Power Series Rings and Laurent Series Rings

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

CHAPTER

ONE

POWER SERIES RINGS

Power series rings are constructed in the standard Sage fashion. See also *Multivariate Power Series Rings*.

EXAMPLES:

Construct rings and elements:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R.random_element(6)  # random
-4 - 1/2*t^2 - 1/95*t^3 + 1/2*t^4 - 12*t^5 + O(t^6)
```

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
sage: p = -t + 1/2*t^3*u - 1/4*t^4*u + 2/3*v^5 + R.O(6); p
-t + 1/2*t^3*u - 1/4*t^4*u + 2/3*v^5 + O(t, u, v)^6
sage: p in R
True
```

The default precision is specified at construction, but does not bound the precision of created elements.

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=5)
sage: R.random_element(6)  # random
1/2 - 1/4*t + 2/3*t^2 - 5/2*t^3 + 2/3*t^5 + O(t^6)
```

Construct univariate power series from a list of coefficients:

```
sage: S = R([1, 3, 5, 7]); S
1 + 3*t + 5*t^2 + 7*t^3
```

The default precision of a power series ring stays fixed and cannot be changed. To work with different default precision, create a new power series ring:

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=10)
sage: sin(x)
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 + O(x^10)
sage: R.<x> = PowerSeriesRing(QQ, default_prec=15)
sage: sin(x)
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 + 1/6227020800*x^
↪13 + O(x^15)
```

An iterated example:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: S.<t2> = PowerSeriesRing(R)
sage: S
Power Series Ring in t2 over Power Series Ring in t over Integer Ring
sage: S.base_ring()
Power Series Ring in t over Integer Ring
```

Sage can compute with power series over the symbolic ring.

```
sage: # needs sage.symbolic
sage: K.<t> = PowerSeriesRing(SR, default_prec=5)
sage: a, b, c = var('a,b,c')
sage: f = a + b*t + c*t^2 + O(t^3)
sage: f*f
a^2 + 2*a*b*t + (b^2 + 2*a*c)*t^2 + O(t^3)
sage: f = sqrt(2) + sqrt(3)*t + O(t^3)
sage: f^2
2 + 2*sqrt(3)*sqrt(2)*t + 3*t^2 + O(t^3)
```

Elements are first coerced to constants in `base_ring`, then coerced into the `PowerSeriesRing`:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: f = Mod(2, 3) * t; (f, f.parent())
(2*t, Power Series Ring in t over Ring of integers modulo 3)
```

We make a sparse power series.

```
sage: R.<x> = PowerSeriesRing(QQ, sparse=True); R
Sparse Power Series Ring in x over Rational Field
sage: f = 1 + x^1000000
sage: g = f*f
sage: g.degree()
2000000
```

We make a sparse Laurent series from a power series generator:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: latex(-2/3*(1/t^3) + 1/t + 3/5*t^2 + O(t^5))
\frac{-\frac{2}{3}}{t^{3}} + \frac{1}{t} + \frac{3}{5}t^{2} + O(t^{5})
sage: S = parent(1/t); S
Sparse Laurent Series Ring in t over Rational Field
```

Choose another implementation of the attached polynomial ring:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: type(t.polynomial())                                                    #␣
↪needs sage.libs.flint
<... 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint
↪'>
sage: S.<s> = PowerSeriesRing(ZZ, implementation='NTL')                       #␣
↪needs sage.libs.ntl
sage: type(s.polynomial())                                                    #␣
↪needs sage.libs.ntl
<... 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl'>
```

AUTHORS:

- William Stein: the code
- Jeremy Cho (2006-05-17): some examples (above)
- Niles Johnson (2010-09): implement multivariate power series
- Simon King (2012-08): use category and coercion framework, github issue #13412

`sage.rings.power_series_ring.`**`PowerSeriesRing`**(*base_ring*, *name=None*, *arg2=None*, *names=None*, *sparse=False*, *default_prec=None*, *order='negdeglex'*, *num_gens=None*, *implementation=None*)

Create a univariate or multivariate power series ring over a given (commutative) base ring.

INPUT:

- `base_ring` – a commutative ring
- `name`, `names` – name(s) of the indeterminate
- **`default_prec` – the default precision used if an exact object must**
  be changed to an approximate object in order to do an arithmetic operation. If left as `None`, it will be set to the global default (20) in the univariate case, and 12 in the multivariate case.
- `sparse` – (default: `False`) whether power series are represented as sparse objects.
- `order` – (default: `negdeglex`) term ordering, for multivariate case
- `num_gens` – number of generators, for multivariate case

There is a unique power series ring over each base ring with given variable name. Two power series over the same base ring with different variable names are not equal or isomorphic.

EXAMPLES (Univariate):

```
sage: R = PowerSeriesRing(QQ, 'x'); R
Power Series Ring in x over Rational Field
```

```
sage: S = PowerSeriesRing(QQ, 'y'); S
Power Series Ring in y over Rational Field
```

```
sage: R = PowerSeriesRing(QQ, 10)
Traceback (most recent call last):
...
ValueError: variable name '10' does not start with a letter
```

```
sage: S = PowerSeriesRing(QQ, 'x', default_prec=15); S
Power Series Ring in x over Rational Field
sage: S.default_prec()
15
```

EXAMPLES (Multivariate) See also *Multivariate Power Series Rings*:

```
sage: R = PowerSeriesRing(QQ, 't,u,v'); R
Multivariate Power Series Ring in t, u, v over Rational Field
```

```
sage: N = PowerSeriesRing(QQ,'w',num_gens=5); N
Multivariate Power Series Ring in w0, w1, w2, w3, w4 over Rational Field
```

Number of generators can be specified before variable name without using keyword:

```
sage: M = PowerSeriesRing(QQ,4,'k'); M
Multivariate Power Series Ring in k0, k1, k2, k3 over Rational Field
```

Multivariate power series can be constructed using angle bracket or double square bracket notation:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ, 't,u,v'); R
Multivariate Power Series Ring in t, u, v over Rational Field

sage: ZZ[['s,t,u']]
Multivariate Power Series Ring in s, t, u over Integer Ring
```

Sparse multivariate power series ring:

```
sage: M = PowerSeriesRing(QQ,4,'k',sparse=True); M
Sparse Multivariate Power Series Ring in k0, k1, k2, k3 over
Rational Field
```

Power series ring over polynomial ring:

```
sage: H = PowerSeriesRing(PolynomialRing(ZZ,3,'z'), 4, 'f'); H
Multivariate Power Series Ring in f0, f1, f2, f3 over Multivariate
Polynomial Ring in z0, z1, z2 over Integer Ring
```

Power series ring over finite field:

```
sage: S = PowerSeriesRing(GF(65537),'x,y'); S                          #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size
65537
```

Power series ring with many variables:

```
sage: R = PowerSeriesRing(ZZ, ['x%s'%p for p in primes(100)]); R        #␣
↪needs sage.libs.pari
Multivariate Power Series Ring in x2, x3, x5, x7, x11, x13, x17, x19,
x23, x29, x31, x37, x41, x43, x47, x53, x59, x61, x67, x71, x73, x79,
x83, x89, x97 over Integer Ring
```

- Use `inject_variables()` to make the variables available for interactive use.

  ```
  sage: R.inject_variables()
  ↪# needs sage.libs.pari
  Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37,
  x41, x43, x47, x53, x59, x61, x67, x71, x73, x79, x83, x89, x97

  sage: f = x47 + 3*x11*x29 - x19 + R.O(3)
  ↪# needs sage.libs.pari
  sage: f in R
  ↪# needs sage.libs.pari
  True
  ```

Variable ordering determines how series are displayed:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,order='deglex'); T
Multivariate Power Series Ring in a, b over Integer Ring
sage: T.term_order()
Degree lexicographic term order
sage: p = - 2*b^6 + a^5*b^2 + a^7 - b^2 - a*b^3 + T.O(9); p
a^7 + a^5*b^2 - 2*b^6 - a*b^3 - b^2 + O(a, b)^9

sage: U = PowerSeriesRing(ZZ,'a,b',order='negdeglex'); U
Multivariate Power Series Ring in a, b over Integer Ring
sage: U.term_order()
Negative degree lexicographic term order
sage: U(p)
-b^2 - a*b^3 - 2*b^6 + a^7 + a^5*b^2 + O(a, b)^9
```

See also:

- `sage.misc.defaults.set_series_precision()`

class `sage.rings.power_series_ring.`**`PowerSeriesRing_domain`**(*base_ring*, *name=None*, *default_prec=None*, *sparse=False*, *implementation=None*, *category=None*)

Bases: *`PowerSeriesRing_generic`*, `IntegralDomain`

`fraction_field()`

Return the Laurent series ring over the fraction field of the base ring.

This is actually *not* the fraction field of this ring, but its completion with respect to the topology defined by the valuation. When we are working at finite precision, these two fields are indistinguishable; that is the reason why we allow ourselves to make this confusion here.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.fraction_field()
Laurent Series Ring in t over Rational Field
sage: Frac(R)
Laurent Series Ring in t over Rational Field
```

class `sage.rings.power_series_ring.`**`PowerSeriesRing_generic`**(*base_ring*, *name=None*, *default_prec=None*, *sparse=False*, *implementation=None*, *category=None*)

Bases: `UniqueRepresentation`, `CommutativeRing`, `Nonexact`

A power series ring.

`base_extend`(*R*)

Return the power series ring over R in the same variable as `self`, assuming there is a canonical coerce map from the base ring of `self` to R.

EXAMPLES:

```
sage: R.<T> = GF(7)[[]]; R
Power Series Ring in T over Finite Field of size 7
sage: R.change_ring(ZZ)
```

(continues on next page)

(continued from previous page)

```
Power Series Ring in T over Integer Ring
sage: R.base_extend(ZZ)
Traceback (most recent call last):
...
TypeError: no base extension defined
```

change_ring(*R*)

Return the power series ring over R in the same variable as `self`.

EXAMPLES:

```
sage: R.<T> = QQ[[]]; R
Power Series Ring in T over Rational Field
sage: R.change_ring(GF(7))
Power Series Ring in T over Finite Field of size 7
sage: R.base_extend(GF(7))
Traceback (most recent call last):
...
TypeError: no base extension defined
sage: R.base_extend(QuadraticField(3,'a'))                                   #␣
↪needs sage.rings.number_field
Power Series Ring in T over Number Field in a
 with defining polynomial x^2 - 3 with a = 1.732050807568878?
```

change_var(*var*)

Return the power series ring in variable `var` over the same base ring.

EXAMPLES:

```
sage: R.<T> = QQ[[]]; R
Power Series Ring in T over Rational Field
sage: R.change_var('D')
Power Series Ring in D over Rational Field
```

characteristic()

Return the characteristic of this power series ring, which is the same as the characteristic of the base ring of the power series ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.characteristic()
0
sage: R.<w> = Integers(2^50)[[]]; R
Power Series Ring in w over Ring of integers modulo 1125899906842624
sage: R.characteristic()
1125899906842624
```

construction()

Return the functorial construction of `self`, namely, completion of the univariate polynomial ring with respect to the indeterminate (to a given precision).

EXAMPLES:

```
sage: R = PowerSeriesRing(ZZ, 'x')
sage: c, S = R.construction(); S
Univariate Polynomial Ring in x over Integer Ring
sage: R == c(S)
True
sage: R = PowerSeriesRing(ZZ, 'x', sparse=True)
sage: c, S = R.construction()
sage: R == c(S)
True
```

gen(*n=0*)

Return the generator of this power series ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.gen()
t
sage: R.gen(3)
Traceback (most recent call last):
...
IndexError: generator n>0 not defined
```

is_dense()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_dense()
True
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_dense()
False
```

is_exact()

Return `False` since the ring of power series over any ring is not exact.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.is_exact()
False
```

is_field(*proof=True*)

Return `False` since the ring of power series over any ring is never a field.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.is_field()
False
```

is_finite()

Return `False` since the ring of power series over any ring is never finite.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.is_finite()
False
```

is_sparse()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_sparse()
False
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_sparse()
True
```

laurent_series_ring()

If this is the power series ring $R[[t]]$, return the Laurent series ring $R((t))$.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ, default_prec=5)
sage: S = R.laurent_series_ring(); S
Laurent Series Ring in t over Integer Ring
sage: S.default_prec()
5
sage: f = 1 + t; g = 1/f; g
1 - t + t^2 - t^3 + t^4 + O(t^5)
```

ngens()

Return the number of generators of this power series ring.

This is always 1.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: R.ngens()
1
```

random_element(*prec=None*, **args*, ***kwds*)

Return a random power series.

INPUT:

- `prec` – Integer specifying precision of output (default: default precision of `self`)
- `*args`, `**kwds` – Passed on to the `random_element` method for the base ring

OUTPUT:

- Power series with precision `prec` whose coefficients are random elements from the base ring, randomized subject to the arguments `*args` and `**kwds`

ALGORITHM:

Call the `random_element` method on the underlying polynomial ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R.random_element(5)  # random
-4 - 1/2*t^2 - 1/95*t^3 + 1/2*t^4 + O(t^5)
sage: R.random_element(10)  # random
-1/2 + 2*t - 2/7*t^2 - 25*t^3 - t^4 + 2*t^5 - 4*t^7 - 1/3*t^8 - t^9 + O(t^10)
```

If given no argument, `random_element` uses default precision of self:

```
sage: T = PowerSeriesRing(ZZ,'t')
sage: T.default_prec()
20
sage: T.random_element()  # random
4 + 2*t - t^2 - t^3 + 2*t^4 + t^5 + t^6 - 2*t^7 - t^8 - t^9 + t^11
 - 6*t^12 + 2*t^14 + 2*t^16 - t^17 - 3*t^18 + O(t^20)
sage: S = PowerSeriesRing(ZZ,'t', default_prec=4)
sage: S.random_element()  # random
2 - t - 5*t^2 + t^3 + O(t^4)
```

Further arguments are passed to the underlying base ring (github issue #9481):

```
sage: SZ = PowerSeriesRing(ZZ,'v')
sage: SQ = PowerSeriesRing(QQ,'v')
sage: SR = PowerSeriesRing(RR,'v')

sage: SZ.random_element(x=4, y=6)  # random
4 + 5*v + 5*v^2 + 5*v^3 + 4*v^4 + 5*v^5 + 5*v^6 + 5*v^7 + 4*v^8
 + 5*v^9 + 4*v^10 + 4*v^11 + 5*v^12 + 5*v^13 + 5*v^14 + 5*v^15
 + 5*v^16 + 5*v^17 + 4*v^18 + 5*v^19 + O(v^20)
sage: SZ.random_element(3, x=4, y=6)  # random
5 + 4*v + 5*v^2 + O(v^3)
sage: SQ.random_element(3, num_bound=3, den_bound=100)  # random
1/87 - 3/70*v - 3/44*v^2 + O(v^3)
sage: SR.random_element(3, max=10, min=-10)  # random
2.85948321262904 - 9.73071330911226*v - 6.60414378519265*v^2 + O(v^3)
```

residue_field()

Return the residue field of this power series ring.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(GF(17))
sage: R.residue_field()
Finite Field of size 17
sage: R.<x> = PowerSeriesRing(Zp(5))                                            #␣
↪needs sage.rings.padics
sage: R.residue_field()                                                         #␣
↪needs sage.rings.padics
Finite Field of size 5
```

uniformizer()

Return a uniformizer of this power series ring if it is a discrete valuation ring (i.e., if the base ring is actually a field). Otherwise, an error is raised.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R.uniformizer()
t

sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.uniformizer()
Traceback (most recent call last):
...
TypeError: The base ring is not a field
```

variable_names_recursive(*depth=None*)

Return the list of variable names of this and its base rings.

EXAMPLES:

```
sage: R = QQ[['x']][['y']][['z']]
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```

class sage.rings.power_series_ring.**PowerSeriesRing_over_field**(*base_ring*, *name=None*, *default_prec=None*, *sparse=False*, *implementation=None*, *category=None*)

Bases: *PowerSeriesRing_domain*

fraction_field()

Return the fraction field of this power series ring, which is defined since this is over a field.

This fraction field is just the Laurent series ring over the base field.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(GF(7))
sage: R.fraction_field()
Laurent Series Ring in t over Finite Field of size 7
sage: Frac(R)
Laurent Series Ring in t over Finite Field of size 7
```

sage.rings.power_series_ring.**is_PowerSeriesRing**(*R*)

Return `True` if this is a *univariate* power series ring. This is in keeping with the behavior of `is_PolynomialRing` versus `is_MPolynomialRing`.

EXAMPLES:

```
sage: from sage.rings.power_series_ring import is_PowerSeriesRing
sage: is_PowerSeriesRing(10)
False
sage: is_PowerSeriesRing(QQ[['x']])
True
```

sage.rings.power_series_ring.**unpickle_power_series_ring_v0**(*base_ring*, *name*, *default_prec*, *sparse*)

Unpickle (deserialize) a univariate power series ring according to the given inputs.

EXAMPLES:

```
sage: P.<x> = PowerSeriesRing(QQ)
sage: loads(dumps(P)) == P  # indirect doctest
True
```

CHAPTER

TWO

POWER SERIES

Sage provides an implementation of dense and sparse power series over any Sage base ring. This is the base class of the implementations of univariate and multivariate power series ring elements in Sage (see also *Power Series Methods*, *Multivariate Power Series*).

AUTHORS:

- William Stein
- David Harvey (2006-09-11): added solve_linear_de() method
- Robert Bradshaw (2007-04): sqrt, rmul, lmul, shifting
- Robert Bradshaw (2007-04): Cython version
- Simon King (2012-08): use category and coercion framework, github issue #13412

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: TestSuite(R).run()
sage: R([1,2,3])
1 + 2*x + 3*x^2
sage: R([1,2,3], 10)
1 + 2*x + 3*x^2 + O(x^10)
sage: f = 1 + 2*x - 3*x^3 + O(x^4); f
1 + 2*x - 3*x^3 + O(x^4)
sage: f^10
1 + 20*x + 180*x^2 + 930*x^3 + O(x^4)
sage: g = 1/f; g
1 - 2*x + 4*x^2 - 5*x^3 + O(x^4)
sage: g * f
1 + O(x^4)
```

In Python (as opposed to Sage) create the power series ring and its generator as follows:

```
sage: R = PowerSeriesRing(ZZ, 'x')
sage: x = R.gen()
sage: parent(x)
Power Series Ring in x over Integer Ring
```

EXAMPLES:

This example illustrates that coercion for power series rings is consistent with coercion for polynomial rings.

```
sage: poly_ring1.<gen1> = PolynomialRing(QQ)
sage: poly_ring2.<gen2> = PolynomialRing(QQ)
sage: huge_ring.<x> = PolynomialRing(poly_ring1)
```

The generator of the first ring gets coerced in as itself, since it is the base ring.

```
sage: huge_ring(gen1)
gen1
```

The generator of the second ring gets mapped via the natural map sending one generator to the other.

```
sage: huge_ring(gen2)
x
```

With power series the behavior is the same.

```
sage: power_ring1.<gen1> = PowerSeriesRing(QQ)
sage: power_ring2.<gen2> = PowerSeriesRing(QQ)
sage: huge_power_ring.<x> = PowerSeriesRing(power_ring1)
sage: huge_power_ring(gen1)
gen1
sage: huge_power_ring(gen2)
x
```

class `sage.rings.power_series_ring_element.`**`PowerSeries`**

Bases: `AlgebraElement`

A power series. Base class of univariate and multivariate power series. The following methods are available with both types of objects.

`O`(*prec*)

Return this series plus $O(x^{\text{prec}})$. Does not change `self`.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: p = 1 + x^2 + x^10; p
1 + x^2 + x^10
sage: p.O(15)
1 + x^2 + x^10 + O(x^15)
sage: p.O(5)
1 + x^2 + O(x^5)
sage: p.O(-5)
Traceback (most recent call last):
...
ValueError: prec (= -5) must be non-negative
```

`V`(*n*)

If $f = \sum a_m x^m$, then this function returns $\sum a_m x^{nm}$.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: p = 1 + x^2 + x^10; p
1 + x^2 + x^10
```

(continues on next page)

(continued from previous page)

```
sage: p.V(3)
1 + x^6 + x^30
sage: (p + O(x^20)).V(3)
1 + x^6 + x^30 + O(x^60)
```

add_bigoh(*prec*)

Return the power series of precision at most `prec` got by adding $O(q^{\text{prec}})$ to f, where q is the variable.

EXAMPLES:

```
sage: R.<A> = RDF[[]]
sage: f = (1+A+O(A^5))^5; f
1.0 + 5.0*A + 10.0*A^2 + 10.0*A^3 + 5.0*A^4 + O(A^5)
sage: f.add_bigoh(3)
1.0 + 5.0*A + 10.0*A^2 + O(A^3)
sage: f.add_bigoh(5)
1.0 + 5.0*A + 10.0*A^2 + 10.0*A^3 + 5.0*A^4 + O(A^5)
```

base_extend(*R*)

Return a copy of this power series but with coefficients in R.

The following coercion uses `base_extend` implicitly:

```
sage: R.<t> = ZZ[['t']]
sage: (t - t^2) * Mod(1, 3)
t + 2*t^2
```

base_ring()

Return the base ring that this power series is defined over.

EXAMPLES:

```
sage: R.<t> = GF(49,'alpha')[[]]                                        #␣
↪needs sage.rings.finite_rings
sage: (t^2 + O(t^3)).base_ring()                                        #␣
↪needs sage.rings.finite_rings
Finite Field in alpha of size 7^2
```

change_ring(*R*)

Change if possible the coefficients of self to lie in R.

EXAMPLES:

```
sage: R.<T> = QQ[[]]; R
Power Series Ring in T over Rational Field
sage: f = 1 - 1/2*T + 1/3*T^2 + O(T^3)
sage: f.base_extend(GF(5))
Traceback (most recent call last):
...
TypeError: no base extension defined
sage: f.change_ring(GF(5))
1 + 2*T + 2*T^2 + O(T^3)
sage: f.change_ring(GF(3))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ZeroDivisionError: inverse of Mod(0, 3) does not exist
```

We can only change the ring if there is a `__call__` coercion defined. The following succeeds because `ZZ(K(4))` is defined.

```
sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')                     #␣
↪needs sage.rings.number_field
sage: R.<t> = K[['t']]                                                        #␣
↪needs sage.rings.number_field
sage: (4*t).change_ring(ZZ)                                                   #␣
↪needs sage.rings.number_field
4*t
```

This does not succeed because `ZZ(K(a+1))` is not defined.

```
sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')                     #␣
↪needs sage.rings.number_field
sage: R.<t> = K[['t']]                                                        #␣
↪needs sage.rings.number_field
sage: ((a+1)*t).change_ring(ZZ)                                               #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: Unable to coerce a + 1 to an integer
```

coefficients()

Return the nonzero coefficients of self.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 - 10/3*t^3
sage: f.coefficients()
[1, 1, -10/3]
```

common_prec(*f*)

Return minimum precision of f and `self`.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
```

```
sage: f = t + t^2 + O(t^3)
sage: g = t + t^3 + t^4 + O(t^4)
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3
```

```
sage: f = t + t^2 + O(t^3)
sage: g = t^2
sage: f.common_prec(g)
```

(continues on next page)

(continued from previous page)

```
3
sage: g.common_prec(f)
3
```

```
sage: f = t + t^2
sage: f = t^2
sage: f.common_prec(g)
+Infinity
```

cos(*prec='infinity'*)

Apply cos to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: cos(f)
1 - 1/2*t^2 - t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: cos(f)
1 - 1/2*a^2 - a*b - 1/2*b^2 + O(a, b)^3
sage: f.cos()
1 - 1/2*a^2 - a*b - 1/2*b^2 + O(a, b)^3
sage: f.cos(prec=2)
1 + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2+f
sage: cos(g)
Traceback (most recent call last):
...
ValueError: can only apply cos to formal power series with zero constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: cos(a)
1 - 1/2*a^2 + 1/24*a^4 - 1/720*a^6 + 1/40320*a^8 - 1/3628800*a^10 + O(a, b)^12
sage: a.cos(prec=5)
```

(continues on next page)

(continued from previous page)

```
1 - 1/2*a^2 + 1/24*a^4 + O(a, b)^5
sage: cos(a + T.O(5))
1 - 1/2*a^2 + 1/24*a^4 + O(a, b)^5
```

cosh(*prec='infinity'*)

Apply cosh to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: cosh(f)
1 + 1/2*t^2 + t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: cosh(f)
1 + 1/2*a^2 + a*b + 1/2*b^2 + O(a, b)^3
sage: f.cosh()
1 + 1/2*a^2 + a*b + 1/2*b^2 + O(a, b)^3
sage: f.cosh(prec=2)
1 + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2 + f
sage: cosh(g)
Traceback (most recent call last):
...
ValueError: can only apply cosh to formal power series with zero
constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: cosh(a)
1 + 1/2*a^2 + 1/24*a^4 + 1/720*a^6 + 1/40320*a^8 + 1/3628800*a^10 +
 O(a, b)^12
sage: a.cosh(prec=5)
1 + 1/2*a^2 + 1/24*a^4 + O(a, b)^5
sage: cosh(a + T.O(5))
1 + 1/2*a^2 + 1/24*a^4 + O(a, b)^5
```

degree()

Return the degree of this power series, which is by definition the degree of the underlying polynomial.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^100000 + O(t^10000000)
sage: f.degree()
100000
```

derivative(**args*)

The formal derivative of this power series, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:

`_derivative()`

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(QQ)
sage: g = -x + x^2/2 - x^4 + O(x^6)
sage: g.derivative()
-1 + x - 4*x^3 + O(x^5)
sage: g.derivative(x)
-1 + x - 4*x^3 + O(x^5)
sage: g.derivative(x, x)
1 - 12*x^2 + O(x^4)
sage: g.derivative(x, 2)
1 - 12*x^2 + O(x^4)
```

egf_to_ogf()

Return the ordinary generating function power series, assuming self is an exponential generating function power series.

This function is known as `serlaplace` in PARI/GP.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2/factorial(2) + 2*t^3/factorial(3)
sage: f.egf_to_ogf()
t + t^2 + 2*t^3
```

exp(*prec=None*)

Return exp of this power series to the indicated precision.

INPUT:

- `prec` - integer; default is `self.parent().default_prec`

ALGORITHM: See *solve_linear_de()*.

Note:

- Screwy things can happen if the coefficient ring is not a field of characteristic zero. See *solve_linear_de()*.

AUTHORS:

- David Harvey (2006-09-08): rewrote to use simplest possible "lazy" algorithm.
- David Harvey (2006-09-10): rewrote to use divide-and-conquer strategy.
- David Harvey (2006-09-11): factored functionality out to solve_linear_de().
- Sourav Sen Gupta, David Harvey (2008-11): handle constant term

EXAMPLES:

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

Check that $\exp(t)$ is, well, $\exp(t)$:

```
sage: (t + O(t^10)).exp()
1 + t + 1/2*t^2 + 1/6*t^3 + 1/24*t^4 + 1/120*t^5 + 1/720*t^6
  + 1/5040*t^7 + 1/40320*t^8 + 1/362880*t^9 + O(t^10)
```

Check that $\exp(\log(1+t))$ is $1+t$:

```
sage: (sum([-(-t)^n/n for n in range(1, 10)]) + O(t^10)).exp()
1 + t + O(t^10)
```

Check that $\exp(2t+t^2-t^5)$ is whatever it is:

```
sage: (2*t + t^2 - t^5 + O(t^10)).exp()
1 + 2*t + 3*t^2 + 10/3*t^3 + 19/6*t^4 + 8/5*t^5 - 7/90*t^6
  - 538/315*t^7 - 425/168*t^8 - 30629/11340*t^9 + O(t^10)
```

Check requesting lower precision:

```
sage: (t + t^2 - t^5 + O(t^10)).exp(5)
1 + t + 3/2*t^2 + 7/6*t^3 + 25/24*t^4 + O(t^5)
```

Can't get more precision than the input:

```
sage: (t + t^2 + O(t^3)).exp(10)
1 + t + 3/2*t^2 + O(t^3)
```

Check some boundary cases:

```
sage: (t + O(t^2)).exp(1)
1 + O(t)
sage: (t + O(t^2)).exp(0)
O(t^0)
```

Handle nonzero constant term (fixes github issue #4477):

```
sage: # needs sage.rings.real_mpfr
sage: R.<x> = PowerSeriesRing(RR)
sage: (1 + x + x^2 + O(x^3)).exp()
2.71828182845905 + 2.71828182845905*x + 4.07742274268857*x^2 + O(x^3)
```

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: (1 + x + O(x^2)).exp()                                              #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ArithmeticError: exponential of constant term does not belong
to coefficient ring (consider working in a larger ring)
```

```
sage: R.<x> = PowerSeriesRing(GF(5))
sage: (1 + x + O(x^2)).exp()
Traceback (most recent call last):
...
ArithmeticError: constant term of power series does not support exponentiation
```

exponents()

Return the exponents appearing in self with nonzero coefficients.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 - 10/3*t^3
sage: f.exponents()
[1, 2, 3]
```

inverse()

Return the inverse of self, i.e., self^(-1).

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: t.inverse()
t^-1
sage: type(_)
<class 'sage.rings.laurent_series_ring_element.LaurentSeries'>
sage: (1-t).inverse()
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + ...
```

is_dense()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_dense()
True
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_dense()
False
```

is_gen()

Return `True` if this is the generator (the variable) of the power series ring.

EXAMPLES:

```
sage: R.<t> = QQ[[]]
sage: t.is_gen()
```

(continues on next page)

(continued from previous page)

```
True
sage: (1 + 2*t).is_gen()
False
```

Note that this only returns True on the actual generator, not on something that happens to be equal to it.

```
sage: (1*t).is_gen()
False
sage: 1*t == t
True
```

is_monomial()

Return True if this element is a monomial. That is, if self is x^n for some non-negative integer n.

EXAMPLES:

```
sage: k.<z> = PowerSeriesRing(QQ, 'z')
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
False
sage: (z^2909).is_monomial()
True
sage: (3*z^2909).is_monomial()
False
```

is_sparse()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_sparse()
False
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_sparse()
True
```

is_square()

Return True if this function has a square root in this ring, e.g., there is an element y in self.parent() such that y^2 equals self.

ALGORITHM: If the base ring is a field, this is true whenever the power series has even valuation and the leading coefficient is a perfect square.

For an integral domain, it attempts the square root in the fraction field and tests whether or not the result lies in the original ring.

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: (1+t).is_square()
True
sage: (2+t).is_square()
```

(continues on next page)

(continued from previous page)

```
False
sage: (2+t.change_ring(RR)).is_square()
True
sage: t.is_square()
False
sage: K.<t> = PowerSeriesRing(ZZ, 't', 5)
sage: (1+t).is_square()
False
sage: f = (1+t)^100
sage: f.is_square()
True
```

is_unit()

Return True if this power series is invertible.

A power series is invertible precisely when the constant term is invertible.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: (-1 + t - t^5).is_unit()
True
sage: (3 + t - t^5).is_unit()
False
sage: O(t^0).is_unit()
False
```

AUTHORS:

- David Harvey (2006-09-03)

jacobi_continued_fraction()

Return the Jacobi continued fraction of self.

The J-fraction or Jacobi continued fraction of a power series is a continued fraction expansion with steps of size two. We use the following convention

$$1/(1 + A_0 t + B_0 t^2/(1 + A_1 t + B_1 t^2/(1 + \cdots)))$$

OUTPUT:

tuple of pairs (A_n, B_n) for $n \geq 0$

The expansion is done as long as possible given the precision. Whenever the expansion is not well-defined, because it would require to divide by zero, an exception is raised.

See section 2.7 of [Kra1999det] for the close relationship of this kind of expansion with Hankel determinants and orthogonal polynomials.

EXAMPLES:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: s = sum(factorial(k) * t**k for k in range(12)).O(12)
sage: s.jacobi_continued_fraction()
((-1, -1), (-3, -4), (-5, -9), (-7, -16), (-9, -25))
```

Another example:

```
sage: (log(1+t)/t).jacobi_continued_fraction()
((1/2, -1/12),
 (1/2, -1/15),
 (1/2, -9/140),
 (1/2, -4/63),
 (1/2, -25/396),
 (1/2, -9/143),
 (1/2, -49/780),
 (1/2, -16/255),
 (1/2, -81/1292))
```

laurent_series()

Return the Laurent series associated to this power series, i.e., this series considered as a Laurent series.

EXAMPLES:

```
sage: k.<w> = QQ[[]]
sage: f = 1 + 17*w + 15*w^3 + O(w^5)
sage: parent(f)
Power Series Ring in w over Rational Field
sage: g = f.laurent_series(); g
1 + 17*w + 15*w^3 + O(w^5)
```

lift_to_precision(*absprec=None*)

Return a congruent power series with absolute precision at least `absprec`.

INPUT:

- `absprec` – an integer or `None` (default: `None`), the absolute precision of the result. If `None`, lifts to an exact element.

EXAMPLES:

```
sage: A.<t> = PowerSeriesRing(GF(5))
sage: x = t + t^2 + O(t^5)
sage: x.lift_to_precision(10)
t + t^2 + O(t^10)
sage: x.lift_to_precision()
t + t^2
```

list()

See this method in derived classes:

- *sage.rings.power_series_poly.PowerSeries_poly.list()*,
- *sage.rings.multi_power_series_ring_element.MPowerSeries.list()*

Implementations *MUST* override this in the derived class.

EXAMPLES:

```
sage: from sage.rings.power_series_ring_element import PowerSeries
sage: R.<x> = PowerSeriesRing(ZZ)
sage: PowerSeries.list(1 + x^2)
Traceback (most recent call last):
...
NotImplementedError
```

`log`(*prec=None*)

Return log of this power series to the indicated precision.

This works only if the constant term of the power series is 1 or the base ring can take the logarithm of the constant coefficient.

INPUT:

- `prec` – integer; default is `self.parent().default_prec()`

ALGORITHM: See *solve_linear_de()*.

Warning: Screwy things can happen if the coefficient ring is not a field of characteristic zero. See *solve_linear_de()*.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=10)
sage: (1 + t + O(t^10)).log()
t - 1/2*t^2 + 1/3*t^3 - 1/4*t^4 + 1/5*t^5 - 1/6*t^6 + 1/7*t^7
  - 1/8*t^8 + 1/9*t^9 + O(t^10)

sage: t.exp().log()
t + O(t^10)

sage: (1 + t).log().exp()
1 + t + O(t^10)

sage: (-1 + t + O(t^10)).log()
Traceback (most recent call last):
...
ArithmeticError: constant term of power series is not 1

sage: # needs sage.rings.real_mpfr
sage: R.<t> = PowerSeriesRing(RR)
sage: (2 + t).log().exp()
2.00000000000000 + 1.00000000000000*t + O(t^20)
```

`map_coefficients`(*f*, *new_base_ring=None*)

Return the series obtained by applying `f` to the non-zero coefficients of `self`.

If `f` is a `sage.categories.map.Map`, then the resulting series will be defined over the codomain of `f`. Otherwise, the resulting polynomial will be over the same ring as self. Set `new_base_ring` to override this behaviour.

INPUT:

- `f` – a callable that will be applied to the coefficients of self.
- `new_base_ring` (optional) – if given, the resulting polynomial will be defined over this ring.

EXAMPLES:

```
sage: R.<x> = SR[[]]                                                    #␣
↪needs sage.symbolic
sage: f = (1+I)*x^2 + 3*x - I                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
sage: f.map_coefficients(lambda z: z.conjugate())                          #␣
↪needs sage.symbolic
I + 3*x + (-I + 1)*x^2
sage: R.<x> = ZZ[[]]
sage: f = x^2 + 2
sage: f.map_coefficients(lambda a: a + 42)
44 + 43*x^2
```

Examples with different base ring:

```
sage: R.<x> = ZZ[[]]
sage: k = GF(2)
sage: residue = lambda x: k(x)
sage: f = 4*x^2+x+3
sage: g = f.map_coefficients(residue); g
1 + x
sage: g.parent()
Power Series Ring in x over Integer Ring
sage: g = f.map_coefficients(residue, new_base_ring=k); g
1 + x
sage: g.parent()
Power Series Ring in x over Finite Field of size 2
sage: residue = k.coerce_map_from(ZZ)
sage: g = f.map_coefficients(residue); g
1 + x
sage: g.parent()
Power Series Ring in x over Finite Field of size 2
```

Tests other implementations:

```
sage: # needs sage.libs.pari
sage: R.<q> = PowerSeriesRing(GF(11), implementation='pari')
sage: f = q - q^3 + O(q^10)
sage: f.map_coefficients(lambda c: c - 2)
10*q + 8*q^3 + O(q^10)
```

nth_root(*n*, *prec=None*)

Return the n-th root of this power series.

INPUT:

- n – integer
- prec – integer (optional) - precision of the result. Though, if this series has finite precision, then the result cannot have larger precision.

EXAMPLES:

```
sage: R.<x> = QQ[[]]
sage: (1+x).nth_root(5)
1 + 1/5*x - 2/25*x^2 + ... + 12039376311816/2384185791015625*x^19 + O(x^20)

sage: (1 + x + O(x^5)).nth_root(5)
1 + 1/5*x - 2/25*x^2 + 6/125*x^3 - 21/625*x^4 + O(x^5)
```

Check that the results are consistent with taking log and exponential:

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=100)
sage: p = (1 + 2*x - x^4)**200
sage: p1 = p.nth_root(1000, prec=100)
sage: p2 = (p.log()/1000).exp()
sage: p1.prec() == p2.prec() == 100
True
sage: p1.polynomial() == p2.polynomial()
True
```

Positive characteristic:

```
sage: R.<u> = GF(3)[[]]
sage: p = 1 + 2 * u^2
sage: p.nth_root(4)
1 + 2*u^2 + u^6 + 2*u^8 + u^12 + 2*u^14 + O(u^20)
sage: p.nth_root(4)**4
1 + 2*u^2 + O(u^20)
```

ogf_to_egf()

Return the exponential generating function power series, assuming self is an ordinary generating function power series.

This can also be computed as `serconvol(f,exp(t))` in PARI/GP.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 + 2*t^3
sage: f.ogf_to_egf()
t + 1/2*t^2 + 1/3*t^3
```

padded_list(*n=None*)

Return a list of coefficients of self up to (but not including) q^n.

Includes 0's in the list on the right so that the list has length n.

INPUT:

- `n` - (optional) an integer that is at least 0. If `n` is not given, it will be taken to be the precision of self, unless this is `+Infinity`, in which case we just return `self.list()`.

EXAMPLES:

```
sage: R.<q> = PowerSeriesRing(QQ)
sage: f = 1 - 17*q + 13*q^2 + 10*q^4 + O(q^7)
sage: f.list()
[1, -17, 13, 0, 10]
sage: f.padded_list(7)
[1, -17, 13, 0, 10, 0, 0]
sage: f.padded_list(10)
[1, -17, 13, 0, 10, 0, 0, 0, 0, 0]
sage: f.padded_list(3)
[1, -17, 13]
sage: f.padded_list()
```

(continues on next page)

(continued from previous page)

```
[1, -17, 13, 0, 10, 0, 0]
sage: g = 1 - 17*q + 13*q^2 + 10*q^4
sage: g.list()
[1, -17, 13, 0, 10]
sage: g.padded_list()
[1, -17, 13, 0, 10]
sage: g.padded_list(10)
[1, -17, 13, 0, 10, 0, 0, 0, 0, 0]
```

polynomial()

See this method in derived classes:

- *sage.rings.power_series_poly.PowerSeries_poly.polynomial()*,
- *sage.rings.multi_power_series_ring_element.MPowerSeries.polynomial()*

Implementations *MUST* override this in the derived class.

