> M.gen(Integer(1))
d/dy
```
**gens()**

Return the generators of this module of derivations.

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y over Integer...
˓→Ring
sage: M.gens()
(d/dx, d/dy)
```

We check that, for a nontrivial twist over a field, the module of twisted derivation is a vector space of dimension 1 generated by twist - id:

```
sage: K = R.fraction_field()
sage: theta = K.hom([K(y),K(x)])
sage: M = K.derivation_module(twist=theta); M
Module of twisted derivations over Fraction Field of Multivariate Polynomial...
˓→Ring (twisting morphism: x |--> y, y |--> x)
sage: M.gens()
([x |--> y, y |--> x] - id,)
```

**ngens()**

Return the number of generators of this module of derivations.

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module(); M
Module of derivations over Multivariate Polynomial Ring in x, y over Integer...
˓→Ring
sage: M.ngens()
2
```

```
Indeed, generators are:

\[
\text{sage: } M.\text{gens()}
\]
\[
(d/dx, d/dy)
\]

We check that, for a nontrivial twist over a field, the module of twisted derivation is a vector space of dimension 1 generated by \(\text{twist} - \text{id}\):

\[
\text{sage: } K = R.\text{fraction_field()}
\]
\[
\text{sage: } \theta = K.\text{hom([K(y),K(x)])}
\]
\[
\text{sage: } M = K.\text{derivation_module(twist=\theta)}; M
\]
\[
\text{Module of twisted derivations over Fraction Field of Multivariate Polynomial Ring in x, y over Integer Ring (twisting morphism: x |--> y, y |--> x)}
\]
\[
\text{sage: } M.\text{ngens()}
\]
\[
1
\]
\[
\text{sage: } M.\text{gen()}
\]
\[
[x |--> y, y |--> x] - \text{id}
\]

\begin{verbatim}
random_element(*args, **kwds)

Return a random derivation in this module.

EXAMPLES:
\end{verbatim}

\[
\text{sage: } R.\langle x,y \rangle = \mathbb{Z}[]
\]
\[
\text{sage: } M = R.\text{derivation_module()}
\]
\[
\text{sage: } M.\text{random_element()}  \# \text{random}
\]
\[
(x^2 + x*y - 3*y^2 + x + 1)*d/dx + (-2*x^2 + 3*x*y + 10*y^2 + 2*x + 8)*d/dy
\]

\begin{verbatim}
ring_of_constants()

Return the subring of the domain consisting of elements \(x\) such that \(d(x) = 0\) for all derivation \(d\) in this module.
\end{verbatim}
EXEMPLARY:

```
sage: R.<x,y> = QQ[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)
sage: M.ring_of_constants()
Rational Field
```

```python
>>> from sage.all import *
>>> R = QQ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> M = R.derivation_module()
>>> M.basis()
Family (d/dx, d/dy)
>>> M.ring_of_constants()
Rational Field
```

**some_elements()**

Return a list of elements of this module.

EXEMPLARY:

```
sage: R.<x,y> = ZZ[]
sage: M = R.derivation_module()
sage: M.some_elements()
[d/dx, d/dy, x*d/dx, x*d/dy, y*d/dx, y*d/dy]
```

```python
>>> from sage.all import *
>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> M = R.derivation_module()
>>> M.some_elements()
[d/dx, d/dy, x*d/dx, x*d/dy, y*d/dx, y*d/dy]
```

**twisting_morphism()**

Return the twisting homomorphism of the derivations in this module.

EXEMPLARY:

```
sage: R.<x,y> = ZZ[]
sage: theta = R.hom([y,x])
sage: M = R.derivation_module(twist=theta); M
Module of twisted derivations over Multivariate Polynomial Ring in x, y
over Integer Ring (twisting morphism: x |--> y, y |--> x)
sage: M.twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
   Defn: x |--> y
   y |--> x
```

```python
>>> from sage.all import *
>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> theta = R.hom([y,x])
>>> M = R.derivation_module(twist=theta); M
Module of twisted derivations over Multivariate Polynomial Ring in x, y
over Integer Ring (twisting morphism: x |--> y, y |--> x)
>>> M.twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
```

(continues on next page)
When the derivations are untwisted, this method returns nothing:

```python
sage: M = R.derivation_module()
sage: M.twisting_morphism()
```

```python
>>> from sage.all import *
>>> M = R.derivation_module()
>>> M.twisting_morphism()
```

**class** `sage.rings.derivation.RingDerivationWithTwist_generic` *(parent, scalar=0)*

**Bases:** `RingDerivation`

The class handles $\theta$-derivations of the form $\lambda(\theta - \iota)$ (where $\iota$ is the defining morphism of the codomain over the domain) for a scalar $\lambda$ varying in the codomain.

**extend_to_fraction_field()**

Return the extension of this derivation to fraction fields of the domain and the codomain.