EXAMPLES:

```
sage: from sage.rings.power_series_ring_element import PowerSeries
sage: R.<x> = PowerSeriesRing(ZZ)
sage: PowerSeries.polynomial(1 + x^2)
Traceback (most recent call last):
...
NotImplementedError
```

prec()

The precision of $... + O(x^r)$ is by definition r.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).prec()
3
sage: (1 - t^2 + O(t^100)).prec()
100
```

precision_absolute()

Return the absolute precision of this series.

By definition, the absolute precision of $... + O(x^r)$ is r.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).precision_absolute()
3
sage: (1 - t^2 + O(t^100)).precision_absolute()
100
```

precision_relative()

Return the relative precision of this series, that is the difference between its absolute precision and its valuation.

By convention, the relative precision of 0 (or $O(x^r)$ for any r) is 0.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).precision_relative()
1
sage: (1 - t^2 + O(t^100)).precision_relative()
100
sage: O(t^4).precision_relative()
0
```

shift(*n*)

Return this power series multiplied by the power t^n.

If n is negative, terms below t^{-n} are discarded.

This power series is left unchanged.

Note: Despite the fact that higher order terms are printed to the right in a power series, right shifting decreases the powers of t, while left shifting increases them. This is to be consistent with polynomials, integers, etc.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ['y'], 't', 5)
sage: f = ~(1+t); f
1 - t + t^2 - t^3 + t^4 + O(t^5)
sage: f.shift(3)
t^3 - t^4 + t^5 - t^6 + t^7 + O(t^8)
sage: f >> 2
1 - t + t^2 + O(t^3)
sage: f << 10
t^10 - t^11 + t^12 - t^13 + t^14 + O(t^15)
sage: t << 29
t^30
```

AUTHORS:

- Robert Bradshaw (2007-04-18)

sin(*prec='infinity'*)

Apply sin to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: sin(f)
t + t^2 - 1/6*t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: sin(f)
a + b + a*b + O(a, b)^3
sage: f.sin()
a + b + a*b + O(a, b)^3
sage: f.sin(prec=2)
a + b + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2+f
sage: sin(g)
Traceback (most recent call last):
...
ValueError: can only apply sin to formal power series with zero constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: sin(a)
a - 1/6*a^3 + 1/120*a^5 - 1/5040*a^7 + 1/362880*a^9 - 1/39916800*a^11 + O(a, b)^
↪12
sage: a.sin(prec=5)
a - 1/6*a^3 + O(a, b)^5
sage: sin(a + T.O(5))
a - 1/6*a^3 + O(a, b)^5
```

sinh(*prec='infinity'*)

Apply sinh to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: sinh(f)
t + t^2 + 1/6*t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: sinh(f)
a + b + a*b + O(a, b)^3
```

(continues on next page)

(continued from previous page)

```
sage: f.sinh()
a + b + a*b + O(a, b)^3
sage: f.sinh(prec=2)
a + b + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2 + f
sage: sinh(g)
Traceback (most recent call last):
...
ValueError: can only apply sinh to formal power series with zero
constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: sinh(a)
a + 1/6*a^3 + 1/120*a^5 + 1/5040*a^7 + 1/362880*a^9 +
 1/39916800*a^11 + O(a, b)^12
sage: a.sinh(prec=5)
a + 1/6*a^3 + O(a, b)^5
sage: sinh(a + T.O(5))
a + 1/6*a^3 + O(a, b)^5
```

solve_linear_de(*prec='infinity'*, *b=None*, *f0=None*)

Obtain a power series solution to an inhomogeneous linear differential equation of the form:

$$f'(t) = a(t)f(t) + b(t).$$

INPUT:

- `self` - the power series $a(t)$
- `b` - the power series $b(t)$ (default is zero)
- `f0` - the constant term of f (“initial condition”) (default is 1)
- `prec` - desired precision of result (this will be reduced if either a or b have less precision available)

OUTPUT: the power series f, to indicated precision

ALGORITHM: A divide-and-conquer strategy; see the source code. Running time is approximately $M(n)\log n$, where $M(n)$ is the time required for a polynomial multiplication of length n over the coefficient ring. (If you’re working over something like $\mathbf{Q}$, running time analysis can be a little complicated because the coefficients tend to explode.)

Note:

- If the coefficient ring is a field of characteristic zero, then the solution will exist and is unique.
- For other coefficient rings, things are more complicated. A solution may not exist, and if it does it may not be unique. Generally, by the time the nth term has been computed, the algorithm will have attempted divisions by $n!$ in the coefficient ring. So if your coefficient ring has enough ‘precision’, and if your coefficient ring can perform divisions even when the answer is not unique, and if you know in advance that a solution exists, then this function will find a solution (otherwise it will probably crash).

AUTHORS:

- David Harvey (2006-09-11): factored functionality out from exp() function, cleaned up precision tests a bit

EXAMPLES:

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

```
sage: a = 2 - 3*t + 4*t^2 + O(t^10)
sage: b = 3 - 4*t^2 + O(t^7)
sage: f = a.solve_linear_de(prec=5, b=b, f0=3/5)
sage: f
 3/5 + 21/5*t + 33/10*t^2 - 38/15*t^3 + 11/24*t^4 + O(t^5)
sage: f.derivative() - a*f - b
 O(t^4)
```

```
sage: a = 2 - 3*t + 4*t^2
sage: b = b = 3 - 4*t^2
sage: f = a.solve_linear_de(b=b, f0=3/5)
Traceback (most recent call last):
...
ValueError: cannot solve differential equation to infinite precision
```

```
sage: a.solve_linear_de(prec=5, b=b, f0=3/5)
 3/5 + 21/5*t + 33/10*t^2 - 38/15*t^3 + 11/24*t^4 + O(t^5)
```

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

Return a square root of `self`.

INPUT:

- `prec` – integer (default: `None`): if not `None` and the series has infinite precision, truncates series at precision `prec`.
- `extend` – bool (default: `False`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square root is not in the base power series ring. For example, if `extend` is `True`, the square root of a power series with odd degree leading coefficient is defined as an element of a formal extension ring.
- `name` – string; if `extend` is `True`, you must also specify the print name of the formal square root.
- `all` – bool (default: `False`); if `True`, return all square roots of `self`, instead of just one.

ALGORITHM: Newton's method

$$x_{i+1} = \frac{1}{2}(x_i + \mathrm{self}/x_i)$$

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: sqrt(t^2)
t
sage: sqrt(1 + t)
1 + 1/2*t - 1/8*t^2 + 1/16*t^3 - 5/128*t^4 + O(t^5)
```

(continues on next page)

(continued from previous page)

```
sage: sqrt(4 + t)
2 + 1/4*t - 1/64*t^2 + 1/512*t^3 - 5/16384*t^4 + O(t^5)
sage: u = sqrt(2 + t, prec=2, extend=True, name = 'alpha'); u
alpha
sage: u^2
2 + t
sage: u.parent()
Univariate Quotient Polynomial Ring in alpha
 over Power Series Ring in t over Rational Field
 with modulus x^2 - 2 - t
sage: K.<t> = PowerSeriesRing(QQ, 't', 50)
sage: sqrt(1 + 2*t + t^2)
1 + t
sage: sqrt(t^2 + 2*t^4 + t^6)
t + t^3
sage: sqrt(1 + t + t^2 + 7*t^3)^2
1 + t + t^2 + 7*t^3 + O(t^50)
sage: sqrt(K(0))
0
sage: sqrt(t^2)
t
```

```
sage: # needs sage.rings.complex_double
sage: K.<t> = PowerSeriesRing(CDF, 5)
sage: v = sqrt(-1 + t + t^3, all=True); v
[1.0*I - 0.5*I*t - 0.125*I*t^2 - 0.5625*I*t^3 - 0.2890625*I*t^4 + O(t^5),
 -1.0*I + 0.5*I*t + 0.125*I*t^2 + 0.5625*I*t^3 + 0.2890625*I*t^4 + O(t^5)]
sage: [a^2 for a in v]
[-1.0 + 1.0*t + 0.0*t^2 + 1.0*t^3 + O(t^5), -1.0 + 1.0*t + 0.0*t^2 + 1.0*t^3 +␣
↪O(t^5)]
```

A formal square root:

```
sage: K.<t> = PowerSeriesRing(QQ, 5)
sage: f = 2*t + t^3 + O(t^4)
sage: s = f.sqrt(extend=True, name='sqrtf'); s
sqrtf
sage: s^2
2*t + t^3 + O(t^4)
sage: parent(s)
Univariate Quotient Polynomial Ring in sqrtf
 over Power Series Ring in t over Rational Field
 with modulus x^2 - 2*t - t^3 + O(t^4)
```

AUTHORS:

- Robert Bradshaw
- William Stein

square_root()

Return the square root of `self` in this ring. If this cannot be done, then an error will be raised.

This function succeeds if and only if `self`. *is_square()*

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: (1 + t).square_root()
1 + 1/2*t - 1/8*t^2 + 1/16*t^3 - 5/128*t^4 + O(t^5)
sage: (2 + t).square_root()
Traceback (most recent call last):
...
ValueError: Square root does not live in this ring.
sage: (2 + t.change_ring(RR)).square_root()                                    #␣
↪needs sage.rings.real_mpfr
1.41421356237309 + 0.353553390593274*t - 0.0441941738241592*t^2
 + 0.0110485434560398*t^3 - 0.00345266983001244*t^4 + O(t^5)
sage: t.square_root()
Traceback (most recent call last):
...
ValueError: Square root not defined for power series of odd valuation.
sage: K.<t> = PowerSeriesRing(ZZ, 't', 5)
sage: f = (1+t)^20
sage: f.square_root()
1 + 10*t + 45*t^2 + 120*t^3 + 210*t^4 + O(t^5)
sage: f = 1 + t
sage: f.square_root()
Traceback (most recent call last):
...
ValueError: Square root does not live in this ring.
```

AUTHORS:

- Robert Bradshaw

stieltjes_continued_fraction()

Return the Stieltjes continued fraction of `self`.

The S-fraction or Stieltjes continued fraction of a power series is a continued fraction expansion with steps of size one. We use the following convention

$$1/(1 - A_1 t/(1 - A_2 t/(1 - A_3 t/(1 - \cdots))))$$

OUTPUT:

A_n for $n \geq 1$

The expansion is done as long as possible given the precision. Whenever the expansion is not well-defined, because it would require to divide by zero, an exception is raised.

EXAMPLES:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: s = sum(catalan_number(k) * t**k for k in range(12)).O(12)
sage: s.stieltjes_continued_fraction()
(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

Another example:

```
sage: (exp(t)).stieltjes_continued_fraction()
(1,
```

(continues on next page)

(continued from previous page)

```
-1/2,
1/6,
-1/6,
1/10,
-1/10,
1/14,
-1/14,
1/18,
-1/18,
1/22,
-1/22,
1/26,
-1/26,
1/30,
-1/30,
1/34,
-1/34,
1/38)
```

tan(*prec='infinity'*)

Apply tan to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: tan(f)
t + t^2 + 1/3*t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: tan(f)
a + b + a*b + O(a, b)^3
sage: f.tan()
a + b + a*b + O(a, b)^3
sage: f.tan(prec=2)
a + b + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2 + f
sage: tan(g)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: can only apply tan to formal power series with zero constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: tan(a)
a + 1/3*a^3 + 2/15*a^5 + 17/315*a^7 + 62/2835*a^9 + 1382/155925*a^11 + O(a, b)^
↪12
sage: a.tan(prec=5)
a + 1/3*a^3 + O(a, b)^5
sage: tan(a + T.O(5))
a + 1/3*a^3 + O(a, b)^5
```

tanh(*prec='infinity'*)

Apply tanh to the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```
sage: t = PowerSeriesRing(QQ, 't').gen()
sage: f = (t + t**2).O(4)
sage: tanh(f)
t + t^2 - 1/3*t^3 + O(t^4)
```

For several variables:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: tanh(f)
a + b + a*b + O(a, b)^3
sage: f.tanh()
a + b + a*b + O(a, b)^3
sage: f.tanh(prec=2)
a + b + O(a, b)^2
```

If the power series has a non-zero constant coefficient c, one raises an error:

```
sage: g = 2 + f
sage: tanh(g)
Traceback (most recent call last):
...
ValueError: can only apply tanh to formal power series with zero
 constant term
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: tanh(a)
a - 1/3*a^3 + 2/15*a^5 - 17/315*a^7 + 62/2835*a^9 -
1382/155925*a^11 + O(a, b)^12
sage: a.tanh(prec=5)
a - 1/3*a^3 + O(a, b)^5
sage: tanh(a + T.O(5))
a - 1/3*a^3 + O(a, b)^5
```

truncate(*prec='infinity'*)

The polynomial obtained from power series by truncation.

EXAMPLES:

```
sage: R.<I> = GF(2)[[]]
sage: f = 1/(1+I+O(I^8)); f
1 + I + I^2 + I^3 + I^4 + I^5 + I^6 + I^7 + O(I^8)
sage: f.truncate(5)
I^4 + I^3 + I^2 + I + 1
```

valuation()

Return the valuation of this power series.

This is equal to the valuation of the underlying polynomial.

EXAMPLES:

Sparse examples:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^100000 + O(t^10000000)
sage: f.valuation()
100000
sage: R(0).valuation()
+Infinity
```

Dense examples:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: f = 17*t^100 + O(t^110)
sage: f.valuation()
100
sage: t.valuation()
1
```

valuation_zero_part()

Factor self as $q^n \cdot (a_0 + a_1 q + \cdots)$ with a_0 nonzero. Then this function returns $a_0 + a_1 q + \cdots$.

Note: This valuation zero part need not be a unit if, e.g., a_0 is not invertible in the base ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: ((1/3)*t^5*(17-2/3*t^3)).valuation_zero_part()
17/3 - 2/9*t^3
```

In this example the valuation 0 part is not a unit:

```
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: u = (-2*t^5*(17-t^3)).valuation_zero_part(); u
-34 + 2*t^3
sage: u.is_unit()
False
sage: u.valuation()
0
```

variable()

Return a string with the name of the variable of this power series.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(Rationals())
sage: f = x^2 + 3*x^4 + O(x^7)
sage: f.variable()
'x'
```

AUTHORS:

- David Harvey (2006-08-08)

sage.rings.power_series_ring_element.**is_PowerSeries**(*x*)

Return True if x is an instance of a univariate or multivariate power series.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: from sage.rings.power_series_ring_element import is_PowerSeries
sage: is_PowerSeries(1 + x^2)
True
sage: is_PowerSeries(x - x)
True
sage: is_PowerSeries(0)
False
sage: var('x')                                                           #␣
↪needs sage.symbolic
x
sage: is_PowerSeries(1 + x^2)                                            #␣
↪needs sage.symbolic
False
```

sage.rings.power_series_ring_element.**make_element_from_parent_v0**(*parent*, **args*)

sage.rings.power_series_ring_element.**make_powerseries_poly_v0**(*parent*, *f*, *prec*, *is_gen*)

CHAPTER

THREE

POWER SERIES METHODS

The class `PowerSeries_poly` provides additional methods for univariate power series.

class `sage.rings.power_series_poly.`**`BaseRingFloorDivAction`**

Bases: `Action`

The floor division action of the base ring on a formal power series.

class `sage.rings.power_series_poly.`**`PowerSeries_poly`**

Bases: *`PowerSeries`*

EXAMPLES:

```
sage: R.<q> = PowerSeriesRing(CC); R                                    # needs␣
↪sage.rings.real_mpfr
Power Series Ring in q over Complex Field with 53 bits of precision
sage: loads(q.dumps()) == q                                             # needs␣
↪sage.rings.real_mpfr
True

sage: R.<t> = QQ[[]]
sage: f = 3 - t^3 + O(t^5)
sage: a = f^3; a
27 - 27*t^3 + O(t^5)
sage: b = f^-3; b
1/27 + 1/27*t^3 + O(t^5)
sage: a*b
1 + O(t^5)
```

Check that github issue #22216 is fixed:

```
sage: R.<T> = PowerSeriesRing(QQ)
sage: R(pari('1 + O(T)'))                                               # needs␣
↪sage.libs.pari
1 + O(T)
sage: R(pari('1/T + O(T)'))                                             # needs␣
↪sage.libs.pari
Traceback (most recent call last):
...
ValueError: series has negative valuation
```

`degree`()

Return the degree of the underlying polynomial of `self`.

That is, if `self` is of the form $f(x) + O(x^n)$, we return the degree of $f(x)$. Note that if $f(x)$ is 0, we return -1, just as with polynomials.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (5 + t^3 + O(t^4)).degree()
3
sage: (5 + O(t^4)).degree()
0
sage: O(t^4).degree()
-1
```

dict()

Return a dictionary of coefficients for `self`.

This is simply a dict for the underlying polynomial, so need not have keys corresponding to every number smaller than `self.prec()`.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: f = 1 + t^10 + O(t^12)
sage: f.dict()
{0: 1, 10: 1}
```

integral(*var=None*)

Return the integral of this power series.

By default, the integration variable is the variable of the power series.

Otherwise, the integration variable is the optional parameter `var`.

Note: The integral is always chosen so the constant term is 0.

EXAMPLES:

```
sage: k.<w> = QQ[[]]
sage: (1+17*w+15*w^3+O(w^5)).integral()
w + 17/2*w^2 + 15/4*w^4 + O(w^6)
sage: (w^3 + 4*w^4 + O(w^7)).integral()
1/4*w^4 + 4/5*w^5 + O(w^8)
sage: (3*w^2).integral()
w^3
```

list()

Return the list of known coefficients for `self`.

This is just the list of coefficients of the underlying polynomial, so in particular, need not have length equal to `self.prec()`.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: f = 1 - 5*t^3 + t^5 + O(t^7)
```

(continues on next page)

(continued from previous page)

```
sage: f.list()
[1, 0, 0, -5, 0, 1]
```

pade(m, n)

Return the Padé approximant of `self` of index (m, n).

The Padé approximant of index (m, n) of a formal power series f is the quotient Q/P of two polynomials Q and P such that $\deg(Q) \leq m$, $\deg(P) \leq n$ and

$$f(z) - Q(z)/P(z) = O(z^{m+n+1}).$$

The formal power series f must be known up to order $n + m$.

See Wikipedia article Padé_approximant

INPUT:

- `m`, `n` – integers, describing the degrees of the polynomials

OUTPUT:

a ratio of two polynomials

ALGORITHM:

This method uses the formula as a quotient of two determinants.

See also:

- `sage.matrix.berlekamp_massey`,
- `sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint.rational_reconstruction()`

EXAMPLES:

```
sage: z = PowerSeriesRing(QQ, 'z').gen()
sage: exp(z).pade(4, 0)
1/24*z^4 + 1/6*z^3 + 1/2*z^2 + z + 1
sage: exp(z).pade(1, 1)
(-z - 2)/(z - 2)
sage: exp(z).pade(3, 3)
(-z^3 - 12*z^2 - 60*z - 120)/(z^3 - 12*z^2 + 60*z - 120)
sage: log(1-z).pade(4, 4)
(25/6*z^4 - 130/3*z^3 + 105*z^2 - 70*z)/(z^4 - 20*z^3 + 90*z^2
- 140*z + 70)
sage: sqrt(1+z).pade(3, 2)
(1/6*z^3 + 3*z^2 + 8*z + 16/3)/(z^2 + 16/3*z + 16/3)
sage: exp(2*z).pade(3, 3)
(-z^3 - 6*z^2 - 15*z - 15)/(z^3 - 6*z^2 + 15*z - 15)
```

polynomial()

Return the underlying polynomial of `self`.

EXAMPLES:

```
sage: R.<t> = GF(7)[[]]
sage: f = 3 - t^3 + O(t^5)
sage: f.polynomial()
6*t^3 + 3
```

reverse(*precision=None*)

Return the reverse of f, i.e., the series g such that $g(f(x)) = x$.

Given an optional argument `precision`, return the reverse with given precision (note that the reverse can have precision at most `f.prec()`). If f has infinite precision, and the argument `precision` is not given, then the precision of the reverse defaults to the default precision of `f.parent()`.

Note that this is only possible if the valuation of `self` is exactly 1.

ALGORITHM:

We first attempt to pass the computation to pari; if this fails, we use Lagrange inversion. Using `sage: set_verbose(1)` will print a message if passing to pari fails.

If the base ring has positive characteristic, then we attempt to lift to a characteristic zero ring and perform the reverse there. If this fails, an error is raised.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(QQ)
sage: f = 2*x + 3*x^2 - x^4 + O(x^5)
sage: g = f.reverse()
sage: g
1/2*x - 3/8*x^2 + 9/16*x^3 - 131/128*x^4 + O(x^5)
sage: f(g)
x + O(x^5)
sage: g(f)
x + O(x^5)

sage: A.<t> = PowerSeriesRing(ZZ)
sage: a = t - t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
t + t^2 + 2*t^3 + 7*t^4 + 25*t^5 + O(t^6)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)

sage: B.<b,c> = PolynomialRing(ZZ)
sage: A.<t> = PowerSeriesRing(B)
sage: f = t + b*t^2 + c*t^3 + O(t^4)
sage: g = f.reverse(); g
t - b*t^2 + (2*b^2 - c)*t^3 + O(t^4)
sage: f(g)
t + O(t^4)
sage: g(f)
t + O(t^4)

sage: A.<t> = PowerSeriesRing(ZZ)
sage: B.<s> = A[[]]
sage: f = (1 - 3*t + 4*t^3 + O(t^4))*s + (2 + t + t^2 + O(t^3))*s^2 + O(s^3)
```

(continues on next page)

(continued from previous page)

```
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(1)
sage: g = f.reverse(); g
verbose 1 (<module>) passing to pari failed; trying Lagrange inversion
(1 + 3*t + 9*t^2 + 23*t^3 + O(t^4))*s + (-2 - 19*t - 118*t^2 + O(t^3))*s^2 +␣
↪O(s^3)
sage: set_verbose(0)
sage: f(g) == g(f) == s
True
```

If the leading coefficient is not a unit, we pass to its fraction field if possible:

```
sage: A.<t> = PowerSeriesRing(ZZ)
sage: a = 2*t - 4*t^2 + t^4 - t^5 + O(t^6)
sage: a.reverse()
1/2*t + 1/2*t^2 + t^3 + 79/32*t^4 + 437/64*t^5 + O(t^6)

sage: B.<b> = PolynomialRing(ZZ)
sage: A.<t> = PowerSeriesRing(B)
sage: f = 2*b*t + b*t^2 + 3*b^2*t^3 + O(t^4)
sage: g = f.reverse(); g
1/(2*b)*t - 1/(8*b^2)*t^2 + ((-3*b + 1)/(16*b^3))*t^3 + O(t^4)
sage: f(g)
t + O(t^4)
sage: g(f)
t + O(t^4)
```

We can handle some base rings of positive characteristic:

```
sage: A8.<t> = PowerSeriesRing(Zmod(8))
sage: a = t - 15*t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
t + 7*t^2 + 2*t^3 + 5*t^4 + t^5 + O(t^6)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)
```

The optional argument `precision` sets the precision of the output:

```
sage: R.<x> = PowerSeriesRing(QQ)
sage: f = 2*x + 3*x^2 - 7*x^3 + x^4 + O(x^5)
sage: g = f.reverse(precision=3); g
1/2*x - 3/8*x^2 + O(x^3)
sage: f(g)
x + O(x^3)
sage: g(f)
x + O(x^3)
```

If the input series has infinite precision, the precision of the output is automatically set to the default precision of the parent ring:

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=20)
sage: (x - x^2).reverse() # get some Catalan numbers
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + 42*x^6 + 132*x^7 + 429*x^8 + 1430*x^9
 + 4862*x^10 + 16796*x^11 + 58786*x^12 + 208012*x^13 + 742900*x^14
 + 2674440*x^15 + 9694845*x^16 + 35357670*x^17 + 129644790*x^18
 + 477638700*x^19 + O(x^20)
sage: (x - x^2).reverse(precision=3)
x + x^2 + O(x^3)
```

truncate(*prec='infinity'*)

The polynomial obtained from power series by truncation at precision prec.

EXAMPLES:

```
sage: R.<I> = GF(2)[[]]
sage: f = 1/(1+I+O(I^8)); f
1 + I + I^2 + I^3 + I^4 + I^5 + I^6 + I^7 + O(I^8)
sage: f.truncate(5)
I^4 + I^3 + I^2 + I + 1
```

truncate_powerseries(*prec*)

Given input prec = n, returns the power series of degree $< n$ which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: R.<I> = GF(2)[[]]
sage: f = 1/(1+I+O(I^8)); f
1 + I + I^2 + I^3 + I^4 + I^5 + I^6 + I^7 + O(I^8)
sage: f.truncate_powerseries(5)
1 + I + I^2 + I^3 + I^4 + O(I^5)
```

valuation()

Return the valuation of self.

EXAMPLES:

```
sage: R.<t> = QQ[[]]
sage: (5 - t^8 + O(t^11)).valuation()
0
sage: (-t^8 + O(t^11)).valuation()
8
sage: O(t^7).valuation()
7
sage: R(0).valuation()
+Infinity
```

sage.rings.power_series_poly.**make_powerseries_poly_v0**(*parent*, *f*, *prec*, *is_gen*)

Return the power series specified by f, prec, and is_gen.

This function exists for the purposes of pickling. Do not delete this function – if you change the internal representation, instead make a new function and make sure that both kinds of objects correctly unpickle as the new type.

EXAMPLES:

```
sage: R.<t> = QQ[[]]
sage: sage.rings.power_series_poly.make_powerseries_poly_v0(R, t, infinity, True)
t
```

CHAPTER

FOUR

POWER SERIES IMPLEMENTED USING PARI

EXAMPLES:

This implementation can be selected for any base ring supported by PARI by passing the keyword `implementation='pari'` to the *PowerSeriesRing()* constructor:

```
sage: R.<q> = PowerSeriesRing(ZZ, implementation='pari'); R
Power Series Ring in q over Integer Ring
sage: S.<t> = PowerSeriesRing(CC, implementation='pari'); S                    #␣
↪needs sage.rings.real_mpfr
Power Series Ring in t over Complex Field with 53 bits of precision
```

Note that only the type of the elements depends on the implementation, not the type of the parents:

```
sage: type(R)
<class 'sage.rings.power_series_ring.PowerSeriesRing_domain_with_category'>
sage: type(q)
<class 'sage.rings.power_series_pari.PowerSeries_pari'>
sage: type(S)                                                                  #␣
↪needs sage.rings.real_mpfr
<class 'sage.rings.power_series_ring.PowerSeriesRing_over_field_with_category'>
sage: type(t)                                                                  #␣
↪needs sage.rings.real_mpfr
<class 'sage.rings.power_series_pari.PowerSeries_pari'>
```

If k is a finite field implemented using PARI, this is the default implementation for power series over k:

```
sage: k.<c> = GF(5^12)
sage: type(c)
<class 'sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt'>
sage: A.<x> = k[[]]
sage: type(x)
<class 'sage.rings.power_series_pari.PowerSeries_pari'>
```

Warning: Because this implementation uses the PARI interface, the PARI variable ordering must be respected in the sense that the variable name of the power series ring must have higher priority than any variable names occurring in the base ring:

```
sage: R.<y> = QQ[]
sage: S.<x> = PowerSeriesRing(R, implementation='pari'); S
Power Series Ring in x over Univariate Polynomial Ring in y over Rational Field
```

Reversing the variable ordering leads to errors:

```
sage: R.<x> = QQ[]
sage: S.<y> = PowerSeriesRing(R, implementation='pari')
Traceback (most recent call last):
...
PariError: incorrect priority in gtopoly: variable x <= y
```

AUTHORS:

- Peter Bruin (December 2013): initial version

class sage.rings.power_series_pari.**PowerSeries_pari**

Bases: *PowerSeries*

A power series implemented using PARI.

INPUT:

- `parent` – the power series ring to use as the parent
- `f` – object from which to construct a power series
- `prec` – (default: infinity) precision of the element to be constructed
- `check` – ignored, but accepted for compatibility with *PowerSeries_poly*

dict()

Return a dictionary of coefficients for `self`.

This is simply a dict for the underlying polynomial; it need not have keys corresponding to every number smaller than `self.prec()`.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ, implementation='pari')
sage: f = 1 + t^10 + O(t^12)
sage: f.dict()
{0: 1, 10: 1}
```

integral(*var=None*)

Return the formal integral of `self`.

By default, the integration variable is the variable of the power series. Otherwise, the integration variable is the optional parameter `var`.

Note: The integral is always chosen so the constant term is 0.

EXAMPLES:

```
sage: k.<w> = PowerSeriesRing(QQ, implementation='pari')
sage: (1+17*w+15*w^3+O(w^5)).integral()
w + 17/2*w^2 + 15/4*w^4 + O(w^6)
sage: (w^3 + 4*w^4 + O(w^7)).integral()
1/4*w^4 + 4/5*w^5 + O(w^8)
sage: (3*w^2).integral()
w^3
```

`list()`

Return the list of known coefficients for `self`.

This is just the list of coefficients of the underlying polynomial; it need not have length equal to `self.prec()`.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ, implementation='pari')
sage: f = 1 - 5*t^3 + t^5 + O(t^7)
sage: f.list()
[1, 0, 0, -5, 0, 1]

sage: # needs sage.rings.padics
sage: S.<u> = PowerSeriesRing(pAdicRing(5), implementation='pari')
sage: (2 + u).list()
[2 + O(5^20), 1 + O(5^20)]
```

`padded_list(n=None)`

Return a list of coefficients of `self` up to (but not including) q^n.

The list is padded with zeroes on the right so that it has length n.

INPUT:

- **n – a non-negative integer (optional); if n is not**
  given, it will be taken to be the precision of `self`` , unless this is ``+Infinity, in which case we just return `self.list()`

EXAMPLES:

```
sage: R.<q> = PowerSeriesRing(QQ, implementation='pari')
sage: f = 1 - 17*q + 13*q^2 + 10*q^4 + O(q^7)
sage: f.list()
[1, -17, 13, 0, 10]
sage: f.padded_list(7)
[1, -17, 13, 0, 10, 0, 0]
sage: f.padded_list(10)
[1, -17, 13, 0, 10, 0, 0, 0, 0, 0]
sage: f.padded_list(3)
[1, -17, 13]
sage: f.padded_list()
[1, -17, 13, 0, 10, 0, 0]
sage: g = 1 - 17*q + 13*q^2 + 10*q^4
sage: g.list()
[1, -17, 13, 0, 10]
sage: g.padded_list()
[1, -17, 13, 0, 10]
sage: g.padded_list(10)
[1, -17, 13, 0, 10, 0, 0, 0, 0, 0]
```

`polynomial()`

Convert `self` to a polynomial.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(GF(7), implementation='pari')
sage: f = 3 - t^3 + O(t^5)
sage: f.polynomial()
6*t^3 + 3
```

reverse(*precision=None*)

Return the reverse of `self`.

The reverse of a power series f is the power series g such that $g(f(x)) = x$. This exists if and only if the valuation of `self` is exactly 1 and the coefficient of x is a unit.

If the optional argument `precision` is given, the reverse is returned with this precision. If `f` has infinite precision and the argument `precision` is not given, then the reverse is returned with the default precision of `f.parent()`.

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(QQ, implementation='pari')
sage: f = 2*x + 3*x^2 - x^4 + O(x^5)
sage: g = f.reverse()
sage: g
1/2*x - 3/8*x^2 + 9/16*x^3 - 131/128*x^4 + O(x^5)
sage: f(g)
x + O(x^5)
sage: g(f)
x + O(x^5)

sage: A.<t> = PowerSeriesRing(ZZ, implementation='pari')
sage: a = t - t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
t + t^2 + 2*t^3 + 7*t^4 + 25*t^5 + O(t^6)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)

sage: B.<b,c> = PolynomialRing(ZZ)
sage: A.<t> = PowerSeriesRing(B, implementation='pari')
sage: f = t + b*t^2 + c*t^3 + O(t^4)
sage: g = f.reverse(); g
t - b*t^2 + (2*b^2 - c)*t^3 + O(t^4)
sage: f(g)
t + O(t^4)
sage: g(f)
t + O(t^4)

sage: A.<t> = PowerSeriesRing(ZZ, implementation='pari')
sage: B.<x> = PowerSeriesRing(A, implementation='pari')
sage: f = (1 - 3*t + 4*t^3 + O(t^4))*x + (2 + t + t^2 + O(t^3))*x^2 + O(x^3)
sage: g = f.reverse(); g
(1 + 3*t + 9*t^2 + 23*t^3 + O(t^4))*x + (-2 - 19*t - 118*t^2 + O(t^3))*x^2 +␣
↪O(x^3)
```

The optional argument `precision` sets the precision of the output:

```
sage: R.<x> = PowerSeriesRing(QQ, implementation='pari')
sage: f = 2*x + 3*x^2 - 7*x^3 + x^4 + O(x^5)
sage: g = f.reverse(precision=3); g
1/2*x - 3/8*x^2 + O(x^3)
sage: f(g)
x + O(x^3)
sage: g(f)
x + O(x^3)
```

If the input series has infinite precision, the precision of the output is automatically set to the default precision of the parent ring:

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=20, implementation='pari')
sage: (x - x^2).reverse()  # get some Catalan numbers
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + 42*x^6 + 132*x^7 + 429*x^8
 + 1430*x^9 + 4862*x^10 + 16796*x^11 + 58786*x^12 + 208012*x^13
 + 742900*x^14 + 2674440*x^15 + 9694845*x^16 + 35357670*x^17
 + 129644790*x^18 + 477638700*x^19 + O(x^20)
sage: (x - x^2).reverse(precision=3)
x + x^2 + O(x^3)
```

valuation()

Return the valuation of `self`.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, implementation='pari')
sage: (5 - t^8 + O(t^11)).valuation()
0
sage: (-t^8 + O(t^11)).valuation()
8
sage: O(t^7).valuation()
7
sage: R(0).valuation()
+Infinity
```

CHAPTER

FIVE

MULTIVARIATE POWER SERIES RINGS

Construct a multivariate power series ring (in finitely many variables) over a given (commutative) base ring.

EXAMPLES:

Construct rings and elements:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
sage: TestSuite(R).run()
sage: p = -t + 1/2*t^3*u - 1/4*t^4*u + 2/3*v^5 + R.O(6); p
-t + 1/2*t^3*u - 1/4*t^4*u + 2/3*v^5 + O(t, u, v)^6
sage: p in R
True

sage: g = 1 + v + 3*u*t^2 - 2*v^2*t^2; g
1 + v + 3*t^2*u - 2*t^2*v^2
sage: g in R
True
```

Add big O as with single variable power series:

```
sage: g.add_bigoh(3)
1 + v + O(t, u, v)^3
sage: g = g.O(5); g
1 + v + 3*t^2*u - 2*t^2*v^2 + O(t, u, v)^5
```

Sage keeps track of total-degree precision:

```
sage: f = (g-1)^2 - g + 1; f
-v + v^2 - 3*t^2*u + 6*t^2*u*v + 2*t^2*v^2 + O(t, u, v)^5
sage: f in R
True
sage: f.prec()
5
sage: ((g-1-v)^2).prec()
8
```

Construct multivariate power series rings over various base rings.

```
sage: M = PowerSeriesRing(QQ, 4, 'k'); M
Multivariate Power Series Ring in k0, k1, k2, k3 over Rational Field
sage: loads(dumps(M)) is M
```

(continues on next page)

(continued from previous page)

```
True
sage: TestSuite(M).run()

sage: H = PowerSeriesRing(PolynomialRing(ZZ, 3, 'z'), 4, 'f'); H
Multivariate Power Series Ring in f0, f1, f2, f3
 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring
sage: TestSuite(H).run()
sage: loads(dumps(H)) is H
True

sage: z = H.base_ring().gens()
sage: f = H.gens()
sage: h = 4*z[1]^2 + 2*z[0]*z[2] + z[1]*z[2] + z[2]^2 \
+ (-z[2]^2 - 2*z[0] + z[2])*f[0]*f[2] \
+ (-22*z[0]^2 + 2*z[1]^2 - z[0]*z[2] + z[2]^2 - 1955*z[2])*f[1]*f[2] \
+ (-z[0]*z[1] - 2*z[1]^2)*f[2]*f[3] \
+ (2*z[0]*z[1] + z[1]*z[2] - z[2]^2 - z[1] + 3*z[2])*f[3]^2 \
+ H.O(3)
sage: h in H
True
sage: h
4*z1^2 + 2*z0*z2 + z1*z2 + z2^2 + (-z2^2 - 2*z0 + z2)*f0*f2
+ (-22*z0^2 + 2*z1^2 - z0*z2 + z2^2 - 1955*z2)*f1*f2
+ (-z0*z1 - 2*z1^2)*f2*f3 + (2*z0*z1 + z1*z2 - z2^2 - z1 + 3*z2)*f3^2
+ O(f0, f1, f2, f3)^3
```

- Use angle-bracket notation:

```
sage: # needs sage.rings.finite_rings
sage: S.<x,y> = PowerSeriesRing(GF(65537)); S
Multivariate Power Series Ring in x, y over Finite Field of size 65537
sage: s = -30077*x + 9485*x*y - 6260*y^3 + 12870*x^2*y^2 - 20289*y^4 + S.O(5); s
-30077*x + 9485*x*y - 6260*y^3 + 12870*x^2*y^2 - 20289*y^4 + O(x, y)^5
sage: s in S
True
sage: TestSuite(S).run()
sage: loads(dumps(S)) is S
True
```

- Use double square bracket notation:

```
sage: ZZ[['s,t,u']]
Multivariate Power Series Ring in s, t, u over Integer Ring
sage: GF(127931)[['x,y']]
↪# needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size 127931
```

Variable ordering determines how series are displayed.

```
sage: T.<a,b> = PowerSeriesRing(ZZ,order='deglex'); T
Multivariate Power Series Ring in a, b over Integer Ring
sage: TestSuite(T).run()
sage: loads(dumps(T)) is T
```

(continues on next page)

(continued from previous page)

```
True
sage: T.term_order()
Degree lexicographic term order
sage: p = - 2*b^6 + a^5*b^2 + a^7 - b^2 - a*b^3 + T.O(9); p
a^7 + a^5*b^2 - 2*b^6 - a*b^3 - b^2 + O(a, b)^9

sage: U = PowerSeriesRing(ZZ,'a,b',order='negdeglex'); U
Multivariate Power Series Ring in a, b over Integer Ring
sage: U.term_order()
Negative degree lexicographic term order
sage: U(p)
-b^2 - a*b^3 - 2*b^6 + a^7 + a^5*b^2 + O(a, b)^9
```

Change from one base ring to another:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
sage: R.base_extend(RR)                                                        #␣
↪needs sage.rings.real_mpfr
Multivariate Power Series Ring in t, u, v
 over Real Field with 53 bits of precision
sage: R.change_ring(IntegerModRing(10))
Multivariate Power Series Ring in t, u, v
 over Ring of integers modulo 10

sage: S = PowerSeriesRing(GF(65537),2,'x,y'); S                                #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size 65537
sage: S.change_ring(GF(5))                                                     #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size 5
```

Coercion from polynomial ring:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
sage: A = PolynomialRing(ZZ,3,'t,u,v')
sage: g = A.gens()
sage: a = 2*g[0]*g[2] - 2*g[0] - 2; a
2*t*v - 2*t - 2
sage: R(a)
-2 - 2*t + 2*t*v
sage: R(a).O(4)
-2 - 2*t + 2*t*v + O(t, u, v)^4
sage: a.parent()
Multivariate Polynomial Ring in t, u, v over Integer Ring
sage: a in R
True
```

Coercion from polynomial ring in subset of variables:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: A = PolynomialRing(QQ,2,'t,v')
sage: g = A.gens()
sage: a = -2*g[0]*g[1] - 1/27*g[1]^2 + g[0] - 1/2*g[1]; a
-2*t*v - 1/27*v^2 + t - 1/2*v
sage: a in R
True
```

Coercion from symbolic ring:

```
sage: # needs sage.symbolic
sage: x,y = var('x,y')
sage: S = PowerSeriesRing(GF(11),2,'x,y'); S
Multivariate Power Series Ring in x, y over Finite Field of size 11
sage: type(x)
<class 'sage.symbolic.expression.Expression'>
sage: type(S(x))
<class 'sage.rings.multi_power_series_ring.MPowerSeriesRing_generic_with_category.
↪element_class'>
sage: f = S(2/7 -100*x^2 + 1/3*x*y + y^2).O(3); f
5 - x^2 + 4*x*y + y^2 + O(x, y)^3
sage: f.parent()
Multivariate Power Series Ring in x, y over Finite Field of size 11
sage: f.parent() == S
True
```

The implementation of the multivariate power series ring uses a combination of multivariate polynomials and univariate power series. Namely, in order to construct the multivariate power series ring $R[[x_1, x_2, \cdots, x_n]]$, we consider the univariate power series ring $S[[T]]$ over the multivariate polynomial ring $S := R[x_1, x_2, \cdots, x_n]$, and in it we take the subring formed by all power series whose i-th coefficient has degree i for all $i \geq 0$. This subring is isomorphic to $R[[x_1, x_2, \cdots, x_n]]$. This is how $R[[x_1, x_2, \cdots, x_n]]$ is implemented in this class. The ring S is called the foreground polynomial ring, and the ring $S[[T]]$ is called the background univariate power series ring.

AUTHORS:

- Niles Johnson (2010-07): initial code
- Simon King (2012-08, 2013-02): Use category and coercion framework, github issue #13412 and github issue #14084

class `sage.rings.multi_power_series_ring.`**`MPowerSeriesRing_generic`**(*base_ring*, *num_gens*, *name_list*, *order='negdeglex'*, *default_prec=10*, *sparse=False*)

Bases: *`PowerSeriesRing_generic`*, `Nonexact`

A multivariate power series ring. This class is implemented as a single variable power series ring in the variable `T` over a multivariable polynomial ring in the specified generators. Each generator `g` of the multivariable polynomial ring (called the "foreground ring") is mapped to `g*T` in the single variable power series ring (called the "background ring"). The background power series ring is used to do arithmetic and track total-degree precision. The foreground polynomial ring is used to display elements.

For usage and examples, see above, and *`PowerSeriesRing()`*.

`Element`

alias of *`MPowerSeries`*

`O`(*prec*)

Return big oh with precision `prec`. This function is an alias for `bigoh`.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2); T
Multivariate Power Series Ring in a, b over Integer Ring
sage: T.O(10)
0 + O(a, b)^10
sage: T.bigoh(10)
0 + O(a, b)^10
```

bigoh(*prec*)

Return big oh with precision `prec`. The function `O` does the same thing.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(ZZ,2); T
Multivariate Power Series Ring in a, b over Integer Ring
sage: T.bigoh(10)
0 + O(a, b)^10
sage: T.O(10)
0 + O(a, b)^10
```

change_ring(*R*)

Returns the power series ring over R in the same variable as self. This function ignores the question of whether the base ring of self is or can extend to the base ring of R; for the latter, use base_extend.

EXAMPLES:

```
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
sage: R.base_extend(RR)                                                    #␣
↪needs sage.rings.real_mpfr
Multivariate Power Series Ring in t, u, v over Real Field with
53 bits of precision
sage: R.change_ring(IntegerModRing(10))
Multivariate Power Series Ring in t, u, v over Ring of integers
modulo 10
sage: R.base_extend(IntegerModRing(10))
Traceback (most recent call last):
...
TypeError: no base extension defined


sage: S = PowerSeriesRing(GF(65537),2,'x,y'); S                            #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size
65537
sage: S.change_ring(GF(5))                                                 #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in x, y over Finite Field of size 5
```

characteristic()

Return characteristic of base ring, which is characteristic of self.

EXAMPLES:

```
sage: H = PowerSeriesRing(GF(65537),4,'f'); H                          #␣
↪needs sage.rings.finite_rings
Multivariate Power Series Ring in f0, f1, f2, f3 over
Finite Field of size 65537
sage: H.characteristic()                                                #␣
↪needs sage.rings.finite_rings
65537
```

construction()

Returns a functor F and base ring R such that F(R) == self.

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ, 4, 'f'); M
Multivariate Power Series Ring in f0, f1, f2, f3 over Rational Field

sage: (c,R) = M.construction(); (c,R)
(Completion[('f0', 'f1', 'f2', 'f3'), prec=12],
 Multivariate Polynomial Ring in f0, f1, f2, f3 over Rational Field)
sage: c
Completion[('f0', 'f1', 'f2', 'f3'), prec=12]
sage: c(R)
Multivariate Power Series Ring in f0, f1, f2, f3 over Rational Field
sage: c(R) == M
True
```

gen(*n=0*)

Return the nth generator of self.

EXAMPLES:

```
sage: M = PowerSeriesRing(ZZ, 10, 'v')
sage: M.gen(6)
v6
```

is_dense()

Is self dense? (opposite of sparse)

EXAMPLES:

```
sage: M = PowerSeriesRing(ZZ, 3, 's,t,u'); M
Multivariate Power Series Ring in s, t, u over Integer Ring
sage: M.is_dense()
True
sage: N = PowerSeriesRing(ZZ, 3, 's,t,u', sparse=True); N
Sparse Multivariate Power Series Ring in s, t, u over Integer Ring
sage: N.is_dense()
False
```

is_integral_domain(*proof=False*)

Return `True` if the base ring is an integral domain; otherwise return False.

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ,4,'v'); M
Multivariate Power Series Ring in v0, v1, v2, v3 over Rational Field
sage: M.is_integral_domain()
True
```

is_noetherian(*proof=False*)

Power series over a Noetherian ring are Noetherian.

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ,4,'v'); M
Multivariate Power Series Ring in v0, v1, v2, v3 over Rational Field
sage: M.is_noetherian()
True

sage: W = PowerSeriesRing(InfinitePolynomialRing(ZZ,'a'),2,'x,y')
sage: W.is_noetherian()
False
```

is_sparse()

Is self sparse?

EXAMPLES:

```
sage: M = PowerSeriesRing(ZZ, 3, 's,t,u'); M
Multivariate Power Series Ring in s, t, u over Integer Ring
sage: M.is_sparse()
False
sage: N = PowerSeriesRing(ZZ, 3, 's,t,u', sparse=True); N
Sparse Multivariate Power Series Ring in s, t, u over Integer Ring
sage: N.is_sparse()
True
```

laurent_series_ring()

Laurent series not yet implemented for multivariate power series rings

ngens()

Return number of generators of self.

EXAMPLES:

```
sage: M = PowerSeriesRing(ZZ, 10, 'v')
sage: M.ngens()
10
```

prec_ideal()

Return the ideal which determines precision; this is the ideal generated by all of the generators of our background polynomial ring.

EXAMPLES:

```
sage: A.<s,t,u> = PowerSeriesRing(ZZ)
sage: A.prec_ideal()
Ideal (s, t, u) of
 Multivariate Polynomial Ring in s, t, u over Integer Ring
```

remove_var(**var*)

Remove given variable or sequence of variables from self.

EXAMPLES:

```
sage: A.<s,t,u> = PowerSeriesRing(ZZ)
sage: A.remove_var(t)
Multivariate Power Series Ring in s, u over Integer Ring
sage: A.remove_var(s,t)
Power Series Ring in u over Integer Ring


sage: M = PowerSeriesRing(GF(5),5,'t'); M
Multivariate Power Series Ring in t0, t1, t2, t3, t4
 over Finite Field of size 5
sage: M.remove_var(M.gens()[3])
Multivariate Power Series Ring in t0, t1, t2, t4
 over Finite Field of size 5
```

Removing all variables results in the base ring:

```
sage: M.remove_var(*M.gens())
Finite Field of size 5
```

term_order()

Print term ordering of self. Term orderings are implemented by the TermOrder class.

EXAMPLES:

```
sage: M.<x,y,z> = PowerSeriesRing(ZZ,3)
sage: M.term_order()
Negative degree lexicographic term order
sage: m = y*z^12 - y^6*z^8 - x^7*y^5*z^2 + x*y^2*z + M.O(15); m
x*y^2*z + y*z^12 - x^7*y^5*z^2 - y^6*z^8 + O(x, y, z)^15

sage: N = PowerSeriesRing(ZZ,3,'x,y,z', order="deglex")
sage: N.term_order()
Degree lexicographic term order
sage: N(m)
-x^7*y^5*z^2 - y^6*z^8 + y*z^12 + x*y^2*z + O(x, y, z)^15
```

sage.rings.multi_power_series_ring.**is_MPowerSeriesRing**(*x*)

Return True if input is a multivariate power series ring.

sage.rings.multi_power_series_ring.**unpickle_multi_power_series_ring_v0**(*base_ring*, *num_gens*, *names*, *order*, *default_prec*, *sparse*)

Unpickle (deserialize) a multivariate power series ring according to the given inputs.

EXAMPLES:

```
sage: P.<x,y> = PowerSeriesRing(QQ)
sage: loads(dumps(P)) == P # indirect doctest
True
```

CHAPTER

SIX

MULTIVARIATE POWER SERIES

Construct and manipulate multivariate power series (in finitely many variables) over a given commutative ring. Multivariate power series are implemented with total-degree precision.

EXAMPLES:

Power series arithmetic, tracking precision:

```
sage: R.<s,t> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in s, t over Integer Ring

sage: f = 1 + s + 3*s^2; f
1 + s + 3*s^2
sage: g = t^2*s + 3*t^2*s^2 + R.O(5); g
s*t^2 + 3*s^2*t^2 + O(s, t)^5
sage: g = t^2*s + 3*t^2*s^2 + O(s, t)^5; g
s*t^2 + 3*s^2*t^2 + O(s, t)^5
sage: f = f.O(7); f
1 + s + 3*s^2 + O(s, t)^7
sage: f += s; f
1 + 2*s + 3*s^2 + O(s, t)^7
sage: f*g
s*t^2 + 5*s^2*t^2 + O(s, t)^5
sage: (f-1)*g
2*s^2*t^2 + 9*s^3*t^2 + O(s, t)^6
sage: f*g - g
2*s^2*t^2 + O(s, t)^5

sage: f *= s; f
s + 2*s^2 + 3*s^3 + O(s, t)^8
sage: f%2
s + s^3 + O(s, t)^8
sage: (f%2).parent()
Multivariate Power Series Ring in s, t over Ring of integers modulo 2
```

As with univariate power series, comparison of f and g is done up to the minimum precision of f and g:

```
sage: f = 1 + t + s + s*t + R.O(3); f
1 + s + t + s*t + O(s, t)^3
sage: g = s^2 + 2*s^4 - s^5 + s^2*t^3 + R.O(6); g
s^2 + 2*s^4 - s^5 + s^2*t^3 + O(s, t)^6
sage: f == g
False
```

(continues on next page)

(continued from previous page)

```
sage: g == g.add_bigoh(3)
True
sage: f < g
False
sage: f > g
True
```

Calling:

```
sage: f = s^2 + s*t + s^3 + s^2*t + 3*s^4 + 3*s^3*t + R.O(5); f
s^2 + s*t + s^3 + s^2*t + 3*s^4 + 3*s^3*t + O(s, t)^5
sage: f(t, s)
s*t + t^2 + s*t^2 + t^3 + 3*s*t^3 + 3*t^4 + O(s, t)^5
sage: f(t^2, s^2)
s^2*t^2 + t^4 + s^2*t^4 + t^6 + 3*s^2*t^6 + 3*t^8 + O(s, t)^10
```

Substitution is defined only for elements of positive valuation, unless f has infinite precision:

```
sage: f(t^2, s^2 + 1)
Traceback (most recent call last):
...
TypeError: Substitution defined only for elements of positive valuation,
unless self has infinite precision.

sage: g = f.truncate()
sage: g(t^2, s^2 + 1)
t^2 + s^2*t^2 + 2*t^4 + s^2*t^4 + 4*t^6 + 3*s^2*t^6 + 3*t^8
sage: g(t^2, (s^2+1).O(3))
t^2 + s^2*t^2 + 2*t^4 + O(s, t)^5
```

0 has valuation +`Infinity`:

```
sage: f(t^2, 0)
t^4 + t^6 + 3*t^8 + O(s, t)^10
sage: f(t^2, s^2 + s)
s*t^2 + s^2*t^2 + t^4 + O(s, t)^5
```

Substitution of power series with finite precision works too:

```
sage: f(s.O(2), t)
s^2 + s*t + O(s, t)^3
sage: f(f, f)
2*s^4 + 4*s^3*t + 2*s^2*t^2 + 4*s^5 + 8*s^4*t + 4*s^3*t^2 + 16*s^6 +
34*s^5*t + 20*s^4*t^2 + 2*s^3*t^3 + O(s, t)^7
sage: t(f, f)
s^2 + s*t + s^3 + s^2*t + 3*s^4 + 3*s^3*t + O(s, t)^5
sage: t(0, f) == s(f, 0)
True
```

The subs syntax works as expected:

```
sage: r0 = -t^2 - s*t^3 - 2*t^6 + s^7 + s^5*t^2 + R.O(10)
sage: r1 = s^4 - s*t^4 + s^6*t - 4*s^2*t^5 - 6*s^3*t^5 + R.O(10)
```

(continues on next page)

(continued from previous page)

```
sage: r2 = 2*s^3*t^2 - 2*s*t^4 - 2*s^3*t^4 + s*t^7 + R.O(10)
sage: r0.subs({t: r2, s: r1})
-4*s^6*t^4 + 8*s^4*t^6 - 4*s^2*t^8 + 8*s^6*t^6 - 8*s^4*t^8 - 4*s^4*t^9
+ 4*s^2*t^11 - 4*s^6*t^8 + O(s, t)^15
sage: r0.subs({t: r2, s: r1}) == r0(r1, r2)
True
```

Construct ring homomorphisms from one power series ring to another:

```
sage: A.<a,b> = PowerSeriesRing(QQ)
sage: X.<x,y> = PowerSeriesRing(QQ)

sage: phi = Hom(A,X)([x,2*y]); phi
Ring morphism:
  From: Multivariate Power Series Ring in a, b over Rational Field
  To:   Multivariate Power Series Ring in x, y over Rational Field
  Defn: a |--> x
        b |--> 2*y

sage: phi(a+b+3*a*b^2 + A.O(5))
x + 2*y + 12*x*y^2 + O(x, y)^5
```

Multiplicative inversion of power series:

```
sage: h = 1 + s + t + s*t + s^2*t^2 + 3*s^4 + 3*s^3*t + R.O(5)
sage: k = h^-1; k
1 - s - t + s^2 + s*t + t^2 - s^3 - s^2*t - s*t^2 - t^3 - 2*s^4 -
2*s^3*t + s*t^3 + t^4 + O(s, t)^5
sage: h*k
1 + O(s, t)^5

sage: f = 1 - 5*s^29 - 5*s^28*t + 4*s^18*t^35 + \
....: 4*s^17*t^36 - s^45*t^25 - s^44*t^26 + s^7*t^83 + \
....: s^6*t^84 + R.O(101)
sage: h = ~f; h
1 + 5*s^29 + 5*s^28*t - 4*s^18*t^35 - 4*s^17*t^36 + 25*s^58 + 50*s^57*t
+ 25*s^56*t^2 + s^45*t^25 + s^44*t^26 - 40*s^47*t^35 - 80*s^46*t^36
- 40*s^45*t^37 + 125*s^87 + 375*s^86*t + 375*s^85*t^2 + 125*s^84*t^3
- s^7*t^83 - s^6*t^84 + 10*s^74*t^25 + 20*s^73*t^26 + 10*s^72*t^27
+ O(s, t)^101
sage: h*f
1 + O(s, t)^101
```

AUTHORS:

- Niles Johnson (07/2010): initial code
- Simon King (08/2012): Use category and coercion framework, github issue #13412

class `sage.rings.multi_power_series_ring_element.MO(x)`

Bases: `object`

Object representing a zero element with given precision.

EXAMPLES:

```
sage: R.<u,v> = QQ[[]]
sage: m = O(u, v)
sage: m^4
0 + O(u, v)^4
sage: m^1
0 + O(u, v)^1

sage: T.<a,b,c> = PowerSeriesRing(ZZ, 3)
sage: z = O(a, b, c)
sage: z^1
0 + O(a, b, c)^1
sage: 1 + a + z^1
1 + O(a, b, c)^1

sage: w = 1 + a + O(a, b, c)^2; w
1 + a + O(a, b, c)^2
sage: w^2
1 + 2*a + O(a, b, c)^2
```

class sage.rings.multi_power_series_ring_element.**MPowerSeries**(*parent*, *x=0*, *prec=+Infinity*, *is_gen=False*, *check=False*)

Bases: *PowerSeries*

Multivariate power series; these are the elements of Multivariate Power Series Rings.

INPUT:

- parent – A multivariate power series.
- x – The element (default: 0). This can be another *MPowerSeries* object, or an element of one of the following:
  - the background univariate power series ring
  - the foreground polynomial ring
  - a ring that coerces to one of the above two
- prec – (default: infinity) The precision
- is_gen – (default: False) Is this element one of the generators?
- check – (default: False) Needed by univariate power series class

EXAMPLES:

Construct multivariate power series from generators:

```
sage: S.<s,t> = PowerSeriesRing(ZZ)
sage: f = s + 4*t + 3*s*t
sage: f in S
True
sage: f = f.add_bigoh(4); f
s + 4*t + 3*s*t + O(s, t)^4
sage: g = 1 + s + t - s*t + S.O(5); g
1 + s + t - s*t + O(s, t)^5

sage: T = PowerSeriesRing(GF(3),5,'t'); T
```

(continues on next page)

(continued from previous page)

```
Multivariate Power Series Ring in t0, t1, t2, t3, t4
 over Finite Field of size 3
sage: t = T.gens()
sage: w = t[0] - 2*t[1]*t[3] + 5*t[4]^3 - t[0]^3*t[2]^2; w
t0 + t1*t3 - t4^3 - t0^3*t2^2
sage: w = w.add_bigoh(5); w
t0 + t1*t3 - t4^3 + O(t0, t1, t2, t3, t4)^5
sage: w in T
True

sage: w = t[0] - 2*t[0]*t[2] + 5*t[4]^3 - t[0]^3*t[2]^2 + T.O(6)
sage: w
t0 + t0*t2 - t4^3 - t0^3*t2^2 + O(t0, t1, t2, t3, t4)^6
```

Get random elements:

```
sage: S.random_element(4)   # random
-2*t + t^2 - 12*s^3 + O(s, t)^4

sage: T.random_element(10)  # random
-t1^2*t3^2*t4^2 + t1^5*t3^3*t4 + O(t0, t1, t2, t3, t4)^10
```

Convert elements from polynomial rings:

```
sage: # needs sage.rings.finite_rings
sage: R = PolynomialRing(ZZ, 5, T.variable_names())
sage: t = R.gens()
sage: r = -t[2]*t[3] + t[3]^2 + t[4]^2
sage: T(r)
-t2*t3 + t3^2 + t4^2
sage: r.parent()
Multivariate Polynomial Ring in t0, t1, t2, t3, t4 over Integer Ring
sage: r in T
True
```

O(*prec*)

Return a multivariate power series of precision `prec` obtained by truncating `self` at precision `prec`.

This is the same as *add_bigoh()*.

EXAMPLES:

```
sage: B.<x,y> = PowerSeriesRing(QQ); B
Multivariate Power Series Ring in x, y over Rational Field
sage: r = 1 - x*y + x^2
sage: r.O(4)
1 + x^2 - x*y + O(x, y)^4
sage: r.O(2)
1 + O(x, y)^2
```

Note that this does not change `self`:

```
sage: r
1 + x^2 - x*y
```

V(*n*)

If

$$f = \sum a_{m_0,\ldots,m_k} x_0^{m_0} \cdots x_k^{m_k},$$

then this function returns

$$\sum a_{m_0,\ldots,m_k} x_0^{n m_0} \cdots x_k^{n m_k}.$$

The total-degree precision of the output is `n` times the precision of `self`.

EXAMPLES:

```
sage: H = QQ[['x,y,z']]
sage: (x,y,z) = H.gens()
sage: h = -x*y^4*z^7 - 1/4*y*z^12 + 1/2*x^7*y^5*z^2 \
+ 2/3*y^6*z^8 + H.O(15)
sage: h.V(3)
-x^3*y^12*z^21 - 1/4*y^3*z^36 + 1/2*x^21*y^15*z^6 + 2/3*y^18*z^24 + O(x, y, z)^
↪45
```

add_bigoh(*prec*)

Return a multivariate power series of precision `prec` obtained by truncating `self` at precision `prec`.

This is the same as *O()*.

EXAMPLES:

```
sage: B.<x,y> = PowerSeriesRing(QQ); B
Multivariate Power Series Ring in x, y over Rational Field
sage: r = 1 - x*y + x^2
sage: r.add_bigoh(4)
1 + x^2 - x*y + O(x, y)^4
sage: r.add_bigoh(2)
1 + O(x, y)^2
```

Note that this does not change `self`:

```
sage: r
1 + x^2 - x*y
```

coefficients()

Return a dict of monomials and coefficients.

EXAMPLES:

```
sage: R.<s,t> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in s, t over Integer Ring
sage: f = 1 + t + s + s*t + R.O(3)
sage: f.coefficients()
{s*t: 1, t: 1, s: 1, 1: 1}
sage: (f^2).coefficients()
{t^2: 1, s*t: 4, s^2: 1, t: 2, s: 2, 1: 1}

sage: g = f^2 + f - 2; g
3*s + 3*t + s^2 + 5*s*t + t^2 + O(s, t)^3
```

(continues on next page)

(continued from previous page)

```
sage: cd = g.coefficients()
sage: g2 = sum(k*v for (k,v) in cd.items()); g2
3*s + 3*t + s^2 + 5*s*t + t^2
sage: g2 == g.truncate()
True
```

constant_coefficient()

Return constant coefficient of `self`.

EXAMPLES:

```
sage: R.<a,b,c> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in a, b, c over Integer Ring
sage: f = 3 + a + b - a*b - b*c - a*c + R.O(4)
sage: f.constant_coefficient()
3
sage: f.constant_coefficient().parent()
Integer Ring
```

degree()

Return degree of underlying polynomial of `self`.

EXAMPLES:

```
sage: B.<x,y> = PowerSeriesRing(QQ)
sage: B
Multivariate Power Series Ring in x, y over Rational Field
sage: r = 1 - x*y + x^2
sage: r = r.add_bigoh(4); r
1 + x^2 - x*y + O(x, y)^4
sage: r.degree()
2
```

derivative(**args*)

The formal derivative of this power series, with respect to variables supplied in `args`.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(ZZ, 2)
sage: f = a + b + a^2*b + T.O(5)
sage: f.derivative(a)
1 + 2*a*b + O(a, b)^4
sage: f.derivative(a,2)
2*b + O(a, b)^3
sage: f.derivative(a,a)
2*b + O(a, b)^3
sage: f.derivative([a,a])
2*b + O(a, b)^3
sage: f.derivative(a,5)
0 + O(a, b)^0
sage: f.derivative(a,6)
0 + O(a, b)^0
```

dict()

Return underlying dictionary with keys the exponents and values the coefficients of this power series.

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ,4,'t',sparse=True); M
Sparse Multivariate Power Series Ring in t0, t1, t2, t3 over
Rational Field

sage: M.inject_variables()
Defining t0, t1, t2, t3

sage: m = 2/3*t0*t1^15*t3^48 - t0^15*t1^21*t2^28*t3^5
sage: m2 = 1/2*t0^12*t1^29*t2^46*t3^6 - 1/4*t0^39*t1^5*t2^23*t3^30 + M.O(100)
sage: s = m + m2
sage: s.dict()
{(1, 15, 0, 48): 2/3,
 (12, 29, 46, 6): 1/2,
 (15, 21, 28, 5): -1,
 (39, 5, 23, 30): -1/4}
```

egf()

Method from univariate power series not yet implemented

exp(*prec=+Infinity*)

Exponentiate the formal power series.

INPUT:

- `prec` – Integer or `infinity`. The degree to truncate the result to.

OUTPUT:

The exponentiated multivariate power series as a new multivariate power series.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(ZZ, 2)
sage: f = a + b + a*b + T.O(3)
sage: exp(f)
1 + a + b + 1/2*a^2 + 2*a*b + 1/2*b^2 + O(a, b)^3
sage: f.exp()
1 + a + b + 1/2*a^2 + 2*a*b + 1/2*b^2 + O(a, b)^3
sage: f.exp(prec=2)
1 + a + b + O(a, b)^2
sage: log(exp(f)) - f
0 + O(a, b)^3
```

If the power series has a constant coefficient c and $\exp(c)$ is transcendental, then $\exp(f)$ would have to be a power series over the `SymbolicRing`. These are not yet implemented and therefore such cases raise an error:

```
sage: g = 2 + f
sage: exp(g)                                                              #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: unsupported operand parent(s) for *: 'Symbolic Ring' and
'Power Series Ring in Tbg over Multivariate Polynomial Ring in a, b
over Rational Field'
```

Another workaround for this limitation is to change base ring to one which is closed under exponentiation, such as $\mathbf{R}$ or $\mathbf{C}$:

```
sage: exp(g.change_ring(RDF))
7.38905609... + 7.38905609...*a + 7.38905609...*b + 3.69452804...*a^2 +
14.7781121...*a*b + 3.69452804...*b^2 + O(a, b)^3
```

If no precision is specified, the default precision is used:

```
sage: T.default_prec()
12
sage: exp(a)
1 + a + 1/2*a^2 + 1/6*a^3 + 1/24*a^4 + 1/120*a^5 + 1/720*a^6 + 1/5040*a^7 +
1/40320*a^8 + 1/362880*a^9 + 1/3628800*a^10 + 1/39916800*a^11 + O(a, b)^12
sage: a.exp(prec=5)
1 + a + 1/2*a^2 + 1/6*a^3 + 1/24*a^4 + O(a, b)^5
sage: exp(a + T.O(5))
1 + a + 1/2*a^2 + 1/6*a^3 + 1/24*a^4 + O(a, b)^5
```

exponents()

Return a list of tuples which hold the exponents of each monomial of `self`.

EXAMPLES:

```
sage: H = QQ[['x,y']]
sage: (x,y) = H.gens()
sage: h = -y^2 - x*y^3 - 6/5*y^6 - x^7 + 2*x^5*y^2 + H.O(10)
sage: h
-y^2 - x*y^3 - 6/5*y^6 - x^7 + 2*x^5*y^2 + O(x, y)^10
sage: h.exponents()
[(0, 2), (1, 3), (0, 6), (7, 0), (5, 2)]
```

integral(**args*)

The formal integral of this multivariate power series, with respect to variables supplied in `args`.

The variable sequence `args` can contain both variables and counts; for the syntax, see `derivative_parse()`.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(QQ, 2)
sage: f = a + b + a^2*b + T.O(5)
sage: f.integral(a, 2)
1/6*a^3 + 1/2*a^2*b + 1/12*a^4*b + O(a, b)^7
sage: f.integral(a, b)
1/2*a^2*b + 1/2*a*b^2 + 1/6*a^3*b^2 + O(a, b)^7
sage: f.integral(a, 5)
1/720*a^6 + 1/120*a^5*b + 1/2520*a^7*b + O(a, b)^10
```

Only integration with respect to variables works:

```
sage: f.integral(a + b)
Traceback (most recent call last):
...
ValueError: a + b is not a variable
```

Warning: Coefficient division.

If the base ring is not a field (e.g. ZZ), or if it has a non-zero characteristic, (e.g. $ZZ/3ZZ$), integration is not always possible while staying with the same base ring. In the first case, Sage will report that it has not been able to coerce some coefficient to the base ring:

```
sage: T.<a,b> = PowerSeriesRing(ZZ, 2)
sage: f = a + T.O(5)
sage: f.integral(a)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

One can get the correct result by changing the base ring first:

```
sage: f.change_ring(QQ).integral(a)
1/2*a^2 + O(a, b)^6
```

However, a correct result is returned even without base change if the denominator cancels:

```
sage: f = 2*b + T.O(5)
sage: f.integral(b)
b^2 + O(a, b)^6
```

In non-zero characteristic, Sage will report that a zero division occurred

```
sage: T.<a,b> = PowerSeriesRing(Zmod(3), 2)
sage: (a^3).integral(a)
a^4
sage: (a^2).integral(a)
Traceback (most recent call last):
...
ZeroDivisionError: inverse of Mod(0, 3) does not exist
```

is_nilpotent()

Return `True` if `self` is nilpotent. This occurs if

- `self` has finite precision and positive valuation, or
- `self` is constant and nilpotent in base ring.

Otherwise, return `False`.

Warning: This is so far just a sufficient condition, so don't trust a `False` output to be legit!

Todo: What should we do about this method? Is nilpotency of a power series even decidable (assuming a nilpotency oracle in the base ring)? And I am not sure that returning `True` just because the series has finite precision and zero constant term is a good idea.

EXAMPLES:

```
sage: R.<a,b,c> = PowerSeriesRing(Zmod(8)); R
Multivariate Power Series Ring in a, b, c over Ring of integers modulo 8
sage: f = a + b + c + a^2*c
sage: f.is_nilpotent()
False
sage: f = f.O(4); f
a + b + c + a^2*c + O(a, b, c)^4
sage: f.is_nilpotent()
True

sage: g = R(2)
sage: g.is_nilpotent()
True
sage: (g.O(4)).is_nilpotent()
True

sage: S = R.change_ring(QQ)
sage: S(g).is_nilpotent()
False
sage: S(g.O(4)).is_nilpotent()
False
```

is_square()

Method from univariate power series not yet implemented.

is_unit()

A multivariate power series is a unit if and only if its constant coefficient is a unit.

EXAMPLES:

```
sage: R.<a,b> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in a, b over Integer Ring
sage: f = 2 + a^2 + a*b + a^3 + R.O(9)
sage: f.is_unit()
False
sage: f.base_extend(QQ).is_unit()
True
sage: (O(a,b)^0).is_unit()
False
```

laurent_series()

Not implemented for multivariate power series.

list()

Doesn't make sense for multivariate power series. Multivariate polynomials don't have list of coefficients either.

log(*prec=+Infinity*)

Return the logarithm of the formal power series.

INPUT:

- prec – Integer or infinity. The degree to truncate the result to.

OUTPUT:

The logarithm of the multivariate power series as a new multivariate power series.

EXAMPLES:

```
sage: T.<a,b> = PowerSeriesRing(ZZ, 2)
sage: f = 1 + a + b + a*b + T.O(5)
sage: f.log()
a + b - 1/2*a^2 - 1/2*b^2 + 1/3*a^3 + 1/3*b^3 - 1/4*a^4 - 1/4*b^4 + O(a, b)^5
sage: log(f)
a + b - 1/2*a^2 - 1/2*b^2 + 1/3*a^3 + 1/3*b^3 - 1/4*a^4 - 1/4*b^4 + O(a, b)^5
sage: exp(log(f)) - f
0 + O(a, b)^5
```

If the power series has a constant coefficient c and $\exp(c)$ is transcendental, then $\exp(f)$ would have to be a power series over the SymbolicRing. These are not yet implemented and therefore such cases raise an error:

```
sage: g = 2 + f
sage: log(g)                                                                    #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for -: 'Symbolic Ring' and 'Power
Series Ring in Tbg over Multivariate Polynomial Ring in a, b over Rational Field
↪'
```

Another workaround for this limitation is to change base ring to one which is closed under exponentiation, such as $\mathbf{R}$ or $\mathbf{C}$:

```
sage: log(g.change_ring(RDF))
1.09861228... + 0.333333333...*a + 0.333333333...*b - 0.0555555555...*a^2
+ 0.222222222...*a*b - 0.0555555555...*b^2 + 0.0123456790...*a^3
- 0.0740740740...*a^2*b - 0.0740740740...*a*b^2 + 0.0123456790...*b^3
- 0.00308641975...*a^4 + 0.0246913580...*a^3*b + 0.0246913580...*a*b^3
- 0.00308641975...*b^4 + O(a, b)^5
```

monomials()

Return a list of monomials of self.

These are the keys of the dict returned by *coefficients()*.

EXAMPLES:

```
sage: R.<a,b,c> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in a, b, c over Integer Ring
sage: f = 1 + a + b - a*b - b*c - a*c + R.O(4)
sage: sorted(f.monomials())
[b*c, a*c, a*b, b, a, 1]
sage: f = 1 + 2*a + 7*b - 2*a*b - 4*b*c - 13*a*c + R.O(4)
sage: sorted(f.monomials())
[b*c, a*c, a*b, b, a, 1]
sage: f = R.zero()
```

(continues on next page)

(continued from previous page)

```
sage: f.monomials()
[]
```

ogf()

Method from univariate power series not yet implemented

padded_list()

Method from univariate power series not yet implemented.

polynomial()

Return the underlying polynomial of `self` as an element of the underlying multivariate polynomial ring (the "foreground polynomial ring").

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ,4,'t'); M
Multivariate Power Series Ring in t0, t1, t2, t3 over Rational
Field
sage: t = M.gens()
sage: f = 1/2*t[0]^3*t[1]^3*t[2]^2 + 2/3*t[0]*t[2]^6*t[3]             - t[0]^
↪3*t[1]^3*t[3]^3 - 1/4*t[0]*t[1]*t[2]^7 + M.O(10)
sage: f
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3
- 1/4*t0*t1*t2^7 + O(t0, t1, t2, t3)^10

sage: f.polynomial()
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3
- 1/4*t0*t1*t2^7

sage: f.polynomial().parent()
Multivariate Polynomial Ring in t0, t1, t2, t3 over Rational Field
```

Contrast with *truncate()*:

```
sage: f.truncate()
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3 - 1/4*t0*t1*t2^7
sage: f.truncate().parent()
Multivariate Power Series Ring in t0, t1, t2, t3 over Rational Field
```

prec()

Return precision of `self`.

EXAMPLES:

```
sage: R.<a,b,c> = PowerSeriesRing(ZZ); R
Multivariate Power Series Ring in a, b, c over Integer Ring
sage: f = 3 + a + b - a*b - b*c - a*c + R.O(4)
sage: f.prec()
4
sage: f.truncate().prec()
+Infinity
```

quo_rem(*other*, *precision=None*)

Return the pair of quotient and remainder for the increasing power division of `self` by `other`.

If a and b are two elements of a power series ring $R[[x_1, x_2, \cdots, x_n]]$ such that the trailing term of b is invertible in R, then the pair of quotient and remainder for the increasing power division of a by b is the unique pair $(u, v) \in R[[x_1, x_2, \cdots, x_n]] \times R[x_1, x_2, \cdots, x_n]$ such that $a = bu + v$ and such that no monomial appearing in v divides the trailing monomial (`trailing_monomial()`) of b. Note that this depends on the order of the variables.

This method returns both quotient and remainder as power series, even though in mathematics, the remainder for the increasing power division of two power series is a polynomial. This is because Sage's power series come with a precision, and that precision is not always sufficient to determine the remainder completely. Disregarding this issue, the `polynomial()` method can be used to recast the remainder as an actual polynomial.

INPUT:

- `other` – an element of the same power series ring as `self` such that the trailing term of `other` is invertible in `self` (this is automatically satisfied if the base ring is a field, unless `other` is zero)
- `precision` – (default: the default precision of the parent of `self`) nonnegative integer, determining the precision to be cast on the resulting quotient and remainder if both `self` and `other` have infinite precision (ignored otherwise); note that the resulting precision might be lower than this integer

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: R.<a,b,c> = PowerSeriesRing(ZZ)
sage: f = 1 + a + b - a*b + R.O(3)
sage: g = 1 + 2*a - 3*a*b + R.O(3)
sage: q, r = f.quo_rem(g); q, r
(1 - a + b + 2*a^2 + O(a, b, c)^3, 0 + O(a, b, c)^3)
sage: f == q*g + r
True
sage: q, r = (a*f).quo_rem(g); q, r
(a - a^2 + a*b + 2*a^3 + O(a, b, c)^4, 0 + O(a, b, c)^4)
sage: a*f == q*g + r
True
sage: q, r = (a*f).quo_rem(a*g); q, r
(1 - a + b + 2*a^2 + O(a, b, c)^3, 0 + O(a, b, c)^4)
sage: a*f == q*(a*g) + r
True
sage: q, r = (a*f).quo_rem(b*g); q, r
(a - 3*a^2 + O(a, b, c)^3, a + a^2 + O(a, b, c)^4)
sage: a*f == q*(b*g) + r
True
```

Trying to divide two polynomials, we run into the issue that there is no natural setting for the precision of the quotient and remainder (and if we wouldn't set a precision, the algorithm would never terminate). Here, default precision comes to our help:

```
sage: # needs sage.libs.singular
sage: (1 + a^3).quo_rem(a + a^2)
(a^2 - a^3 + a^4 - a^5 + a^6 - a^7 + a^8 - a^9 + a^10 + O(a, b, c)^11,
 1 + O(a, b, c)^12)
sage: (1 + a^3 + a*b).quo_rem(b + c)
(a + O(a, b, c)^11, 1 - a*c + a^3 + O(a, b, c)^12)
sage: (1 + a^3 + a*b).quo_rem(b + c, precision=17)
(a + O(a, b, c)^16, 1 - a*c + a^3 + O(a, b, c)^17)
```

(continues on next page)

(continued from previous page)

```
sage: (a^2 + b^2 + c^2).quo_rem(a + b + c)
(a - b - c + O(a, b, c)^11, 2*b^2 + 2*b*c + 2*c^2 + O(a, b, c)^12)
sage: (a^2 + b^2 + c^2).quo_rem(1/(1+a+b+c))
(a^2 + b^2 + c^2 + a^3 + a^2*b + a^2*c + a*b^2 + a*c^2
   + b^3 + b^2*c + b*c^2 + c^3 + O(a, b, c)^14,
 0)
sage: (a^2 + b^2 + c^2).quo_rem(a/(1+a+b+c))
(a + a^2 + a*b + a*c + O(a, b, c)^13, b^2 + c^2)
sage: (1 + a + a^15).quo_rem(a^2)
(0 + O(a, b, c)^10, 1 + a + O(a, b, c)^12)
sage: (1 + a + a^15).quo_rem(a^2, precision=15)
(0 + O(a, b, c)^13, 1 + a + O(a, b, c)^15)
sage: (1 + a + a^15).quo_rem(a^2, precision=16)
(a^13 + O(a, b, c)^14, 1 + a + O(a, b, c)^16)
```

Illustrating the dependency on the ordering of variables:

```
sage: # needs sage.libs.singular
sage: (1 + a + b).quo_rem(b + c)
(1 + O(a, b, c)^11, 1 + a - c + O(a, b, c)^12)
sage: (1 + b + c).quo_rem(c + a)
(0 + O(a, b, c)^11, 1 + b + c + O(a, b, c)^12)
sage: (1 + c + a).quo_rem(a + b)
(1 + O(a, b, c)^11, 1 - b + c + O(a, b, c)^12)
```

shift(*n*)

Doesn't make sense for multivariate power series.

solve_linear_de(*prec=+Infinity*, *b=None*, *f0=None*)

Not implemented for multivariate power series.

sqrt()

Method from univariate power series not yet implemented. Depends on square root method for multivariate polynomials.

square_root()

Method from univariate power series not yet implemented. Depends on square root method for multivariate polynomials.

trailing_monomial()

Return the trailing monomial of `self`.

This is defined here as the lowest term of the underlying polynomial.

EXAMPLES:

```
sage: R.<a,b,c> = PowerSeriesRing(ZZ)
sage: f = 1 + a + b - a*b + R.O(3)
sage: f.trailing_monomial()
1
sage: f = a^2*b^3*f; f
a^2*b^3 + a^3*b^3 + a^2*b^4 - a^3*b^4 + O(a, b, c)^8
sage: f.trailing_monomial()
a^2*b^3
```

truncate(*prec=+Infinity*)

Return infinite precision multivariate power series formed by truncating `self` at precision `prec`.