**EXAMPLES:**

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

sage: theta = R.hom([y,x])

sage: d = R.derivation(x, twist=theta)

sage: D = d.extend_to_fraction_field(); D
```

```python
x*(\{x |--> y, y |--> x\} - id)
```

```python
sage: D.domain()
```

```python
Fraction Field of Multivariate Polynomial Ring in x, y over Integer Ring
```

```python
sage: D(Integer(1)/x)
```

```python
(x - y)/y
```

```python
>>> from sage.all import *

>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)

>>> theta = R.hom([y,x])

>>> d = R.derivation(x, twist=theta)

>>> d
```

```python
x*(\{x |--> y, y |--> x\} - id)
```

```python
>>> D = d.extend_to_fraction_field(); D
```

```python
x*(\{x |--> y, y |--> x\} - id)
```

```python
>>> D.domain()
```

```python
Fraction Field of Multivariate Polynomial Ring in x, y over Integer Ring
```

```python
>>> D(Integer(1)/x)
```

```python
(x - y)/y
```
list()

Return the list of coefficient of this twisted derivation on the canonical basis.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: K = R.fraction_field()
sage: theta = K.hom([y,x])
sage: M = K.derivation_module(twist=theta)
sage: M.basis()
Family (twisting_morphism - id,)
sage: f = (x+y) * M.gen()
sage: f
(x + y)*(twisting_morphism - id)
sage: f.list()  
[x + y]
```

postcompose (morphism)

Return the twisted derivation obtained by applying first this twisted derivation and then morphism.

INPUT:

- morphism—a homomorphism of rings whose domain is the codomain of this derivation or a ring into which the codomain of this derivation

EXAMPLES:

```python
sage: R.<x,y> = ZZ[]
sage: theta = R.hom([y,x])
sage: D = R.derivation(x, twist=theta); D
x*([x |--> y, y |--> x] - id)
sage: f = R.hom([x^2, y^3])
sage: g = D.precompose(f); g
x*([x |--> y^2, y |--> x^3] - [x |--> x^2, y |--> y^3])
```

```python
>>> from sage.all import *
>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> theta = R.hom([y,x])
>>> D = R.derivation(x, twist=theta); D
x*([x |--> y, y |--> x] - id)
>>> f = R.hom([x**Integer(2), y**Integer(3)])
>>> g = D.precompose(f); g
x*([x |--> y^2, y |--> x^3] - [x |--> x^2, y |--> y^3])
```
Observe that the $g$ is no longer a $\theta$-derivation but a $(\theta \circ f)$-derivation:

```python
sage: g.parent().twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> y^2
  y |--> x^3
```

```python
>>> from sage.all import *
```

```python
>>> g.parent().twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> y^2
  y |--> x^3
```

precompose (morphism)

Return the twisted derivation obtained by applying first morphism and then this twisted derivation.

INPUT:

- morphism – a homomorphism of rings whose codomain is the domain of this derivation or a ring that coerces to the domain of this derivation

EXAMPLES:

```python
sage: R.<x,y> = ZZ[]
sage: theta = R.hom([y,x])
sage: D = R.derivation(x, twist=theta); D
x*(x |--> y, y |--> x) - id
sage: f = R.hom([x**Integer(2), y**Integer(3)])
sage: g = D.precompose(f); g
x^2*(x |--> y^3, y |--> x^2) - [x |--> x^2, y |--> y^3])
```

```python
>>> from sage.all import *
```

```python
>>> R = ZZ['x', 'y']; (x, y,) = R._first_ngens(2)
>>> theta = R.hom([y,x])
>>> D = R.derivation(x, twist=theta); D
x*(x |--> y, y |--> x) - id
>>> f = R.hom([x**Integer(2), y**Integer(3)])
>>> g = D.precompose(f); g
x^2*(x |--> y^3, y |--> x^2) - [x |--> x^2, y |--> y^3])
```

Observe that the $g$ is no longer a $\theta$-derivation but a $(f \circ \theta)$-derivation:

```python
sage: g.parent().twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> y^3
  y |--> x^2
```

```python
>>> from sage.all import *
```

```python
>>> g.parent().twisting_morphism()
Ring endomorphism of Multivariate Polynomial Ring in x, y over Integer Ring
  Defn: x |--> y^3
  y |--> x^2
```

class sage.rings.derivation.RingDerivationWithoutTwist

Bases: RingDerivation
An abstract class for untwisted derivations.

**extend_to_fraction_field()**

Return the extension of this derivation to fraction fields of the domain and the codomain.

**EXAMPLES:**

```
sage: S.<x> = QQ[]
sage: d = S.derivation()
d/dx
sage: D = d.extend_to_fraction_field()
sage: D
d/dx
sage: D.domain()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: D(1/x)
-1/x^2
```

```
>>> from sage.all import *
>>> S = QQ['x']; (x,) = S._first_ngens(1)
>>> d = S.derivation()
>>> d
d/dx
>>> D = d.extend_to_fraction_field()
>>> D
d/dx
>>> D.domain()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> D(Integer(1)/x)
-1/x^2
```

**is_zero()**

Return True if this derivation is zero.