EXAMPLES:

```
sage: M = PowerSeriesRing(QQ,4,'t'); M
Multivariate Power Series Ring in t0, t1, t2, t3 over Rational Field
sage: t = M.gens()
sage: f = 1/2*t[0]^3*t[1]^3*t[2]^2 + 2/3*t[0]*t[2]^6*t[3]             - t[0]^
↪3*t[1]^3*t[3]^3 - 1/4*t[0]*t[1]*t[2]^7 + M.O(10)
sage: f
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3
- 1/4*t0*t1*t2^7 + O(t0, t1, t2, t3)^10

sage: f.truncate()
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3
- 1/4*t0*t1*t2^7
sage: f.truncate().parent()
Multivariate Power Series Ring in t0, t1, t2, t3 over Rational Field
```

Contrast with polynomial:

```
sage: f.polynomial()
1/2*t0^3*t1^3*t2^2 + 2/3*t0*t2^6*t3 - t0^3*t1^3*t3^3 - 1/4*t0*t1*t2^7
sage: f.polynomial().parent()
Multivariate Polynomial Ring in t0, t1, t2, t3 over Rational Field
```

valuation()

Return the valuation of `self`.

The valuation of a power series f is the highest nonnegative integer k less or equal to the precision of f and such that the coefficient of f before each term of degree $< k$ is zero. (If such an integer does not exist, then the valuation is the precision of f itself.)

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: R.<a,b> = PowerSeriesRing(GF(4949717)); R
Multivariate Power Series Ring in a, b
 over Finite Field of size 4949717
sage: f = a^2 + a*b + a^3 + R.O(9)
sage: f.valuation()
2
sage: g = 1 + a + a^3
sage: g.valuation()
0
sage: R.zero().valuation()
+Infinity
```

valuation_zero_part()

Doesn't make sense for multivariate power series; valuation zero with respect to which variable?

variable()

Doesn't make sense for multivariate power series.

variables()

Return tuple of variables occurring in `self`.

EXAMPLES:

```
sage: T = PowerSeriesRing(GF(3),5,'t'); T
Multivariate Power Series Ring in t0, t1, t2, t3, t4 over
Finite Field of size 3
sage: t = T.gens()
sage: w = t[0] - 2*t[0]*t[2] + 5*t[4]^3 - t[0]^3*t[2]^2 + T.O(6)
sage: w
t0 + t0*t2 - t4^3 - t0^3*t2^2 + O(t0, t1, t2, t3, t4)^6
sage: w.variables()
(t0, t2, t4)
```

`sage.rings.multi_power_series_ring_element.`**is_MPowerSeries**(*f*)

Return `True` if `f` is a multivariate power series.

CHAPTER

SEVEN

LAURENT SERIES RINGS

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.base_ring()
Rational Field
sage: S = LaurentSeriesRing(GF(17)['x'], 'y')
sage: S
Laurent Series Ring in y over
 Univariate Polynomial Ring in x over Finite Field of size 17
sage: S.base_ring()
Univariate Polynomial Ring in x over Finite Field of size 17
```

See also:

- `sage.misc.defaults.set_series_precision()`

class `sage.rings.laurent_series_ring.`**`LaurentSeriesRing`**(*power_series*)

Bases: `UniqueRepresentation`, `CommutativeRing`

Univariate Laurent Series Ring.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, 'x'); R
Laurent Series Ring in x over Rational Field
sage: x = R.0
sage: g = 1 - x + x^2 - x^4 + O(x^8); g
1 - x + x^2 - x^4 + O(x^8)
sage: g = 10*x^(-3) + 2006 - 19*x + x^2 - x^4 +O(x^8); g
10*x^-3 + 2006 - 19*x + x^2 - x^4 + O(x^8)
```

You can also use more mathematical notation when the base is a field:

```
sage: Frac(QQ[['x']])
Laurent Series Ring in x over Rational Field
sage: Frac(GF(5)['y'])
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
```

When the base ring is a domain, the fraction field is the Laurent series ring over the fraction field of the base ring:

```
sage: Frac(ZZ[['t']])
Laurent Series Ring in t over Rational Field
```

Laurent series rings are determined by their variable and the base ring, and are globally unique:

```
sage: # needs sage.rings.padics
sage: K = Qp(5, prec=5)
sage: L = Qp(5, prec=200)
sage: R.<x> = LaurentSeriesRing(K)
sage: S.<y> = LaurentSeriesRing(L)
sage: R is S
False
sage: T.<y> = LaurentSeriesRing(Qp(5, prec=200))
sage: S is T
True
sage: W.<y> = LaurentSeriesRing(Qp(5, prec=199))
sage: W is T
False

sage: K = LaurentSeriesRing(CC, 'q'); K                                  #␣
↪needs sage.rings.real_mpfr
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: loads(K.dumps()) == K                                              #␣
↪needs sage.rings.real_mpfr
True
sage: P = QQ[['x']]
sage: F = Frac(P)
sage: TestSuite(F).run()
```

When the base ring k is a field, the ring $k((x))$ is a CDVF, that is a field equipped with a discrete valuation for which it is complete. The appropriate (sub)category is automatically set in this case:

```
sage: k = GF(11)
sage: R.<x> = k[[]]
sage: F = Frac(R)
sage: F.category()
Join of
 Category of complete discrete valuation fields and
 Category of commutative algebras over (finite enumerated fields and
 subquotients of monoids and quotients of semigroups) and
 Category of infinite sets
sage: TestSuite(F).run()
```

Element

alias of *LaurentSeries*

base_extend(*R*)

Return the Laurent series ring over R in the same variable as self, assuming there is a canonical coerce map from the base ring of self to R.

EXAMPLES:

```
sage: K.<x> = LaurentSeriesRing(QQ, default_prec=4)
sage: K.base_extend(QQ['t'])
Laurent Series Ring in x over Univariate Polynomial Ring in t over Rational␣
↪Field
```

change_ring(*R*)

EXAMPLES:

```
sage: K.<x> = LaurentSeriesRing(QQ, default_prec=4)
sage: R = K.change_ring(ZZ); R
Laurent Series Ring in x over Integer Ring
sage: R.default_prec()
4
```

characteristic()

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(GF(17))
sage: R.characteristic()
17
```

construction()

Return the functorial construction of this Laurent power series ring.

The construction is given as the completion of the Laurent polynomials.

EXAMPLES:

```
sage: L.<t> = LaurentSeriesRing(ZZ, default_prec=42)
sage: phi, arg = L.construction()
sage: phi
Completion[t, prec=42]
sage: arg
Univariate Laurent Polynomial Ring in t over Integer Ring
sage: phi(arg) is L
True
```

Because of this construction, pushout is automatically available:

```
sage: 1/2 * t
1/2*t
sage: parent(1/2 * t)
Laurent Series Ring in t over Rational Field

sage: QQbar.gen() * t                                                    #␣
↪needs sage.rings.number_field
I*t
sage: parent(QQbar.gen() * t)                                            #␣
↪needs sage.rings.number_field
Laurent Series Ring in t over Algebraic Field
```

default_prec()

Get the precision to which exact elements are truncated when necessary (most frequently when inverting).

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ, default_prec=5)
sage: R.default_prec()
5
```

fraction_field()

Return the fraction field of this ring of Laurent series.

If the base ring is a field, then Laurent series are already a field. If the base ring is a domain, then the Laurent series over its fraction field is returned. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: R = LaurentSeriesRing(ZZ, 't', 30).fraction_field()
sage: R
Laurent Series Ring in t over Rational Field
sage: R.default_prec()
30

sage: LaurentSeriesRing(Zmod(4), 't').fraction_field()
Traceback (most recent call last):
...
ValueError: must be an integral domain
```

gen(*n=0*)

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.gen()
x
```

is_dense()

EXAMPLES:

```
sage: K.<x> = LaurentSeriesRing(QQ, sparse=True)
sage: K.is_dense()
False
```

is_exact()

Laurent series rings are inexact.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.is_exact()
False
```

is_field(*proof=True*)

A Laurent series ring is a field if and only if the base ring is a field.

is_sparse()

Return if `self` is a sparse implementation.

EXAMPLES:

```
sage: K.<x> = LaurentSeriesRing(QQ, sparse=True)
sage: K.is_sparse()
True
```

laurent_polynomial_ring()

If this is the Laurent series ring $R((t))$, return the Laurent polynomial ring $R[t, 1/t]$.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.laurent_polynomial_ring()
Univariate Laurent Polynomial Ring in x over Rational Field
```

ngens()

Laurent series rings are univariate.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.ngens()
1
```

polynomial_ring()

If this is the Laurent series ring $R((t))$, return the polynomial ring $R[t]$.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

power_series_ring()

If this is the Laurent series ring $R((t))$, return the power series ring $R[[t]]$.

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.power_series_ring()
Power Series Ring in x over Rational Field
```

random_element(*algorithm='default'*)

Return a random element of this Laurent series ring.

The optional `algorithm` parameter decides how elements are generated. Algorithms currently implemented:

- `'default'`: Choose an integer `shift` using the standard distribution on the integers. Then choose a list of coefficients using the `random_element` function of the base ring, and construct a new element based on those coefficients, so that the i-th coefficient corresponds to the (i+shift)-th power of the uniformizer. The amount of coefficients is determined by the `default_prec` of the ring. Note that this method only creates non-exact elements.

EXAMPLES:

```
sage: S.<s> = LaurentSeriesRing(GF(3))
sage: S.random_element()  # random
s^-8 + s^-7 + s^-6 + s^-5 + s^-1 + s + s^3 + s^4
+ s^5 + 2*s^6 + s^7 + s^11 + O(s^12)
```

residue_field()

Return the residue field of this Laurent series field if it is a complete discrete valuation field (i.e. if the base ring is a field, in which base it is also the residue field).

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(GF(17))
sage: R.residue_field()
Finite Field of size 17

sage: R.<x> = LaurentSeriesRing(ZZ)
sage: R.residue_field()
Traceback (most recent call last):
...
TypeError: the base ring is not a field
```

uniformizer()

Return a uniformizer of this Laurent series field if it is a discrete valuation field (i.e. if the base ring is actually a field). Otherwise, an error is raised.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: R.uniformizer()
t

sage: R.<t> = LaurentSeriesRing(ZZ)
sage: R.uniformizer()
Traceback (most recent call last):
...
TypeError: the base ring is not a field
```

sage.rings.laurent_series_ring.**is_LaurentSeriesRing**(*x*)

Return `True` if this is a *univariate* Laurent series ring.

This is in keeping with the behavior of `is_PolynomialRing` versus `is_MPolynomialRing`.

CHAPTER

EIGHT

LAURENT SERIES

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(GF(7), 't'); R
Laurent Series Ring in t over Finite Field of size 7
sage: f = 1/(1-t+O(t^10)); f
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)
```

Laurent series are immutable:

```
sage: f[2]
1
sage: f[2] = 5
Traceback (most recent call last):
...
IndexError: Laurent series are immutable
```

We compute with a Laurent series over the complex mpfr numbers.

```
sage: K.<q> = Frac(CC[['q']]); K                                        #␣
↪needs sage.rings.real_mpfr
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: q                                                                 #␣
↪needs sage.rings.real_mpfr
1.00000000000000*q
```

Saving and loading.

```
sage: loads(q.dumps()) == q                                             #␣
↪needs sage.rings.real_mpfr
True
sage: loads(K.dumps()) == K                                             #␣
↪needs sage.rings.real_mpfr
True
```

IMPLEMENTATION: Laurent series in Sage are represented internally as a power of the variable times the unit part (which need not be a unit - it's a polynomial with nonzero constant term). The zero Laurent series has unit part 0.

AUTHORS:

- William Stein: original version
- David Joyner (2006-01-22): added examples
- Robert Bradshaw (2007-04): optimizations, shifting

- Robert Bradshaw: Cython version

class sage.rings.laurent_series_ring_element.**LaurentSeries**

Bases: AlgebraElement

A Laurent Series.

We consider a Laurent series of the form $t^n \cdot f$ where f is a power series.

INPUT:

- parent – a Laurent series ring
- f – a power series (or something can be coerced to one); note that f does *not* have to be a unit
- n – (default: 0) integer

O(*prec*)

Return the Laurent series of precision at most prec obtained by adding $O(q^{\text{prec}})$, where q is the variable.

The precision of self and the integer prec can be arbitrary. The resulting Laurent series will have precision equal to the minimum of the precision of self and prec. The term $O(q^{\text{prec}})$ is the zero series with precision prec.

See also *add_bigoh()*.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = t^-5 + t^-4 + t^3 + O(t^10); f
t^-5 + t^-4 + t^3 + O(t^10)
sage: f.O(-4)
t^-5 + O(t^-4)
sage: f.O(15)
t^-5 + t^-4 + t^3 + O(t^10)
```

V(*n*)

Return the n-th Verschiebung of self.

If $f = \sum a_m x^m$ then this function returns $\sum a_m x^{mn}$.

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: f = -1/x + 1 + 2*x^2 + 5*x^5
sage: f.V(2)
-x^-2 + 1 + 2*x^4 + 5*x^10
sage: f.V(-1)
5*x^-5 + 2*x^-2 + 1 - x
sage: h = f.add_bigoh(7)
sage: h.V(2)
-x^-2 + 1 + 2*x^4 + 5*x^10 + O(x^14)
sage: h.V(-2)
Traceback (most recent call last):
...
ValueError: For finite precision only positive arguments allowed
```

add_bigoh(*prec*)

Return the truncated series at chosen precision prec.

See also `O()`.

INPUT:

- `prec` – the precision of the series as an integer

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = t^2 + t^3 + O(t^10); f
t^2 + t^3 + O(t^10)
sage: f.add_bigoh(5)
t^2 + t^3 + O(t^5)
```

change_ring(*R*)

Change the base ring of `self`.

EXAMPLES:

```
sage: R.<q> = LaurentSeriesRing(ZZ)
sage: p = R([1,2,3]); p
1 + 2*q + 3*q^2
sage: p.change_ring(GF(2))
1 + q^2
```

coefficients()

Return the nonzero coefficients of `self`.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.coefficients()
[-5, 1, 1, -10/3]
```

common_prec(*other*)

Return the minimum precision of `self` and `other`.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
```

```
sage: f = t^(-1) + t + t^2 + O(t^3)
sage: g = t + t^3 + t^4 + O(t^4)
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3
```

```
sage: f = t + t^2 + O(t^3)
sage: g = t^(-3) + t^2
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3
```

```
sage: f = t + t^2
sage: g = t^2
sage: f.common_prec(g)
+Infinity
```

```
sage: f = t^(-3) + O(t^(-2))
sage: g = t^(-5) + O(t^(-1))
sage: f.common_prec(g)
-2
```

```
sage: f = O(t^2)
sage: g = O(t^5)
sage: f.common_prec(g)
2
```

common_valuation(*other*)

Return the minimum valuation of `self` and `other`.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
```

```
sage: f = t^(-1) + t + t^2 + O(t^3)
sage: g = t + t^3 + t^4 + O(t^4)
sage: f.common_valuation(g)
-1
sage: g.common_valuation(f)
-1
```

```
sage: f = t + t^2 + O(t^3)
sage: g = t^(-3) + t^2
sage: f.common_valuation(g)
-3
sage: g.common_valuation(f)
-3
```

```
sage: f = t + t^2
sage: g = t^2
sage: f.common_valuation(g)
1
```

```
sage: f = t^(-3) + O(t^(-2))
sage: g = t^(-5) + O(t^(-1))
sage: f.common_valuation(g)
-5
```

```
sage: f = O(t^2)
sage: g = O(t^5)
sage: f.common_valuation(g)
+Infinity
```

degree()

Return the degree of a polynomial equivalent to this power series modulo big oh of the precision.

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: g = x^2 - x^4 + O(x^8)
sage: g.degree()
4
sage: g = -10/x^5 + x^2 - x^4 + O(x^8)
sage: g.degree()
4
sage: (x^-2 + O(x^0)).degree()
-2
```

derivative(**args*)

The formal derivative of this Laurent series, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:

`_derivative()`

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: g = 1/x^10 - x + x^2 - x^4 + O(x^8)
sage: g.derivative()
-10*x^-11 - 1 + 2*x - 4*x^3 + O(x^7)
sage: g.derivative(x)
-10*x^-11 - 1 + 2*x - 4*x^3 + O(x^7)
```

```
sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = LaurentSeriesRing(R)
sage: f = 2*t/x + (3*t^2 + 6*t)*x + O(x^2)
sage: f.derivative()
-2*t*x^-2 + (3*t^2 + 6*t) + O(x)
sage: f.derivative(x)
-2*t*x^-2 + (3*t^2 + 6*t) + O(x)
sage: f.derivative(t)
2*x^-1 + (6*t + 6)*x + O(x^2)
```

exponents()

Return the exponents appearing in self with nonzero coefficients.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.exponents()
[-2, 1, 2, 3]
```

integral()

The formal integral of this Laurent series with 0 constant term.

EXAMPLES: The integral may or may not be defined if the base ring is not a field.

```
sage: t = LaurentSeriesRing(ZZ, 't').0
sage: f = 2*t^-3 + 3*t^2 + O(t^4)
sage: f.integral()
-t^-2 + t^3 + O(t^5)
```

```
sage: f = t^3
sage: f.integral()
Traceback (most recent call last):
...
ArithmeticError: Coefficients of integral cannot be coerced into the base ring
```

The integral of 1/t is $\log(t)$, which is not given by a Laurent series:

```
sage: t = Frac(QQ[['t']]).0
sage: f = -1/t^3 - 31/t + O(t^3)
sage: f.integral()
Traceback (most recent call last):
...
ArithmeticError: The integral of is not a Laurent series, since t^-1 has␣
↪nonzero coefficient.
```

Another example with just one negative coefficient:

```
sage: A.<t> = QQ[[]]
sage: f = -2*t^(-4) + O(t^8)
sage: f.integral()
2/3*t^-3 + O(t^9)
sage: f.integral().derivative() == f
True
```

inverse()

Return the inverse of self, i.e., self^(-1).

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(ZZ)
sage: t.inverse()
t^-1
sage: (1-t).inverse()
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + ...
```

is_monomial()

Return `True` if this element is a monomial. That is, if self is x^n for some integer n.

EXAMPLES:

```
sage: k.<z> = LaurentSeriesRing(QQ, 'z')
sage: (30*z).is_monomial()
False
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
```

(continues on next page)

(continued from previous page)

```
False
sage: (z^-2909).is_monomial()
True
sage: (3*z^-2909).is_monomial()
False
```

is_unit()

Return True if this is Laurent series is a unit in this ring.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: (2 + t).is_unit()
True
sage: f = 2 + t^2 + O(t^10); f.is_unit()
True
sage: 1/f
1/2 - 1/4*t^2 + 1/8*t^4 - 1/16*t^6 + 1/32*t^8 + O(t^10)
sage: R(0).is_unit()
False
sage: R.<s> = LaurentSeriesRing(ZZ)
sage: f = 2 + s^2 + O(s^10)
sage: f.is_unit()
False
sage: 1/f
Traceback (most recent call last):
...
ValueError: constant term 2 is not a unit
```

ALGORITHM: A Laurent series is a unit if and only if its "unit part" is a unit.

is_zero()

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = 1/x + x + x^2 + 3*x^4 + O(x^7)
sage: f.is_zero()
0
sage: z = 0*f
sage: z.is_zero()
1
```

laurent_polynomial()

Return the corresponding Laurent polynomial.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = t^-3 + t + 7*t^2 + O(t^5)
sage: g = f.laurent_polynomial(); g
t^-3 + t + 7*t^2
sage: g.parent()
Univariate Laurent Polynomial Ring in t over Rational Field
```

lift_to_precision(*absprec=None*)

Return a congruent Laurent series with absolute precision at least `absprec`.

INPUT:

- `absprec` – an integer or `None` (default: `None`), the absolute precision of the result. If `None`, lifts to an exact element.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: A.<t> = LaurentSeriesRing(GF(5))
sage: x = t^(-1) + t^2 + O(t^5)
sage: x.lift_to_precision(10)
t^-1 + t^2 + O(t^10)
sage: x.lift_to_precision()
t^-1 + t^2
```

list()

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.list()
[-5, 0, 0, 1, 1, -10/3]
```

nth_root(*n*, *prec=None*)

Return the `n`-th root of this Laurent power series.

INPUT:

- `n` – integer
- `prec` – integer (optional) - precision of the result. Though, if this series has finite precision, then the result cannot have larger precision.

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: (x^-2 + 1 + x).nth_root(2)
x^-1 + 1/2*x + 1/2*x^2 - ... - 19437/65536*x^18 + O(x^19)
sage: (x^-2 + 1 + x).nth_root(2)**2
x^-2 + 1 + x + O(x^18)

sage: # needs sage.modular
sage: j = j_invariant_qexp()
sage: q = j.parent().gen()
sage: j(q^3).nth_root(3)
q^-1 + 248*q^2 + 4124*q^5 + ... + O(q^29)
sage: (j(q^2) - 1728).nth_root(2)
q^-1 - 492*q - 22590*q^3 - ... + O(q^19)
```

power_series()

Convert this Laurent series to a power series.

An error is raised if the Laurent series has a term (or an error term $O(x^k)$) whose exponent is negative.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(ZZ)
sage: f = 1/(1-t+O(t^10)); f.parent()
Laurent Series Ring in t over Integer Ring
sage: g = f.power_series(); g
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)
sage: parent(g)
Power Series Ring in t over Integer Ring
sage: f = 3/t^2 +  t^2 + t^3 + O(t^10)
sage: f.power_series()
Traceback (most recent call last):
...
TypeError: self is not a power series
```

prec()

This function returns the n so that the Laurent series is of the form (stuff) + $O(t^n)$. It doesn't matter how many negative powers appear in the expansion. In particular, prec could be negative.

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = x^2 + 3*x^4 + O(x^7)
sage: f.prec()
7
sage: g = 1/x^10 - x + x^2 - x^4 + O(x^8)
sage: g.prec()
8
```

precision_absolute()

Return the absolute precision of this series.

By definition, the absolute precision of $... + O(x^r)$ is r.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).precision_absolute()
3
sage: (1 - t^2 + O(t^100)).precision_absolute()
100
```

precision_relative()

Return the relative precision of this series, that is the difference between its absolute precision and its valuation.

By convention, the relative precision of 0 (or $O(x^r)$ for any r) is 0.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).precision_relative()
1
sage: (1 - t^2 + O(t^100)).precision_relative()
100
sage: O(t^4).precision_relative()
0
```

residue()

Return the residue of `self`.

Consider the Laurent series

$$f = \sum_{n \in \mathbf{Z}} a_n t^n = \cdots + \frac{a_{-2}}{t^2} + \frac{a_{-1}}{t} + a_0 + a_1 t + a_2 t^2 + \cdots ,$$

then the residue of f is a_{-1}. Alternatively this is the coefficient of $1/t$.

EXAMPLES:

```
sage: t = LaurentSeriesRing(ZZ,'t').gen()
sage: f = 1/t**2 + 2/t + 3 + 4*t
sage: f.residue()
2
sage: f = t + t**2
sage: f.residue()
0
sage: f.residue().parent()
Integer Ring
```

reverse(*precision=None*)

Return the reverse of f, i.e., the series g such that g(f(x)) = x. Given an optional argument `precision`, return the reverse with given precision (note that the reverse can have precision at most `f.prec()`). If `f` has infinite precision, and the argument `precision` is not given, then the precision of the reverse defaults to the default precision of `f.parent()`.

Note that this is only possible if the valuation of self is exactly 1.

The implementation depends on the underlying power series element implementing a reverse method.

EXAMPLES:

```
sage: R.<x> = Frac(QQ[['x']])
sage: f = 2*x + 3*x^2 - x^4 + O(x^5)
sage: g = f.reverse()
sage: g
1/2*x - 3/8*x^2 + 9/16*x^3 - 131/128*x^4 + O(x^5)
sage: f(g)
x + O(x^5)
sage: g(f)
x + O(x^5)

sage: A.<t> = LaurentSeriesRing(ZZ)
sage: a = t - t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
t + t^2 + 2*t^3 + 7*t^4 + 25*t^5 + O(t^6)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)

sage: B.<b,c> = ZZ[ ]
sage: A.<t> = LaurentSeriesRing(B)
sage: f = t + b*t^2 + c*t^3 + O(t^4)
```

(continues on next page)

(continued from previous page)

```
sage: g = f.reverse(); g
t - b*t^2 + (2*b^2 - c)*t^3 + O(t^4)
sage: f(g)
t + O(t^4)
sage: g(f)
t + O(t^4)

sage: A.<t> = PowerSeriesRing(ZZ)
sage: B.<s> = LaurentSeriesRing(A)
sage: f = (1 - 3*t + 4*t^3 + O(t^4))*s + (2 + t + t^2 + O(t^3))*s^2 + O(s^3)
sage: set_verbose(1)
sage: g = f.reverse(); g
verbose 1 (<module>) passing to pari failed; trying Lagrange inversion
(1 + 3*t + 9*t^2 + 23*t^3 + O(t^4))*s + (-2 - 19*t - 118*t^2 + O(t^3))*s^2 +
↪O(s^3)
sage: set_verbose(0)
sage: f(g) == g(f) == s
True
```

If the leading coefficient is not a unit, we pass to its fraction field if possible:

```
sage: A.<t> = LaurentSeriesRing(ZZ)
sage: a = 2*t - 4*t^2 + t^4 - t^5 + O(t^6)
sage: a.reverse()
1/2*t + 1/2*t^2 + t^3 + 79/32*t^4 + 437/64*t^5 + O(t^6)

sage: B.<b> = PolynomialRing(ZZ)
sage: A.<t> = LaurentSeriesRing(B)
sage: f = 2*b*t + b*t^2 + 3*b^2*t^3 + O(t^4)
sage: g = f.reverse(); g
1/(2*b)*t - 1/(8*b^2)*t^2 + ((-3*b + 1)/(16*b^3))*t^3 + O(t^4)
sage: f(g)
t + O(t^4)
sage: g(f)
t + O(t^4)
```

We can handle some base rings of positive characteristic:

```
sage: A8.<t> = LaurentSeriesRing(Zmod(8))
sage: a = t - 15*t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
t + 7*t^2 + 2*t^3 + 5*t^4 + t^5 + O(t^6)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)
```

The optional argument `precision` sets the precision of the output:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: f = 2*x + 3*x^2 - 7*x^3 + x^4 + O(x^5)
sage: g = f.reverse(precision=3); g
1/2*x - 3/8*x^2 + O(x^3)
```

(continues on next page)

(continued from previous page)

```
sage: f(g)
x + O(x^3)
sage: g(f)
x + O(x^3)
```

If the input series has infinite precision, the precision of the output is automatically set to the default precision of the parent ring:

```
sage: R.<x> = LaurentSeriesRing(QQ, default_prec=20)
sage: (x - x^2).reverse()  # get some Catalan numbers
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + 42*x^6 + 132*x^7 + 429*x^8 + 1430*x^9
 + 4862*x^10 + 16796*x^11 + 58786*x^12 + 208012*x^13 + 742900*x^14
 + 2674440*x^15 + 9694845*x^16 + 35357670*x^17 + 129644790*x^18
 + 477638700*x^19 + O(x^20)
sage: (x - x^2).reverse(precision=3)
x + x^2 + O(x^3)
```

shift(k)

Returns this Laurent series multiplied by the power t^n. Does not change this series.

Note: Despite the fact that higher order terms are printed to the right in a power series, right shifting decreases the powers of t, while left shifting increases them. This is to be consistent with polynomials, integers, etc.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ['y'])
sage: f = (t+t^-1)^4; f
t^-4 + 4*t^-2 + 6 + 4*t^2 + t^4
sage: f.shift(10)
t^6 + 4*t^8 + 6*t^10 + 4*t^12 + t^14
sage: f >> 10
t^-14 + 4*t^-12 + 6*t^-10 + 4*t^-8 + t^-6
sage: t << 4
t^5
sage: t + O(t^3) >> 4
t^-3 + O(t^-1)
```

AUTHORS:

- Robert Bradshaw (2007-04-18)

truncate(n)

Return the Laurent series of degree ` < n` which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: A.<x> = LaurentSeriesRing(ZZ)
sage: f = 1/(1-x)
sage: f
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^10 + x^11
 + x^12 + x^13 + x^14 + x^15 + x^16 + x^17 + x^18 + x^19 + O(x^20)
```

(continues on next page)

(continued from previous page)

```
sage: f.truncate(10)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9
```

truncate_laurentseries(*n*)

Replace any terms of degree >= n by big oh.

EXAMPLES:

```
sage: A.<x> = LaurentSeriesRing(ZZ)
sage: f = 1/(1-x)
sage: f
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^10 + x^11
 + x^12 + x^13 + x^14 + x^15 + x^16 + x^17 + x^18 + x^19 + O(x^20)
sage: f.truncate_laurentseries(10)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + O(x^10)
```

truncate_neg(*n*)

Return the Laurent series equivalent to `self` except without any degree `n` terms.

This is equivalent to:

```
self - self.truncate(n)
```

EXAMPLES:

```
sage: A.<t> = LaurentSeriesRing(ZZ)
sage: f = 1/(1-t)
sage: f.truncate_neg(15)
t^15 + t^16 + t^17 + t^18 + t^19 + O(t^20)
```

valuation()

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: f = 1/x + x^2 + 3*x^4 + O(x^7)
sage: g = 1 - x + x^2 - x^4 + O(x^8)
sage: f.valuation()
-1
sage: g.valuation()
0
```

Note that the valuation of an element undistinguishable from zero is infinite:

```
sage: h = f - f; h
O(x^7)
sage: h.valuation()
+Infinity
```

valuation_zero_part()

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = x + x^2 + 3*x^4 + O(x^7)
```

(continues on next page)

(continued from previous page)

```
sage: f/x
1 + x + 3*x^3 + O(x^6)
sage: f.valuation_zero_part()
1 + x + 3*x^3 + O(x^6)
sage: g = 1/x^7 - x + x^2 - x^4 + O(x^8)
sage: g.valuation_zero_part()
1 - x^8 + x^9 - x^11 + O(x^15)
```

variable()

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = 1/x + x^2 + 3*x^4 + O(x^7)
sage: f.variable()
'x'
```

verschiebung(n)

Return the n-th Verschiebung of `self`.

If $f = \sum a_m x^m$ then this function returns $\sum a_m x^{mn}$.

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: f = -1/x + 1 + 2*x^2 + 5*x^5
sage: f.V(2)
-x^-2 + 1 + 2*x^4 + 5*x^10
sage: f.V(-1)
5*x^-5 + 2*x^-2 + 1 - x
sage: h = f.add_bigoh(7)
sage: h.V(2)
-x^-2 + 1 + 2*x^4 + 5*x^10 + O(x^14)
sage: h.V(-2)
Traceback (most recent call last):
...
ValueError: For finite precision only positive arguments allowed
```

sage.rings.laurent_series_ring_element.**is_LaurentSeries**(x)

CHAPTER NINE

LAZY SERIES

Coefficients of lazy series are computed on demand. They have infinite precision, although equality can only be decided in special cases.

AUTHORS:

- Kwankyu Lee (2019-02-24): initial version
- Tejasvi Chebrolu, Martin Rubey, Travis Scrimshaw (2021-08): refactored and expanded functionality

EXAMPLES:

Laurent series over the integer ring are particularly useful as generating functions for sequences arising in combinatorics.

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
```

The generating function of the Fibonacci sequence is:

```
sage: f = 1 / (1 - z - z^2)
sage: f
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + O(z^7)
```

In principle, we can now compute any coefficient of f:

```
sage: f.coefficient(100)
573147844013817084101
```

Which coefficients are actually computed depends on the type of implementation. For the sparse implementation, only the coefficients which are needed are computed.

```
sage: s = L(lambda n: n, valuation=0); s
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: s.coefficient(10)
10
sage: s._coeff_stream._cache
{1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 10: 10}
```

Using the dense implementation, all coefficients up to the required coefficient are computed.

```
sage: L.<x> = LazyLaurentSeriesRing(ZZ, sparse=False)
sage: s = L(lambda n: n, valuation=0); s
x + 2*x^2 + 3*x^3 + 4*x^4 + 5*x^5 + 6*x^6 + O(x^7)
sage: s.coefficient(10)
10
```

(continues on next page)

(continued from previous page)

```
sage: s._coeff_stream._cache
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can do arithmetic with lazy power series:

```
sage: f
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + O(z^7)
sage: f^-1
1 - z - z^2 + O(z^7)
sage: f + f^-1
2 + z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + O(z^7)
sage: g = (f + f^-1)*(f - f^-1); g
4*z + 6*z^2 + 8*z^3 + 19*z^4 + 38*z^5 + 71*z^6 + O(z^7)
```

We call lazy power series whose coefficients are known to be eventually constant 'exact'. In some cases, computations with such series are much faster. Moreover, these are the series where equality can be decided. For example:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: f = 1 + 2*z^2 / (1 - z)
sage: f - 2 / (1 - z) + 1 + 2*z
0
```

However, multivariate Taylor series are actually represented as streams of multivariate polynomials. Therefore, the only exact series in this case are polynomials:

```
sage: L.<x,y> = LazyPowerSeriesRing(ZZ)
sage: 1 / (1-x)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + O(x,y)^7
```

A similar statement is true for lazy symmetric functions:

```
sage: h = SymmetricFunctions(QQ).h()                                      #␣
↪needs sage.combinat
sage: L = LazySymmetricFunctions(h)                                       #␣
↪needs sage.combinat
sage: 1 / (1-L(h[1]))                                                     #␣
↪needs sage.combinat
h[] + h[1] + (h[1,1]) + (h[1,1,1]) + (h[1,1,1,1]) + (h[1,1,1,1,1]) + (h[1,1,1,1,1,1]) +␣
↪O^7
```

We can change the base ring:

```
sage: h = g.change_ring(QQ)
sage: h.parent()                                                          #␣
↪needs sage.combinat
Lazy Laurent Series Ring in z over Rational Field
sage: h                                                                   #␣
↪needs sage.combinat
4*z + 6*z^2 + 8*z^3 + 19*z^4 + 38*z^5 + 71*z^6 + 130*z^7 + O(z^8)
sage: hinv = h^-1; hinv                                                   #␣
↪needs sage.combinat
1/4*z^-1 - 3/8 + 1/16*z - 17/32*z^2 + 5/64*z^3 - 29/128*z^4 + 165/256*z^5 + O(z^6)
sage: hinv.valuation()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
-1
```

class sage.rings.lazy_series.**LazyCauchyProductSeries**(*parent*, *coeff_stream*)

Bases: *LazyModuleElement*

A class for series where multiplication is the Cauchy product.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = 1 / (1 - z)
sage: f
1 + z + z^2 + O(z^3)
sage: f * (1 - z)
1

sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=True)
sage: f = 1 / (1 - z)
sage: f
1 + z + z^2 + O(z^3)
```

exp()

Return the exponential series of self.

We use the identity

$$\exp(s) = 1 + \int s' \exp(s).$$

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: exp(z)
1 + z + 1/2*z^2 + 1/6*z^3 + 1/24*z^4 + 1/120*z^5 + 1/720*z^6 + O(z^7)
sage: exp(z + z^2)
1 + z + 3/2*z^2 + 7/6*z^3 + 25/24*z^4 + 27/40*z^5 + 331/720*z^6 + O(z^7)
sage: exp(0)                                                            #␣
↪needs sage.symbolic
1
sage: exp(1 + z)
Traceback (most recent call last):
...
ValueError: can only compose with a positive valuation series

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: exp(x+y)[4].factor()
(1/24) * (x + y)^4
sage: exp(x/(1-y)).polynomial(3)
1/6*x^3 + x^2*y + x*y^2 + 1/2*x^2 + x*y + x + 1
```

log()

Return the series for the natural logarithm of self.

We use the identity

$$\log(s) = \int s'/s.$$

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: log(1/(1-z))
z + 1/2*z^2 + 1/3*z^3 + 1/4*z^4 + 1/5*z^5 + 1/6*z^6 + 1/7*z^7 + O(z^8)

sage: L.<x, y> = LazyPowerSeriesRing(QQ)
sage: log((1 + x/(1-y))).polynomial(3)
1/3*x^3 - x^2*y + x*y^2 - 1/2*x^2 + x*y + x
```

valuation()

Return the valuation of `self`.

This method determines the valuation of the series by looking for a nonzero coefficient. Hence if the series happens to be zero, then it may run forever.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: s = 1/(1 - z) - 1/(1 - 2*z)
sage: s.valuation()
1
sage: t = z - z
sage: t.valuation()
+Infinity
sage: M = L(lambda n: n^2, 0)
sage: M.valuation()
1
sage: (M - M).valuation()
+Infinity
```

class sage.rings.lazy_series.**LazyCompletionGradedAlgebraElement**(*parent*, *coeff_stream*)

Bases: *LazyCauchyProductSeries*

An element of a completion of a graded algebra that is computed lazily.

class sage.rings.lazy_series.**LazyDirichletSeries**(*parent*, *coeff_stream*)

Bases: *LazyModuleElement*

A Dirichlet series where the coefficients are computed lazily.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: f = L(constant=1)^2
sage: f                                                                   #␣
↪needs sage.symbolic
1 + 2/2^z + 2/3^z + 3/4^z + 2/5^z + 4/6^z + 2/7^z + O(1/(8^z))
sage: f.coefficient(100) == number_of_divisors(100)                       #␣
↪needs sage.libs.pari
True
```

Lazy Dirichlet series is picklable:

```
sage: g = loads(dumps(f))
sage: g                                                                   #␣
↪needs sage.symbolic
1 + 2/2^z + 2/3^z + 3/4^z + 2/5^z + 4/6^z + 2/7^z + O(1/(8^z))
sage: g == f
True
```

is_unit()

Return whether this element is a unit in the ring.

EXAMPLES:

```
sage: D = LazyDirichletSeriesRing(ZZ, "s")
sage: D([0, 2]).is_unit()
False

sage: D([-1, 2]).is_unit()
True

sage: D([3, 2]).is_unit()
False

sage: D = LazyDirichletSeriesRing(QQ, "s")
sage: D([3, 2]).is_unit()
True
```

valuation()

Return the valuation of `self`.

This method determines the valuation of the series by looking for a nonzero coefficient. Hence if the series happens to be zero, then it may run forever.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: mu = L(moebius); mu.valuation()                                #␣
↪needs sage.libs.pari
0
sage: (mu - mu).valuation()                                          #␣
↪needs sage.libs.pari
+Infinity
sage: g = L(constant=1, valuation=2)
sage: g.valuation()                                                  #␣
↪needs sage.symbolic
log(2)
sage: (g*g).valuation()                                              #␣
↪needs sage.symbolic
2*log(2)
```

class sage.rings.lazy_series.**LazyLaurentSeries**(*parent*, *coeff_stream*)

Bases: *LazyCauchyProductSeries*

A Laurent series where the coefficients are computed lazily.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
```

We can build a series from a function and specify if the series eventually takes a constant value:

```
sage: f = L(lambda i: i, valuation=-3, constant=-1, degree=3)
sage: f
-3*z^-3 - 2*z^-2 - z^-1 + z + 2*z^2 - z^3 - z^4 - z^5 + O(z^6)
sage: f[-2]
-2
sage: f[10]
-1
sage: f[-5]
0

sage: f = L(lambda i: i, valuation=-3)
sage: f
-3*z^-3 - 2*z^-2 - z^-1 + z + 2*z^2 + 3*z^3 + O(z^4)
sage: f[20]
20
```

Anything that converts into a polynomial can be input, where we can also specify the valuation or if the series eventually takes a constant value:

```
sage: L([-5,2,0,5])
-5 + 2*z + 5*z^3
sage: L([-5,2,0,5], constant=6)
-5 + 2*z + 5*z^3 + 6*z^4 + 6*z^5 + 6*z^6 + O(z^7)
sage: L([-5,2,0,5], degree=6, constant=6)
-5 + 2*z + 5*z^3 + 6*z^6 + 6*z^7 + 6*z^8 + O(z^9)
sage: L([-5,2,0,5], valuation=-2, degree=3, constant=6)
-5*z^-2 + 2*z^-1 + 5*z + 6*z^3 + 6*z^4 + 6*z^5 + O(z^6)
sage: L([-5,2,0,5], valuation=5)
-5*z^5 + 2*z^6 + 5*z^8
sage: L({-2:9, 3:4}, constant=2, degree=5)
9*z^-2 + 4*z^3 + 2*z^5 + 2*z^6 + 2*z^7 + O(z^8)
```

We can also perform arithmetic:

```
sage: f = 1 / (1 - z - z^2)
sage: f
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + O(z^7)
sage: f.coefficient(100)
573147844013817084101
sage: f = (z^-2 - 1 + 2*z) / (z^-1 - z + 3*z^2)
sage: f
z^-1 - z^2 - z^4 + 3*z^5 + O(z^6)
```

However, we may not always be able to know when a result is exactly a polynomial:

```
sage: f * (z^-1 - z + 3*z^2)
z^-2 - 1 + 2*z + O(z^5)
```

approximate_series(*prec*, *name=None*)

Return the Laurent series with absolute precision `prec` approximated from this series.

INPUT:

- `prec` – an integer
- `name` – name of the variable; if it is `None`, the name of the variable of the series is used

OUTPUT: a Laurent series with absolute precision `prec`

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: z = L.gen()
sage: f = (z - 2*z^3)^5/(1 - 2*z)
sage: f
z^5 + 2*z^6 - 6*z^7 - 12*z^8 + 16*z^9 + 32*z^10 - 16*z^11 + O(z^12)
sage: g = f.approximate_series(10)
sage: g
z^5 + 2*z^6 - 6*z^7 - 12*z^8 + 16*z^9 + O(z^10)
sage: g.parent()
Power Series Ring in z over Integer Ring
sage: h = (f^-1).approximate_series(3)
sage: h
z^-5 - 2*z^-4 + 10*z^-3 - 20*z^-2 + 60*z^-1 - 120 + 280*z - 560*z^2 + O(z^3)
sage: h.parent()
Laurent Series Ring in z over Integer Ring
```

compose(*g*)

Return the composition of `self` with `g`.