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: f = R.derivation(); f
d/dx
sage: f.is_zero()
False
sage: (f-f).is_zero()
True
```

```
>>> from sage.all import *
>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> f = R.derivation(); f
d/dx
>>> f.is_zero()
False
```
list()  
Return the list of coefficient of this derivation on the canonical basis.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)
sage: R.derivation(x).list()
[1, 0]
sage: R.derivation(y).list()
[0, 1]
sage: f = x*R.derivation(x) + y*R.derivation(y); f
x*d/dx + y*d/dy
sage: f.list()
[x, y]
```

monomial_coefficients()  
Return dictionary of nonzero coordinates (on the canonical basis) of this derivation. More precisely, this returns a dictionary whose keys are indices of basis elements and whose values are the corresponding coefficients.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)
sage: R.derivation(x).monomial_coefficients()
{0: 1}
sage: R.derivation(y).monomial_coefficients()
{1: 1}
```
sage: f = x*R.derivation(x) + y*R.derivation(y); f
x*d/dx + y*d/dy
sage: f.monomial_coefficients()
{0: x, 1: y}

>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> M = R.derivation_module()
>>> M.basis()
Family (d/dx, d/dy)

R.derivation(x).monomial_coefficients()
{0: 1}

R.derivation(y).monomial_coefficients()
{1: 1}

>>> f = x*R.derivation(x) + y*R.derivation(y); f
x*d/dx + y*d/dy
>>> f.monomial_coefficients()
{0: x, 1: y}

postcompose (morphism)

Return the derivation obtained by applying first this derivation and then morphism.

INPUT:

- morphism—a homomorphism of rings whose domain is the codomain of this derivation or a ring into which the codomain of this derivation coerces

EXAMPLES:

sage: A.<x,y>= QQ[]
sage: ev = A.hom([QQ(0), QQ(1)])
sage: Dx = A.derivation(x)
sage: Dy = A.derivation(y)

>>> from sage.all import *
>>> A = QQ['x, y']; (x, y,) = A._first_ngens(2)
>>> ev = A.hom([QQ(Integer(0)), QQ(Integer(1))])
>>> Dx = A.derivation(x)
>>> Dy = A.derivation(y)

We can define the derivation at (0, 1) just by postcomposing with ev:

sage: dx = Dx.postcompose(ev)
sage: dy = Dy.postcompose(ev)
sage: f = x^2 + y^2
sage: dx(f)
0
sage: dy(f)
2

(continues on next page)
Note that we cannot avoid the creation of the evaluation morphism: if we pass in \( \mathbb{Q} \) instead, an error is raised since there is no coercion morphism from \( A \) to \( \mathbb{Q} \):

```python
sage: Dx.postcompose(QQ)
Traceback (most recent call last):
  ...TypeError: the codomain of the derivation does not coerce to the given ring
```

Note that this method cannot be used to compose derivations:

```python
sage: Dx.precompose(Dy)
Traceback (most recent call last):
  ...TypeError: you must give a homomorphism of rings
```

**precompose** *(morphism)*

Return the derivation obtained by applying first \( \text{morphism} \) and then this derivation.

**INPUT:**

- \( \text{morphism} \) — a homomorphism of rings whose codomain is the domain of this derivation or a ring that coerces to the domain of this derivation

**EXAMPLES:**

```python
sage: A.<x> = QQ[]
sage: B.<x,y> = QQ[]
sage: D = B.derivation(x) - 2*x*B.derivation(y); D
d/dx - 2*x*d/dy
```

When restricting to \( A \), the term \( d/dy \) disappears (since it vanishes on \( A \)): 

```python
>>> A = QQ['x']; (x,) = A._first_ngens(1)
>>> B = QQ['x, y']; (x, y,) = B._first_ngens(2)
>>> D = B.derivation(x) - Integer(2)*x*B.derivation(y); D
d/dx - 2*x*d/dy
```
If we restrict to another well chosen subring, the derivation vanishes:

```python
C.<t> = QQ[]
f = C.hom([x^2 + y]); f
Ring morphism:
  From: Univariate Polynomial Ring in t over Rational Field
  To:  Multivariate Polynomial Ring in x, y over Rational Field
  Defn: t |--> x^2 + y
D.precompose(f)
```

Note that this method cannot be used to compose derivations:

```python
D.precompose(D)
```

### pth_power()

Return the $p$-th power of this derivation where $p$ is the characteristic of the domain.

**Note:** Leibniz rule implies that this is again a derivation.

**EXAMPLES:**

```python
# needs sage.rings.finite_rings
R.<x,y> = GF(5)[]
Dx = R.derivation(x)
Dx.pth_power()
```
An error is raised if the domain has characteristic zero:

```python
sage: R.<x,y> = QQ[]
sage: Dx = R.derivation(x)
sage: Dx.pth_power()
Traceback (most recent call last):
  ...  
TypeError: the domain of the derivation must have positive and prime characteristic
```

or if the characteristic is not a prime number:

```python
sage: R.<x,y> = Integers(10)[]
sage: Dx = R.derivation(x)
sage: Dx.pth_power()
Traceback (most recent call last):
  ...  
TypeError: the domain of the derivation must have positive and prime characteristic
```
```python
>>> from sage.all import *
>>> R = Integers(Integer(10))['x', 'y']; (x, y,) = R._first_ngens(2)
>>> Dx = R.derivation(x)
>>> Dx.pth_power()
Traceback (most recent call last):
...
TypeError: the domain of the derivation must have positive and prime-
→characteristic
```

class sage.rings.derivation.RingDerivationWithoutTwist_fraction_field(parent, arg=None)

Bases: RingDerivationWithoutTwist_wrapper

This class handles derivations over fraction fields.

class sage.rings.derivation.RingDerivationWithoutTwist_function(parent, arg=None)

Bases: RingDerivationWithoutTwist

A class for untwisted derivations over rings whose elements are either polynomials, rational fractions, power series or Laurent series.

**is_zero()**

Return True if this derivation is zero.

**EXAMPLES:**

```
sage: R.<x,y> = ZZ[]
sage: f = R.derivation(); f
d/dx
sage: f.is_zero()
False
sage: (f-f).is_zero()
True
```

**list()**

Return the list of coefficient of this derivation on the canonical basis.

**EXAMPLES:**