Given two Laurent series f and g over the same base ring, the composition $(f \circ g)(z) = f(g(z))$ is defined if and only if:

- $g = 0$ and $\mathrm{val}(f) \geq 0$,
- g is non-zero and f has only finitely many non-zero coefficients,
- g is non-zero and $\mathrm{val}(g) > 0$.

INPUT:

- `g` – other series

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: f = z^2 + 1 + z
sage: f(0)
1
sage: f(L(0))
1
sage: f(f)
3 + 3*z + 4*z^2 + 2*z^3 + z^4
sage: g = z^-3/(1-2*z); g
z^-3 + 2*z^-2 + 4*z^-1 + 8 + 16*z + 32*z^2 + 64*z^3 + O(z^4)
sage: f(g)
z^-6 + 4*z^-5 + 12*z^-4 + 33*z^-3 + 82*z^-2 + 196*z^-1 + 457 + O(z)
sage: g^2 + 1 + g
z^-6 + 4*z^-5 + 12*z^-4 + 33*z^-3 + 82*z^-2 + 196*z^-1 + 457 + O(z)
sage: f(int(2))
```

(continues on next page)

(continued from previous page)

```
7

sage: f = z^-2 + z + 4*z^3
sage: f(f)
4*z^-6 + 12*z^-3 + z^-2 + 48*z^-1 + 12 + O(z)
sage: f^-2 + f + 4*f^3
4*z^-6 + 12*z^-3 + z^-2 + 48*z^-1 + 12 + O(z)
sage: f(g)
4*z^-9 + 24*z^-8 + 96*z^-7 + 320*z^-6 + 960*z^-5 + 2688*z^-4 + 7169*z^-3 + O(z^-
↪2)
sage: g^-2 + g + 4*g^3
4*z^-9 + 24*z^-8 + 96*z^-7 + 320*z^-6 + 960*z^-5 + 2688*z^-4 + 7169*z^-3 + O(z^-
↪2)

sage: f = z^-3 + z^-2 + 1 / (1 + z^2); f
z^-3 + z^-2 + 1 - z^2 + O(z^4)
sage: g = z^3 / (1 + z - z^3); g
z^3 - z^4 + z^5 - z^7 + 2*z^8 - 2*z^9 + O(z^10)
sage: f(g)
z^-9 + 3*z^-8 + 3*z^-7 - z^-6 - 4*z^-5 - 2*z^-4 + z^-3 + O(z^-2)
sage: g^-3 + g^-2 + 1 / (1 + g^2)
z^-9 + 3*z^-8 + 3*z^-7 - z^-6 - 4*z^-5 - 2*z^-4 + z^-3 + O(z^-2)

sage: f = z^-3
sage: g = z^-2 + z^-1
sage: g^(-3)
z^6 - 3*z^7 + 6*z^8 - 10*z^9 + 15*z^10 - 21*z^11 + 28*z^12 + O(z^13)
sage: f(g)
z^6 - 3*z^7 + 6*z^8 - 10*z^9 + 15*z^10 - 21*z^11 + 28*z^12 + O(z^13)

sage: f = z^2 + z^3
sage: g = z^-3 + z^-2
sage: f^-3 + f^-2
z^-6 - 3*z^-5 + 7*z^-4 - 12*z^-3 + 18*z^-2 - 25*z^-1 + 33 + O(z)
sage: g(f)
z^-6 - 3*z^-5 + 7*z^-4 - 12*z^-3 + 18*z^-2 - 25*z^-1 + 33 + O(z)
sage: g^2 + g^3
z^-9 + 3*z^-8 + 3*z^-7 + 2*z^-6 + 2*z^-5 + z^-4
sage: f(g)
z^-9 + 3*z^-8 + 3*z^-7 + 2*z^-6 + 2*z^-5 + z^-4

sage: f = L(lambda n: n, valuation=0); f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: f(z^2)
z^2 + 2*z^4 + 3*z^6 + 4*z^8 + O(z^9)

sage: f = L(lambda n: n, valuation=-2); f
-2*z^-2 - z^-1 + z + 2*z^2 + 3*z^3 + 4*z^4 + O(z^5)
sage: f3 = f(z^3); f3
-2*z^-6 - z^-3 + O(z)
sage: [f3[i] for i in range(-6,13)]
[-2, 0, 0, -1, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 3, 0, 0, 4]
```

We compose a Laurent polynomial with a generic element:

```
sage: R.<x> = QQ[]
sage: f = z^2 + 1 + z^-1
sage: g = x^2 + x + 3
sage: f(g)
(x^6 + 3*x^5 + 12*x^4 + 19*x^3 + 37*x^2 + 28*x + 31)/(x^2 + x + 3)
sage: f(g) == g^2 + 1 + g^-1
True
```

We compose with another lazy Laurent series:

```
sage: LS.<y> = LazyLaurentSeriesRing(QQ)
sage: f = z^2 + 1 + z^-1
sage: fy = f(y); fy
y^-1 + 1 + y^2
sage: fy.parent() is LS
True
sage: g = y - y
sage: f(g)
Traceback (most recent call last):
...
ZeroDivisionError: the valuation of the series must be nonnegative

sage: g = 1 - y
sage: f(g)
3 - y + 2*y^2 + y^3 + y^4 + y^5 + O(y^6)
sage: g^2 + 1 + g^-1
3 - y + 2*y^2 + y^3 + y^4 + y^5 + O(y^6)

sage: f = L(lambda n: n, valuation=0); f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: f(0)
0
sage: f(y)
y + 2*y^2 + 3*y^3 + 4*y^4 + 5*y^5 + 6*y^6 + 7*y^7 + O(y^8)
sage: fp = f(y - y)
sage: fp == 0
True
sage: fp.parent() is LS
True

sage: f = z^2 + 3 + z
sage: f(y - y)
3
```

With both of them sparse:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ, sparse=True)
sage: LS.<y> = LazyLaurentSeriesRing(QQ, sparse=True)
sage: f = L(lambda n: 1, valuation=0); f
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + O(z^7)
sage: f(y^2)
1 + y^2 + y^4 + y^6 + O(y^7)
```

(continues on next page)

(continued from previous page)

```
sage: fp = f - 1 + z^-2; fp
z^-2 + z + z^2 + z^3 + z^4 + O(z^5)
sage: fpy = fp(y^2); fpy
y^-4 + y^2 + O(y^3)
sage: fpy.parent() is LS
True
sage: [fpy[i] for i in range(-4,11)]
[1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

sage: g = LS(valuation=2, constant=1); g
y^2 + y^3 + y^4 + O(y^5)
sage: fg = f(g); fg
1 + y^2 + y^3 + 2*y^4 + 3*y^5 + 5*y^6 + O(y^7)
sage: 1 + g + g^2 + g^3 + g^4 + g^5 + g^6
1 + y^2 + y^3 + 2*y^4 + 3*y^5 + 5*y^6 + O(y^7)

sage: h = LS(lambda n: 1 if n % 2 else 0, valuation=2); h
y^3 + y^5 + y^7 + O(y^9)
sage: fgh = fg(h); fgh
1 + y^6 + O(y^7)
sage: [fgh[i] for i in range(0, 15)]
[1, 0, 0, 0, 0, 0, 1, 0, 2, 1, 3, 3, 6, 6, 13]
sage: t = 1 + h^2 + h^3 + 2*h^4 + 3*h^5 + 5*h^6
sage: [t[i] for i in range(0, 15)]
[1, 0, 0, 0, 0, 0, 1, 0, 2, 1, 3, 3, 6, 6, 13]
```

We look at mixing the sparse and the dense:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: f = L(lambda n: 1, valuation=0); f
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + O(z^7)
sage: g = LS(lambda n: 1, valuation=1); g
y + y^2 + y^3 + y^4 + y^5 + y^6 + y^7 + O(y^8)
sage: f(g)
1 + y + 2*y^2 + 4*y^3 + 8*y^4 + 16*y^5 + 32*y^6 + O(y^7)

sage: f = z^-2 + 1 + z
sage: g = 1/(y*(1-y)); g
y^-1 + 1 + y + O(y^2)
sage: f(g)
y^-1 + 2 + y + 2*y^2 - y^3 + 2*y^4 + y^5 + y^6 + y^7 + O(y^8)
sage: g^-2 + 1 + g == f(g)
True

sage: f = z^-3 + z^-2 + 1
sage: g = 1/(y^2*(1-y)); g
y^-2 + y^-1 + 1 + O(y)
sage: f(g)
1 + y^4 - 2*y^5 + 2*y^6 - 3*y^7 + 3*y^8 - y^9
sage: g^-3 + g^-2 + 1 == f(g)
True
```

(continues on next page)

(continued from previous page)

```
sage: z(y)
y
```

We look at cases where the composition does not exist. $g = 0$ and $\mathrm{val}(f) < 0$:

```
sage: g = L(0)
sage: f = z^-1 + z^-2
sage: f.valuation() < 0
True
sage: f(g)
Traceback (most recent call last):
...
ZeroDivisionError: the valuation of the series must be nonnegative
```

$g \neq 0$ and $\mathrm{val}(g) \leq 0$ and f has infinitely many non-zero coefficients:

```
sage: g = z^-1 + z^-2
sage: g.valuation() <= 0
True
sage: f = L(lambda n: n, valuation=0)
sage: f(g)
Traceback (most recent call last):
...
ValueError: can only compose with a positive valuation series

sage: f = L(lambda n: n, valuation=1)
sage: f(1 + z)
Traceback (most recent call last):
...
ValueError: can only compose with a positive valuation series
```

We compose the exponential with a Dirichlet series:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: e = L(lambda n: 1/factorial(n), 0)
sage: D = LazyDirichletSeriesRing(QQ, "s")
sage: g = D(constant=1)-1
sage: g                                                                 #␣
↪needs sage.symbolic
1/(2^s) + 1/(3^s) + 1/(4^s) + O(1/(5^s))

sage: e(g)[0:10]
[0, 1, 1, 1, 3/2, 1, 2, 1, 13/6, 3/2]

sage: sum(g^k/factorial(k) for k in range(10))[0:10]
[0, 1, 1, 1, 3/2, 1, 2, 1, 13/6, 3/2]

sage: g = D([0,1,0,1,1,2])
sage: g                                                                 #␣
↪needs sage.symbolic
1/(2^s) + 1/(4^s) + 1/(5^s) + 2/6^s
sage: e(g)[0:10]
[0, 1, 1, 0, 3/2, 1, 2, 0, 7/6, 0]
```

(continues on next page)

(continued from previous page)

```
sage: sum(g^k/factorial(k) for k in range(10))[0:10]
[0, 1, 1, 0, 3/2, 1, 2, 0, 7/6, 0]

sage: e(D([1,0,1]))
Traceback (most recent call last):
...
ValueError: can only compose with a positive valuation series

sage: e5 = L(e, degree=5)
sage: e5
1 + z + 1/2*z^2 + 1/6*z^3 + 1/24*z^4
sage: e5(g)                                                            #␣
↪needs sage.symbolic
1 + 1/(2^s) + 3/2/4^s + 1/(5^s) + 2/6^s + O(1/(8^s))
sage: sum(e5[k] * g^k for k in range(5))                               #␣
↪needs sage.symbolic
1 + 1/(2^s) + 3/2/4^s + 1/(5^s) + 2/6^s + O(1/(8^s))
```

The output parent is always the common parent between the base ring of f and the parent of g or extended to the corresponding lazy series:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: R.<x> = ZZ[]
sage: parent(z(x))
Univariate Polynomial Ring in x over Rational Field
sage: parent(z(R.zero()))
Univariate Polynomial Ring in x over Rational Field
sage: parent(z(0))
Rational Field
sage: f = 1 / (1 - z)
sage: f(x)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + O(x^7)
sage: three = L(3)(x^2); three
3
sage: parent(three)
Univariate Polynomial Ring in x over Rational Field
```

Consistency check when g is an uninitialized series between a polynomial f as both a polynomial and a lazy series:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: f = 1 + z
sage: g = L.undefined(valuation=0)
sage: f(g) == f.polynomial()(g)
True
```

compositional_inverse()

Return the compositional inverse of `self`.

Given a Laurent series f, the compositional inverse is a Laurent series g over the same base ring, such that $(f \circ g)(z) = f(g(z)) = z$.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bz$ with $a, b \neq 0$, or
- $f = a/z$ with $a \neq 0$.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: (2*z).revert()
1/2*z
sage: (2/z).revert()
2*z^-1
sage: (z-z^2).revert()
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)

sage: s = L(degree=1, constant=-1)
sage: s.revert()
-z - z^2 - z^3 + O(z^4)

sage: s = L(degree=1, constant=1)
sage: s.revert()
z - z^2 + z^3 - z^4 + z^5 - z^6 + z^7 + O(z^8)
```

Warning: For series not known to be eventually constant (e.g., being defined by a function) with approximate valuation ≤ 1 (but not necessarily its true valuation), this assumes that this is the actual valuation:

```
sage: f = L(lambda n: n if n > 2 else 0, valuation=1)
sage: f.revert()
<repr(... failed: ValueError: inverse does not exist>
```

derivative(**args*)

Return the derivative of the Laurent series.

Multiple variables and iteration counts may be supplied; see the documentation of `sage.calculus.functional.derivative()` function for details.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: z.derivative()
1
sage: (1+z+z^2).derivative(3)
0
sage: (1/z).derivative()
-z^-2
sage: (1/(1-z)).derivative(z)
1 + 2*z + 3*z^2 + 4*z^3 + 5*z^4 + 6*z^5 + 7*z^6 + O(z^7)
```

is_unit()

Return whether this element is a unit in the ring.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: (2*z).is_unit()
False

sage: (1 + 2*z).is_unit()
True

sage: (1 + 2*z^-1).is_unit()
False

sage: (z^3 + 4 - z^-2).is_unit()
True
```

polynomial(*degree=None*, *name=None*)

Return `self` as a Laurent polynomial if `self` is actually so.

INPUT:

- `degree` – `None` or an integer
- `name` – name of the variable; if it is `None`, the name of the variable of the series is used

OUTPUT:

A Laurent polynomial if the valuation of the series is negative or a polynomial otherwise.

If `degree` is not `None`, the terms of the series of degree greater than `degree` are first truncated. If `degree` is `None` and the series is not a polynomial or a Laurent polynomial, a `ValueError` is raised.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = L([1,0,0,2,0,0,0,3], valuation=5); f
z^5 + 2*z^8 + 3*z^12
sage: f.polynomial()
3*z^12 + 2*z^8 + z^5
```

revert()

Return the compositional inverse of `self`.

Given a Laurent series f, the compositional inverse is a Laurent series g over the same base ring, such that $(f \circ g)(z) = f(g(z)) = z$.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bz$ with $a, b \neq 0$, or
- $f = a/z$ with $a \neq 0$.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: (2*z).revert()
1/2*z
sage: (2/z).revert()
2*z^-1
sage: (z-z^2).revert()
```

(continues on next page)

(continued from previous page)

```
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)

sage: s = L(degree=1, constant=-1)
sage: s.revert()
-z - z^2 - z^3 + O(z^4)

sage: s = L(degree=1, constant=1)
sage: s.revert()
z - z^2 + z^3 - z^4 + z^5 - z^6 + z^7 + O(z^8)
```

Warning: For series not known to be eventually constant (e.g., being defined by a function) with approximate valuation ≤ 1 (but not necessarily its true valuation), this assumes that this is the actual valuation:

```
sage: f = L(lambda n: n if n > 2 else 0, valuation=1)
sage: f.revert()
<repr(... failed: ValueError: inverse does not exist>
```

class sage.rings.lazy_series.**LazyModuleElement**(*parent*, *coeff_stream*)

Bases: `Element`

A lazy sequence with a module structure given by term-wise addition and scalar multiplication.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: M = L(lambda n: n, valuation=0)
sage: N = L(lambda n: 1, valuation=0)
sage: M[0:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: N[0:10]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Two sequences can be added:

```
sage: O = M + N
sage: O[0:10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Two sequences can be subtracted:

```
sage: P = M - N
sage: P[0:10]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

A sequence can be multiplied by a scalar:

```
sage: Q = 2 * M
sage: Q[0:10]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The negation of a sequence can also be found:

```
sage: R = -M
sage: R[0:10]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

arccos()

Return the arccosine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(RR)
sage: arccos(z)                                                          #␣
↪needs sage.symbolic
1.57079632679490 - 1.00000000000000*z + 0.000000000000000*z^2
 - 0.166666666666667*z^3 + 0.000000000000000*z^4
 - 0.0750000000000000*z^5 + O(1.00000000000000*z^7)

sage: L.<z> = LazyLaurentSeriesRing(SR)                                  #␣
↪needs sage.symbolic
sage: arccos(z/(1-z))                                                    #␣
↪needs sage.symbolic
1/2*pi - z - z^2 - 7/6*z^3 - 3/2*z^4 - 83/40*z^5 - 73/24*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(SR)                                  #␣
↪needs sage.symbolic
sage: arccos(x/(1-y))                                                    #␣
↪needs sage.symbolic
1/2*pi + (-x) + (-x*y) + ((-1/6)*x^3-x*y^2) + ((-1/2)*x^3*y-x*y^3)
 + ((-3/40)*x^5-x^3*y^2-x*y^4) + ((-3/8)*x^5*y+(-5/3)*x^3*y^3-x*y^5) + O(x,y)^7
```

arccot()

Return the arctangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(RR)
sage: arccot(z)                                                          #␣
↪needs sage.symbolic
1.57079632679490 - 1.00000000000000*z + 0.000000000000000*z^2
 + 0.333333333333333*z^3 + 0.000000000000000*z^4
 - 0.200000000000000*z^5 + O(1.00000000000000*z^7)

sage: L.<z> = LazyLaurentSeriesRing(SR)                                  #␣
↪needs sage.symbolic
sage: arccot(z/(1-z))                                                    #␣
↪needs sage.symbolic
1/2*pi - z - z^2 - 2/3*z^3 + 4/5*z^5 + 4/3*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(SR)                                  #␣
↪needs sage.symbolic
sage: acot(x/(1-y))                                                      #␣
↪needs sage.symbolic
1/2*pi + (-x) + (-x*y) + (1/3*x^3-x*y^2) + (x^3*y-x*y^3)
 + ((-1/5)*x^5+2*x^3*y^2-x*y^4) + (-x^5*y+10/3*x^3*y^3-x*y^5) + O(x,y)^7
```

arcsin()

Return the arcsine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: arcsin(z)
z + 1/6*z^3 + 3/40*z^5 + 5/112*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: asin(x/(1-y))
x + x*y + (1/6*x^3+x*y^2) + (1/2*x^3*y+x*y^3)
 + (3/40*x^5+x^3*y^2+x*y^4) + (3/8*x^5*y+5/3*x^3*y^3+x*y^5)
 + (5/112*x^7+9/8*x^5*y^2+5/2*x^3*y^4+x*y^6) + O(x,y)^8
```

arcsinh()

Return the inverse of the hyperbolic sine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: asinh(z)
z - 1/6*z^3 + 3/40*z^5 - 5/112*z^7 + O(z^8)
```

`arcsinh` is an alias:

```
sage: arcsinh(z)
z - 1/6*z^3 + 3/40*z^5 - 5/112*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: asinh(x/(1-y))
x + x*y + (-1/6*x^3+x*y^2) + (-1/2*x^3*y+x*y^3) + (3/40*x^5-x^3*y^2+x*y^4)
 + (3/8*x^5*y-5/3*x^3*y^3+x*y^5) + (-5/112*x^7+9/8*x^5*y^2-5/2*x^3*y^4+x*y^6) +␣
↪O(x,y)^8
```

arctan()

Return the arctangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: arctan(z)
z - 1/3*z^3 + 1/5*z^5 - 1/7*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: atan(x/(1-y))
x + x*y + (-1/3*x^3+x*y^2) + (-x^3*y+x*y^3) + (1/5*x^5-2*x^3*y^2+x*y^4)
 + (x^5*y-10/3*x^3*y^3+x*y^5) + (-1/7*x^7+3*x^5*y^2-5*x^3*y^4+x*y^6) + O(x,y)^8
```

arctanh()

Return the inverse of the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: atanh(z)
z + 1/3*z^3 + 1/5*z^5 + 1/7*z^7 + O(z^8)
```

`arctanh` is an alias:

```
sage: arctanh(z)
z + 1/3*z^3 + 1/5*z^5 + 1/7*z^7 + O(z^8)

sage: L.<x, y> = LazyPowerSeriesRing(QQ)
sage: atanh(x/(1-y))
x + x*y + (1/3*x^3+x*y^2) + (x^3*y+x*y^3) + (1/5*x^5+2*x^3*y^2+x*y^4)
 + (x^5*y+10/3*x^3*y^3+x*y^5) + (1/7*x^7+3*x^5*y^2+5*x^3*y^4+x*y^6) + O(x,y)^8
```

change_ring(*ring*)

Return `self` with coefficients converted to elements of `ring`.

INPUT:

- `ring` – a ring

EXAMPLES:

Dense Implementation:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=False)
sage: s = 2 + z
sage: t = s.change_ring(QQ)
sage: t^-1
1/2 - 1/4*z + 1/8*z^2 - 1/16*z^3 + 1/32*z^4 - 1/64*z^5 + 1/128*z^6 + O(z^7)
sage: M = L(lambda n: n, valuation=0); M
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: N = M.change_ring(QQ)
sage: N.parent()
Lazy Laurent Series Ring in z over Rational Field
sage: M.parent()
Lazy Laurent Series Ring in z over Integer Ring
```

Sparse Implementation:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=True)
sage: M = L(lambda n: n, valuation=0); M
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: M.parent()
Lazy Laurent Series Ring in z over Integer Ring
sage: N = M.change_ring(QQ)
sage: N.parent()
Lazy Laurent Series Ring in z over Rational Field
sage: M^-1
z^-1 - 2 + z + O(z^6)
```

A Dirichlet series example:

```
sage: L = LazyDirichletSeriesRing(ZZ, 'z')
sage: s = L(constant=2)
```

(continues on next page)

(continued from previous page)

```
sage: t = s.change_ring(QQ)
sage: t.parent()
Lazy Dirichlet Series Ring in z over Rational Field
sage: it = t^-1
sage: it                                                                          #␣
↪needs sage.symbolic
1/2 - 1/2/2^z - 1/2/3^z - 1/2/5^z + 1/2/6^z - 1/2/7^z + O(1/(8^z))
```

A Taylor series example:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: s = 2 + z
sage: t = s.change_ring(QQ)
sage: t^-1
1/2 - 1/4*z + 1/8*z^2 - 1/16*z^3 + 1/32*z^4 - 1/64*z^5 + 1/128*z^6 + O(z^7)
sage: t.parent()
Lazy Taylor Series Ring in z over Rational Field
```

coefficient(*n*)

Return the homogeneous degree `n` part of the series.

INPUT:

- `n` – integer; the degree

For a series `f`, the slice `f[start:stop]` produces the following:

- if `start` and `stop` are integers, return the list of terms with given degrees
- if `start` is `None`, return the list of terms beginning with the valuation
- if `stop` is `None`, return a `lazy_list_generic` instead.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = z / (1 - 2*z^3)
sage: [f[n] for n in range(20)]
[0, 1, 0, 0, 2, 0, 0, 4, 0, 0, 8, 0, 0, 16, 0, 0, 32, 0, 0, 64]
sage: f[0:20]
[0, 1, 0, 0, 2, 0, 0, 4, 0, 0, 8, 0, 0, 16, 0, 0, 32, 0, 0, 64]
sage: f[:20]
[1, 0, 0, 2, 0, 0, 4, 0, 0, 8, 0, 0, 16, 0, 0, 32, 0, 0, 64]
sage: f[::3]
lazy list [1, 2, 4, ...]

sage: M = L(lambda n: n, valuation=0)
sage: [M[n] for n in range(20)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=True)
sage: M = L(lambda n: n, valuation=0)
sage: [M[n] for n in range(20)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Similarly for multivariate series:

```
sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: sin(x*y)[:11]
[x*y, 0, 0, 0, -1/6*x^3*y^3, 0, 0, 0, 1/120*x^5*y^5]
sage: sin(x*y)[2::4]
lazy list [x*y, -1/6*x^3*y^3, 1/120*x^5*y^5, ...]
```

Similarly for Dirichlet series:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: L(lambda n: n)[1:11]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

coefficients(*n=None*)

Return the first n non-zero coefficients of `self`.

INPUT:

- `n` – (optional) the number of non-zero coefficients to return

If the series has fewer than n non-zero coefficients, only these are returned.

If `n` is `None`, a `lazy_list_generic` with all non-zero coefficients is returned instead.

Warning: If there are fewer than n non-zero coefficients, but this cannot be detected, this method will not return.

EXAMPLES:

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: f = L([1,2,3])
sage: f.coefficients(5)
doctest:...: DeprecationWarning: the method coefficients now only returns the␣
↪non-zero coefficients. Use __getitem__ instead.
See https://github.com/sagemath/sage/issues/32367 for details.
[1, 2, 3]

sage: f = sin(x)
sage: f.coefficients(5)
[1, -1/6, 1/120, -1/5040, 1/362880]

sage: L.<x, y> = LazyPowerSeriesRing(QQ)
sage: f = sin(x^2+y^2)
sage: f.coefficients(5)
[1, 1, -1/6, -1/2, -1/2]

sage: f.coefficients()
lazy list [1, 1, -1/6, ...]

sage: L.<x> = LazyPowerSeriesRing(GF(2))
sage: f = L(lambda n: n)
sage: f.coefficients(5)
[1, 1, 1, 1, 1]
```

cos()

Return the cosine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: cos(z)
1 - 1/2*z^2 + 1/24*z^4 - 1/720*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: cos(x/(1-y)).polynomial(4)
1/24*x^4 - 3/2*x^2*y^2 - x^2*y - 1/2*x^2 + 1
```

cosh()

Return the hyperbolic cosine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: cosh(z)
1 + 1/2*z^2 + 1/24*z^4 + 1/720*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: cosh(x/(1-y))
1 + 1/2*x^2 + x^2*y + (1/24*x^4+3/2*x^2*y^2) + (1/6*x^4*y+2*x^2*y^3)
 + (1/720*x^6+5/12*x^4*y^2+5/2*x^2*y^4) + O(x,y)^7
```

cot()

Return the cotangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: cot(z)
z^-1 - 1/3*z - 1/45*z^3 - 2/945*z^5 + O(z^6)

sage: L.<x> = LazyLaurentSeriesRing(QQ)
sage: cot(x/(1-x)).polynomial(4)
x^-1 - 1 - 1/3*x - 1/3*x^2 - 16/45*x^3 - 2/5*x^4
```

coth()

Return the hyperbolic cotangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: coth(z)                                                              #␣
↪needs sage.libs.flint
z^-1 + 1/3*z - 1/45*z^3 + 2/945*z^5 + O(z^6)

sage: coth(z + z^2)                                                        #␣
↪needs sage.libs.flint
z^-1 - 1 + 4/3*z - 2/3*z^2 + 44/45*z^3 - 16/15*z^4 + 884/945*z^5 + O(z^6)
```

csc()

Return the cosecant of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: csc(z)
z^-1 + 1/6*z + 7/360*z^3 + 31/15120*z^5 + O(z^6)

sage: L.<x> = LazyLaurentSeriesRing(QQ)
sage: csc(x/(1-x)).polynomial(4)
x^-1 - 1 + 1/6*x + 1/6*x^2 + 67/360*x^3 + 9/40*x^4
```

csch()

Return the hyperbolic cosecant of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: csch(z)                                                              #␣
↪needs sage.libs.flint
z^-1 - 1/6*z + 7/360*z^3 - 31/15120*z^5 + O(z^6)

sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: csch(z/(1-z))                                                        #␣
↪needs sage.libs.flint
z^-1 - 1 - 1/6*z - 1/6*z^2 - 53/360*z^3 - 13/120*z^4 - 787/15120*z^5 + O(z^6)
```

define(*s*)

Define an equation by `self = s`.

INPUT:

- `s` – a lazy series

EXAMPLES:

We begin by constructing the Catalan numbers:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: C = L.undefined()
sage: C.define(1 + z*C^2)
sage: C
1 + z + 2*z^2 + 5*z^3 + 14*z^4 + 42*z^5 + 132*z^6 + O(z^7)
sage: binomial(2000, 1000) / C[1000]                                       #␣
↪needs sage.symbolic
1001
```

The Catalan numbers but with a valuation 1:

```
sage: B = L.undefined(valuation=1)
sage: B.define(z + B^2)
sage: B
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)
```

We can define multiple series that are linked:

```
sage: s = L.undefined()
sage: t = L.undefined()
```

(continues on next page)

(continued from previous page)

```
sage: s.define(1 + z*t^3)
sage: t.define(1 + z*s^2)
sage: s[0:9]
[1, 1, 3, 9, 34, 132, 546, 2327, 10191]
sage: t[0:9]
[1, 1, 2, 7, 24, 95, 386, 1641, 7150]
```

A bigger example:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: A = L.undefined(valuation=5)
sage: B = L.undefined()
sage: C = L.undefined(valuation=2)
sage: A.define(z^5 + B^2)
sage: B.define(z^5 + C^2)
sage: C.define(z^2 + C^2 + A^2)
sage: A[0:15]
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2, 5, 4, 14, 10, 48]
sage: B[0:15]
[0, 0, 0, 0, 1, 1, 2, 0, 5, 0, 14, 0, 44, 0, 138]
sage: C[0:15]
[0, 0, 1, 0, 1, 0, 2, 0, 5, 0, 15, 0, 44, 2, 142]
```

Counting binary trees:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: s = L.undefined(valuation=1)
sage: s.define(z + (s^2+s(z^2))/2)
sage: s[0:9]
[0, 1, 1, 1, 2, 3, 6, 11, 23]
```

The q-Catalan numbers:

```
sage: R.<q> = ZZ[]
sage: L.<z> = LazyLaurentSeriesRing(R)
sage: s = L.undefined(valuation=0)
sage: s.define(1+z*s*s(q*z))
sage: s
1 + z + (q + 1)*z^2 + (q^3 + q^2 + 2*q + 1)*z^3
 + (q^6 + q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 3*q + 1)*z^4
 + (q^10 + q^9 + 2*q^8 + 3*q^7 + 5*q^6 + 5*q^5 + 7*q^4 + 7*q^3 + 6*q^2 + 4*q +␣
↪1)*z^5
 + (q^15 + q^14 + 2*q^13 + 3*q^12 + 5*q^11 + 7*q^10 + 9*q^9 + 11*q^8
    + 14*q^7 + 16*q^6 + 16*q^5 + 17*q^4 + 14*q^3 + 10*q^2 + 5*q + 1)*z^6 + O(z^
↪7)
```

We count unlabeled ordered trees by total number of nodes and number of internal nodes:

```
sage: R.<q> = QQ[]
sage: Q.<z> = LazyPowerSeriesRing(R)
sage: leaf = z
sage: internal_node = q * z
sage: L = Q(constant=1, degree=1)
```

(continues on next page)

(continued from previous page)

```
sage: T = Q.undefined(valuation=1)
sage: T.define(leaf + internal_node * L(T))
sage: T[0:6]
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

Similarly for Dirichlet series:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: g = L(constant=1, valuation=2)
sage: F = L.undefined()
sage: F.define(1 + g*F)
sage: F[:16]
[1, 1, 1, 2, 1, 3, 1, 4, 2, 3, 1, 8, 1, 3, 3]
sage: oeis(_)                                                # optional -
↪ internet
0: A002033: Number of perfect partitions of n.
1: A074206: Kalmár's [Kalmar's] problem: number of ordered factorizations of n.
...

sage: F = L.undefined()
sage: F.define(1 + g*F*F)
sage: F[:16]
[1, 1, 1, 3, 1, 5, 1, 10, 3, 5, 1, 24, 1, 5, 5]
```

We can compute the Frobenius character of unlabeled trees:

```
sage: # needs sage.combinat
sage: m = SymmetricFunctions(QQ).m()
sage: s = SymmetricFunctions(QQ).s()
sage: L = LazySymmetricFunctions(m)
sage: E = L(lambda n: s[n], valuation=0)
sage: X = L(s[1])
sage: A = L.undefined()
sage: A.define(X*E(A))
sage: A[:6]
[m[1],
 2*m[1, 1] + m[2],
 9*m[1, 1, 1] + 5*m[2, 1] + 2*m[3],
 64*m[1, 1, 1, 1] + 34*m[2, 1, 1] + 18*m[2, 2] + 13*m[3, 1] + 4*m[4],
 625*m[1, 1, 1, 1, 1] + 326*m[2, 1, 1, 1] + 171*m[2, 2, 1] + 119*m[3, 1, 1] +␣
↪63*m[3, 2] + 35*m[4, 1] + 9*m[5]]
```

euler()

Return the Euler function evaluated at `self`.

The *Euler function* is defined as

$$\phi(z) = (z; z)_\infty = \sum_{n=0}^{\infty} (-1)^n q^{(3n^2-n)/2}.$$

See also:

sage.rings.lazy_series_ring.LazyLaurentSeriesRing.euler()

EXAMPLES:

```
sage: L.<q> = LazyLaurentSeriesRing(ZZ)
sage: phi = L.euler()
sage: (q + q^2).euler() - phi(q + q^2)
O(q^7)
```

exp()

Return the exponential series of `self`.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(QQ, "s")
sage: Z = L(constant=1, valuation=2)
sage: exp(Z)                                                               #␣
↪needs sage.symbolic
1 + 1/(2^s) + 1/(3^s) + 3/2/4^s + 1/(5^s) + 2/6^s + 1/(7^s) + O(1/(8^s))
```

hypergeometric(*a*, *b*)

Return the ${}_pF_q$-hypergeometric function ${}_pF_q$ where (p, q) is the parameterization of `self`.

INPUT:

- `a` – the first parameter of the hypergeometric function
- `b` – the second parameter of the hypergeometric function

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: z.hypergeometric([1, 1], [1])
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + O(z^7)
sage: z.hypergeometric([], []) - exp(z)
O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: (x+y).hypergeometric([1, 1], [1]).polynomial(4)
x^4 + 4*x^3*y + 6*x^2*y^2 + 4*x*y^3 + y^4 + x^3 + 3*x^2*y
 + 3*x*y^2 + y^3 + x^2 + 2*x*y + y^2 + x + y + 1
```

is_nonzero(*proof=False*)

Return `True` if `self` is *known* to be nonzero.

INPUT:

- `proof` – (default: `False`) if `True`, this will also return an index such that `self` has a nonzero coefficient

Warning: If the stream is exactly zero, this will run forever.

EXAMPLES:

A series that it not known to be nonzero with no halting precision:

```
sage: L.<z> = LazyLaurentSeriesRing(GF(2))
sage: f = L(lambda n: 0, valuation=0)
sage: f.is_nonzero()
```

(continues on next page)

(continued from previous page)

```
False
sage: bool(f)
True
sage: g = L(lambda n: 0 if n < 50 else 1, valuation=2)
sage: g.is_nonzero()
False
sage: g[60]
1
sage: g.is_nonzero()
True
```

With finite halting precision, it can be considered to be indistinguishable from zero until possibly enough coefficients are computed:

```
sage: L.options.halting_precision = 20
sage: f = L(lambda n: 0, valuation=0)
sage: f.is_zero()
True

sage: g = L(lambda n: 0 if n < 50 else 1, valuation=2)
sage: g.is_nonzero()  # checks up to degree 22 = 2 + 20
False
sage: g.is_nonzero()  # checks up to degree 42 = 22 + 20
False
sage: g.is_nonzero()  # checks up to degree 62 = 42 + 20
True
sage: L.options._reset()
```

With a proof:

```
sage: L.<z> = LazyLaurentSeriesRing(GF(5))
sage: g = L(lambda n: 5 if n < 50 else 1, valuation=2)
sage: g.is_nonzero(proof=True)
(True, 50)

sage: L.zero().is_nonzero(proof=True)
(False, None)
```

is_trivial_zero()

Return whether `self` is known to be trivially zero.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = L(lambda n: 0, valuation=2)
sage: f.is_trivial_zero()
False

sage: L.zero().is_trivial_zero()
True
```

lift_to_precision(*absprec=None*)

Return another element of the same parent with absolute precision at least `absprec`, congruent to this element modulo the precision of this element.

Since the precision of a lazy series is infinity, this method returns the series itself, and the argument is ignored.

EXAMPLES:

```
sage: P.<t> = PowerSeriesRing(QQ, default_prec=2)
sage: R.<z> = LazyPowerSeriesRing(P)
sage: f = R(lambda n: 1/(1-t)^n)
sage: f
1 + ((1+t+O(t^2))*z) + ((1+2*t+O(t^2))*z^2)
  + ((1+3*t+O(t^2))*z^3)
  + ((1+4*t+O(t^2))*z^4)
  + ((1+5*t+O(t^2))*z^5)
  + ((1+6*t+O(t^2))*z^6) + O(z^7)
sage: f.lift_to_precision()
1 + ((1+t+O(t^2))*z) + ((1+2*t+O(t^2))*z^2)
  + ((1+3*t+O(t^2))*z^3)
  + ((1+4*t+O(t^2))*z^4)
  + ((1+5*t+O(t^2))*z^5)
  + ((1+6*t+O(t^2))*z^6) + O(z^7)
```

log()

Return the series for the natural logarithm of `self`.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(QQ, "s")
sage: Z = L(constant=1)
sage: log(Z)                                                              #␣
↪needs sage.symbolic
1/(2^s) + 1/(3^s) + 1/2/4^s + 1/(5^s) + 1/(7^s) + O(1/(8^s))
```

map_coefficients(*f*)

Return the series with `f` applied to each nonzero coefficient of `self`.

INPUT:

- `func` – function that takes in a coefficient and returns a new coefficient

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: m = L(lambda n: n, valuation=0); m
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: m.map_coefficients(lambda c: c + 1)
2*z + 3*z^2 + 4*z^3 + 5*z^4 + 6*z^5 + 7*z^6 + 8*z^7 + O(z^8)
```

Similarly for Dirichlet series:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: s = L(lambda n: n-1)
sage: s                                                                   #␣
↪needs sage.symbolic
1/(2^z) + 2/3^z + 3/4^z + 4/5^z + 5/6^z + 6/7^z + O(1/(8^z))
sage: ms = s.map_coefficients(lambda c: c + 1)                            #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
sage: ms                                                                  #␣
↪needs sage.symbolic
2/2^z + 3/3^z + 4/4^z + 5/5^z + 6/6^z + 7/7^z + 8/8^z + O(1/(9^z))
```

Similarly for multivariate power series:

```
sage: L.<x, y> = LazyPowerSeriesRing(QQ)
sage: f = 1/(1-(x+y)); f
1 + (x+y) + (x^2+2*x*y+y^2) + (x^3+3*x^2*y+3*x*y^2+y^3)
 + (x^4+4*x^3*y+6*x^2*y^2+4*x*y^3+y^4)
 + (x^5+5*x^4*y+10*x^3*y^2+10*x^2*y^3+5*x*y^4+y^5)
 + (x^6+6*x^5*y+15*x^4*y^2+20*x^3*y^3+15*x^2*y^4+6*x*y^5+y^6)
 + O(x,y)^7
sage: f.map_coefficients(lambda c: c^2)
1 + (x+y) + (x^2+4*x*y+y^2) + (x^3+9*x^2*y+9*x*y^2+y^3)
 + (x^4+16*x^3*y+36*x^2*y^2+16*x*y^3+y^4)
 + (x^5+25*x^4*y+100*x^3*y^2+100*x^2*y^3+25*x*y^4+y^5)
 + (x^6+36*x^5*y+225*x^4*y^2+400*x^3*y^3+225*x^2*y^4+36*x*y^5+y^6)
 + O(x,y)^7
```

Similarly for lazy symmetric functions:

```
sage: # needs sage.combinat
sage: p = SymmetricFunctions(QQ).p()
sage: L = LazySymmetricFunctions(p)
sage: f = 1/(1-2*L(p[1])); f
p[] + 2*p[1] + (4*p[1,1]) + (8*p[1,1,1]) + (16*p[1,1,1,1])
 + (32*p[1,1,1,1,1]) + (64*p[1,1,1,1,1,1]) + O^7
sage: f.map_coefficients(lambda c: log(c, 2))
p[1] + (2*p[1,1]) + (3*p[1,1,1]) + (4*p[1,1,1,1])
 + (5*p[1,1,1,1,1]) + (6*p[1,1,1,1,1,1]) + O^7
```

prec()

Return the precision of the series, which is infinity.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = 1/(1 - z)
sage: f.prec()
+Infinity
```

q_pochhammer(*q=None*)

Return the infinite q-Pochhammer symbol $(a; q)_\infty$, where a is `self`.

This is also one version of the quantum dilogarithm or the q-Exponential function.

See also:

sage.rings.lazy_series_ring.LazyLaurentSeriesRing.q_pochhammer()

INPUT:

- q – (default: $q \in \mathbf{Q}(q)$) the parameter q

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: L.<z> = LazyLaurentSeriesRing(q.parent())
sage: qp = L.q_pochhammer(q)
sage: (z + z^2).q_pochhammer(q) - qp(z + z^2)
O(z^7)
```

sec()

Return the secant of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: sec(z)
1 + 1/2*z^2 + 5/24*z^4 + 61/720*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: sec(x/(1-y)).polynomial(4)
5/24*x^4 + 3/2*x^2*y^2 + x^2*y + 1/2*x^2 + 1
```

sech()

Return the hyperbolic secant of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: sech(z)                                                          #␣
↪needs sage.libs.flint
1 - 1/2*z^2 + 5/24*z^4 - 61/720*z^6 + O(z^7)

sage: L.<x, y> = LazyPowerSeriesRing(QQ)
sage: sech(x/(1-y))                                                    #␣
↪needs sage.libs.flint
1 + (-1/2*x^2) + (-x^2*y) + (5/24*x^4-3/2*x^2*y^2) + (5/6*x^4*y-2*x^2*y^3)
 + (-61/720*x^6+25/12*x^4*y^2-5/2*x^2*y^4) + O(x,y)^7
```

set(*s*)

Define an equation by `self = s`.

INPUT:

- `s` – a lazy series

EXAMPLES:

We begin by constructing the Catalan numbers:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: C = L.undefined()
sage: C.define(1 + z*C^2)
sage: C
1 + z + 2*z^2 + 5*z^3 + 14*z^4 + 42*z^5 + 132*z^6 + O(z^7)
sage: binomial(2000, 1000) / C[1000]                                   #␣
↪needs sage.symbolic
1001
```

The Catalan numbers but with a valuation 1:

```
sage: B = L.undefined(valuation=1)
sage: B.define(z + B^2)
sage: B
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)
```

We can define multiple series that are linked:

```
sage: s = L.undefined()
sage: t = L.undefined()
sage: s.define(1 + z*t^3)
sage: t.define(1 + z*s^2)
sage: s[0:9]
[1, 1, 3, 9, 34, 132, 546, 2327, 10191]
sage: t[0:9]
[1, 1, 2, 7, 24, 95, 386, 1641, 7150]
```

A bigger example:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: A = L.undefined(valuation=5)
sage: B = L.undefined()
sage: C = L.undefined(valuation=2)
sage: A.define(z^5 + B^2)
sage: B.define(z^5 + C^2)
sage: C.define(z^2 + C^2 + A^2)
sage: A[0:15]
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2, 5, 4, 14, 10, 48]
sage: B[0:15]
[0, 0, 0, 0, 1, 1, 2, 0, 5, 0, 14, 0, 44, 0, 138]
sage: C[0:15]
[0, 0, 1, 0, 1, 0, 2, 0, 5, 0, 15, 0, 44, 2, 142]
```

Counting binary trees:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: s = L.undefined(valuation=1)
sage: s.define(z + (s^2+s(z^2))/2)
sage: s[0:9]
[0, 1, 1, 1, 2, 3, 6, 11, 23]
```

The q-Catalan numbers:

```
sage: R.<q> = ZZ[]
sage: L.<z> = LazyLaurentSeriesRing(R)
sage: s = L.undefined(valuation=0)
sage: s.define(1+z*s*s(q*z))
sage: s
1 + z + (q + 1)*z^2 + (q^3 + q^2 + 2*q + 1)*z^3
 + (q^6 + q^5 + 2*q^4 + 3*q^3 + 3*q^2 + 3*q + 1)*z^4
 + (q^10 + q^9 + 2*q^8 + 3*q^7 + 5*q^6 + 5*q^5 + 7*q^4 + 7*q^3 + 6*q^2 + 4*q +␣
↪1)*z^5
 + (q^15 + q^14 + 2*q^13 + 3*q^12 + 5*q^11 + 7*q^10 + 9*q^9 + 11*q^8
    + 14*q^7 + 16*q^6 + 16*q^5 + 17*q^4 + 14*q^3 + 10*q^2 + 5*q + 1)*z^6 + O(z^
↪7)
```

We count unlabeled ordered trees by total number of nodes and number of internal nodes:

```
sage: R.<q> = QQ[]
sage: Q.<z> = LazyPowerSeriesRing(R)
sage: leaf = z
sage: internal_node = q * z
sage: L = Q(constant=1, degree=1)
sage: T = Q.undefined(valuation=1)
sage: T.define(leaf + internal_node * L(T))
sage: T[0:6]
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

Similarly for Dirichlet series:

```
sage: L = LazyDirichletSeriesRing(ZZ, "z")
sage: g = L(constant=1, valuation=2)
sage: F = L.undefined()
sage: F.define(1 + g*F)
sage: F[:16]
[1, 1, 1, 2, 1, 3, 1, 4, 2, 3, 1, 8, 1, 3, 3]
sage: oeis(_)                                                        # optional -
↪ internet
0: A002033: Number of perfect partitions of n.
1: A074206: Kalmár's [Kalmar's] problem: number of ordered factorizations of n.
...

sage: F = L.undefined()
sage: F.define(1 + g*F*F)
sage: F[:16]
[1, 1, 1, 3, 1, 5, 1, 10, 3, 5, 1, 24, 1, 5, 5]
```

We can compute the Frobenius character of unlabeled trees:

```
sage: # needs sage.combinat
sage: m = SymmetricFunctions(QQ).m()
sage: s = SymmetricFunctions(QQ).s()
sage: L = LazySymmetricFunctions(m)
sage: E = L(lambda n: s[n], valuation=0)
sage: X = L(s[1])
sage: A = L.undefined()
sage: A.define(X*E(A))
sage: A[:6]
[m[1],
 2*m[1, 1] + m[2],
 9*m[1, 1, 1] + 5*m[2, 1] + 2*m[3],
 64*m[1, 1, 1, 1] + 34*m[2, 1, 1] + 18*m[2, 2] + 13*m[3, 1] + 4*m[4],
 625*m[1, 1, 1, 1, 1] + 326*m[2, 1, 1, 1] + 171*m[2, 2, 1] + 119*m[3, 1, 1] +␣
↪63*m[3, 2] + 35*m[4, 1] + 9*m[5]]
```

shift(*n*)

Return `self` with the indices shifted by `n`.

For example, a Laurent series is multiplied by the power z^n, where z is the variable of `self`. For series with a fixed minimal valuation (e.g., power series), this removes any terms that are less than the minimal valuation.

INPUT:

- n – the amount to shift

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = 1 / (1 + 2*z)
sage: f
1 - 2*z + 4*z^2 - 8*z^3 + 16*z^4 - 32*z^5 + 64*z^6 + O(z^7)
sage: f.shift(3)
z^3 - 2*z^4 + 4*z^5 - 8*z^6 + 16*z^7 - 32*z^8 + 64*z^9 + O(z^10)
sage: f << -3  # shorthand
z^-3 - 2*z^-2 + 4*z^-1 - 8 + 16*z - 32*z^2 + 64*z^3 + O(z^4)
sage: g = z^-3 + 3 + z^2
sage: g.shift(5)
z^2 + 3*z^5 + z^7
sage: L([2,0,3], valuation=2, degree=7, constant=1) << -2
2 + 3*z^2 + z^5 + z^6 + z^7 + O(z^8)

sage: D = LazyDirichletSeriesRing(QQ, 't')
sage: f = D([0,1,2])
sage: f                                                                 #␣
↪needs sage.symbolic
1/(2^t) + 2/3^t
sage: sf = f.shift(3)
sage: sf                                                                #␣
↪needs sage.symbolic
1/(5^t) + 2/6^t
```

Examples with power series (where the minimal valuation is 0):

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: f = 1 / (1 - x)
sage: f.shift(2)
x^2 + x^3 + x^4 + O(x^5)
sage: g = f.shift(-1); g
1 + x + x^2 + O(x^3)
sage: f == g
True
sage: g[-1]
0
sage: h = L(lambda n: 1)
sage: LazyPowerSeriesRing.options.halting_precision(20)  # verify up to degree␣
↪20
sage: f == h
True
sage: h == f
True
sage: h.shift(-1) == h
True
sage: LazyPowerSeriesRing.options._reset()

sage: fp = L([3,3,3], constant=1)
sage: fp.shift(2)
```

(continues on next page)

(continued from previous page)

```
3*x^2 + 3*x^3 + 3*x^4 + x^5 + x^6 + x^7 + O(x^8)
sage: fp.shift(-2)
3 + x + x^2 + x^3 + O(x^4)
sage: fp.shift(-7)
1 + x + x^2 + O(x^3)
sage: fp.shift(-5) == g
True
```

We compare the shifting with converting to the fraction field (see also github issue #35293):

```
sage: M = L.fraction_field()
sage: f = L([1,2,3,4]); f
1 + 2*x + 3*x^2 + 4*x^3
sage: f.shift(-3)
4
sage: M(f).shift(-3)
x^-3 + 2*x^-2 + 3*x^-1 + 4
```

An example with a more general function:

```
sage: fun = lambda n: 1 if ZZ(n).is_power_of(2) else 0
sage: f = L(fun); f
x + x^2 + x^4 + O(x^7)
sage: fs = f.shift(-4)
sage: fs
1 + x^4 + O(x^7)
sage: fs.shift(4)
x^4 + x^8 + O(x^11)
sage: M(f).shift(-4)
x^-3 + x^-2 + 1 + O(x^4)
```

sin()

Return the sine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: sin(z)
z - 1/6*z^3 + 1/120*z^5 - 1/5040*z^7 + O(z^8)

sage: sin(1 + z)
Traceback (most recent call last):
...
ValueError: can only compose with a positive valuation series

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: sin(x/(1-y)).polynomial(3)
-1/6*x^3 + x*y^2 + x*y + x
```

sinh()

Return the hyperbolic sine of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: sinh(z)
z + 1/6*z^3 + 1/120*z^5 + 1/5040*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: sinh(x/(1-y))
x + x*y + (1/6*x^3+x*y^2) + (1/2*x^3*y+x*y^3)
 + (1/120*x^5+x^3*y^2+x*y^4) + (1/24*x^5*y+5/3*x^3*y^3+x*y^5)
 + (1/5040*x^7+1/8*x^5*y^2+5/2*x^3*y^4+x*y^6) + O(x,y)^8
```

sqrt()

Return `self^(1/2)`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: sqrt(1+z)
1 + 1/2*z - 1/8*z^2 + 1/16*z^3 - 5/128*z^4 + 7/256*z^5 - 21/1024*z^6 + O(z^7)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: sqrt(1+x/(1-y))
1 + 1/2*x + (-1/8*x^2+1/2*x*y) + (1/16*x^3-1/4*x^2*y+1/2*x*y^2)
 + (-5/128*x^4+3/16*x^3*y-3/8*x^2*y^2+1/2*x*y^3)
 + (7/256*x^5-5/32*x^4*y+3/8*x^3*y^2-1/2*x^2*y^3+1/2*x*y^4)
 + (-21/1024*x^6+35/256*x^5*y-25/64*x^4*y^2+5/8*x^3*y^3-5/8*x^2*y^4+1/2*x*y^5)
 + O(x,y)^7
```

This also works for Dirichlet series:

```
sage: # needs sage.symbolic
sage: D = LazyDirichletSeriesRing(SR, "s")
sage: Z = D(constant=1)
sage: f = sqrt(Z);  f
1 + 1/2/2^s + 1/2/3^s + 3/8/4^s + 1/2/5^s + 1/4/6^s + 1/2/7^s + O(1/(8^s))
sage: f*f - Z
O(1/(8^s))
```

tan()

Return the tangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: tan(z)
z + 1/3*z^3 + 2/15*z^5 + 17/315*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: tan(x/(1-y)).polynomial(5)
2/15*x^5 + 2*x^3*y^2 + x*y^4 + x^3*y + x*y^3 + 1/3*x^3 + x*y^2 + x*y + x
```

tanh()

Return the hyperbolic tangent of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: tanh(z)                                                          #␣
↪needs sage.libs.flint
z - 1/3*z^3 + 2/15*z^5 - 17/315*z^7 + O(z^8)

sage: L.<x,y> = LazyPowerSeriesRing(QQ)
sage: tanh(x/(1-y))                                                    #␣
↪needs sage.libs.flint
x + x*y + (-1/3*x^3+x*y^2) + (-x^3*y+x*y^3) + (2/15*x^5-2*x^3*y^2+x*y^4)
 + (2/3*x^5*y-10/3*x^3*y^3+x*y^5) + (-17/315*x^7+2*x^5*y^2-5*x^3*y^4+x*y^6) +␣
↪O(x,y)^8
```

truncate(*d*)

Return the series obtained by removing all terms of degree at least d.

INPUT:

- d – integer; the degree from which the series is truncated

EXAMPLES:

Dense implementation:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=False)
sage: alpha = 1/(1-z)
sage: alpha
1 + z + z^2 + O(z^3)
sage: beta = alpha.truncate(5)
sage: beta
1 + z + z^2 + z^3 + z^4
sage: alpha - beta
z^5 + z^6 + z^7 + O(z^8)
sage: M = L(lambda n: n, valuation=0); M
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: M.truncate(4)
z + 2*z^2 + 3*z^3
```

Sparse Implementation:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=True)
sage: M = L(lambda n: n, valuation=0); M
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: M.truncate(4)
z + 2*z^2 + 3*z^3
```

Series which are known to be exact can also be truncated:

```
sage: M = z + z^2 + z^3 + z^4
sage: M.truncate(4)
z + z^2 + z^3
```

class sage.rings.lazy_series.**LazyPowerSeries**(*parent*, *coeff_stream*)

Bases: *LazyCauchyProductSeries*

A Taylor series where the coefficients are computed lazily.

EXAMPLES:

```
sage: L.<x, y> = LazyPowerSeriesRing(ZZ)
sage: f = 1 / (1 - x^2 + y^3); f
1 + x^2 + (-y^3) + x^4 + (-2*x^2*y^3) + (x^6+y^6) + O(x,y)^7
sage: P.<x, y> = PowerSeriesRing(ZZ, default_prec=101)
sage: g = 1 / (1 - x^2 + y^3); f[100] - g[100]
0
```

Lazy Taylor series is picklable:

```
sage: g = loads(dumps(f))
sage: g
1 + x^2 + (-y^3) + x^4 + (-2*x^2*y^3) + (x^6+y^6) + O(x,y)^7
sage: g == f
True
```

compose(**g*)

Return the composition of `self` with `g`.

The arity of `self` must be equal to the number of arguments provided.

Given a Taylor series f of arity n and a tuple of Taylor series $g = (g_1, \ldots, g_n)$ over the same base ring, the composition $f \circ g$ is defined if and only if for each $1 \leq i \leq n$:

- g_i is zero, or
- setting all variables except the i-th in f to zero yields a polynomial, or
- $\mathrm{val}(g_i) > 0$.

If f is a univariate 'exact' series, we can check whether f is a actually a polynomial. However, if f is a multivariate series, we have no way to test whether setting all but one variable of f to zero yields a polynomial, except if f itself is 'exact' and therefore a polynomial.

INPUT:

- `g` – other series, all can be coerced into the same parent

EXAMPLES:

```
sage: L.<x, y, z> = LazyPowerSeriesRing(QQ)
sage: M.<a, b> = LazyPowerSeriesRing(ZZ)
sage: g1 = 1 / (1 - x)
sage: g2 = x + y^2
sage: p = a^2 + b + 1
sage: p(g1, g2) - g1^2 - g2 - 1
O(x,y,z)^7
```

The number of mappings from a set with m elements to a set with n elements:

```
sage: M.<a> = LazyPowerSeriesRing(QQ)
sage: Ea = M(lambda n: 1/factorial(n))
sage: Ex = L(lambda n: 1/factorial(n)*x^n)
sage: Ea(Ex*y)[5]
1/24*x^4*y + 2/3*x^3*y^2 + 3/4*x^2*y^3 + 1/6*x*y^4 + 1/120*y^5
```

So, there are $3!2!2/3 = 8$ mappings from a three element set to a two element set.

We perform the composition with a lazy Laurent series:

```
sage: N.<w> = LazyLaurentSeriesRing(QQ)
sage: f1 = 1 / (1 - w)
sage: f2 = cot(w / (1 - w))
sage: p(f1, f2)
w^-1 + 1 + 5/3*w + 8/3*w^2 + 164/45*w^3 + 23/5*w^4 + 5227/945*w^5 + O(w^6)
```

We perform the composition with a lazy Dirichlet series:

```
sage: # needs sage.symbolic
sage: D = LazyDirichletSeriesRing(QQ, "s")
sage: g = D(constant=1)-1
sage: g
1/(2^s) + 1/(3^s) + 1/(4^s) + O(1/(5^s))
sage: f = 1 / (1 - x - y*z); f
1 + x + (x^2+y*z) + (x^3+2*x*y*z) + (x^4+3*x^2*y*z+y^2*z^2)
 + (x^5+4*x^3*y*z+3*x*y^2*z^2)
 + (x^6+5*x^4*y*z+6*x^2*y^2*z^2+y^3*z^3)
 + O(x,y,z)^7
sage: fog = f(g, g, g)
sage: fog
1 + 1/(2^s) + 1/(3^s) + 3/4^s + 1/(5^s) + 5/6^s + O(1/(7^s))
sage: fg = 1 / (1 - g - g*g)
sage: fg
1 + 1/(2^s) + 1/(3^s) + 3/4^s + 1/(5^s) + 5/6^s + 1/(7^s) + O(1/(8^s))
sage: fog - fg
O(1/(8^s))

sage: f = 1 / (1 - 2*a)
sage: f(g)                                                              #␣
↪needs sage.symbolic
1 + 2/2^s + 2/3^s + 6/4^s + 2/5^s + 10/6^s + 2/7^s + O(1/(8^s))
sage: 1 / (1 - 2*g)                                                     #␣
↪needs sage.symbolic
1 + 2/2^s + 2/3^s + 6/4^s + 2/5^s + 10/6^s + 2/7^s + O(1/(8^s))
```

The output parent is always the common parent between the base ring of f and the parent of g or extended to the corresponding lazy series:

```
sage: T.<x,y> = LazyPowerSeriesRing(QQ)
sage: R.<a,b,c> = ZZ[]
sage: S.<v> = R[]
sage: L.<z> = LaurentPolynomialRing(ZZ)
sage: parent(x(a, b))
Multivariate Polynomial Ring in a, b, c over Rational Field
sage: parent(x(CC(2), a))
Multivariate Polynomial Ring in a, b, c over Complex Field with 53 bits of␣
↪precision
sage: parent(x(0, 0))
Rational Field
sage: f = 1 / (1 - x - y); f
1 + (x+y) + (x^2+2*x*y+y^2) + (x^3+3*x^2*y+3*x*y^2+y^3)
 + (x^4+4*x^3*y+6*x^2*y^2+4*x*y^3+y^4)
 + (x^5+5*x^4*y+10*x^3*y^2+10*x^2*y^3+5*x*y^4+y^5)
```

(continues on next page)

(continued from previous page)

```
 + (x^6+6*x^5*y+15*x^4*y^2+20*x^3*y^3+15*x^2*y^4+6*x*y^5+y^6)
 + O(x,y)^7
sage: f(a^2, b*c)
1 + (a^2+b*c) + (a^4+2*a^2*b*c+b^2*c^2) + (a^6+3*a^4*b*c+3*a^2*b^2*c^2+b^3*c^3)␣
↪+ O(a,b,c)^7
sage: f(v, v^2)
1 + v + 2*v^2 + 3*v^3 + 5*v^4 + 8*v^5 + 13*v^6 + O(v^7)
sage: f(z, z^2 + z)
1 + 2*z + 5*z^2 + 12*z^3 + 29*z^4 + 70*z^5 + 169*z^6 + O(z^7)
sage: three = T(3)(a^2, b); three
3
sage: parent(three)
Multivariate Polynomial Ring in a, b, c over Rational Field
```

compositional_inverse()

Return the compositional inverse of `self`.

Given a Taylor series f in one variable, the compositional inverse is a power series g over the same base ring, such that $(f \circ g)(z) = f(g(z)) = z$.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bz$ with $a, b \neq 0$.

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: (2*z).revert()
1/2*z
sage: (z-z^2).revert()
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)

sage: s = L(degree=1, constant=-1)
sage: s.revert()
-z - z^2 - z^3 + O(z^4)

sage: s = L(degree=1, constant=1)
sage: s.revert()
z - z^2 + z^3 - z^4 + z^5 - z^6 + z^7 + O(z^8)
```

Warning: For series not known to be eventually constant (e.g., being defined by a function) with approximate valuation ≤ 1 (but not necessarily its true valuation), this assumes that this is the actual valuation:

```
sage: f = L(lambda n: n if n > 2 else 0)
sage: f.revert()
<repr(... failed: ValueError: generator already executing>
```

compute_coefficients(*i*)

Computes all the coefficients of `self` up to `i`.

This method is deprecated, it has no effect anymore.

derivative(**args*)

Return the derivative of the Taylor series.

Multiple variables and iteration counts may be supplied; see the documentation of `sage.calculus.functional.derivative()` function for details.

EXAMPLES:

```
sage: T.<z> = LazyPowerSeriesRing(ZZ)
sage: z.derivative()
1
sage: (1+z+z^2).derivative(3)
0
sage: (1/(1-z)).derivative()
1 + 2*z + 3*z^2 + 4*z^3 + 5*z^4 + 6*z^5 + 7*z^6 + O(z^7)

sage: R.<q> = QQ[]
sage: L.<x, y> = LazyPowerSeriesRing(R)
sage: f = 1/(1-q*x+y); f
1 + (q*x-y) + (q^2*x^2+(-2*q)*x*y+y^2)
 + (q^3*x^3+(-3*q^2)*x^2*y+3*q*x*y^2-y^3)
 + (q^4*x^4+(-4*q^3)*x^3*y+6*q^2*x^2*y^2+(-4*q)*x*y^3+y^4)
 + (q^5*x^5+(-5*q^4)*x^4*y+10*q^3*x^3*y^2+(-10*q^2)*x^2*y^3+5*q*x*y^4-y^5)
 + (q^6*x^6+(-6*q^5)*x^5*y+15*q^4*x^4*y^2+(-20*q^3)*x^3*y^3+15*q^2*x^2*y^4+(-
↪6*q)*x*y^5+y^6)
 + O(x,y)^7
sage: f.derivative(q)
x + (2*q*x^2+(-2)*x*y) + (3*q^2*x^3+(-6*q)*x^2*y+3*x*y^2)
 + (4*q^3*x^4+(-12*q^2)*x^3*y+12*q*x^2*y^2+(-4)*x*y^3)
 + (5*q^4*x^5+(-20*q^3)*x^4*y+30*q^2*x^3*y^2+(-20*q)*x^2*y^3+5*x*y^4)
 + (6*q^5*x^6+(-30*q^4)*x^5*y+60*q^3*x^4*y^2+(-60*q^2)*x^3*y^3+30*q*x^2*y^4+(-
↪6)*x*y^5)
 + O(x,y)^7
```

exponential()

Return the exponential series of `self`.

This method is deprecated, use `exp()` instead.

is_unit()

Return whether this element is a unit in the ring.

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: (2*z).is_unit()
False

sage: (1 + 2*z).is_unit()
True

sage: (3 + 2*z).is_unit()
False

sage: L.<x,y> = LazyPowerSeriesRing(ZZ)
```

(continues on next page)

(continued from previous page)

```
sage: (-1 + 2*x + 3*x*y).is_unit()
True
```

polynomial(*degree=None*, *names=None*)

Return `self` as a polynomial if `self` is actually so.

INPUT:

- `degree` – `None` or an integer
- `names` – names of the variables; if it is `None`, the name of the variables of the series is used

OUTPUT:

If `degree` is not `None`, the terms of the series of degree greater than `degree` are first truncated. If `degree` is `None` and the series is not a polynomial polynomial, a `ValueError` is raised.

EXAMPLES:

```
sage: L.<x,y> = LazyPowerSeriesRing(ZZ)
sage: f = x^2 + y*x - x + 2; f
2 + (-x) + (x^2+x*y)
sage: f.polynomial()
x^2 + x*y - x + 2
```

revert()

Return the compositional inverse of `self`.

Given a Taylor series f in one variable, the compositional inverse is a power series g over the same base ring, such that $(f \circ g)(z) = f(g(z)) = z$.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bz$ with $a, b \neq 0$.

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: (2*z).revert()
1/2*z
sage: (z-z^2).revert()
z + z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 + 132*z^7 + O(z^8)

sage: s = L(degree=1, constant=-1)
sage: s.revert()
-z - z^2 - z^3 + O(z^4)

sage: s = L(degree=1, constant=1)
sage: s.revert()
z - z^2 + z^3 - z^4 + z^5 - z^6 + z^7 + O(z^8)
```

Warning: For series not known to be eventually constant (e.g., being defined by a function) with approximate valuation ≤ 1 (but not necessarily its true valuation), this assumes that this is the actual valuation:

```
sage: f = L(lambda n: n if n > 2 else 0)
sage: f.revert()
<repr(... failed: ValueError: generator already executing>
```

class sage.rings.lazy_series.**LazyPowerSeries_gcd_mixin**

Bases: object

A lazy power series that also implements the GCD algorithm.

gcd(*other*)

Return the greatest common divisor of self and other.

EXAMPLES:

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: a = 16*x^5 / (1 - 5*x)
sage: b = (22*x^2 + x^8) / (1 - 4*x^2)
sage: a.gcd(b)
x^2
```

xgcd(*f*)

Return the extended gcd of self and f.

OUTPUT:

A triple (g, s, t) such that g is the gcd of self and f, and s and t are cofactors satisfying the Bezout identity

$$g = s \cdot \mathrm{self} + t \cdot f.$$

EXAMPLES:

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: a = 16*x^5 / (1 - 2*x)
sage: b = (22*x^3 + x^8) / (1 - 3*x^2)
sage: g, s, t = a.xgcd(b)
sage: g
x^3
sage: s
1/22 - 41/242*x^2 - 8/121*x^3 + 120/1331*x^4 + 1205/5324*x^5 + 316/14641*x^6 +␣
↪O(x^7)
sage: t
1/22 - 41/242*x^2 - 8/121*x^3 + 120/1331*x^4 + 1205/5324*x^5 + 316/14641*x^6 +␣
↪O(x^7)

sage: LazyPowerSeriesRing.options.halting_precision(20)  # verify up to degree␣
↪20

sage: g == s * a + t * b
True

sage: a = 16*x^5 / (1 - 2*x)
sage: b = (-16*x^5 + x^8) / (1 - 3*x^2)
sage: g, s, t = a.xgcd(b)
```

(continues on next page)

(continued from previous page)

```
sage: g
x^5
sage: s
1/16 - 1/16*x - 3/16*x^2 + 1/8*x^3 - 17/256*x^4 + 9/128*x^5 + 1/128*x^6 + O(x^7)
sage: t
1/16*x - 1/16*x^2 - 3/16*x^3 + 1/8*x^4 - 17/256*x^5 + 9/128*x^6 + 1/128*x^7 +␣
↪O(x^8)
sage: g == s * a + t * b
True

sage: # needs sage.rings.finite_rings
sage: L.<x> = LazyPowerSeriesRing(GF(2))
sage: a = L(lambda n: n % 2, valuation=3); a
x^3 + x^5 + x^7 + x^9 + O(x^10)
sage: b = L(lambda n: binomial(n,2) % 2, valuation=3); b
x^3 + x^6 + x^7 + O(x^10)
sage: g, s, t = a.xgcd(b)
sage: g
x^3
sage: s
1 + x + x^3 + x^4 + x^5 + O(x^7)
sage: t
x + x^2 + x^4 + x^5 + x^6 + O(x^8)
sage: g == s * a + t * b
True

sage: LazyPowerSeriesRing.options._reset()  # reset the options
```

class sage.rings.lazy_series.**LazySymmetricFunction**(*parent*, *coeff_stream*)

Bases: *LazyCompletionGradedAlgebraElement*

A symmetric function where each degree is computed lazily.

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()                                          #␣
↪needs sage.modules
sage: L = LazySymmetricFunctions(s)                                           #␣
↪needs sage.modules
```

arithmetic_product(**args*)

Return the arithmetic product of `self` with `g`.

The arithmetic product is a binary operation $\boxdot$ on the ring of symmetric functions which is bilinear in its two arguments and satisfies

$$p_\lambda \boxdot p_\mu = \prod_{i\geq 1, j\geq 1} p_{\mathrm{lcm}(\lambda_i,\mu_j)}^{\mathrm{gcd}(\lambda_i,\mu_j)}$$

for any two partitions $\lambda = (\lambda_1, \lambda_2, \lambda_3, \dots)$ and $\mu = (\mu_1, \mu_2, \mu_3, \dots)$ (where p_ν denotes the power-sum symmetric function indexed by the partition ν, and p_i denotes the i-th power-sum symmetric function). This is enough to define the arithmetic product if the base ring is torsion-free as a $\mathbf{Z}$-module; for all other cases the arithmetic product is uniquely determined by requiring it to be functorial in the base ring. See http://mathoverflow.net/questions/138148/ for a discussion of this arithmetic product.

Warning: The operation $f \boxdot g$ was originally defined only for symmetric functions f and g without constant term. We extend this definition using the convention that the least common multiple of any integer with 0 is 0.

If f and g are two symmetric functions which are homogeneous of degrees a and b, respectively, then $f \boxdot g$ is homogeneous of degree ab.

The arithmetic product is commutative and associative and has unity $e_1 = p_1 = h_1$.

For species M and N such that $M[\varnothing] = N[\varnothing] = \varnothing$, their arithmetic product is the species $M \boxdot N$ of "M-assemblies of cloned N-structures". This operation is defined and several examples are given in [MM2008].

INPUT:

- g – a cycle index series having the same parent as `self`

OUTPUT:

The arithmetic product of `self` with `g`.

See also:

`sage.combinat.sf.sfa.SymmetricFunctionAlgebra_generic_Element.arithmetic_product()`

EXAMPLES:

For C the species of (oriented) cycles and L_+ the species of nonempty linear orders, $C \boxdot L_+$ corresponds to the species of "regular octopuses"; a $(C \boxdot L_+)$-structure is a cycle of some length, each of whose elements is an ordered list of a length which is consistent for all the lists in the structure.

```
sage: R.<q> = QQ[]
sage: p = SymmetricFunctions(R).p()                                          #␣
↪needs sage.modules
sage: m = SymmetricFunctions(R).m()                                          #␣
↪needs sage.modules
sage: L = LazySymmetricFunctions(m)                                          #␣
↪needs sage.modules

sage: # needs sage.modules
sage: C = species.CycleSpecies().cycle_index_series()
sage: c = L(lambda n: C[n])
sage: Lplus = L(lambda n: p([1]*n), valuation=1)
sage: r = c.arithmetic_product(Lplus); r                                     #␣
↪needs sage.libs.pari
m[1] + (3*m[1,1]+2*m[2])
 + (8*m[1,1,1]+4*m[2,1]+2*m[3])
 + (42*m[1,1,1,1]+21*m[2,1,1]+12*m[2,2]+7*m[3,1]+3*m[4])
 + (144*m[1,1,1,1,1]+72*m[2,1,1,1]+36*m[2,2,1]+24*m[3,1,1]+12*m[3,2]+6*m[4,
↪1]+2*m[5])
 + ...
 + O^7
```

In particular, the number of regular octopuses is:

```
sage: [r[n].coefficient([1]*n) for n in range(8)]                    #␣
↪needs sage.libs.pari sage.modules
[0, 1, 3, 8, 42, 144, 1440, 5760]
```

It is shown in [MM2008] that the exponential generating function for regular octopuses satisfies $(C \boxdot L_+)(x) = \sum_{n \geq 1} \sigma(n)(n-1)! \frac{x^n}{n!}$ (where $\sigma(n)$ is the sum of the divisors of n).