```
sage: R.<x,y> = GF(5)[]
sage: M = R.derivation_module()
sage: M.basis()
Family (d/dx, d/dy)
sage: R.derivation(x).list()
[1, 0]
```

(continues on next page)
sage: R.derivation(y).list()
[0, 1]

sage: f = x*R.derivation(x) + y*R.derivation(y); f
x*d/dx + y*d/dy
sage: f.list()
[x, y]

>>> from sage.all import *
>>> R = GF(Integer(5))[['x', 'y']]; (x, y,) = R._first_ngens(2)
>>> M = R.derivation_module()
>>> M.basis()
Family (d/dx, d/dy)

>>> R.derivation(x).list()
[1, 0]
>>> R.derivation(y).list()
[0, 1]

>>> f = x*R.derivation(x) + y*R.derivation(y); f
x*d/dx + y*d/dy
>>> f.list()
[x, y]

class sage.rings.derivation.RingDerivationWithoutTwist_quotient (parent, arg=None)

Bases: RingDerivationWithoutTwist_wrapper

This class handles derivations over quotient rings.

class sage.rings.derivation.RingDerivationWithoutTwist_wrapper (parent, arg=None)

Bases: RingDerivationWithoutTwist

This class is a wrapper for derivation.

It is useful for changing the parent without changing the computation rules for derivations. It is used for derivations over fraction fields and quotient rings.

list()

Return the list of coefficient of this derivation on the canonical basis.

EXAMPLES:
>>> from sage.all import *
>>> # needs sage.libs.singular
>>> R = GF(Integer(5))['X, Y']; (X, Y,) = R._first_ngens(2)
>>> S = R.quotient([X**Integer(5), Y**Integer(5)], names=('x', 'y',)); (x, y,) = S._first_ngens(2)
>>> M = S.derivation_module()
>>> M.basis()
Family (d/dx, d/dy)
>>> S.derivation(x).list()
[1, 0]
>>> S.derivation(y).list()
[0, 1]
>>> f = x*S.derivation(x) + y*S.derivation(y); f
x*d/dx + y*d/dy
>>> f.list()
[x, y]

class sage.rings.derivation.RingDerivationWithoutTwist_zero (parent, arg=None)

Bases: RingDerivationWithoutTwist

This class can only represent the zero derivation.

It is used when the parent is the zero derivation module (e.g., when its domain is \(\mathbb{Z}/p\mathbb{Z}\), \(\mathbb{Q}\), a finite field, etc.)

is_zero()

Return True if this derivation vanishes.

EXAMPLES:

sage: M = QQ.derivation_module()
sage: M().is_zero()
True

list()

Return the list of coefficient of this derivation on the canonical basis.

EXAMPLES:

sage: M = QQ.derivation_module()
sage: M().list()
[]

>>> from sage.all import *
>>> M = QQ.derivation_module()
>>> M().list()
[]
• Index
• Module Index
• Search Page
sage.rings.abc, 21
sage.rings.big_oh, 241
sage.rings.derivation, 259
sage.rings.fraction_field, 137
sage.rings.fraction_field_element, 147
sage.rings.generic, 235
sage.rings.homset, 102
sage.rings.ideal, 37
sage.rings.ideal_monoid, 63
sage.rings.infinity, 244
sage.rings.localization, 157
sage.rings.morphism, 69
sage.rings.nc_ideals, 64
sage.rings.numbers_abc, 257
sage.rings.quotient_ring, 107
sage.rings.quotient_ring_element, 130
sage.rings.ring, 1
sage.rings.ring_extension, 173
sage.rings.ring_extension_element, 209
sage.rings.ring_extension_morphism, 228
<table>
<thead>
<tr>
<th>Method</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute_base()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>186</td>
</tr>
<tr>
<td>absolute_degree()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>186</td>
</tr>
<tr>
<td>absolute_norm()</td>
<td>sage.rings.ideal.Ideal_generic method</td>
<td>42</td>
</tr>
<tr>
<td>additive_order()</td>
<td>sage.rings.ring_extension_element.RingExtensionElement method</td>
<td>209</td>
</tr>
<tr>
<td>Algebra</td>
<td>class in sage.rings.ring</td>
<td>2</td>
</tr>
<tr>
<td>algebraic_closure()</td>
<td>sage.rings.ring.Field method</td>
<td>4</td>
</tr>
<tr>
<td>AlgebraicField</td>
<td>class in sage.rings.abc</td>
<td>21</td>
</tr>
<tr>
<td>AlgebraicField_common</td>
<td>class in sage.rings.abc</td>
<td>22</td>
</tr>
<tr>
<td>AlgebraicRealField</td>
<td>class in sage.rings.abc</td>
<td>23</td>
</tr>
<tr>
<td>ambient()</td>
<td>sage.rings.quotient_ring.QuotientRing_nc method</td>
<td>116</td>
</tr>
<tr>
<td>AnInfinity</td>
<td>class in sage.rings.infinity</td>
<td>248</td>
</tr>
<tr>
<td>apply_morphism()</td>
<td>sage.rings.ideal.Ideal_generic method</td>
<td>42</td>
</tr>
<tr>
<td>associated_primes()</td>
<td>sage.rings.ideal.Ideal_generic method</td>
<td>43</td>
</tr>
<tr>
<td>backend()</td>
<td>sage.rings.ring_extension_element.RingExtensionElement method</td>
<td>209</td>
</tr>
<tr>
<td>backend()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>187</td>
</tr>
<tr>
<td>base()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>187</td>
</tr>
<tr>
<td>base_extend()</td>
<td>sage.rings.ring.Ring method</td>
<td>9</td>
</tr>
<tr>
<td>base_map()</td>
<td>sage.rings.morphism.RingHomomorphism_im_gens method</td>
<td>99</td>
</tr>
<tr>
<td>base_map()</td>
<td>sage.rings.ring_extension_morphism.RingExtensionHomomorphism method</td>
<td>230</td>
</tr>
<tr>
<td>base_ring()</td>
<td>sage.rings.fraction_field.FractionField_generic method</td>
<td>142</td>
</tr>
<tr>
<td>base_ring()</td>
<td>sage.rings.ideal.Ideal_generic method</td>
<td>43</td>
</tr>
<tr>
<td>bases()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>188</td>
</tr>
<tr>
<td>basis()</td>
<td>sage.rings.derivation.RingDerivationModule method</td>
<td>265</td>
</tr>
<tr>
<td>basis_over()</td>
<td>sage.rings.ring_extension.RingExtensionWithBasis method</td>
<td>177</td>
</tr>
<tr>
<td>Characteristic</td>
<td>class in sage.rings.abc</td>
<td>22</td>
</tr>
<tr>
<td>CallableSymbolicExpressionRing</td>
<td>class in sage.rings.abc</td>
<td>23</td>
</tr>
<tr>
<td>category()</td>
<td>sage.rings.ideal.Ideal_generic method</td>
<td>45</td>
</tr>
<tr>
<td>category()</td>
<td>sage.rings.ring.Ring method</td>
<td>10</td>
</tr>
<tr>
<td>characteristic()</td>
<td>sage.rings.fraction_field.FractionField_generic method</td>
<td>142</td>
</tr>
<tr>
<td>characteristic()</td>
<td>sage.rings.localization.Localization method</td>
<td>165</td>
</tr>
<tr>
<td>characteristic()</td>
<td>sage.rings.quotient_ring.QuotientRing_nc method</td>
<td>116</td>
</tr>
<tr>
<td>characteristic()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>189</td>
</tr>
<tr>
<td>charpoly()</td>
<td>sage.rings.ring_extension_element.RingExtensionWithBasisElement method</td>
<td>216</td>
</tr>
<tr>
<td>class_number()</td>
<td>sage.rings.fraction_field.FractionField_Ipoly_field method</td>
<td>140</td>
</tr>
<tr>
<td>codomain()</td>
<td>sage.rings.derivation.RingDerivationModule method</td>
<td>263</td>
</tr>
<tr>
<td>codomain()</td>
<td>sage.rings.derivation.RingDerivationModule method</td>
<td>265</td>
</tr>
<tr>
<td>common_base()</td>
<td>in module sage.rings.ring_extension</td>
<td>206</td>
</tr>
<tr>
<td>CommutativeAlgebra</td>
<td>class in sage.rings.ring</td>
<td>2</td>
</tr>
<tr>
<td>CommutativeRing</td>
<td>class in sage.rings.abc</td>
<td>24</td>
</tr>
<tr>
<td>ComplexBallField</td>
<td>class in sage.rings.abc</td>
<td>25</td>
</tr>
<tr>
<td>ComplexDoubleField</td>
<td>class in sage.rings.abc</td>
<td>25</td>
</tr>
<tr>
<td>ComplexField</td>
<td>class in sage.rings.abc</td>
<td>25</td>
</tr>
<tr>
<td>ComplexIntervalField</td>
<td>class in sage.rings.abc</td>
<td>26</td>
</tr>
<tr>
<td>construction()</td>
<td>sage.rings.fraction_field.FractionField_generic method</td>
<td>142</td>
</tr>
<tr>
<td>construction()</td>
<td>sage.rings.quotient_ring.QuotientRing_nc method</td>
<td>117</td>
</tr>
<tr>
<td>construction()</td>
<td>sage.rings.ring_extension.RingExtension_generic method</td>
<td>190</td>
</tr>
</tbody>
</table>
function_field() (sage.rings.fraction_field.FractionField_1poly_field method), 141

G

gcd() (sage.rings.ideal.Ideal_pid method), 55
gen() (sage.rings.derivation.RingDerivationModule method), 267
gen() (sage.rings.fraction_field.FractionField_generic method), 143
gen() (sage.rings.ideal.Idéal_generic method), 46
gen() (sage.rings.ideal.Idéal_principal method), 59
gen() (sage.rings.infinity.InfinityRing_class method), 250
gen() (sage.rings.infinity.UnsignedInfinityRing_class method), 253
gen() (sage.rings.localization.Localization method), 166
gen() (sage.rings.quotient_ring.QuotientRing_nc method), 120
gen() (sage.rings.ring_extension.RingExtension_generic method), 196
generators() (in module sage.rings.ring_extension), 207
gens() (sage.rings.derivation.RingDerivationModule method), 267
gens() (sage.rings.ideal.Idéal_generic method), 46
gens() (sage.rings.infinity.InfinityRing_class method), 250
gens() (sage.rings.infinity.UnsignedInfinityRing_class method), 253
gen() (sage.rings.localization.Localization method), 166
gen() (sage.rings.ring_extension.RingExtension_generic method), 197
gen() (sage.rings.ring_extension.RingExtensionWithGen method), 184
gens_reduced() (sage.rings.ideal.Idéal_generic method), 47
graded_free_resolution() (sage.rings.ideal.Idéal_generic method), 47

H

has_coerce_map_from() (sage.rings.homset.RingHomset_generic method), 102
hom() (sage.rings.ring_extension.RingExtension_generic method), 197

I

Ideal() (in module sage.rings.ideal), 39
ideal() (sage.rings.quotient_ring.QuotientRing_nc method), 121
ideal() (sage.rings.ring.Field method), 6
ideal() (sage.rings.ring.Ring method), 12
Ideal_fractional (class in sage.rings.ideal), 41
Ideal_generic (class in sage.rings.ideal), 42
ideal_monoid() (sage.rings.ring.CommutativeRing method), 3
ideal_monoid() (sage.rings.ring.Ring method), 13
Ideal_nc (class in sage.rings.commutative_ideals), 65
Ideal_pid (class in sage.rings.ideal), 55
Ideal_principal (class in sage.rings.ideal), 59
IdealMonoid() (in module sage.rings.ideal_monoid), 63
IdealMonoid_c (class in sage.rings.ideal_monoid), 63
IdealMonoid_nc (class in sage.rings.noncommutative_ideals), 65
im gens() (sage.rings.morphism.RingHomomorphism_im_gens method), 100
in_base() (sage.rings.ring_extension_element.RingExtensionElement method), 210
InfinityRing_class (class in sage.rings.infinity), 250
IntegerModRing (class in sage.rings.abe), 27
integral_closure() (sage.rings.ring.Field method), 6
IntegralDomain (class in sage.rings.ring), 8
interpolation() (sage.rings.generic.ProductTree method), 237
inverse() (sage.rings.morphism.RingHomomorphism method), 81
inverse() (sage.rings.morphism.RingHomomorphism_from_base method), 95
inverse() (sage.rings.morphism.RingHomomorphism_from_fraction_field method), 97
inverse_image() (sage.rings.morphism.RingHomomorphism method), 86
inverse_of_unit() (sage.rings.localization.LocalizationElement method), 169
is_commutative() (sage.rings.infinity.InfinityRing_class method), 251
is_commutative() (sage.rings.quotient_ring.QuotientRing_nc method), 121
is_commutative() (sage.rings.ring.CommutativeRing method), 3
is_defined_over() (sage.rings.ring_extension.RingExtension_generic method), 199
is_exact() (sage.rings.fraction_field.FractionField_extension_generic method), 143
is_exact() (sage.rings.ring.Ring method), 13
is_field() (sage.rings.fraction_field.FractionField_extension_generic method), 144
is_field() (sage.rings.localization.Localization method), 167
is_field() (sage.rings.quotient_ring.QuotientRing_nc method), 122
is_field() (sage.rings.ring_extension.RingExtension_generic method), 200
is_field() (sage.rings.ring.Field method), 6
is_field() (sage.rings.ring.IntegralDomain method), 8
is_field() (sage.rings.ring.Ring method), 14
is_finite()  (sage.rings.fraction_field.FractionField_generic method), 144
is_finite_over()  (sage.rings.ring_extension.RingExtension_generic method), 201
is_FractionField()  (in module sage.rings.fraction_field), 146
is_FractionFieldElement()  (in module sage.rings.fraction_field_element), 153
is_free_over()  (sage.rings.ring_extension.RingExtension_generic method), 202
is_Ideal()  (in module sage.rings.ideal), 62
is_identity()  (sage.rings.ring_extension_morphism.RingExtensionHomomorphism method), 231
is_Infinite()  (in module sage.rings.infinity), 254
is_injective()  (sage.rings.fraction_field.FractionFieldEmbedding method), 139
is_injective()  (sage.rings.ring_extension_morphism.MapFreeModuleToRelativeRing method), 228
is_injective()  (sage.rings.ring_extension_morphism.MapRelativeRingToFreeModule method), 229
is_injective()  (sage.rings.ring_extension_morphism.RingExtensionHomomorphism method), 232
is_integral()  (sage.rings.fraction_field_element.FractionFieldElement_poly_field method), 152
is_integral_domain()  (sage.rings.quotient_ring.QuotientRing_nc method), 122
is_integrally_closed()  (sage.rings.ring.Field method), 7
is_integrally_closed()  (sage.rings.ring.IntegralDomain method), 8
is_invertible()  (sage.rings.morphism.RingHomomorphism method), 88
is_maximal()  (sage.rings.ideal.Ideal_generic method), 47
is_maximal()  (sage.rings.ideal.Ideal_pid method), 56
is_nilpotent()  (sage.rings.ring_extension_element.RingExtensionElement method), 211
is_noetherian()  (sage.rings.quotient_ring.QuotientRing_nc method), 123
is_noetherian()  (sage.rings.ring.Field method), 7
is_one()  (sage.rings.fraction_field_element.FractionFieldElement method), 147
is_primary()  (sage.rings.ideal.Ideal_generic method), 48
is_prime()  (sage.rings.ideal.Ideal_generic method), 49
is_prime()  (sage.rings.ideal.Ideal_pid method), 57
is_prime()  (sage.rings.ring_extension_element.RingExtensionElement method), 211
is_prime_field()  (sage.rings.ring.Ring method), 15
is_principal()  (sage.rings.ideal.Ideal_generic method), 51
is_principal()  (sage.rings.ideal.Ideal_principal method), 60
is_QuotientRing()  (in module sage.rings.quotient_ring), 129
is_Ring()  (in module sage.rings.ring), 21
is_RingHomset()  (in module sage.rings.homset), 104
is_square()  (sage.rings.fraction_field_element.FractionFieldElement method), 148
is_square()  (sage.rings.ring_extension_element.RingExtensionElement method), 212
is_subring()  (sage.rings.ring.Ring method), 16
is_surjective()  (sage.rings.fraction_field.FractionFieldEmbedding method), 139
is_surjective()  (sage.rings.morphism.RingHomomorphism method), 88
is_surjective()  (sage.rings.ring_extension_morphism.MapRelativeRingToFreeModule method), 228
is_surjective()  (sage.rings.ring_extension_morphism.RingExtensionHomomorphism method), 232
is_trivial()  (sage.rings.ideal.Ideal_generic method), 51
is_unit()  (sage.rings.localization.LocalizationElement method), 169
is_unit()  (sage.rings.quotient_ring_element.QuotientRingElement method), 131
is_unit()  (sage.rings.ring_extension_element.RingExtensionElement method), 212
is_zero()  (sage.rings.derivation.RingDerivationWithoutTwist method), 274
is_zero()  (sage.rings.derivation.RingDerivationWithoutTwist_function method), 280
is_zero()  (sage.rings.derivation.RingDerivationWithoutTwist_zero method), 282
is_zero()  (sage.rings.fraction_field_element.FractionFieldElement method), 149
is_zero()  (sage.rings.infinity.InfinityRing_class method), 251

K
Katsura()  (in module sage.rings.ideal), 61
kernel()  (sage.rings.morphism.RingHomomorphism method), 89
kernel()  (sage.rings.morphism.RingHomomorphism_cover method), 92
krull_dimension()  (sage.rings.localization.Localization method), 167
krull_dimension() (sage.rings.commutative_ring method), 4
krull_dimension() (sage.rings.field method), 7

L
lc() (sage.rings.quotient_ring_element.QuotientRingElement method), 132
lcm() (sage.rings.infinity.AnInfinity method), 248
leaves() (sage.rings.generic.ProductTree method), 238
less_than_infinity() (sage.rings.infinity.UnsignedInfinityRing_class method), 254
LessThanInfinity (class in sage.rings.infinity), 251
lift() (sage.rings.morphism.RingHomomorphism method), 90
lift() (sage.rings.quotient_ring_element.QuotientRingElement method), 133
lift() (sage.rings.quotient_ring.QuotientRing_nc method), 124
lifting_map() (sage.rings.quotient_ring.QuotientRing_nc method), 125
list() (sage.rings.derivation.RingDerivationWithoutTwist method), 275
list() (sage.rings.derivation.RingDerivationWithoutTwist_function method), 280
list() (sage.rings.derivation.RingDerivationWithoutTwist_wrapper method), 281
list() (sage.rings.derivation.RingDerivationWithoutTwist_zero method), 282
list() (sage.rings.derivation.RingDerivationWithTwist_generic method), 271
lm() (sage.rings.quotient_ring_element.QuotientRingElement method), 133
Localization (class in sage.rings.localization), 162
localization() (sage.rings.commutative_ring method), 4
LocalizationElement (class in sage.rings.localization), 168
lt() (sage.rings.quotient_ring_element.QuotientRingElement method), 134

M
make_element() (in module sage.rings.fraction_field_element), 154
make_element_old() (in module sage.rings.fraction_field_element), 154
MapFreeModuleToRelativeRing (class in sage.rings.ring_extension_morphism), 228
MapRelativeRingToFreeModule (class in sage.rings.ring_extension_morphism), 228
matrix() (sage.rings.ring_extension_element.RingExtensionWithBasisElement method), 218
maximal_order() (sage.rings.ring_extension_element.RingExtensionWithBasisElement method), 218
minimal_associated_primes() (sage.rings.ideal.Ideal_generic method), 51
minpoly() (sage.rings.ring_extension_element.RingExtensionWithBasisElement method), 219
MinusInfinity (class in sage.rings.infinity), 252
module
sage.rings.abc, 21
sage.rings.big_oh, 241
sage.rings.derivation, 259
sage.rings.fraction_field, 137
sage.rings.fraction_field_element, 147
sage.rings.generic, 235
sage.rings.homset, 102
sage.rings.ideal, 37
sage.rings.ideal_monoid, 63
sage.rings.infinity, 244
sage.rings.localization, 157
sage.rings.morphism, 69
sage.rings.noncommutative_ideals, 64
sage.rings.numbers_abcs, 257
sage.rings.quotient_ring, 107
sage.rings.quotient_ring_element, 130
sage.rings.ring, 1
sage.rings.ring_extension, 173
sage.rings.ring_extension_element, 209
sage.rings.ring_extension_morphism, 228
modulus() (sage.rings.ring_extension.RingExtensionWithGen method), 185
monomial_coefficients() (sage.rings.derivation.RingDerivationWithoutTwist method), 275
monomials() (sage.rings.quotient_ring_element.QuotientRingElement method), 134
morphism_from_cover() (sage.rings.morphism.RingHomomorphism_from_quotient method), 98
multiplicative_order() (sage.rings.ring_extension_element.RingExtensionElement method), 213

N
natural_map() (sage.rings.homset.RingHomset_generic method), 103
ngens() (sage.rings.derivation.RingDerivationModule method), 268
ngens() (sage.rings.fraction_field.FractionField_generic method), 144
ngens() (sage.rings.ideal.Ideal_generic method), 52
ngens() (sage.rings.infinity.InfinityRing_class method), 251
relative_degree() (sage.rings.ring_extension.RingExtension_generic method), 205
reminders() (sage.rings.generic.ProductTree method), 238
residue_field() (sage.rings.ideal.Ideal_pid method), 58
retract() (sage.rings.quotient_ring.QuotientRing_nc method), 128
Ring (class in sage.rings.ring), 9
ring() (sage.rings.fraction_field.FractionField_generic method), 145
ring() (sage.rings.ideal_monoid.IdealMonoid_c method), 63
ring() (sage.rings.ideal.Ideal_generic method), 54
ring() (sage.rings.ring_extension.RingExtensionFractionField method), 177
ring_of_constants() (sage.rings.derivation.RingDerivationModule method), 269
root() (sage.rings.generic.ProductTree method), 239
S
sage.rings.abc module, 21
sage.rings.big_oh module, 241
sage.rings.derivation module, 259
sage.rings.fraction_field module, 137
sage.rings.fraction_field_element module, 147
sage.rings.generic module, 235
sage.rings.homset module, 102
sage.rings.ideal module, 37
sage.rings.ideal_monoid module, 63
sage.rings.infinity module, 244
sage.rings.localization module, 157
sage.rings.morphism module, 69
sage.rings.noncommutative_ideals module