```
sage: [sum(divisors(i))*factorial(i-1) for i in range(1,8)]          #␣
↪needs sage.modules
[1, 3, 8, 42, 144, 1440, 5760]
```

AUTHORS:

- Andrew Gainer-Dewar (2013)

REFERENCES:

- [MM2008]

compositional_inverse()

Return the compositional inverse of `self`.

Given a symmetric function f, the compositional inverse is a symmetric function g over the same base ring, such that $f \circ g = p_1$. Thus, it is the inverse with respect to plethystic substitution.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bp_1$ with $a, b \neq 0$.

EXAMPLES:

```
sage: # needs sage.modules
sage: h = SymmetricFunctions(QQ).h()
sage: L = LazySymmetricFunctions(h)
sage: f = L(lambda n: h[n]) - 1
sage: f(f.revert())
h[1] + O^8
```

ALGORITHM:

Let F be a symmetric function with valuation 1, i.e., whose constant term vanishes and whose degree one term equals bp_1. Then

$$(F - bp_1) \circ G = F \circ G - bp_1 \circ G = p_1 - bG,$$

and therefore $G = (p_1 - (F - bp_1) \circ G)/b$, which allows recursive computation of G.

See also:

The compositional inverse Ω of the symmetric function $h_1 + h_2 + \ldots$ can be handled much more efficiently using specialized methods. See `LogarithmCycleIndexSeries()`

AUTHORS:

- Andrew Gainer-Dewar
- Martin Rubey

derivative_with_respect_to_p1(*n=1*)

Return the symmetric function obtained by taking the derivative of `self` with respect to the power-sum symmetric function p_1 when the expansion of `self` in the power-sum basis is considered as a polynomial in p_k's (with $k \geq 1$).

This is the same as skewing `self` by the first power-sum symmetric function p_1.

INPUT:

- `n` – (default: 1) nonnegative integer which determines which power of the derivative is taken

EXAMPLES:

The species E of sets satisfies the relationship $E' = E$:

```
sage: # needs sage.modules
sage: h = SymmetricFunctions(QQ).h()
sage: T = LazySymmetricFunctions(h)
sage: E = T(lambda n: h[n])
sage: E - E.derivative_with_respect_to_p1()
O^6
```

The species C of cyclic orderings and the species L of linear orderings satisfy the relationship $C' = L$:

```
sage: # needs sage.modules
sage: p = SymmetricFunctions(QQ).p()
sage: C = T(lambda n: (sum(euler_phi(k)*p([k])**(n//k)
....:                         for k in divisors(n))/n if n > 0 else 0))
sage: L = T(lambda n: p([1]*n))
sage: L - C.derivative_with_respect_to_p1()                                  #␣
↪needs sage.libs.pari
O^6
```

functorial_composition(**args*)

Return the functorial composition of `self` and `g`.

Let X be a finite set of cardinality m. For a group action of the symmetric group $g : S_n \to S_X$ and a (possibly virtual) representation of the symmetric group on X, $f : S_X \to GL(V)$, the functorial composition is the (virtual) representation of the symmetric group $f \Box g : S_n \to GL(V)$ given by $\sigma \mapsto f(g(\sigma))$.

This is more naturally phrased in the language of combinatorial species. Let F and G be species, then their functorial composition is the species $F \Box G$ with $(F \Box G)[A] = F[G[A]]$. In other words, an $(F \Box G)$-structure on a set A of labels is an F-structure whose labels are the set of all G-structures on A.

The Frobenius character (or cycle index series) of $F \Box G$ can be computed as follows, see section 2.2 of [BLL1998]):

$$\sum_{n \geq 0} \frac{1}{n!} \sum_{\sigma \in \mathfrak{S}_n} \operatorname{fix} F[(G[\sigma])_1, (G[\sigma])_2, \ldots] \, p_1^{\sigma_1} p_2^{\sigma_2} \cdots .$$

Warning: The operation $f \Box g$ only makes sense when g corresponds to a permutation representation, i.e., a group action.

EXAMPLES:

The species G of simple graphs can be expressed in terms of a functorial composition: $G = \mathfrak{p} \Box \mathfrak{p}_2$, where $\mathfrak{p}$ is the `SubsetSpecies`.:

```
sage: # needs sage.modules
sage: R.<q> = QQ[]
sage: h = SymmetricFunctions(R).h()
sage: m = SymmetricFunctions(R).m()
sage: L = LazySymmetricFunctions(m)
sage: P = L(lambda n: sum(q^k*h[n-k]*h[k] for k in range(n+1)))
sage: P2 = L(lambda n: h[2]*h[n-2], valuation=2)
sage: P.functorial_composition(P2)[:4]                                   #␣
↪needs sage.libs.pari
[m[],
 m[1],
 (q+1)*m[1, 1] + (q+1)*m[2],
 (q^3+3*q^2+3*q+1)*m[1, 1, 1] + (q^3+2*q^2+2*q+1)*m[2, 1] + (q^3+q^2+q+1)*m[3]]
```

For example, there are:

```
sage: P.functorial_composition(P2)[4].coefficient([4])[3]                #␣
↪needs sage.libs.pari sage.modules
3
```

unlabelled graphs on 4 vertices and 3 edges, and:

```
sage: P.functorial_composition(P2)[4].coefficient([2,2])[3]              #␣
↪needs sage.libs.pari sage.modules
8
```

labellings of their vertices with two 1's and two 2's.

The symmetric function $h_1 \sum_n h_n$ is the neutral element with respect to functorial composition:

```
sage: # needs sage.modules
sage: p = SymmetricFunctions(QQ).p()
sage: h = SymmetricFunctions(QQ).h()
sage: e = SymmetricFunctions(QQ).e()
sage: L = LazySymmetricFunctions(h)
sage: H = L(lambda n: h[n])
sage: Ep = p[1]*H.derivative_with_respect_to_p1(); Ep
h[1] + (h[1,1]) + (h[2,1]) + (h[3,1]) + (h[4,1]) + (h[5,1]) + O^7
sage: f = L(lambda n: h[n-n//2, n//2])
sage: f - Ep.functorial_composition(f)                                   #␣
↪needs sage.libs.pari
O^7
```

The symmetric function $\sum_n h_n$ is a left absorbing element:

```
sage: # needs sage.modules
sage: H.functorial_composition(f) - H
O^7
```

The functorial composition distributes over the sum:

```
sage: # needs sage.modules
sage: F1 = L(lambda n: h[n])
sage: F2 = L(lambda n: e[n])
```

(continues on next page)

(continued from previous page)

```
sage: f1 = F1.functorial_composition(f)
sage: f2 = F2.functorial_composition(f)
sage: (F1 + F2).functorial_composition(f) - f1 - f2         # long time
O^7
```

is_unit()

Return whether this element is a unit in the ring.

EXAMPLES:

```
sage: # needs sage.modules
sage: m = SymmetricFunctions(ZZ).m()
sage: L = LazySymmetricFunctions(m)
sage: L(2*m[1]).is_unit()
False
sage: L(-1 + 2*m[1]).is_unit()
True
sage: L(2 + m[1]).is_unit()
False
sage: m = SymmetricFunctions(QQ).m()
sage: L = LazySymmetricFunctions(m)
sage: L(2 + 3*m[1]).is_unit()
True
```

plethysm(**args*)

Return the composition of `self` with `g`.

The arity of `self` must be equal to the number of arguments provided.

Given a lazy symmetric function f of arity n and a tuple of lazy symmetric functions $g = (g_1, \ldots, g_n)$ over the same base ring, the composition (or plethysm) $(f \circ g)$ is defined if and only if for each $1 \leq i \leq n$:

- $g_i = 0$, or
- setting all alphabets except the i-th in f to zero yields a symmetric function with only finitely many non-zero coefficients, or
- $\mathrm{val}(g) > 0$.

If f is a univariate 'exact' lazy symmetric function, we can check whether f has only finitely many non-zero coefficients. However, if f has larger arity, we have no way to test whether setting all but one alphabets of f to zero yields a polynomial, except if f itself is 'exact' and therefore a symmetric function with only finitely many non-zero coefficients.

INPUT:

- `g` – other (lazy) symmetric functions

Todo: Allow specification of degree one elements.

EXAMPLES:

```
sage: # needs sage.modules
sage: P.<q> = QQ[]
sage: s = SymmetricFunctions(P).s()
```

(continues on next page)

(continued from previous page)

```
sage: L = LazySymmetricFunctions(s)
sage: f = s[2]
sage: g = s[3]
sage: L(f)(L(g)) - L(f(g))
0
sage: f = s[2] + s[2,1]
sage: g = s[1] + s[2,2]
sage: L(f)(L(g)) - L(f(g))
0
sage: L(f)(g) - L(f(g))
0
sage: f = s[2] + s[2,1]
sage: g = s[1] + s[2,2]
sage: L(f)(L(q*g)) - L(f(q*g))
0
```

The Frobenius character of the permutation action on set partitions is a plethysm:

```
sage: # needs sage.modules
sage: s = SymmetricFunctions(QQ).s()
sage: S = LazySymmetricFunctions(s)
sage: E1 = S(lambda n: s[n], valuation=1)
sage: E = 1 + E1
sage: P = E(E1)
sage: P[:5]
[s[], s[1], 2*s[2], s[2, 1] + 3*s[3], 2*s[2, 2] + 2*s[3, 1] + 5*s[4]]
```

The plethysm with a tensor product is also implemented:

```
sage: # needs sage.modules
sage: s = SymmetricFunctions(QQ).s()
sage: X = tensor([s[1],s[[]]])
sage: Y = tensor([s[[]],s[1]])
sage: S = LazySymmetricFunctions(s)
sage: S2 = LazySymmetricFunctions(tensor([s, s]))
sage: A = S(s[1,1,1])
sage: B = S2(X+Y)
sage: A(B)                                                                  #␣
↪needs lrcalc_python
(s[]#s[1,1,1]+s[1]#s[1,1]+s[1,1]#s[1]+s[1,1,1]#s[])

sage: H = S(lambda n: s[n])                                                 #␣
↪needs sage.modules
sage: H(S2(X*Y))                                                            #␣
↪needs lrcalc_python sage.modules
(s[]#s[]) + (s[1]#s[1]) + (s[1,1]#s[1,1]+s[2]#s[2])
 + (s[1,1,1]#s[1,1,1]+s[2,1]#s[2,1]+s[3]#s[3]) + O^7
sage: H(S2(X+Y))                                                            #␣
↪needs sage.modules
(s[]#s[]) + (s[]#s[1]+s[1]#s[]) + (s[]#s[2]+s[1]#s[1]+s[2]#s[])
 + (s[]#s[3]+s[1]#s[2]+s[2]#s[1]+s[3]#s[])
 + (s[]#s[4]+s[1]#s[3]+s[2]#s[2]+s[3]#s[1]+s[4]#s[])
```

(continues on next page)

(continued from previous page)

```
+ (s[]#s[5]+s[1]#s[4]+s[2]#s[3]+s[3]#s[2]+s[4]#s[1]+s[5]#s[])
+ (s[]#s[6]+s[1]#s[5]+s[2]#s[4]+s[3]#s[3]+s[4]#s[2]+s[5]#s[1]+s[6]#s[])
+ O^7
```

plethystic_inverse()

Return the compositional inverse of `self`.

Given a symmetric function f, the compositional inverse is a symmetric function g over the same base ring, such that $f \circ g = p_1$. Thus, it is the inverse with respect to plethystic substitution.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bp_1$ with $a, b \neq 0$.

EXAMPLES:

```
sage: # needs sage.modules
sage: h = SymmetricFunctions(QQ).h()
sage: L = LazySymmetricFunctions(h)
sage: f = L(lambda n: h[n]) - 1
sage: f(f.revert())
h[1] + O^8
```

ALGORITHM:

Let F be a symmetric function with valuation 1, i.e., whose constant term vanishes and whose degree one term equals bp_1. Then

$$(F - bp_1) \circ G = F \circ G - bp_1 \circ G = p_1 - bG,$$

and therefore $G = (p_1 - (F - bp_1) \circ G)/b$, which allows recursive computation of G.

See also:

The compositional inverse Ω of the symmetric function $h_1 + h_2 + \dots$ can be handled much more efficiently using specialized methods. See `LogarithmCycleIndexSeries()`

AUTHORS:

- Andrew Gainer-Dewar
- Martin Rubey

revert()

Return the compositional inverse of `self`.

Given a symmetric function f, the compositional inverse is a symmetric function g over the same base ring, such that $f \circ g = p_1$. Thus, it is the inverse with respect to plethystic substitution.

The compositional inverse exists if and only if:

- $\mathrm{val}(f) = 1$, or
- $f = a + bp_1$ with $a, b \neq 0$.

EXAMPLES:

```
sage: # needs sage.modules
sage: h = SymmetricFunctions(QQ).h()
sage: L = LazySymmetricFunctions(h)
sage: f = L(lambda n: h[n]) - 1
sage: f(f.revert())
h[1] + O^8
```

ALGORITHM:

Let F be a symmetric function with valuation 1, i.e., whose constant term vanishes and whose degree one term equals bp_1. Then

$$(F - bp_1) \circ G = F \circ G - bp_1 \circ G = p_1 - bG,$$

and therefore $G = (p_1 - (F - bp_1) \circ G)/b$, which allows recursive computation of G.

See also:

The compositional inverse Ω of the symmetric function $h_1 + h_2 + \dots$ can be handled much more efficiently using specialized methods. See `LogarithmCycleIndexSeries()`

AUTHORS:

- Andrew Gainer-Dewar
- Martin Rubey

`symmetric_function`(*degree=None*)

Return `self` as a symmetric function if `self` is actually so.

INPUT:

- `degree` – `None` or an integer

OUTPUT:

If `degree` is not `None`, the terms of the series of degree greater than `degree` are first truncated. If `degree` is `None` and the series is not a polynomial polynomial, a `ValueError` is raised.

EXAMPLES:

```
sage: # needs sage.modules
sage: s = SymmetricFunctions(QQ).s()
sage: S = LazySymmetricFunctions(s)
sage: elt = S(s[2])
sage: elt.symmetric_function()
s[2]
```

CHAPTER

TEN

LAZY SERIES RINGS

We provide lazy implementations for various **N**-graded rings.

LazyLaurentSeriesRing	The ring of lazy Laurent series.
LazyPowerSeriesRing	The ring of (possibly multivariate) lazy Taylor series.
LazyCompletionGradedAlgebra	The completion of a graded algebra consisting of formal series.
LazySymmetricFunctions	The ring of (possibly multivariate) lazy symmetric functions.
LazyDirichletSeriesRing	The ring of lazy Dirichlet series.

See also:

`sage.rings.padics.generic_nodes.pAdicRelaxedGeneric`, `sage.rings.padics.factory.ZpER()`

Warning: When the halting precision is infinite, the default for `bool(f)` is `True` for any lazy series `f` that is not known to be zero. This could end up resulting in infinite loops:

```
sage: L.<x> = LazyPowerSeriesRing(ZZ)
sage: f = L(lambda n: 0, valuation=0)
sage: 1 / f  # not tested - infinite loop
```

See also:

The examples of *LazyLaurentSeriesRing* contain a discussion about the different methods of comparisons the lazy series can use.

AUTHORS:

- Kwankyu Lee (2019-02-24): initial version
- Tejasvi Chebrolu, Martin Rubey, Travis Scrimshaw (2021-08): refactored and expanded functionality

class `sage.rings.lazy_series_ring.`**`LazyCompletionGradedAlgebra`**(*basis*, *sparse=True*, *category=None*)

Bases: *LazySeriesRing*

The completion of a graded algebra consisting of formal series.

For a graded algebra A, we can form a completion of A consisting of all formal series of A such that each homogeneous component is a finite linear combination of basis elements of A.

INPUT:

- `basis` – a graded algebra
- `names` – name(s) of the alphabets

- sparse – (default: True) whether we use a sparse or a dense representation

EXAMPLES:

```
sage: # needs sage.modules
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete()
sage: L = S.formal_series_ring(); L
Lazy completion of Non-Commutative Symmetric Functions
 over the Rational Field in the Complete basis
sage: f = 1 / (1 - L(S[1])); f
S[] + S[1] + (S[1,1]) + (S[1,1,1]) + (S[1,1,1,1]) + (S[1,1,1,1,1])
 + (S[1,1,1,1,1,1]) + O^7
sage: g = 1 / (1 - L(S[2])); g
S[] + S[2] + (S[2,2]) + (S[2,2,2]) + O^7
sage: f * g
S[] + S[1] + (S[1,1]+S[2]) + (S[1,1,1]+S[1,2])
 + (S[1,1,1,1]+S[1,1,2]+S[2,2]) + (S[1,1,1,1,1]+S[1,1,1,2]+S[1,2,2])
 + (S[1,1,1,1,1,1]+S[1,1,1,1,2]+S[1,1,2,2]+S[2,2,2]) + O^7
sage: g * f
S[] + S[1] + (S[1,1]+S[2]) + (S[1,1,1]+S[2,1])
 + (S[1,1,1,1]+S[2,1,1]+S[2,2]) + (S[1,1,1,1,1]+S[2,1,1,1]+S[2,2,1])
 + (S[1,1,1,1,1,1]+S[2,1,1,1,1]+S[2,2,1,1]+S[2,2,2]) + O^7
sage: f * g - g * f
(S[1,2]-S[2,1]) + (S[1,1,2]-S[2,1,1])
 + (S[1,1,1,2]+S[1,2,2]-S[2,1,1,1]-S[2,2,1])
 + (S[1,1,1,1,2]+S[1,1,2,2]-S[2,1,1,1,1]-S[2,2,1,1]) + O^7
```

Element

alias of *LazyCompletionGradedAlgebraElement*

some_elements()

Return a list of elements of self.

EXAMPLES:

```
sage: m = SymmetricFunctions(GF(5)).m()                                    #␣
↪needs sage.modules
sage: L = LazySymmetricFunctions(m)                                        #␣
↪needs sage.modules
sage: L.some_elements()[:5]                                                #␣
↪needs sage.modules
[0, m[], 2*m[] + 2*m[1] + 3*m[2], 2*m[1] + 3*m[2],
 3*m[] + 2*m[1] + (m[1,1]+m[2])
       + (2*m[1,1,1]+m[3])
       + (2*m[1,1,1,1]+4*m[2,1,1]+2*m[2,2])
       + (3*m[2,1,1,1]+3*m[3,1,1]+4*m[3,2]+m[5])
       + (2*m[2,2,1,1]+m[2,2,2]+2*m[3,2,1]+2*m[3,3]+m[4,1,1]+3*m[4,2]+4*m[5,
↪1]+4*m[6])
       + O^7]

sage: # needs sage.modules
sage: NCSF = NonCommutativeSymmetricFunctions(QQ)
sage: S = NCSF.Complete()
sage: L = S.formal_series_ring()
```

(continues on next page)

(continued from previous page)

```
sage: L.some_elements()[:4]
[0, S[], 2*S[] + 2*S[1] + (3*S[1,1]), 2*S[1] + (3*S[1,1])]
```

class sage.rings.lazy_series_ring.**LazyDirichletSeriesRing**(*base_ring*, *names*, *sparse=True*, *category=None*)

Bases: *LazySeriesRing*

The ring of lazy Dirichlet series.

INPUT:

- `base_ring` – base ring of this Dirichlet series ring
- `names` – name of the generator of this Dirichlet series ring
- `sparse` – (default: `True`) whether this series is sparse or not

Unlike formal univariate Laurent/power series (over a field), the ring of formal Dirichlet series is not a Wikipedia article discrete_valuation_ring. On the other hand, it is a Wikipedia article local_ring. The unique maximal ideal consists of all non-invertible series, i.e., series with vanishing constant term.

Todo: According to the answers in https://mathoverflow.net/questions/5522/dirichlet-series-with-integer-coefficients-as-a-ufd, (which, in particular, references arXiv math/0105219) the ring of formal Dirichlet series is actually a Wikipedia article Unique_factorization_domain over **Z**.

Note: An interesting valuation is described in Emil Daniel Schwab; Gheorghe Silberberg *A note on some discrete valuation rings of arithmetical functions*, Archivum Mathematicum, Vol. 36 (2000), No. 2, 103-109, http://dml.cz/dmlcz/107723. Let J_k be the ideal of Dirichlet series whose coefficient $f[n]$ of n^s vanishes if n has less than k prime factors, counting multiplicities. For any Dirichlet series f, let $D(f)$ be the largest integer k such that f is in J_k. Then D is surjective, $D(fg) = D(f) + D(g)$ for nonzero f and g, and $D(f + g) \geq \min(D(f), D(g))$ provided that $f + g$ is nonzero.

For example, J_1 are series with no constant term, and J_2 are series such that $f[1]$ and $f[p]$ for prime p vanish.

Since this is a chain of increasing ideals, the ring of formal Dirichlet series is not a Wikipedia article Noetherian_ring.

Evidently, this valuation cannot be computed for a given series.

EXAMPLES:

```
sage: LazyDirichletSeriesRing(ZZ, 't')
Lazy Dirichlet Series Ring in t over Integer Ring
```

The ideal generated by 2^-s and 3^-s is not principal:

```
sage: L = LazyDirichletSeriesRing(QQ, 's')
sage: L in PrincipalIdealDomains
False
```

Element

alias of *LazyDirichletSeries*

one()

Return the constant series 1.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(ZZ, 'z')
sage: L.one()                                                             #␣
↪needs sage.symbolic
1
sage: ~L.one()                                                            #␣
↪needs sage.symbolic
1 + O(1/(8^z))
```

some_elements()

Return a list of elements of `self`.

EXAMPLES:

```
sage: L = LazyDirichletSeriesRing(ZZ, 'z')
sage: l = L.some_elements()
sage: l                                                                   #␣
↪needs sage.symbolic
[0, 1,
 1/(4^z) + 1/(5^z) + 1/(6^z) + O(1/(7^z)),
 1/(2^z) - 1/(3^z) + 2/4^z - 2/5^z + 3/6^z - 3/7^z + 4/8^z - 4/9^z,
 1/(2^z) - 1/(3^z) + 2/4^z - 2/5^z + 3/6^z - 3/7^z + 4/8^z - 4/9^z + 1/(10^z) +␣
↪1/(11^z) + 1/(12^z) + O(1/(13^z)),
 1 + 4/2^z + 9/3^z + 16/4^z + 25/5^z + 36/6^z + 49/7^z + O(1/(8^z))]

sage: L = LazyDirichletSeriesRing(QQ, 'z')
sage: l = L.some_elements()
sage: l                                                                   #␣
↪needs sage.symbolic
[0, 1,
 1/2/4^z + 1/2/5^z + 1/2/6^z + O(1/(7^z)),
 1/2 - 1/2/2^z + 2/3^z - 2/4^z + 1/(6^z) - 1/(7^z) + 42/8^z + 2/3/9^z,
 1/2 - 1/2/2^z + 2/3^z - 2/4^z + 1/(6^z) - 1/(7^z) + 42/8^z + 2/3/9^z + 1/2/10^
↪z + 1/2/11^z + 1/2/12^z + O(1/(13^z)),
 1 + 4/2^z + 9/3^z + 16/4^z + 25/5^z + 36/6^z + 49/7^z + O(1/(8^z))]
```

class sage.rings.lazy_series_ring.**LazyLaurentSeriesRing**(*base_ring*, *names*, *sparse=True*, *category=None*)

Bases: *LazySeriesRing*

The ring of lazy Laurent series.

The ring of Laurent series over a ring with the usual arithmetic where the coefficients are computed lazily.

INPUT:

- `base_ring` – base ring
- `names` – name of the generator
- `sparse` – (default: `True`) whether the implementation of the series is sparse or not

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: 1 / (1 - z)
1 + z + z^2 + O(z^3)
sage: 1 / (1 - z) == 1 / (1 - z)
True
sage: L in Fields
True
```

Lazy Laurent series ring over a finite field:

```
sage: # needs sage.rings.finite_rings
sage: L.<z> = LazyLaurentSeriesRing(GF(3)); L
Lazy Laurent Series Ring in z over Finite Field of size 3
sage: e = 1 / (1 + z)
sage: e.coefficient(100)
1
sage: e.coefficient(100).parent()
Finite Field of size 3
```

Series can be defined by specifying a coefficient function and a valuation:

```
sage: R.<x,y> = QQ[]
sage: L.<z> = LazyLaurentSeriesRing(R)
sage: def coeff(n):
....:     if n < 0:
....:         return -2 + n
....:     if n == 0:
....:         return 6
....:     return x + y^n
sage: f = L(coeff, valuation=-5)
sage: f
-7*z^-5 - 6*z^-4 - 5*z^-3 - 4*z^-2 - 3*z^-1 + 6 + (x + y)*z + O(z^2)
sage: 1 / (1 - f)
1/7*z^5 - 6/49*z^6 + 1/343*z^7 + 8/2401*z^8 + 64/16807*z^9
 + 17319/117649*z^10 + (1/49*x + 1/49*y - 180781/823543)*z^11 + O(z^12)
sage: L(coeff, valuation=-3, degree=3, constant=x)
-5*z^-3 - 4*z^-2 - 3*z^-1 + 6 + (x + y)*z + (y^2 + x)*z^2
 + x*z^3 + x*z^4 + x*z^5 + O(z^6)
```

We can also specify a polynomial or the initial coefficients. Additionally, we may specify that all coefficients are equal to a given constant, beginning at a given degree:

```
sage: L([1, x, y, 0, x+y])
1 + x*z + y*z^2 + (x + y)*z^4
sage: L([1, x, y, 0, x+y], constant=2)
1 + x*z + y*z^2 + (x + y)*z^4 + 2*z^5 + 2*z^6 + 2*z^7 + O(z^8)
sage: L([1, x, y, 0, x+y], degree=7, constant=2)
1 + x*z + y*z^2 + (x + y)*z^4 + 2*z^7 + 2*z^8 + 2*z^9 + O(z^10)
sage: L([1, x, y, 0, x+y], valuation=-2)
z^-2 + x*z^-1 + y + (x + y)*z^2
sage: L([1, x, y, 0, x+y], valuation=-2, constant=3)
z^-2 + x*z^-1 + y + (x + y)*z^2 + 3*z^3 + 3*z^4 + 3*z^5 + O(z^6)
sage: L([1, x, y, 0, x+y], valuation=-2, degree=4, constant=3)
z^-2 + x*z^-1 + y + (x + y)*z^2 + 3*z^4 + 3*z^5 + 3*z^6 + O(z^7)
```

Some additional examples over the integer ring:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L in Fields
False
sage: 1 / (1 - 2*z)^3
1 + 6*z + 24*z^2 + 80*z^3 + 240*z^4 + 672*z^5 + 1792*z^6 + O(z^7)

sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: L(x^-2 + 3 + x)
z^-2 + 3 + z
sage: L(x^-2 + 3 + x, valuation=-5, constant=2)
z^-5 + 3*z^-3 + z^-2 + 2*z^-1 + 2 + 2*z + O(z^2)
sage: L(x^-2 + 3 + x, valuation=-5, degree=0, constant=2)
z^-5 + 3*z^-3 + z^-2 + 2 + 2*z + 2*z^2 + O(z^3)
```

We can truncate a series, shift its coefficients, or replace all coefficients beginning with a given degree by a constant:

```
sage: f = 1 / (z + z^2)
sage: f
z^-1 - 1 + z - z^2 + z^3 - z^4 + z^5 + O(z^6)
sage: L(f, valuation=2)
z^2 - z^3 + z^4 - z^5 + z^6 - z^7 + z^8 + O(z^9)
sage: L(f, degree=3)
z^-1 - 1 + z - z^2
sage: L(f, degree=3, constant=2)
z^-1 - 1 + z - z^2 + 2*z^3 + 2*z^4 + 2*z^5 + O(z^6)
sage: L(f, valuation=1, degree=4)
z - z^2 + z^3
sage: L(f, valuation=1, degree=4, constant=5)
z - z^2 + z^3 + 5*z^4 + 5*z^5 + 5*z^6 + O(z^7)
```

Power series can be defined recursively (see `sage.rings.lazy_series.LazyModuleElement.define()` for more examples):

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: s = L.undefined(valuation=0)
sage: s.define(1 + z*s^2)
sage: s
1 + z + 2*z^2 + 5*z^3 + 14*z^4 + 42*z^5 + 132*z^6 + O(z^7)
```

By default, any two series `f` and `g` that are not known to be equal are considered to be different:

```
sage: f = L(lambda n: 0, valuation=0)
sage: f == 0
False

sage: f = L(constant=1, valuation=0).derivative(); f
1 + 2*z + 3*z^2 + 4*z^3 + 5*z^4 + 6*z^5 + 7*z^6 + O(z^7)
sage: g = L(lambda n: (n+1), valuation=0); g
1 + 2*z + 3*z^2 + 4*z^3 + 5*z^4 + 6*z^5 + 7*z^6 + O(z^7)
sage: f == g
False
```

Warning: We have imposed that `(f == g) == not (f != g)`, and so `f != g` returning `True` might not mean that the two series are actually different:

```
sage: f = L(lambda n: 0, valuation=0)
sage: g = L.zero()
sage: f != g
True
```

This can be verified by `is_nonzero()`, which only returns `True` if the series is known to be nonzero:

```
sage: (f - g).is_nonzero()
False
```

The implementation of the ring can be either be a sparse or a dense one. The default is a sparse implementation:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L.is_sparse()
True
sage: L.<z> = LazyLaurentSeriesRing(ZZ, sparse=False)
sage: L.is_sparse()
False
```

We additionally provide two other methods of performing comparisons. The first is raising a `ValueError` and the second uses a check up to a (user set) finite precision. These behaviors are set using the options `secure` and `halting_precision`. In particular, this applies to series that are not specified by a finite number of initial coefficients and a constant for the remaining coefficients. Equality checking will depend on the coefficients which have already been computed. If this information is not enough to check that two series are different, then if `L.options.secure` is set to `True`, then we raise a `ValueError`:

```
sage: L.options.secure = True
sage: f = 1 / (z + z^2); f
z^-1 - 1 + z - z^2 + z^3 - z^4 + z^5 + O(z^6)
sage: f2 = f * 2  # currently no coefficients computed
sage: f3 = f * 3  # currently no coefficients computed
sage: f2 == f3
Traceback (most recent call last):
...
ValueError: undecidable
sage: f2  # computes some of the coefficients of f2
2*z^-1 - 2 + 2*z - 2*z^2 + 2*z^3 - 2*z^4 + 2*z^5 + O(z^6)
sage: f3  # computes some of the coefficients of f3
3*z^-1 - 3 + 3*z - 3*z^2 + 3*z^3 - 3*z^4 + 3*z^5 + O(z^6)
sage: f2 == f3
False
sage: f2a = f + f
sage: f2 == f2a
Traceback (most recent call last):
...
ValueError: undecidable
sage: zf = L(lambda n: 0, valuation=0)
sage: zf == 0
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: undecidable
```

For boolean checks, an error is raised when it is not known to be nonzero:

```
sage: bool(zf)
Traceback (most recent call last):
...
ValueError: undecidable
```

If the halting precision is set to a finite number p (for unlimited precision, it is set to `None`), then it will check up to p values from the current position:

```
sage: L.options.halting_precision = 20
sage: f2 = f * 2  # currently no coefficients computed
sage: f3 = f * 3  # currently no coefficients computed
sage: f2 == f3
False
sage: f2a = f + f
sage: f2 == f2a
True
sage: zf = L(lambda n: 0, valuation=0)
sage: zf == 0
True
```

`Element`

alias of *`LazyLaurentSeries`*

`euler``()`

Return the Euler function as an element of `self`.

The *Euler function* is defined as

$$\phi(z) = (z;z)_\infty = \sum_{n=0}^{\infty} (-1)^n q^{(3n^2-n)/2}.$$

EXAMPLES:

```
sage: L.<q> = LazyLaurentSeriesRing(ZZ)
sage: phi = q.euler()
sage: phi
1 - q - q^2 + q^5 + O(q^7)
```

We verify that $1/phi$ gives the generating function for all partitions:

```
sage: P = 1 / phi; P
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + O(q^7)
sage: P[:20] == [Partitions(n).cardinality() for n in range(20)]        #␣
↪needs sage.libs.flint
True
```

REFERENCES:

- Wikipedia article Euler_function

gen(*n=0*)

Return the n-th generator of `self`.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.gen()
z
sage: L.gen(3)
Traceback (most recent call last):
...
IndexError: there is only one generator
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L.gens()
(z,)
sage: 1/(1 - z)
1 + z + z^2 + O(z^3)
```

ngens()

Return the number of generators of `self`.

This is always 1.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L.ngens()
1
```

q_pochhammer(*q=None*)

Return the infinite q-Pochhammer symbol $(a; q)_\infty$, where a is the variable of `self`.

This is also one version of the quantum dilogarithm or the q-Exponential function.

INPUT:

- q – (default: $q \in \mathbf{Q}(q)$) the parameter q

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: L.<z> = LazyLaurentSeriesRing(q.parent())
sage: qpoch = L.q_pochhammer(q)
sage: qpoch
1
 + (-1/(-q + 1))*z
 + (q/(q^3 - q^2 - q + 1))*z^2
 + (-q^3/(-q^6 + q^5 + q^4 - q^2 - q + 1))*z^3
 + (q^6/(q^10 - q^9 - q^8 + 2*q^5 - q^2 - q + 1))*z^4
 + (-q^10/(-q^15 + q^14 + q^13 - q^10 - q^9 - q^8 + q^7 + q^6 + q^5 - q^2 - q +␣
↪1))*z^5
```

(continues on next page)

(continued from previous page)

```
 + (q^15/(q^21 - q^20 - q^19 + q^16 + 2*q^14 - q^12 - q^11 - q^10 - q^9 + 2*q^7␣
↪+ q^5 - q^2 - q + 1))*z^6
 + O(z^7)
```

We show that $(z;q)_n = \frac{(z;q)_\infty}{(q^n z;q)_\infty}$:

```
sage: qpoch / qpoch(q*z)
1 - z + O(z^7)
sage: qpoch / qpoch(q^2*z)
1 + (-q - 1)*z + q*z^2 + O(z^7)
sage: qpoch / qpoch(q^3*z)
1 + (-q^2 - q - 1)*z + (q^3 + q^2 + q)*z^2 - q^3*z^3 + O(z^7)
sage: qpoch / qpoch(q^4*z)
1 + (-q^3 - q^2 - q - 1)*z + (q^5 + q^4 + 2*q^3 + q^2 + q)*z^2
 + (-q^6 - q^5 - q^4 - q^3)*z^3 + q^6*z^4 + O(z^7)
```

We can also construct part of Euler's function:

```
sage: M.<a> = LazyLaurentSeriesRing(QQ)
sage: phi = sum(qpoch[i](q=a)*a^i for i in range(10))
sage: phi[:20] == M.euler()[:20]
True
```

REFERENCES:

- Wikipedia article Q-Pochhammer_symbol
- Wikipedia article Quantum_dilogarithm
- Wikipedia article Q-exponential

residue_field()

Return the residue field of the ring of integers of `self`.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(QQ, 'z')
sage: L.residue_field()
Rational Field
```

series(*coefficient*, *valuation*, *degree=None*, *constant=None*)

Return a lazy Laurent series.

INPUT:

- `coefficient` – Python function that computes coefficients or a list
- `valuation` – integer; approximate valuation of the series
- `degree` – (optional) integer
- `constant` – (optional) an element of the base ring

Let the coefficient of index i mean the coefficient of the term of the series with exponent i.

Python function `coefficient` returns the value of the coefficient of index i from input s and i where s is the series itself.

Let `valuation` be n. All coefficients of index below n are zero. If `constant` is not specified, then the `coefficient` function is responsible to compute the values of all coefficients of index $\geq n$. If `degree` or `constant` is a pair (c, m), then the `coefficient` function is responsible to compute the values of all coefficients of index $\geq n$ and $< m$ and all the coefficients of index $\geq m$ is the constant c.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.series(lambda s, i: i, 5, (1,10))
5*z^5 + 6*z^6 + 7*z^7 + 8*z^8 + 9*z^9 + z^10 + z^11 + z^12 + O(z^13)

sage: def g(s, i):
....:     if i < 0:
....:         return 1
....:     else:
....:         return s.coefficient(i - 1) + i
sage: e = L.series(g, -5); e
z^-5 + z^-4 + z^-3 + z^-2 + z^-1 + 1 + 2*z + O(z^2)
sage: f = e^-1; f
z^5 - z^6 - z^11 + O(z^12)
sage: f.coefficient(10)
0
sage: f.coefficient(20)
9
sage: f.coefficient(30)
-219
```

Alternatively, the `coefficient` can be a list of elements of the base ring. Then these elements are read as coefficients of the terms of degrees starting from the `valuation`. In this case, `constant` may be just an element of the base ring instead of a tuple or can be simply omitted if it is zero.

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: f = L.series([1,2,3,4], -5); f
z^-5 + 2*z^-4 + 3*z^-3 + 4*z^-2
sage: g = L.series([1,3,5,7,9], 5, constant=-1); g
z^5 + 3*z^6 + 5*z^7 + 7*z^8 + 9*z^9 - z^10 - z^11 - z^12 + O(z^13)
```

some_elements()

Return a list of elements of `self`.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.some_elements()[:7]
[0, 1, z,
 -3*z^-4 + z^-3 - 12*z^-2 - 2*z^-1 - 10 - 8*z + z^2 + z^3,
 z^-2 + z^3 + z^4 + z^5 + O(z^6),
 -2*z^-3 - 2*z^-2 + 4*z^-1 + 11 - z - 34*z^2 - 31*z^3 + O(z^4),
 4*z^-2 + z^-1 + z + 4*z^2 + 9*z^3 + 16*z^4 + O(z^5)]

sage: L = LazyLaurentSeriesRing(GF(2), 'z')
sage: L.some_elements()[:7]
[0, 1, z,
 z^-4 + z^-3 + z^2 + z^3,
 z^-2,
```

(continues on next page)

(continued from previous page)

```
 1 + z + z^3 + z^4 + z^6 + O(z^7),
 z^-1 + z + z^3 + O(z^5)]

sage: L = LazyLaurentSeriesRing(GF(3), 'z')
sage: L.some_elements()[:7]
[0, 1, z,
 z^-3 + z^-1 + 2 + z + z^2 + z^3,
 z^-2,
 z^-3 + z^-2 + z^-1 + 2 + 2*z + 2*z^2 + O(z^3),
 z^-2 + z^-1 + z + z^2 + z^4 + O(z^5)]
```

uniformizer()

Return a uniformizer of `self`..

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(QQ, 'z')
sage: L.uniformizer()
z
```

class sage.rings.lazy_series_ring.**LazyPowerSeriesRing**(*base_ring*, *names*, *sparse=True*, *category=None*)

Bases: *LazySeriesRing*

The ring of (possibly multivariate) lazy Taylor series.

INPUT:

- `base_ring` – base ring of this Taylor series ring
- `names` – name(s) of the generator of this Taylor series ring
- `sparse` – (default: `True`) whether this series is sparse or not

EXAMPLES:

```
sage: LazyPowerSeriesRing(ZZ, 't')
Lazy Taylor Series Ring in t over Integer Ring

sage: L.<x, y> = LazyPowerSeriesRing(QQ); L
Multivariate Lazy Taylor Series Ring in x, y over Rational Field
```

Element

alias of *LazyPowerSeries*

fraction_field()

Return the fraction field of `self`.

If this is with a single variable over a field, then the fraction field is the field of (lazy) formal Laurent series.

Todo: Implement other fraction fields.

EXAMPLES:

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: L.fraction_field()
Lazy Laurent Series Ring in x over Rational Field
```

gen(*n=0*)

Return the n-th generator of `self`.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(ZZ, 'z')
sage: L.gen()
z
sage: L.gen(3)
Traceback (most recent call last):
...
IndexError: there is only one generator
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(ZZ, 'x,y')
sage: L.gens()
(x, y)
```

ngens()

Return the number of generators of `self`.

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(ZZ)
sage: L.ngens()
1
```

residue_field()

Return the residue field of the ring of integers of `self`.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ, 'x')
sage: L.residue_field()
Rational Field
```

some_elements()

Return a list of elements of `self`.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(ZZ, 'z')
sage: L.some_elements()[:6]
[0, 1, z + z^2 + z^3 + O(z^4),
 -12 - 8*z + z^2 + z^3,
 1 + z - 2*z^2 - 7*z^3 - z^4 + 20*z^5 + 23*z^6 + O(z^7),
 z + 4*z^2 + 9*z^3 + 16*z^4 + 25*z^5 + 36*z^6 + O(z^7)]
```

(continues on next page)

(continued from previous page)

```
sage: L = LazyPowerSeriesRing(GF(3)["q"], 'z')
sage: L.some_elements()[:6]
[0, 1, z + q*z^2 + q*z^3 + q*z^4 + O(z^5),
 z + z^2 + z^3,
 1 + z + z^2 + 2*z^3 + 2*z^4 + 2*z^5 + O(z^6),
 z + z^2 + z^4 + z^5 + O(z^7)]

sage: L = LazyPowerSeriesRing(GF(3), 'q, t')
sage: L.some_elements()[:6]
[0, 1, q,
 q + q^2 + q^3,
 1 + q + q^2 + (-q^3) + (-q^4) + (-q^5) + (-q^6) + O(q,t)^7,
 1 + (q+t) + (q^2-q*t+t^2) + (q^3+t^3)
   + (q^4+q^3*t+q*t^3+t^4)
   + (q^5-q^4*t+q^3*t^2+q^2*t^3-q*t^4+t^5)
   + (q^6-q^3*t^3+t^6) + O(q,t)^7]
```

uniformizer()

Return a uniformizer of `self`.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ, 'x')
sage: L.uniformizer()
x
```

class sage.rings.lazy_series_ring.**LazySeriesRing**

Bases: `UniqueRepresentation`, `Parent`

Abstract base class for lazy series.

characteristic()

Return the characteristic of this lazy power series ring, which is the same as the characteristic of its base ring.

EXAMPLES:

```
sage: L.<t> = LazyLaurentSeriesRing(ZZ)
sage: L.characteristic()
0

sage: R.<w> = LazyLaurentSeriesRing(GF(11)); R
Lazy Laurent Series Ring in w over Finite Field of size 11
sage: R.characteristic()
11

sage: R.<x, y> = LazyPowerSeriesRing(GF(7)); R
Multivariate Lazy Taylor Series Ring in x, y over Finite Field of size 7
sage: R.characteristic()
7

sage: L = LazyDirichletSeriesRing(ZZ, "s")
```

(continues on next page)

(continued from previous page)

```
sage: L.characteristic()
0
```

is_exact()

Return if `self` is exact or not.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.is_exact()
True
sage: L = LazyLaurentSeriesRing(RR, 'z')
sage: L.is_exact()
False
```

is_sparse()

Return whether `self` is sparse or not.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z', sparse=False)
sage: L.is_sparse()
False

sage: L = LazyLaurentSeriesRing(ZZ, 'z', sparse=True)
sage: L.is_sparse()
True
```

one()

Return the constant series 1.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.one()
1

sage: L = LazyPowerSeriesRing(ZZ, 'z')
sage: L.one()
1

sage: m = SymmetricFunctions(ZZ).m()                                            #␣
↪needs sage.modules
sage: L = LazySymmetricFunctions(m)                                             #␣
↪needs sage.modules
sage: L.one()                                                                   #␣
↪needs sage.modules
m[]
```

options = Current options for lazy series rings - constant_length: 3 - display_length: 7 - halting_precision: None - secure: False

prod(*f*, *a=None*, *b=+Infinity*, *add_one=False*)

The product of elements of `self`.

INPUT:

- `f` – a list (or iterable) of elements of `self`
- `a`, `b` – optional arguments
- `add_one` – (default: `False`); if `True`, then converts a lazy series p_i from `args` into $1 + p_i$ for the product

If `a` and `b` are both integers, then this returns the product $\prod_{i=a}^{b} f(i)$, where $f(i) = p_i$ if `add_one=False` or $f(i) = 1 + p_i$ otherwise. If `b` is not specified, then we consider $b = \infty$. Note this corresponds to the Python `range(a, b+1)`.

If a is any other iterable, then this returns the product $\prod_{i \in a} f(i)$, where $f(i) = p_i$ if `add_one=False` or $f(i) = 1 + p_i$.

Note: For infinite products, it is faster to use `add_one=True` since the implementation is based on p_i in $\prod_i (1 + p_i)$.

Warning: When `f` is an infinite generator, then the first argument `a` must be `True`. Otherwise this will loop forever.

Warning: For an *infinite* product of the form $\prod_i (1 + p_i)$, if $p_i = 0$, then this will loop forever.

EXAMPLES:

```
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: euler = L.prod(lambda n: 1 - t^n, PositiveIntegers())
sage: euler
1 - t - t^2 + t^5 + O(t^7)
sage: 1 / euler
1 + t + 2*t^2 + 3*t^3 + 5*t^4 + 7*t^5 + 11*t^6 + O(t^7)
sage: euler - L.euler()
O(t^7)
sage: L.prod(lambda n: -t^n, 1, add_one=True)
1 - t - t^2 + t^5 + O(t^7)

sage: L.prod((1 - t^n for n in PositiveIntegers()), True)
1 - t - t^2 + t^5 + O(t^7)
sage: L.prod((-t^n for n in PositiveIntegers()), True, add_one=True)
1 - t - t^2 + t^5 + O(t^7)

sage: L.prod((1 + t^(n-3) for n in PositiveIntegers()), True)
2*t^-3 + 4*t^-2 + 4*t^-1 + 4 + 6*t + 10*t^2 + 16*t^3 + O(t^4)

sage: L.prod(lambda n: 2 + t^n, -3, 5)
96*t^-6 + 240*t^-5 + 336*t^-4 + 840*t^-3 + 984*t^-2 + 1248*t^-1
 + 1980 + 1668*t + 1824*t^2 + 1872*t^3 + 1782*t^4 + 1710*t^5
 + 1314*t^6 + 1122*t^7 + 858*t^8 + 711*t^9 + 438*t^10 + 282*t^11
 + 210*t^12 + 84*t^13 + 60*t^14 + 24*t^15
sage: L.prod(lambda n: t^n / (1 + abs(n)), -2, 2, add_one=True)
```

(continues on next page)

(continued from previous page)

```
1/3*t^-3 + 5/6*t^-2 + 13/9*t^-1 + 25/9 + 13/9*t + 5/6*t^2 + 1/3*t^3
sage: L.prod(lambda n: t^-2 + t^n / n, -4, -2)
1/24*t^-9 - 1/8*t^-8 - 1/6*t^-7 + 1/2*t^-6

sage: D = LazyDirichletSeriesRing(QQ, "s")
sage: D.prod(lambda p: (1+D(1, valuation=p)).inverse(), Primes())
1 - 1/(2^s) - 1/(3^s) + 1/(4^s) - 1/(5^s) + 1/(6^s) - 1/(7^s) + O(1/(8^s))

sage: D.prod(lambda p: D(1, valuation=p), Primes(), add_one=True)
1 + 1/(2^s) + 1/(3^s) + 1/(5^s) + 1/(6^s) + 1/(7^s) + O(1/(8^s))
```

sum(*f*, *a=None*, *b=+Infinity*)

The sum of elements of `self`.

INPUT:

- `f` – a list (or iterable or function) of elements of `self`
- `a, b` – optional arguments

If `a` and `b` are both integers, then this returns the sum $\sum_{i=a}^{b} f(i)$. If `b` is not specified, then we consider $b = \infty$. Note this corresponds to the Python `range(a, b+1)`.

If a is any other iterable, then this returns the sum $\sum i \in af(i)$.

Warning: When `f` is an infinite generator, then the first argument `a` must be `True`. Otherwise this will loop forever.

Warning: For an *infinite* sum of the form $\sum_i s_i$, if $s_i = 0$, then this will loop forever.

EXAMPLES:

```
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: L.sum(lambda n: t^n / (n+1), PositiveIntegers())
1/2*t + 1/3*t^2 + 1/4*t^3 + 1/5*t^4 + 1/6*t^5 + 1/7*t^6 + 1/8*t^7 + O(t^8)

sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: T = L.undefined(1)
sage: D = L.undefined(0)
sage: H = L.sum(lambda k: T(z^k)/k, 2)
sage: T.define(z*exp(T)*D)
sage: D.define(exp(H))
sage: T
z + z^2 + 2*z^3 + 4*z^4 + 9*z^5 + 20*z^6 + 48*z^7 + O(z^8)
sage: D
1 + 1/2*z^2 + 1/3*z^3 + 7/8*z^4 + 11/30*z^5 + 281/144*z^6 + O(z^7)
```

We verify the Rogers-Ramanujan identities up to degree 100:

```
sage: L.<q> = LazyPowerSeriesRing(QQ)
sage: Gpi = L.prod(lambda k: -q^(1+5*k), 0, oo, add_one=True)
```

(continues on next page)

(continued from previous page)

```
sage: Gpi *= L.prod(lambda k: -q^(4+5*k), 0, oo, add_one=True)
sage: Gp = 1 / Gpi
sage: G = L.sum(lambda n: q^(n^2) / prod(1 - q^(k+1) for k in range(n)), 0, oo)
sage: G - Gp
O(q^7)
sage: all(G[k] == Gp[k] for k in range(100))
True

sage: Hpi = L.prod(lambda k: -q^(2+5*k), 0, oo, add_one=True)
sage: Hpi *= L.prod(lambda k: -q^(3+5*k), 0, oo, add_one=True)
sage: Hp = 1 / Hpi
sage: H = L.sum(lambda n: q^(n^2+n) / prod(1 - q^(k+1) for k in range(n)), 0,␣
↪oo)
sage: H - Hp
O(q^7)
sage: all(H[k] == Hp[k] for k in range(100))
True
```

```
sage: D = LazyDirichletSeriesRing(QQ, "s")
sage: D.sum(lambda p: D(1, valuation=p), Primes())
1/(2^s) + 1/(3^s) + 1/(5^s) + 1/(7^s) + O(1/(9^s))
```

undefined(*valuation=None*)

Return an uninitialized series.

INPUT:

- `valuation` – integer; a lower bound for the valuation of the series

Power series can be defined recursively (see *sage.rings.lazy_series.LazyModuleElement.define()* for more examples).

See also:

`sage.rings.padics.generic_nodes.pAdicRelaxedGeneric.unknown()`

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: s = L.undefined(1)
sage: s.define(z + (s^2+s(z^2))/2)
sage: s
z + z^2 + z^3 + 2*z^4 + 3*z^5 + 6*z^6 + 11*z^7 + O(z^8)
```

Alternatively:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: f = L(None, valuation=-1)
sage: f.define(z^-1 + z^2*f^2)
sage: f
z^-1 + 1 + 2*z + 5*z^2 + 14*z^3 + 42*z^4 + 132*z^5 + O(z^6)
```

unknown(*valuation=None*)

Return an uninitialized series.

INPUT:

- `valuation` – integer; a lower bound for the valuation of the series

Power series can be defined recursively (see *sage.rings.lazy_series.LazyModuleElement.define()* for more examples).

See also:

`sage.rings.padics.generic_nodes.pAdicRelaxedGeneric.unknown()`

EXAMPLES:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: s = L.undefined(1)
sage: s.define(z + (s^2+s(z^2))/2)
sage: s
z + z^2 + z^3 + 2*z^4 + 3*z^5 + 6*z^6 + 11*z^7 + O(z^8)
```

Alternatively:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: f = L(None, valuation=-1)
sage: f.define(z^-1 + z^2*f^2)
sage: f
z^-1 + 1 + 2*z + 5*z^2 + 14*z^3 + 42*z^4 + 132*z^5 + O(z^6)
```

zero()

Return the zero series.

EXAMPLES:

```
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.zero()
0

sage: s = SymmetricFunctions(ZZ).s()                                    #␣
↪needs sage.modules
sage: L = LazySymmetricFunctions(s)                                     #␣
↪needs sage.modules
sage: L.zero()                                                          #␣
↪needs sage.modules
0

sage: L = LazyDirichletSeriesRing(ZZ, 'z')
sage: L.zero()
0

sage: L = LazyPowerSeriesRing(ZZ, 'z')
sage: L.zero()
0
```

class sage.rings.lazy_series_ring.**LazySymmetricFunctions**(*basis*, *sparse=True*, *category=None*)

Bases: *LazyCompletionGradedAlgebra*

The ring of lazy symmetric functions.

INPUT:

- `basis` – the ring of symmetric functions

- `names` – name(s) of the alphabets
- `sparse` – (default: `True`) whether we use a sparse or a dense representation

EXAMPLES:

```
sage: s = SymmetricFunctions(ZZ).s()                                    #␣
↪needs sage.modules
sage: LazySymmetricFunctions(s)                                         #␣
↪needs sage.modules
Lazy completion of Symmetric Functions over Integer Ring in the Schur basis

sage: m = SymmetricFunctions(ZZ).m()                                    #␣
↪needs sage.modules
sage: LazySymmetricFunctions(tensor([s, m]))                            #␣
↪needs sage.modules
Lazy completion of
 Symmetric Functions over Integer Ring in the Schur basis
  # Symmetric Functions over Integer Ring in the monomial basis
```

`Element`

alias of *`LazySymmetricFunction`*

CHAPTER

ELEVEN

PUISEUX SERIES RING

The ring of Puiseux series.

AUTHORS:

- Chris Swierczewski 2016: initial version on https://github.com/abelfunctions/abelfunctions/tree/master/abelfunctions
- Frédéric Chapoton 2016: integration of code
- Travis Scrimshaw, Sebastian Oehms 2019-2020: basic improvements and completions

REFERENCES:

- Wikipedia article Puiseux_series

class sage.rings.puiseux_series_ring.**PuiseuxSeriesRing**(*laurent_series*)

Bases: UniqueRepresentation, CommutativeRing

Rings of Puiseux series.

EXAMPLES:

```
sage: P = PuiseuxSeriesRing(QQ, 'y')
sage: y = P.gen()
sage: f = y**(4/3) + y**(-5/6); f
y^(-5/6) + y^(4/3)
sage: f.add_bigoh(2)
y^(-5/6) + y^(4/3) + O(y^2)
sage: f.add_bigoh(1)
y^(-5/6) + O(y)
```

Element

alias of *PuiseuxSeries*

base_extend(*R*)

Extend the coefficients.

INPUT:

- R – a ring

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(ZZ, 'y')
sage: A.base_extend(QQ)
Puiseux Series Ring in y over Rational Field
```

change_ring(*R*)

Return a Puiseux series ring over another ring.

INPUT:

- R – a ring

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(ZZ, 'y')
sage: A.change_ring(QQ)
Puiseux Series Ring in y over Rational Field
```

default_prec()

Return the default precision of `self`.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(AA, 'z')                                    #␣
↪needs sage.rings.number_field
sage: A.default_prec()                                                  #␣
↪needs sage.rings.number_field
20
```

fraction_field()

Return the fraction field of this ring of Laurent series.

If the base ring is a field, then Puiseux series are already a field. If the base ring is a domain, then the Puiseux series over its fraction field is returned. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: R = PuiseuxSeriesRing(ZZ, 't', 30).fraction_field()
sage: R
Puiseux Series Ring in t over Rational Field
sage: R.default_prec()
30

sage: PuiseuxSeriesRing(Zmod(4), 't').fraction_field()
Traceback (most recent call last):
...
ValueError: must be an integral domain
```

gen(*n=0*)

Return the generator of `self`.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(AA, 'z')                                    #␣
↪needs sage.rings.number_field
sage: A.gen()                                                           #␣
↪needs sage.rings.number_field
z
```

is_dense()

Return whether `self` is dense.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(ZZ, 'y')
sage: A.is_dense()
True
```

is_field(*proof=True*)

Return whether `self` is a field.

A Puiseux series ring is a field if and only its base ring is a field.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(ZZ, 'y')
sage: A.is_field()
False
sage: A.change_ring(QQ).is_field()
True
```

is_sparse()

Return whether `self` is sparse.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(ZZ, 'y')
sage: A.is_sparse()
False
```

laurent_series_ring()

Return the underlying Laurent series ring.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(AA, 'z')                                        #␣
↪needs sage.rings.number_field
sage: A.laurent_series_ring()                                               #␣
↪needs sage.rings.number_field
Laurent Series Ring in z over Algebraic Real Field
```

ngens()

Return the number of generators of `self`, namely 1.

EXAMPLES:

```
sage: A = PuiseuxSeriesRing(AA, 'z')                                        #␣
↪needs sage.rings.number_field
sage: A.ngens()                                                             #␣
↪needs sage.rings.number_field
1
```

residue_field()

Return the residue field of this Puiseux series field if it is a complete discrete valuation field (i.e. if the base ring is a field, in which case it is also the residue field).

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(GF(17))
sage: R.residue_field()
Finite Field of size 17

sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: R.residue_field()
Traceback (most recent call last):
...
TypeError: the base ring is not a field
```

uniformizer()

Return a uniformizer of this Puiseux series field if it is a discrete valuation field (i.e. if the base ring is actually a field). Otherwise, an error is raised.

EXAMPLES:

```
sage: R.<t> = PuiseuxSeriesRing(QQ)
sage: R.uniformizer()
t

sage: R.<t> = PuiseuxSeriesRing(ZZ)
sage: R.uniformizer()
Traceback (most recent call last):
...
TypeError: the base ring is not a field
```

CHAPTER

TWELVE

PUISEUX SERIES RING ELEMENT

A Puiseux series is a series of the form

$$p(x) = \sum_{n=N}^{\infty} a_n (x-a)^{n/e},$$

where the integer e is called the *ramification index* of the series and the number a is the *center*. A Puiseux series is essentially a Laurent series but with fractional exponents.

EXAMPLES:

We begin by constructing the ring of Puiseux series in x with coefficients in the rationals:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
```

This command also defines x as the generator of this ring.

When constructing a Puiseux series, the ramification index is automatically determined from the greatest common divisor of the exponents:

```
sage: p = x^(1/2); p
x^(1/2)
sage: p.ramification_index()
2
sage: q = x^(1/2) + x**(1/3); q
x^(1/3) + x^(1/2)
sage: q.ramification_index()
6
```

Other arithmetic can be performed with Puiseux Series:

```
sage: p + q
x^(1/3) + 2*x^(1/2)
sage: p - q
-x^(1/3)
sage: p * q
x^(5/6) + x
sage: (p / q).add_bigoh(4/3)
x^(1/6) - x^(1/3) + x^(1/2) - x^(2/3) + x^(5/6) - x + x^(7/6) + O(x^(4/3))
```

Mind the base ring. However, the base ring can be changed:

```
sage: I*q                                                                    #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Number Field in I with defining␣
↪polynomial x^2 + 1 with I = 1*I' and 'Puiseux Series Ring in x over Rational Field'
sage: qz = q.change_ring(ZZ); qz
x^(1/3) + x^(1/2)
sage: qz.parent()
Puiseux Series Ring in x over Integer Ring
```

Other properties of the Puiseux series can be easily obtained:

```
sage: r = (3*x^(-1/5) + 7*x^(2/5) + (1/2)*x).add_bigoh(6/5); r
3*x^(-1/5) + 7*x^(2/5) + 1/2*x + O(x^(6/5))
sage: r.valuation()
-1/5
sage: r.prec()
6/5
sage: r.precision_absolute()
6/5
sage: r.precision_relative()
7/5
sage: r.exponents()
[-1/5, 2/5, 1]
sage: r.coefficients()
[3, 7, 1/2]
```

Finally, Puiseux series are compatible with other objects in Sage. For example, you can perform arithmetic with Laurent series:

```
sage: L.<x> = LaurentSeriesRing(ZZ)
sage: l = 3*x^(-2) + x^(-1) + 2 + x**3
sage: r + l
3*x^-2 + x^-1 + 3*x^(-1/5) + 2 + 7*x^(2/5) + 1/2*x + O(x^(6/5))
```

AUTHORS:

- Chris Swierczewski 2016: initial version on https://github.com/abelfunctions/abelfunctions/tree/master/abelfunctions
- Frédéric Chapoton 2016: integration of code
- Travis Scrimshaw, Sebastian Oehms 2019-2020: basic improvements and completions

REFERENCES:

- Wikipedia article Puiseux_series

class sage.rings.puiseux_series_ring_element.**PuiseuxSeries**

Bases: `AlgebraElement`

A Puiseux series.

$$\sum_{n=-N}^{\infty} a_n x^{n/e}$$

It is stored as a Laurent series:

$$\sum_{n=-N}^{\infty} a_n t^n$$

where $t = x^{1/e}$.

INPUT:

- `parent` – the parent ring
- `f` – one of the following types of inputs:
  - instance of *PuiseuxSeries*
  - instance that can be coerced into the Laurent series ring of the parent
- `e` – integer (default: 1) the ramification index

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(1/2) + x**3; p
x^(1/2) + x^3
sage: q = x**(1/2) - x**(-1/2)
sage: r = q.add_bigoh(7/2); r
-x^(-1/2) + x^(1/2) + O(x^(7/2))
sage: r**2
x^-1 - 2 + x + O(x^3)
```

add_bigoh(*prec*)

Return the truncated series at chosen precision `prec`.

INPUT:

- `prec` – the precision of the series as a rational number

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: p.add_bigoh(2)
x^(-7/2) + 3 + 5*x^(1/2) + O(x^2)
sage: p.add_bigoh(0)
x^(-7/2) + O(1)
sage: p.add_bigoh(-1)
x^(-7/2) + O(x^-1)
```

Note: The precision passed to the method is adapted to the common ramification index:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x**(-1/3) + 2*x**(1/5)
sage: p.add_bigoh(1/2)
x^(-1/3) + 2*x^(1/5) + O(x^(7/15))
```

change_ring(*R*)

Return `self` over a the new ring R.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: q = p.change_ring(QQ); q
x^(-7/2) + 3 + 5*x^(1/2) - 7*x^3
sage: q.parent()
Puiseux Series Ring in x over Rational Field
```

coefficients()

Return the list of coefficients.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(3/4) + 2*x^(4/5) + 3* x^(5/6)
sage: p.coefficients()
[1, 2, 3]
```

common_prec(*p*)

Return the minimum precision of p and `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = (x**(-1/3) + 2*x**3)**2
sage: q5 = p.add_bigoh(5); q5
x^(-2/3) + 4*x^(8/3) + O(x^5)
sage: q7 = p.add_bigoh(7); q7
x^(-2/3) + 4*x^(8/3) + 4*x^6 + O(x^7)
sage: q5.common_prec(q7)
5
sage: q7.common_prec(q5)
5
```

degree()

Return the degree of `self`.

EXAMPLES:

```
sage: P.<y> = PolynomialRing(GF(5))
sage: R.<x> = PuiseuxSeriesRing(P)
sage: p = 3*y*x**(-2/3) + 2*y**2*x**(1/5); p
3*y*x^(-2/3) + 2*y^2*x^(1/5)
sage: p.degree()
1/5
```

exponents()

Return the list of exponents.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(3/4) + 2*x^(4/5) + 3* x^(5/6)
sage: p.exponents()
[3/4, 4/5, 5/6]
```

inverse()

Return the inverse of `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: 1/p
x^(7/2) - 3*x^7 - 5*x^(15/2) + 7*x^10 + 9*x^(21/2) + 30*x^11 +
25*x^(23/2) + O(x^(27/2))
```

is_monomial()

Return whether `self` is a monomial.

This is `True` if and only if `self` is x^p for some rational p.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(1/2) + 3/4 * x^(2/3)
sage: p.is_monomial()
False
sage: q = x**(11/13)
sage: q.is_monomial()
True
sage: q = 4*x**(11/13)
sage: q.is_monomial()
False
```

is_unit()

Return whether `self` is a unit.

A Puiseux series is a unit if and only if its leading coefficient is.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: p.is_unit()
True
sage: q = 4 * x^(-7/2) + 3 * x**4
sage: q.is_unit()
False
```

is_zero()

Return whether `self` is zero.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(1/2) + 3/4 * x^(2/3)
sage: p.is_zero()
False
sage: R.zero().is_zero()
True
```

laurent_part()

Return the underlying Laurent series.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(1/2) + 3/4 * x^(2/3)
sage: p.laurent_part()
x^3 + 3/4*x^4
```

laurent_series()

If `self` is a Laurent series, return it as a Laurent series.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x**(1/2) - x**(-1/2)
sage: p.laurent_series()
Traceback (most recent call last):
...
ArithmeticError: self is not a Laurent series
sage: q = p**2
sage: q.laurent_series()
x^-1 - 2 + x
```

list()

Return the list of coefficients indexed by the exponents of the the corresponding Laurent series.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(3/4) + 2*x^(4/5) + 3* x^(5/6)
sage: p.list()
[1, 0, 0, 2, 0, 3]
```

power_series()

If `self` is a power series, return it as a power series.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = PuiseuxSeriesRing(QQbar)
sage: p = x**(3/2) - QQbar(I)*x**(1/2)
sage: p.power_series()
Traceback (most recent call last):
...
ArithmeticError: self is not a power series
```

(continues on next page)

(continued from previous page)

```
sage: q = p**2
sage: q.power_series()
-x - 2*I*x^2 + x^3
```

prec()

Return the precision of `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = (x**(-1/3) + 2*x**3)**2; p
x^(-2/3) + 4*x^(8/3) + 4*x^6
sage: q = p.add_bigoh(5); q
x^(-2/3) + 4*x^(8/3) + O(x^5)
sage: q.prec()
5
```

precision_absolute()

Return the precision of `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = (x**(-1/3) + 2*x**3)**2; p
x^(-2/3) + 4*x^(8/3) + 4*x^6
sage: q = p.add_bigoh(5); q
x^(-2/3) + 4*x^(8/3) + O(x^5)
sage: q.prec()
5
```

precision_relative()

Return the relative precision of the series.

The relative precision of the Puiseux series is the difference between its absolute precision and its valuation.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(GF(3))
sage: p = (x**(-1/3) + 2*x**3)**2; p
x^(-2/3) + x^(8/3) + x^6
sage: q = p.add_bigoh(7); q
x^(-2/3) + x^(8/3) + x^6 + O(x^7)
sage: q.precision_relative()
23/3
```

ramification_index()

Return the ramification index.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(1/2) + 3/4 * x^(2/3)
sage: p.ramification_index()
6
```

shift(r)

Return this Puiseux series multiplied by x^r.

EXAMPLES:

```
sage: P.<y> = LaurentPolynomialRing(ZZ)
sage: R.<x> = PuiseuxSeriesRing(P)
sage: p = y*x**(-1/3) + 2*y^(-2)*x**(1/2); p
y*x^(-1/3) + (2*y^-2)*x^(1/2)
sage: p.shift(3)
y*x^(8/3) + (2*y^-2)*x^(7/2)
```

truncate(r)

Return the Puiseux series of degree $< r$.

This is equivalent to `self` modulo x^r.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = (x**(-1/3) + 2*x**3)**2; p
x^(-2/3) + 4*x^(8/3) + 4*x^6
sage: q = p.truncate(5); q
x^(-2/3) + 4*x^(8/3)
sage: q == p.add_bigoh(5)
True
```

valuation()

Return the valuation of `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: p.valuation()
-7/2
```

variable()

Return the variable of `self`.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: p.variable()
'x'
```

CHAPTER

THIRTEEN

TATE ALGEBRAS

Let K be a finite extension of $\mathbf{Q}_p$ for some prime number p and let $(v_1, \dots, v_n)$ be a tuple of real numbers.

The associated Tate algebra consists of series of the form

$$\sum_{i_1,\dots,i_n \in \mathbf{N}} a_{i_1,\dots,i_n} x_1^{i_1} \cdots x_n^{i_n}$$

for which the quantity

$$\operatorname{val}(a_{i_1,\dots,i_n}) - (v_1 i_1 + \cdots + v_n i_n)$$

goes to infinity when the multi-index $(i_1, \dots, i_n)$ goes to infinity.

These series converge on the closed disc defined by the inequalities $\operatorname{val}(x_i) \geq -v_i$ for all $i \in \{1, \dots, n\}$. The v_i's are then the logarithms of the radii of convergence of the series in the above Tate algebra; the will be called the log radii of convergence.

We can create Tate algebras using the constructor `sage.rings.tate_algebra.TateAlgebra()`:

```
sage: K = Qp(2, 5, print_mode='digits')
sage: A.<x,y> = TateAlgebra(K)
sage: A
Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with capped relative␣
↪precision 5
```

As we observe, the default value for the log radii of convergence is 0 (the series then converge on the closed unit disc).

We can specify different log radii using the following syntax:

```
sage: B.<u,v> = TateAlgebra(K, log_radii=[1,2]); B
Tate Algebra in u (val >= -1), v (val >= -2) over 2-adic Field with capped relative␣
↪precision 5
```

Note that if we pass in the ring of integers of p-adic field, the same Tate algebra is returned:

```
sage: A1.<x,y> = TateAlgebra(K.integer_ring()); A1
Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with capped relative␣
↪precision 5
sage: A is A1
True
```

However the method `integer_ring()` constructs the integer ring of a Tate algebra, that is the subring consisting of series bounded by 1 on the domain of convergence:

```
sage: Ao = A.integer_ring()
sage: Ao
Integer ring of the Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with␣
↪capped relative precision 5
```

Now we can build elements:

```
sage: f = 5 + 2*x*y^3 + 4*x^2*y^2; f
...00101 + ...000010*x*y^3 + ...0000100*x^2*y^2
sage: g = x^3*y + 2*x*y; g
...00001*x^3*y + ...000010*x*y
```

and perform all usual arithmetic operations on them:

```
sage: f + g
...00001*x^3*y + ...00101 + ...000010*x*y^3 + ...000010*x*y + ...0000100*x^2*y^2
sage: f * g
...00101*x^3*y + ...000010*x^4*y^4 + ...001010*x*y + ...0000100*x^5*y^3 + ...0000100*x^
↪2*y^4 + ...00001000*x^3*y^3
```

An element in the integer ring is invertible if and only if its reduction modulo p is a nonzero constant. In our example, f is invertible (its reduction modulo 2 is 1) but g is not:

```
sage: f.inverse_of_unit()
...01101 + ...01110*x*y^3 + ...10100*x^2*y^6 + ... + O(2^5 * <x, y>)
sage: g.inverse_of_unit()
Traceback (most recent call last):
...
ValueError: this series in not invertible
```

The notation $O(2^5)$ in the result above hides a series which lies in 2^5 times the integer ring of A, that is a series which is bounded by $|2^5|$ (2-adic norm) on the domain of convergence.

We can also evaluate series in a point of the domain of convergence (in the base field or in an extension):

```
sage: L.<a> = Qq(2^3, 5)
sage: f(a^2, 2*a)
1 + 2^2 + a*2^4 + O(2^5)

sage: u = polygen(ZZ, 'u')
sage: L.<pi> = K.change(print_mode="series").extension(u^3 - 2)
sage: g(pi, 2*pi)
pi^7 + pi^8 + pi^19 + pi^20 + O(pi^21)
```

Computations with ideals in Tate algebras are also supported:

```
sage: f = 7*x^3*y + 2*x*y - x*y^2 - 6*y^5
sage: g = x*y^4 + 8*x^3 - 3*y^3 + 1
sage: I = A.ideal([f, g])
sage: I.groebner_basis()
[...00001*x^2*y^3 + ...00001*y^4 + ...10001*x^2 + ... + O(2^5 * <x, y>),
 ...00001*x*y^4 + ...11101*y^3 + ...00001 + ... + O(2^5 * <x, y>),
 ...00001*y^5 + ...11111*x*y^3 + ...01001*x^2*y + ... + O(2^5 * <x, y>),
 ...00001*x^3 + ...01001*x*y + ...10110*y^4 + ...01110*x + O(2^5 * <x, y>)]
```

(continues on next page)

(continued from previous page)

```
sage: (x^2 + 3*y)*f + 1/2*(x^3*y + x*y)*g in I
True
```

AUTHORS:

- Xavier Caruso, Thibaut Verron (2018-09)

class sage.rings.tate_algebra.**TateAlgebraFactory**

Bases: UniqueFactory

Construct a Tate algebra over a p-adic field.

Given a p-adic field K, variables $X_1, \dots, X_k$ and convergence log radii $v_1, \dots, v_n$ in $\mathbf{R}$, the corresponding Tate algebra $KX_1, \dots, X_k$ consists of power series with coefficients $a_{i_1,\dots,i_n}$ in K such that

$$\mathrm{val}(a_{i_1,\dots,i_n}) - (i_1 v_1 + \cdots + i_n v_n)$$

tends to infinity as $i_1, \dots, i_n$ go towards infinity.

INPUT:

- `base` – a p-adic ring or field; if a ring is given, the Tate algebra over its fraction field will be constructed
- `prec` – an integer or `None` (default: `None`), the precision cap; it is used if an exact object must be truncated in order to do an arithmetic operation. If left as `None`, it will be set to the precision cap of the base field.
- `log_radii` – an integer or a list or a tuple of integers (default: `0`), the value(s) v_i. If an integer is given, this will be the common value for all v_i.
- `names` – names of the indeterminates
- `order` – the monomial ordering (default: `degrevlex`) used to break ties when comparing terms with the same coefficient valuation

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits'); R
2-adic Ring with capped relative precision 10
sage: A.<x,y> = TateAlgebra(R, order='lex'); A
Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with capped relative␣
↪precision 10
```

We observe that the result is the Tate algebra over the fraction field of R and not R itself:

```
sage: A.base_ring()
2-adic Field with capped relative precision 10
sage: A.base_ring() is R.fraction_field()
True
```

If we want to construct the ring of integers of the Tate algebra, we must use the method `integer_ring()`:

```
sage: Ao = A.integer_ring(); Ao
Integer ring of the Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field␣
↪with capped relative precision 10
sage: Ao.base_ring()
2-adic Ring with capped relative precision 10
sage: Ao.base_ring() is R
True
```

The term ordering is used (in particular) to determine how series are displayed. Terms are compared first according to the valuation of their coefficient, and ties are broken using the monomial ordering:

```
sage: A.term_order()
Lexicographic term order
sage: f = 2 + y^5 + x^2; f
...0000000001*x^2 + ...0000000001*y^5 + ...00000000010
sage: B.<x,y> = TateAlgebra(R); B
Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with capped relative␣
↪precision 10
sage: B.term_order()
Degree reverse lexicographic term order
sage: B(f)
...0000000001*y^5 + ...0000000001*x^2 + ...00000000010
```

Here are examples of Tate algebra with smaller radii of convergence:

```
sage: B.<x,y> = TateAlgebra(R, log_radii=-1); B
Tate Algebra in x (val >= 1), y (val >= 1) over 2-adic Field with capped relative␣
↪precision 10
sage: C.<x,y> = TateAlgebra(R, log_radii=[-1,-2]); C
Tate Algebra in x (val >= 1), y (val >= 2) over 2-adic Field with capped relative␣
↪precision 10
```

AUTHORS:

- Xavier Caruso, Thibaut Verron (2018-09)

create_key(*base*, *prec=None*, *log_radii=0*, *names=None*, *order='degrevlex'*)

Create a key from the input parameters.

INPUT:

- `base` – a p-adic ring or field
- `prec` – an integer or `None` (default: `None`)
- `log_radii` – an integer or a list or a tuple of integers (default: `0`)
- `names` – names of the indeterminates
- `order` - a monomial ordering (default: `degrevlex`)

EXAMPLES:

```
sage: TateAlgebra.create_key(Zp(2), names=['x','y'])
(2-adic Field with capped relative precision 20,
 20,
 (0, 0),
 ('x', 'y'),
 Degree reverse lexicographic term order)
```

create_object(*version*, *key*)

Create an object using the given key.

class sage.rings.tate_algebra.**TateAlgebra_generic**(*field*, *prec*, *log_radii*, *names*, *order*, *integral=False*)

Bases: `CommutativeAlgebra`

Initialize the Tate algebra

absolute_e()

Return the absolute index of ramification of this Tate algebra.

It is equal to the absolute index of ramification of the field of coefficients.

EXAMPLES:

```
sage: R = Zp(2)
sage: A.<u,v> = TateAlgebra(R)
sage: A.absolute_e()
1

sage: R.<a> = Zq(2^3)
sage: A.<u,v> = TateAlgebra(R)
sage: A.absolute_e()
1

sage: x = polygen(ZZ, 'x')
sage: S.<a> = R.extension(x^2 - 2)
sage: A.<u,v> = TateAlgebra(S)
sage: A.absolute_e()
2
```

characteristic()

Return the characteristic of this algebra.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.characteristic()
0
```

gen(*n=0*)

Return the n-th generator of this Tate algebra.

INPUT:

- n - an integer (default: 0), the index of the requested generator

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.gen()
...0000000001*x
sage: A.gen(0)
...0000000001*x
sage: A.gen(1)
...0000000001*y
sage: A.gen(2)
Traceback (most recent call last):
...
ValueError: generator not defined
```

gens()

Return the list of generators of this Tate algebra.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.gens()
(...0000000001*x, ...0000000001*y)
```

integer_ring()

Return the ring of integers (consisting of series bounded by 1 in the domain of convergence) of this Tate algebra.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: Ao = A.integer_ring()
sage: Ao
Integer ring of the Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic␣
↪Field with capped relative precision 10

sage: x in Ao
True
sage: x/2 in Ao
False
```

is_integral_domain(*proof=True*)

Return `True` since any Tate algebra is an integral domain.

EXAMPLES:

```
sage: A.<x,y> = TateAlgebra(Zp(3))
sage: A.is_integral_domain()
True
```

log_radii()

Return the list of the log-radii of convergence radii defining this Tate algebra.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: A.log_radii()
(0, 0)

sage: B.<x,y> = TateAlgebra(R, log_radii=1)
sage: B.log_radii()
(1, 1)

sage: C.<x,y> = TateAlgebra(R, log_radii=(1,-1))
sage: C.log_radii()
(1, -1)
```

monoid_of_terms()

Return the monoid of terms of this Tate algebra.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: A.monoid_of_terms()
Monoid of terms in x (val >= 0), y (val >= 0) over 2-adic Field with capped␣
↪relative precision 10
```

ngens()

Return the number of generators of this algebra.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.ngens()
2
```

precision_cap()

Return the precision cap of this Tate algebra.

Note: The precision cap is the truncation precision used for arithmetic operations computed by successive approximations (as inversion).

EXAMPLES:

By default the precision cap is the precision cap of the field of coefficients:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: A.precision_cap()
10
```

But it could be different (either smaller or larger) if we ask to:

```
sage: A.<x,y> = TateAlgebra(R, prec=5)
sage: A.precision_cap()
5

sage: A.<x,y> = TateAlgebra(R, prec=20)
sage: A.precision_cap()
20
```

prime()

Return the prime, that is the characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3)
sage: A.<x,y> = TateAlgebra(R)
sage: A.prime()
3
```

random_element(*degree=2*, *terms=5*, *integral=False*, *prec=None*)

Return a random element of this Tate algebra.

INPUT:

- `degree` – an integer (default: 2), an upper bound on the total degree of the result
- `terms` – an integer (default: 5), the maximal number of terms of the result
- `integral` – a boolean (default: `False`); if `True` the result will be in the ring of integers
- `prec` – (optional) an integer, the precision of the result

EXAMPLES:

```
sage: R = Zp(2, prec=10, print_mode="digits")
sage: A.<x,y> = TateAlgebra(R)
sage: A.random_element()  # random
(...00101000.01)*y + ...1111011111*x^2 + ...0010010001*x*y + ...110000011 + ...
↪010100100*y^2

sage: A.random_element(degree=5, terms=3)  # random
(...0101100.01)*x^2*y + (...01000011.11)*y^2 + ...00111011*x*y

sage: A.random_element(integral=True)  # random
...0001111101*x + ...1101110101 + ...00010010110*y + ...101110001100*x*y + ...
↪000001100100*y^2
```

Note that if we are already working on the ring of integers, specifying `integral=False` has no effect:

```
sage: Ao = A.integer_ring()
sage: f = Ao.random_element(integral=False); f  # random
...1100111011*x^2 + ...1110100101*x + ...1100001101*y + ...1110110001 + ...
↪01011010110*y^2
sage: f in Ao
True
```

When the log radii are negative, integral series may have non integral coefficients:

```
sage: B.<x,y> = TateAlgebra(R, log_radii=[-1,-2])
sage: B.random_element(integral=True)  # random
(...1111111.001)*x*y + (...111000101.1)*x + (...11010111.01)*y^2 + ...
↪0010011011*y + ...0010100011000
```

some_elements()

Return a list of elements in this Tate algebra.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.some_elements()
[0,
 ...00000000010,
 ...0000000001*x,
 ...0000000001*y,
 ...00000000010*x*y,
 ...00000000100,
 ...0000000001*x + ...00000000010,
 ...0000000001*y + ...00000000010,
 ...00000000010*x*y + ...00000000010,
```

(continues on next page)

(continued from previous page)

```
...0000000010*x,
...0000000001*x + ...0000000001*y,
...0000000001*x + ...00000000010*x*y,
...0000000010*y,
...0000000001*y + ...00000000010*x*y,
...00000000100*x*y]
```

term_order()

Return the monomial order used in this algebra.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: A.term_order()
Degree reverse lexicographic term order

sage: A.<x,y> = TateAlgebra(R, order='lex')
sage: A.term_order()
Lexicographic term order
```

variable_names()

Return the names of the variables of this algebra.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.variable_names()
('x', 'y')
```

class sage.rings.tate_algebra.**TateTermMonoid**(*A*)

Bases: Monoid_class, UniqueRepresentation

A base class for Tate algebra terms

A term in a Tate algebra $K\{X_1, \ldots, X_n\}$ (resp. in its ring of integers) is a monomial in this ring.

Those terms form a pre-ordered monoid, with term multiplication and the term order of the parent Tate algebra.

Element

alias of TateAlgebraTerm

algebra_of_series()

Return the Tate algebra corresponding to this Tate term monoid.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.algebra_of_series()
Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic Field with capped␣
↪relative precision 10
sage: T.algebra_of_series() is A
True
```

base_ring()

Return the base ring of this Tate term monoid.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.base_ring()
2-adic Field with capped relative precision 10
```

We observe that the base field is not R but its fraction field:

```
sage: T.base_ring() is R
False
sage: T.base_ring() is R.fraction_field()
True
```

If we really want to create an integral Tate algebra, we have to invoke the method `integer_ring()`:

```
sage: Ao = A.integer_ring(); Ao
Integer ring of the Tate Algebra in x (val >= 0), y (val >= 0) over 2-adic␣
↪Field with capped relative precision 10
sage: Ao.base_ring()
2-adic Ring with capped relative precision 10
sage: Ao.base_ring() is R
True
```

gen(*n=0*)

Return the `n`-th generator of this monoid of terms.

INPUT:

- `n` - an integer (default: `0`), the index of the requested generator

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.gen()
...0000000001*x
sage: T.gen(0)
...0000000001*x
sage: T.gen(1)
...0000000001*y
sage: T.gen(2)
Traceback (most recent call last):
...
ValueError: generator not defined
```

gens()

Return the list of generators of this monoid of terms.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.gens()
(...0000000001*x, ...0000000001*y)
```

log_radii()

Return the log radii of convergence of this Tate term monoid.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.log_radii()
(0, 0)

sage: B.<x,y> = TateAlgebra(R, log_radii=[1,2])
sage: B.monoid_of_terms().log_radii()
(1, 2)
```

ngens()

Return the number of variables in the Tate term monoid

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.ngens()
2
```

prime()

Return the prime, that is the characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.prime()
3
```

some_elements()

Return a list of elements in this monoid of terms.

EXAMPLES:

```
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.some_elements()
[...00000000010, ...0000000001*x, ...0000000001*y, ...00000000010*x*y]
```

term_order()

Return the term order on this Tate term monoid.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.term_order()  # default term order is grevlex
Degree reverse lexicographic term order

sage: A.<x,y> = TateAlgebra(R, order='lex')
sage: T = A.monoid_of_terms()
sage: T.term_order()
Lexicographic term order
```

variable_names()

Return the names of the variables of this Tate term monoid.

EXAMPLES:

```
sage: R = Zp(2, 10)
sage: A.<x,y> = TateAlgebra(R)
sage: T = A.monoid_of_terms()
sage: T.variable_names()
('x', 'y')
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX