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 can compute with power series over the symbolic ring.

```
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})
```

Choose another implementation of the attached polynomial ring:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: type(t.polynomial())
<... 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>
sage: S.<s> = PowerSeriesRing(ZZ, implementation='NTL')
sage: type(s.polynomial())
<... 'sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl'>
```

AUTHORS:

- William Stein: the code
- Jeremy Cho (2006-05-17): some examples (above)
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
Multivariate Power Series Ring in x, y over Finite Field of size 655537
```

Power series ring with many variables:

```
sage: R = PowerSeriesRing(ZZ, [x%p for p in primes(100)]); R
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()
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*x11^8*x29 - x19 + R.O(3)
sage: f in R
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*a^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 + 0(a, b)^9
```
```
sage: U = PowerSeriesRing(ZZ,'a,b',order='negdeglex'); U
Multivariate Power Series Ring in a, b over Integer Ring
```

(continues on next page)
U.term_order()
Negative degree lexicographic term order
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: sage.rings.power_series_ring.PowerSeriesRing_generic, sage.rings.ring.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)


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)
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:**

```python
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'))
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:**

```python
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:**

```python
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:**

```python
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[[t]]
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, random-
  ized 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)
```

(continues on next page)
Further arguments are passed to the underlying base ring (trac ticket #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 + 0(v^20)
sage: SZ.random_element(3, x=4, y=6)  # random
5 + 4*v + 5*v^2 + 0(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))
sage: R.residue_field()
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:**
```python
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`

Bases: `sage.rings.power_series_ring.PowerSeriesRing_domain`

```python
def 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
```

```python
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
```

```python
sage.rings.power_series_ring.unpickle_power_series_ring_v0(base_ring, name=None, default_prec=None, sparse=False, category=None)
    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
```
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
- Simon King (2012-08): use category and coercion framework, trac ticket #13412

EXAMPLES:

```python
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:

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

```sage
sage: huge_ring.gen1
gen1
```

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

```sage
sage: huge_ring.gen2
x
```

With power series the behavior is the same.

```sage
class sage.rings.power_series_ring_element.PowerSeries
Bases: sage.structure.element.AlgebraElement

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

\(O(\text{prec})\)

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

EXAMPLES:

```sage
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
sage: R.<x> = PowerSeriesRing(ZZ)
sage: p = 1 + x^2 + x^10; p
1 + x^2 + x^10
sage: p.V(3)
```

(continues on next page)
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 \(O(q^{prec})\) 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’)[[[]]
sage: (t^2 + O(t^3)).base_ring()
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):
...
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 \(\text{ZZ(K(4))}\) is defined.
sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')
sage: R.<t> = K[[t]]
sage: (4*t).change_ring(ZZ)
4*t

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

sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')
sage: R.<t> = K[[t]]
sage: ((a+1)*t).change_ring(ZZ)
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]

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

```
cos(prec='infinity')
Apply cos to the formal power series.

INPUT:
```
• \( \text{prec} \) – Integer or \( \infty \). The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

```python
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:

```python
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
```

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

```python
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:

```python
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)
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
```

\texttt{cosh}(\text{prec}=\text{\textquoteleft infinity\textquoteleft})

Apply \( \text{cosh} \) to the formal power series.

INPUT:

• \( \text{prec} \) – Integer or \( \infty \). The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:
```sage
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
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 + 0(a, b)^2
```

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

```sage
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
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 +
0(a, b)^12
sage: a.cosh(prec=5)
1 + 1/2*a^2 + 1/24*a^4 + 0(a, b)^5
sage: cosh(a + T.O(5))
1 + 1/2*a^2 + 1/24*a^4 + 0(a, b)^5
```

\textbf{degree()}

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

\textbf{EXAMPLES:}

```sage
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^1000000 + O(t^10000000)
sage: f.degree()
1000000
```

\textbf{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.

\textbf{See also:}

\texttt{_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 trac ticket #4477):

```
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()
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
```

\texttt{exponents()} 
Return the exponents appearing in self with nonzero coefficients.

\textbf{EXAMPLES:}

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

(continues on next page)
```
sage: f.exponents()
[1, 2, 3]
```

**inverse()**

Return the inverse of self, i.e., \( \text{self}^\text{-1} \).

**EXAMPLES:**

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: t.inverse()
t^-1
sage: type(_)
<type '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()
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
```

(continues on next page)
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()
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
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

\[
\frac{1}{1 + A_0 t + B_0 t^2 / (1 + A_1 t + B_1 t^2 / (1 + \cdots))}
\]

**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:**

```python
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:

```python
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:**

```python
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**\((absprec=\text{None})\)

Return a congruent power series with absolute precision at least \(absprec\).

**INPUT:**

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

**EXAMPLES:**

```python
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:**

```python
sage: R.<x> = PowerSeriesRing(ZZ)
sage: PowerSeries.list(1+x^2)
Traceback (most recent call last):
  ... Not ImplementedError
```

**log**\((prec=\text{None})\)

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:**

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

(continues on next page)
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[]
sage: f = (1+I)*x^2 + 3*x - I
sage: f.map_coefficients(lambda z: z.conjugate())
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
```

(continues on next page)
Tests other implementations:

```python
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:**

```python
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:

```python
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:

```python
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:**
```python
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**

```
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:

```python
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()  # n is not given
[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:

```python
sage: R.<x> = PowerSeriesRing(ZZ)
sage: PowerSeries.polynomial(1+x^2)
Traceback (most recent call last):
... Not ImplementedError
```

**prec**

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

EXAMPLES:
sage: R.<t> = ZZ[[t]]
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[[t]]
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[[t]]
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 $t^n$ will be discarded. Does not change this power 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> = 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)
AUTHORS:
• Robert Bradshaw (2007-04-18)

\texttt{sin}(\texttt{prec='infinity'})
Apply \texttt{sin} to the formal power series.

INPUT:
• \texttt{prec} – Integer or \texttt{infinity}. The degree to truncate the result to.

OUTPUT:
A new power series.

EXAMPLES:
For one variable:

\begin{verbatim}
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)
\end{verbatim}

For several variables:

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
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 + 0(a, b)^5
sage: sin(a + T.O(5))
a - 1/6*a^3 + 0(a, b)^5
\end{verbatim}
\textbf{sinh}(\texttt{prec='infinity'})

Apply \textit{sinh} to the formal power series.

INPUT:

- \texttt{prec} – Integer or \textit{infinity}. The degree to truncate the result to.

OUTPUT:

A new power series.

EXAMPLES:

For one variable:

\begin{verbatim}
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)
\end{verbatim}

For several variables:

\begin{verbatim}
sage: T.<a,b> = PowerSeriesRing(ZZ,2)
sage: f = a + b + a*b + T.O(3)
sage: sinh(f)
a + b + O(a, b)^3
sage: f.sinh()
a + b + O(a, b)^3
sage: f.sinh(\texttt{prec=2})
a + b + O(a, b)^2
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
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(\texttt{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
\end{verbatim}

\textbf{solve_linear_de}(\texttt{prec='infinity', b=None, f0=None})

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

\[ f'(t) = a(t)f(t) + b(t). \]
• self - the power series \( a(t) \)
• b - the power series \( b(t) \) (default is zero)
• \( f_0 \) - 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 \( \mathbb{Q} \), 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)

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

```sage```
a = 2 - 3*t + 4*t^2
b = 3 - 4*t^2
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 powers 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

\[ x_{i+1} = \frac{1}{2}(x_i + \text{self}/x_i) \]

**EXAMPLES:**

```python
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)
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
t
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)
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
```

```python
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 + 0(t^5), -1.0 + 1.0*t + 0.0*t^2 + 1.0*t^3 + 0(t^5)]
```

A formal square root:

```python
sage: K.<t> = PowerSeriesRing(QQ, 5)
sage: f = 2*t + t^3 + O(t^4)
```

(continues on next page)
AUTHORS:
• Robert Bradshaw
• William Stein

\texttt{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 \texttt{self. is\_square()}

EXAMPLIES:

\begin{Verbatim}
\texttt{sage: K.<t> = PowerSeriesRing(QQ, 't', 5)}
\texttt{sage: (1+t).square\_root()}
\texttt{1 + 1/2\*t - 1/8\*t^2 + 1/16\*t^3 - 5/128\*t^4 + O(t^5)}
\texttt{sage: (2\*t).square\_root()}
Traceback (most recent call last):
...
ValueError: Square root does not live in this ring.
\texttt{sage: (2\*t.change\_ring(RR)).square\_root()}
1.41421356237309 + 0.353553390593274\*t - 0.0441941738241592\*t^2 + 0.
\rightarrow \ldots
\texttt{sage: t.square\_root()}
Traceback (most recent call last):
...
ValueError: Square root not defined for power series of odd valuation.
\texttt{sage: K.<t> = PowerSeriesRing(ZZ, 't', 5)}
\texttt{sage: f = (1+t)^20}
\texttt{sage: f.square\_root()}
1 + 10\*t + 45\*t^2 + 120\*t^3 + 210\*t^4 + O(t^5)
\texttt{sage: f = 1+t}
\texttt{sage: f.square\_root()}
Traceback (most recent call last):
...
ValueError: Square root does not live in this ring.
\end{Verbatim}

AUTHORS:
• Robert Bradshaw

\texttt{stieltjes\_continued\_fraction()}

Return the Stieltjes continued fraction of \texttt{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

\[ \frac{1}{1 - A_1 t/(1 - A_2 t/(1 - A_3 t/(1 - \cdots)))} \]
OUTPUT:

\[ A_n \text{ for } n \geq 1 \]

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:

```python
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, 1)
```

Another example:

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

```python
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 + 0(a, b)^3
sage: f.tan()
a + b + a^b + 0(a, b)^3
sage: f.tan(prec=2)
a + 0(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):
  ...  
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)^\rightarrow12
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()
```
If the power series has a non-zero constant coefficient \(c\), one raises an error:

```python
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:

```python
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 + 0(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:

```python
sage: R.<I> = GF(2)[[]]
sage: f = 1/(1+I+0(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:

```python
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^100000 + O(t^1000000)
sage: f.valuation()
100000
```

Dense examples:

```python
sage: R(0).valuation() +Infinity
```
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)
```
(continues on next page)
False
\begin{Verbatim}
\texttt{sage: \texttt{var('x')}}
\texttt{x}
\texttt{sage: \texttt{is\_PowerSeries(1+x^2)}}
\texttt{False}
\end{Verbatim}

\texttt{sage.rings.power_series_ring_element.make\_element\_from\_parent\_v0(\texttt{parent}, \ast\texttt{args})}

\texttt{sage.rings.power_series_ring_element.make\_powerseries\_poly\_v0(\texttt{parent}, \texttt{f}, \texttt{prec}, \texttt{is\_gen})}
The class `PowerSeries_poly` provides additional methods for univariate power series.

```python
class sage.rings.power_series_poly.PowerSeries_poly
    Bases: sage.rings.power_series_ring_element.PowerSeries

EXAMPLES:

    sage: R.<q> = PowerSeriesRing(CC)
    sage: R
    Power Series Ring in q over Complex Field with 53 bits of precision
    sage: loads(q.dumps()) == q
    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 trac ticket #22216 is fixed:

    sage: R.<T> = PowerSeriesRing(QQ)
    sage: R(pari('1 + O(T)'))
    1 + O(T)
    sage: R(pari('1/T + O(T)'))
    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:

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

(continues on next page)
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:

```python
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:

```python
sage: k.<w> = QQ[[[]]
```

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:

```python
sage: R.<t> = ZZ[[[]]
```

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

**Warning:** The current implementation uses a very slow algorithm and is not suitable for high orders.

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_reconstruct()`

EXAMPLES:

```python
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:

```python
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:**

```python
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)
```

```python
sage: A.<t> = PowerSeriesRing(ZZ)
sage: a = t - t^2 - 2*t^4 + t^5 + O(t^6)
sage: b = a.reverse(); b
1/2*t - 3/8*t^2 + 9/16*t^3 - 131/128*t^4 + O(t^5)
sage: a(b)
t + O(t^6)
sage: b(a)
t + O(t^6)
```

```python
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
1/2*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)
```

```python
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)
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 + ...
```
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)
```

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)
```

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)
```

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 + O(x^10)
```

```
˓→ 4862*x^10 + 16796*x^11 + 58786*x^12 + 268012*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)
```
\textbf{truncate}(\texttt{prec='infinity'})

The polynomial obtained from power series by truncation at precision \texttt{prec}.

**EXAMPLES:**

\begin{verbatim}
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
\end{verbatim}

\textbf{truncate_powerseries}(\texttt{prec})

Given input \texttt{prec} = \(n\), returns the power series of degree < \(n\) which is equivalent to self modulo \(x^n\).

**EXAMPLES:**

\begin{verbatim}
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)
\end{verbatim}

\textbf{valuation}()

Return the valuation of \texttt{self}.

**EXAMPLES:**

\begin{verbatim}
sage: R.<t> = QQ[[[]]
sage: (5 - t^8 + O(t^11)).valuation()
0
sage: (-t^8 + O(t^11)).valuation()
8
sage: 0(t^7).valuation()
7
sage: R(0).valuation()
+Infinity
\end{verbatim}

\texttt{sage.rings.power_series_poly.make_powerseries_poly_v0}(\texttt{parent}, \texttt{f}, \texttt{prec}, \texttt{is_gen})

Return the power series specified by \texttt{f}, \texttt{prec}, and \texttt{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:**

\begin{verbatim}
sage: R.<t> = QQ[[[]]
sage: sage.rings.power_series_poly.make_powerseries_poly_v0(R, t, infinity, True)
t
\end{verbatim}
**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
sage: R.<q> = PowerSeriesRing(ZZ, implementation='pari'); R
Power Series Ring in q over Integer Ring
sage: S.<t> = PowerSeriesRing(CC, implementation='pari'); S
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
sage: type(R)
<class 'sage.rings.power_series_ring.PowerSeriesRing_domain_with_category'>
sage: type(q)
<type 'sage.rings.power_series_pari.PowerSeries_pari'>
sage: type(S)
<class 'sage.rings.power_series_ring.PowerSeriesRing_over_field_with_category'>
sage: type(t)
<type '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
sage: k.<c> = GF(5^12)
sage: type(c)
<type 'sage.rings.finite_rings.element_pari_ffelt.FiniteFieldElement_pari_ffelt'>
sage: A.<x> = k[[]

sage: type(x)
<type '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
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:
AUTHORS:
  • Peter Bruin (December 2013): initial version

class sage.rings.power_series_pari.PowerSeries_pari
  Bases: sage.rings.power_series_ring_element.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
```

44 Chapter 4. Power series implemented using PARI
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:

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

padded_list(n=\text{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 \(+\text{Infinity}\), in which case we just return self.list()

EXAMPLES:

```python
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]
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]
```

dpolynomial()
Convert self to a polynomial.

EXAMPLES:

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

```python
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:

```python
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)
```

(continues on next page)
x + 0(x^3)
sage: g(f)
x + 0(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 + 0(x^3) |

valuation()

Return the valuation of self.

EXAMPLES:

| sage: R.<t> = PowerSeriesRing(QQ, implementation='pari') |
| sage: (5 - t^8 + 0(t^11)).valuation() |
| 0 |
| sage: (-t^8 + 0(t^11)).valuation() |
| 8 |
| sage: 0(t^7).valuation() |
| 7 |
| sage: R(0).valuation() |
| +Infinity |
Construct a multivariate power series ring (in finitely many variables) over a given (commutative) base ring.

EXAMPLES:

Construct rings and elements:

```sage
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*t^2*u - 2*t^2*v^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
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
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
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)
True

\begin{verbatim}
sage: TestSuite(M).run()

sage: H = PowerSeriesRing(PolynomialRing(ZZ, '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()

  + (-z[2]^2 - 2*z[0] + z[2])*f[0]*f[2]
  + (-z[0]^2 + 2*z[1]^2)*f[2]*f[3]
  + H.O(3)

sage: h in H
True

sage: h
+ (-z[0]^2 + 2*z[1]^2)*f[2]*f[3]
+ O(f[0], f[1], f[2], f[3])^3
\end{verbatim}

- Use angle-bracket notation:

\begin{verbatim}
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
\end{verbatim}

- Use double square bracket notation:

\begin{verbatim}
sage: ZZ[['s,t,u']]
Multivariate Power Series Ring in s, t, u over Integer Ring

sage: GF(127931)[['x,y']]
Multivariate Power Series Ring in x, y over Finite Field of size 127931
\end{verbatim}

Variable ordering determines how series are displayed.

\begin{verbatim}
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
True

sage: T.term_order()
\end{verbatim}
Degree lexicographic term order

```python
sage: p = - 2*b^6 + a^5*b^2 + a^7 - b^2 - a*b^3 + O(a, b)^9; p
a^7 + a^5*b^2 - 2*b^6 - a*b^3 - b^2 + O(a, b)^9
```

```python
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:

```python
sage: R.<t,u,v> = PowerSeriesRing(QQ); R
Multivariate Power Series Ring in t, u, v over Rational Field
```
```
sage: R.base_extend(RR)
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
Multivariate Power Series Ring in x, y over Finite Field of size 65537
```
```
sage: S.change_ring(GF(5))
Multivariate Power Series Ring in x, y over Finite Field of size 5
```

Coercion from polynomial ring:

```python
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
```
```python
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:

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

```python
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)
<type '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, trac ticket #13412 and trac ticket #14084

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


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 `sage.rings.multi_power_series_ring_element.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)
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
Multivariate Power Series Ring in x, y over Finite Field of size 65537
sage: S.change_ring(GF(5))
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
Multivariate Power Series Ring in f0, f1, f2, f3 over Finite Field of size 65537
sage: H.characteristic()
65537
construction()
Returns a functor F and base ring R such that F(R) == self.

EXAMPLES:

```python
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:

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

is_dense()
Is self dense? (opposite of sparse)

EXAMPLES:

```python
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:

```python
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'); W
Multivariate Power Series Ring in x, y over Integer Ring
sage: W.is_noetherian()
False

is_sparse()
Is self sparse?

EXEMPLARY:

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'); M
Multivariate Power Series Ring in v0 over Integer Ring
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

(continues on next page)
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.pickle_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
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 %= 2
s + 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 + 0(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 + 0(s, t)^6
sage: f == g
False
```

(continues on next 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 + s^4 + 3*s^3*t + R.O(5) ; f
s^2 + s*t + s^3 + s^2*t + s^4 + 3*s^3*t + O(s, t)^5

sage: f(t,s)
s^2 + s*t + t^2 + s^3 + 3*s^2*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 + O(s, t)^5

0 has valuation $+\infty$:

sage: f(t^2,0)
t^4 + t^6 + 3*t^8 + O(s, t)^10
sage: f(t^2,s^2+s)
s^2*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^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^3 - 2*t^6 + s^7 + s^5*t^2 + R.O(10)

sage: r1 = s^4 - s^7 + s^6*t - 4*s^2*t^5 - 6*s^3*t^5 + R.O(10)
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 - 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, trac ticket #13412

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

Bases: object

Object representing a zero element with given precision.
class sage.rings.multi_power_series_ring_element.MPowerSeries(parent, x=0, prec=+ Infinity, is_gen=False, check=False)

Bases: sage.rings.power_series_ring_element.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
sage: S.<s,t> = PowerSeriesRing(ZZ)
sage: f = s + 4*t + 3*s^s*t
sage: f in S
True
sage: f = f.add_bigoh(4); f
s + 4*t + 3*s^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
```
Multivariate Power Series Ring in \(t_0, t_1, t_2, t_3, t_4\) over Finite Field of size 3

```
sage: t = T.gens()
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
1 + 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: R = PolynomialRing(ZZ, 5, T.variable_names())
sage: t = R.gens()
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
```

\(O(\text{prec})\)

Return a multivariate power series of precision \(\text{prec}\) obtained by truncating \(\text{self}\) at precision \(\text{prec}\).

This is the same as \(\text{add\_bigoh()}.\)

**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 \(\text{self}\):

```
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

\[
\sum a_{m_0, \ldots, m_k} x_0^{nm_0} \cdots x_k^{nm_k}.
\]

The total-degree precision of the output is \( n \) times the precision of \( \text{self} \).

**EXAMPLES:**

```
sage: H = QQ[[x,y,z]]
sage: (x,y,z) = H.gens()
sage: h = -x*y^4*z^7 - 1/4*y^2*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)^\rightarrow 45
```

**add_bigoh** *(prec)*

Return a multivariate power series of precision \( \text{prec} \) obtained by truncating \( \text{self} \) at precision \( \text{prec} \).

This is the same as \( \text{O()} \).

**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 \( \text{self} \):

```
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()
{t: 1, s: 1, 1: 1}
sage: (f^2).coefficients()
{t^2: 1, s^2: 4, s^t: 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)
\begin{Verbatim}
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
\end{Verbatim}

\textbf{constant\_coefficient()}

Return constant coefficient of \texttt{self}.

\textbf{degree()}

Return degree of underlying polynomial of \texttt{self}.

\textbf{derivative(*args)}

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

\textbf{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)
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 $\mathbb{R}$ or $\mathbb{C}$:

```
sage: exp(g.change_ring(RDF))
7.38905609... + 7.38905609...*a + 3.69452804...*b + 14.7781121...*a*b + 3.69452804...*a^2 + 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 + O(a, b)^7
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.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 + 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., \( \mathbb{Z} \)), or if it has a non-zero characteristic (e.g., \( \mathbb{Z}/3\mathbb{Z} \)), 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
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
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
 sage: R.<a,b,c> = PowerSeriesRing(Zmod(8)); R
Multivariate Power Series Ring in a, b, c over Ring of integers modulo 8
```

(continues on next page)
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:
```python
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:

```python
sage: g = 2+f
sage: log(g)
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 $\mathbb{R}$ or $\mathbb{C}$:

```python
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:

```python
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()
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 \texttt{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*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 \texttt{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 \texttt{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
```

\texttt{quo_rem}(other, \texttt{precision=None})

Return the pair of quotient and remainder for the increasing power division of \texttt{self} by \texttt{other}.

If \(a \) and \(b \) are two elements of a power series ring \(R[[x_1, x_2, \ldots, x_n]]\) 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 \((u, v) \in R[[x_1, x_2, \ldots, x_n]] \times R[x_1, x_2, \ldots, x_n] \) such that \(a = bu + v \) and such that no monomial appearing in \(v \) divides the trailing monomial (\texttt{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 \texttt{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:

```python
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
```

```python
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
```

```python
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
```

```python
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:

```python
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)
```

```python
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)
```

```python
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)
```

```python
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)
```

```python
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)
```
Illustrating the dependency on the ordering of variables:

```sage
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
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 + 0(a, b, c)^8
sage: f.trailing_monomial()
a^2*b^3
```

truncate(Infinity)

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

EXAMPLES:

```sage
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]**3 - 3*t[1]**3*t[3]**3 - 1/4*t[0]**t[1]**3*t[3]**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 \( \leq k \) is zero. (If such an integer does not exist, then the valuation is the precision of \( f \) itself.)

EXAMPLES:

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

(continues on next page)
```python
sage: t = T.gens()
sage: w
```

t₀ + t₀*t₂ - t₄^3 - t₀^3*t₂^2 + O(t₀, t₁, t₂, t₃, t₄)^6

```python
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.
EXAMPLES:

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

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
```

When the base ring is a domain, the fraction field is the Laurent series ring over the fraction field of the base ring:
Laurent series rings are determined by their variable and the base ring, and are globally unique:

```
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')
sage: K
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: loads(K.dumps()) == K
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[[x]]
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 `sage.rings.laurent_series_ring_element.LaurentSeries`

**base_extend**($R$)

Return the Laurent series ring over $R$ in the same variable as self, assuming there is a canonical coercion 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
```

**change_ring**($R$)

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

4
```

**characteristic()**

EXAMPLES:

```python
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:

```python
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
default_prec()

sage: phi(arg) == L
True
```

Because of this construction, pushout is automatically available:

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

```python
sage: QQbar.gen() * t
I*t
sage: parent(QQbar.gen() * t)
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:

```python
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:
```python
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
```

```python
gen(n=0)

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

```python
is_dense()

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

```python
is_exact()

Laurent series rings are inexact.

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

```python
is_field(proof=True)

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

```python
is_sparse()

Return if self is a sparse implementation.

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

```python
laurent_polynomial_ring()

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

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

```python
ngens()

Laurent series rings are univariate.
```
EXAMPLES:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.
EXAMPLES:

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

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

```sage
sage: K.<q> = Frac(CC[[q]])
sage: K
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: q
1.00000000000000*q
```

Saving and loading.

```sage
sage: loads(q.dumps()) == q
True
sage: loads(K.dumps()) == K
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: Cython version
class sage.rings.laurent_series_ring_element.LaurentSeries
Bases: sage.structure.element.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 \( O(q^{prec}) \), 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 \( O(q^{prec}) \) 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:**

```python
gap
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:**

```python
gap
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:**

```python
gap
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:**

```python
gap
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
```
```python
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:**

```python
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)

(continues on next page)
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 a nonzero coefficient.
```

Another example with just one negative coefficient:

```
sage: A.<t> = QQ[['t']]
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., $s^{-1}$. 

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

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

EXAMPLES:

```python
sage: R.<t> = LaurentSeriesRing(QQ)
sage: (2+t).is_unit()
True
```

```python
sage: f = 2*t^2+O(t^10); f.is_unit()
True
```

```python
sage: 1/f
1/2 - 1/4*t^2 + 1/8*t^4 - 1/16*t^6 + 1/32*t^8 + O(t^10)
```

```python
sage: R(0).is_unit()
False
```

```python
sage: R.<s> = LaurentSeriesRing(ZZ)
sage: f = 2 + s^2 + O(s^10)
sage: f.is_unit()
False
```

```python
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:

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

```python
sage: z = 0*f
sage: z.is_zero()
1
```

laurent_polynomial()

Return the corresponding Laurent polynomial.

EXAMPLES:

```python
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = t^-3 + t + 7*t^2 + O(t^5)
sage: g = f.laurent_polynomial(); g
```

```python
t^-3 + t + 7*t^2
```

```python
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:**

```python
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:**

```python
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:**

```python
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: 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:**

```python
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 \(\ldots + O(x^r)\) 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
\[ f = \sum_{n \in \mathbb{Z}} a_n t^n = \cdots + \frac{a_{-2}}{t^2} + \frac{a_{-1}}{t} + a_0 + a_1 t + a_2 t^2 + \cdots , \]
then the residue of \( f \) is \( a_{-1} \). Alternatively this is the coefficient of \( 1/t \).

**EXAMPLES:**

```python
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 \( f\text{.prec()} \)). If \( f \) has infinite precision, and the argument \( precision \) is not given, then the precision of the reverse defaults to the default precision of \( f\text{.parent()} \).

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:**

```python
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)
sage: g = f.reverse(); g
t - b*t^2 + (2*b^2 - c)*t^3 + O(t^4)
sage: f(g)
```
(continues on next page)
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)
sage: f(g)
x + O(x^3)

(continues on next page)
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 + 0(x^20)
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:

```python
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)
```

**truncation_neg(n)**

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

This is equivalent to:

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

EXAMPLES:

```python
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:

```python
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:

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

**valuation_zero_part()**

EXAMPLES:

```python
sage: x = Frac(QQ[['x']]).0
sage: f = x + x^2 + 3*x^4 + O(x^7)
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)
```
A lazy Laurent series is a Laurent series whose coefficients are computed as demanded or needed. Unlike the usual Laurent series in Sage, lazy Laurent series do not have precisions because a lazy Laurent series knows (can be computed, lazily) all its coefficients.

EXAMPLES:

Generating functions are Laurent series over the integer ring:

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

This defines the generating function of Fibonacci sequence:

```
sage: def coeff(s, i):
....:     if i in [0, 1]:
....:         return 1
....:     else:
....:         return s.coefficient(i - 1) + s.coefficient(i - 2)
sage: f = L.series(coeff, valuation=0); f
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + ...
```

The 100th element of Fibonacci sequence can be obtained from the generating function:

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

Coefficients are computed and cached only when necessary:

```
sage: f._cache[100]
573147844013817084101
sage: f._cache[101]
Traceback (most recent call last):
  ...
KeyError: 101
```

You 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 + ...
sage: f^(-1)
1 - z - z^2 + ...
sage: f + f^(-1)
2 + z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + ...
```

(continues on next page)
You may need to change the base ring:

```python
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 + ...
```

```
sage: h = g.change_ring(QQ)
sage: h.parent()
Lazy Laurent Series Ring in z over Rational Field
sage: h
4*z + 6*z^2 + 8*z^3 + 19*z^4 + 38*z^5 + 71*z^6 + ...
```

```
sage: h^-1
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 + ...

sage: _.valuation()
-1
```

AUTHORS:

- Kwankyu Lee (2019-02-24): initial version

```python
class sage.rings.lazy_laurent_series.LazyLaurentSeries(parent, coefficient=None, valuation=0, constant=None)

Bases: sage.structure.element.ModuleElement

Return a lazy Laurent series.

INPUT:

- `coefficient` – Python function that computes coefficients
- `valuation` – integer; approximate valuation of the series
- `constant` – either `None` or pair of an element of the base ring and an integer

Let the coefficient of index `i` mean the coefficient of the term of the series with exponent `i`.

Let `valuation` be `n`. All coefficients of index below `n` are zero. If `constant` is `None`, then the `coefficient` function is responsible to compute the values of all coefficients of index `≥ n`. If `constant` is a pair `(c, m)`, then the `coefficient` function is responsible to compute the values of all coefficients of index `≥ n` and `< m` and all the coefficients of index `≥ m` is the constant `c`.

EXAMPLES:

```python
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.series(lambda s, i: i, valuation=-3, constant=(-1,3))
-3*z^-3 - 2*z^-2 - z^-1 + z + 2*z^2 + 3*z^3 + z^4 - z^5 + ...
```

```python
sage: def coeff(s, i):
.....:     if i in [0, 1]:
.....:         return 1
.....:     else:
.....:         return s.coefficient(i - 1) + s.coefficient(i - 2)
sage: f = L.series(coeff, valuation=0); f
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + ...
```

```python
sage: f.coefficient(100)
573147844013817084101
```
Lazy Laurent series is picklable:

```python
sage: z = L.gen()
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 + ...
sage: g = loads(dumps(f))
sage: g
1 + z + 2*z^2 + 3*z^3 + 5*z^4 + 8*z^5 + 13*z^6 + ...
sage: g == f
True
```

**apply_to_coefficients**(function)

Return the series with function applied to each coefficient of this series.

**INPUT:**

- function – Python function

Python function function returns a new coefficient for input coefficient.

**EXAMPLES:**

```python
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: s = z/(1 - 2*z)
sage: t = s.apply_to_coefficients(lambda c: c + 1)
sage: s
z + 2*z^2 + 4*z^3 + 8*z^4 + 16*z^5 + 32*z^6 + 64*z^7 + ...
sage: t
2*z + 3*z^2 + 5*z^3 + 9*z^4 + 17*z^5 + 33*z^6 + 65*z^7 + ...
```

**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:**

```python
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 + ...
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)
```

(continues on next page)
change_ring(ring)
Return this series with coefficients converted to elements of ring.

INPUT:
• ring – a ring

EXAMPLES:

```python
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: s = 2 + z
def g(s, i):
    if i == 0:
        return 1
    else:
        return sum(s.coefficient(j)*s.coefficient(i - 1 -j) for j in [0..i-1])
sage: e = L.series(g, valuation=0)
sage: e.coefficient(10)
16796
```

coefficient(n)
Return the coefficient of the term with exponent n of the series.

INPUT:
• n – integer

EXAMPLES:

```python
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: g(s, i):
    ...:     return 1

sage: e = L.series(g, valuation=0)
sage: e.coefficient(10)
16796
```

polynomial(degree=None, name=None)
Return the polynomial or Laurent polynomial if the series 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 truncated first. If degree is None and the series is not a polynomial or a Laurent polynomial, a ValueError is raised.

EXAMPLES:

```python
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: f = L.series([1,0,0,2,0,0,0,3], 5); f
```
\[ z^5 + 2z^8 + 3z^{12} \]
\[ \text{sage: } f = \text{f}.\text{polynomial()} \]
\[ 3z^{12} + 2z^8 + z^5 \]

\[ \text{sage: } g = L.\text{series([1,0,0,2,0,0,0,3]}, -5); g \]
\[ z^{-5} + 2z^{-2} + 3z^2 \]
\[ \text{sage: } g = g.\text{polynomial()} \]
\[ z^{-5} + 2z^{-2} + 3z^2 \]

\[ \text{sage: } z = L.\text{gen()} \]
\[ \text{sage: } f = (1 + z)/(z^3 - z^5) \]
\[ \text{sage: } f \]
\[ z^{-3} + z^{-2} + z^{-1} + 1 + z + z^2 + z^3 + ... \]
\[ \text{sage: } f = f.\text{polynomial}(5) \]
\[ z^{-3} + z^{-2} + z^{-1} + 1 + z + z^2 + z^3 + z^4 + z^5 \]
\[ \text{sage: } f = f.\text{polynomial}(0) \]
\[ z^{-3} + z^{-2} + z^{-1} + 1 \]
\[ \text{sage: } f = f.\text{polynomial}(-5) \]
\[ 0 \]

\text{\texttt{prec()}}

Return the precision of the series, which is infinity.

EXAMPLES:

\[ \text{sage: } L.\text{gen()} = \text{LazyLaurentSeriesRing(ZZ)} \]
\[ \text{sage: } f = 1/(1 - z) \]
\[ \text{sage: } f = f.\text{prec()} \]
\[ +\text{Infinity} \]

\text{\texttt{truncate(d)}}

Return this series with its terms of degree >= d truncated.

INPUT:

\begin{itemize}
  \item d – integer
\end{itemize}

EXAMPLES:

\[ \text{sage: } L.\text{gen()} = \text{LazyLaurentSeriesRing(ZZ)} \]
\[ \text{sage: } \alpha = 1/(1 - z) \]
\[ \text{sage: } \alpha = \alpha.\text{truncate}(5) \]
\[ \text{sage: } \beta = \alpha.\text{truncate}(5) \]
\[ \text{sage: } \alpha = \alpha - \beta \]
\[ z^5 + z^6 + z^7 + z^8 + z^9 + z^{10} + z^{11} + ... \]

\text{\texttt{valuation()}}

Return the valuation of the series.

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
CHAPTER TEN

LAZY LAURENT SERIES RINGS

The ring of lazy Laurent series over a ring has usual arithmetic operations, but it is actually not a ring in the usual sense since every arithmetic operation gives a new series.

EXAMPLES:

The definition of Laurent series rings is not initially imported into the global namespace. You need to import it explicitly to use it:

```
sage: L.<z> = LazyLaurentSeriesRing(QQ)
sage: L.category()
Category of magmas and additive magmas
sage: 1/(1 - z)
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + ...
sage: 1/(1 - z) == 1/(1 - z)
True
```

Lazy Laurent series ring over a finite field:

```
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
```

Generating functions of integer sequences are Laurent series over the integer ring:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ); L
Lazy Laurent Series Ring in z over Integer Ring
sage: 1/(1 - 2*z)^3
1 + 6*z + 24*z^2 + 80*z^3 + 240*z^4 + 672*z^5 + 1792*z^6 + ...
```

Power series can be defined recursively:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L.series(lambda s,n: (1 + z*s^2)[n], valuation=0)
1 + z + 2*z^2 + 5*z^3 + 14*z^4 + 42*z^5 + 132*z^6 + ...
```

AUTHORS:

• Kwankyu Lee (2019-02-24): initial version
class sage.rings.lazy_laurent_series_ring.LazyLaurentSeriesRing(base_ring, names, category=None)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

Lazy Laurent series ring.

INPUT:

• base_ring – base ring of this Laurent series ring
• names – name of the generator of this Laurent series ring

EXAMPLES:

```
sage: LazyLaurentSeriesRing(ZZ, 't')
Lazy Laurent Series Ring in t over Integer Ring
```

Element

alias of sage.rings.lazy_laurent_series.LazyLaurentSeries
gen(n=0)

Return the generator of this Laurent series ring.

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 tuple of the generator.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: 1/(1 - z)
1 + z + z^2 + z^3 + z^4 + z^5 + z^6 + ...
```
gens()

Return the number of generators of this Laurent series ring.

This is always 1.

EXAMPLES:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: L.ngens()
1
```
one()

Return the constant series 1.

EXAMPLES:
sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.one()
1

series(coefficient, valuation, constant=None)

Return a lazy Laurent series.

INPUT:

- coefficient – Python function that computes coefficients
- valuation – integer; approximate valuation of the series
- constant – either None or pair of an element of the base ring and an integer

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 None, then the coefficient function is responsible to compute the values of all coefficients of index \( \geq n \). If 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 + ...
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 + 2*z^-4 + 3*z^-3 + 4*z^-2 + ...

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.

EXAMPLES:
sage: g = L.series([1,3,5,7,9], 5, -1)
sage: g
z^5 + 3*z^6 + 5*z^7 + 7*z^8 + 9*z^9 - z^10 - z^11 - z^12 + ...

zero()

Return the zero series.

EXAMPLES:

sage: L = LazyLaurentSeriesRing(ZZ, 'z')
sage: L.zero()
0
LAZY LAURENT SERIES OPERATORS

This module implements operators internally used to construct lazy Laurent series. The job of an operator attached to a series is to compute the $n$-th coefficient of the series if $n$ is not less than the valuation of the series and the $n$-th coefficient is not declared to be a constant.

If a new operator is added to this module, an example of how it is used should be added below.

EXAMPLES:

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

Constructors:

```
sage: L(1)
1
```

```
sage: L.series([1,2,3,4], -10)
z^-10 + 2*z^-9 + 3*z^-8 + 4*z^-7
```

```
sage: L.gen()
z
```

```
sage: P.<x> = LaurentPolynomialRing(ZZ)
sage: p = (1 + 1/x)^3 + (1 + x)^4
sage: L(p)
z^-3 + 3*z^-2 + 3*z^-1 + 2 + 4*z + 6*z^2 + 4*z^3 + z^4
```

Unary operators:

```
sage: ~f
1 - 2*z - 4*z^2 - 8*z^3 - 16*z^4 - 32*z^5 - 64*z^6 + ...
```

```
sage: f + g
2 + 2*z + 3*z^2 + 8*z^3 + 17*z^4 + 32*z^5 + 63*z^6 + ...
```
sage: f - g
2*z + 5*z^2 + 8*z^3 + 15*z^4 + 32*z^5 + 65*z^6 + 128*z^7 + ...

sage: f * g
1 + 2*z + 3*z^2 + 6*z^3 + 13*z^4 + 26*z^5 + 51*z^6 + ...

sage: f / g
1 + 2*z + 5*z^2 + 10*z^3 + 20*z^4 + 40*z^5 + 80*z^6 + ...

Transformers:

sage: 2*f
2 + 4*z + 8*z^2 + 16*z^3 + 32*z^4 + 64*z^5 + 128*z^6 + ...

sage: f.change_ring(GF(3))
1 + 2*z + z^2 + 2*z^3 + z^4 + 2*z^5 + z^6 + ...

sage: f.apply_to_coefficients(lambda c: c^2)
1 + 4*z + 16*z^2 + 64*z^3 + 256*z^4 + 1024*z^5 + 4096*z^6 + ...

sage: f.truncate(5)
1 + 2*z + 4*z^2 + 8*z^3 + 16*z^4

AUTHORS:

• Kwankyu Lee (2019-02-24): initial version

class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesBinaryOperator(left, right)
Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator

Abstract base class for binary operators.

INPUT:

• left – series on the left side of the binary operator
• right – series on the right side of the binary operator

class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator
Bases: object

Base class for operators computing coefficients of a lazy Laurent series.

Subclasses of this class are used to implement arithmetic operations for lazy Laurent series. These classes are not to be used directly by the user.

class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_add(left, right)
Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesBinaryOperator

Operator for addition.

class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_apply(series, function)
Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator

Operator for applying a function.

INPUT:

• series – a lazy Laurent series
• function – a Python function to apply to each coefficient of the series

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_change_ring(series, ring):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator
    Operator for changing the base ring of the series to ring.
    INPUT:
    • series – a lazy Laurent series
    • ring – a ring
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_constant(ring, constant):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator
    Operator for the generator element.
    INPUT:
    • ring – a lazy Laurent series ring
    • constant – a constant of the base ring of ring
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_div(left, right):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesBinaryOperator
    Operator for division.
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_gen(ring):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator
    Operator for the generator element.
    INPUT:
    • ring – a lazy Laurent series ring
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_inv(series):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesUnaryOperator
    Operator for inversion.
    INPUT:
    • series – a lazy Laurent series
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_list(ring, l, v):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator
    Operator for the series defined by a list.
    INPUT:
    • l – list
    • v – integer
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_mul(left, right):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesBinaryOperator
    Operator for multiplication.
```

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_neg(series):
    Bases: sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesUnaryOperator
```
Operator for negation.

**INPUT:**
- `series` – a lazy Laurent series

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_polynomial(ring, poly)
```

Bases: `sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator`

Operator for the series coerced from a polynomial or a Laurent polynomial.

**INPUT:**
- `ring` – a lazy Laurent series ring
- `poly` – a polynomial or a Laurent polynomial

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_scale(series, scalar)
```

Bases: `sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator`

Operator for scalar multiplication of `series` with `scalar`.

**INPUT:**
- `series` – a lazy Laurent series
- `scalar` – an element of the base ring

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_sub(left, right)
```

Bases: `sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesBinaryOperator`

Operator for subtraction.

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator_truncate(series, d)
```

Bases: `sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator`

Operator for truncation.

**INPUT:**
- `series` – a lazy Laurent series
- `d` – an integer; the series is truncated the terms of degree > `d`

```python
class sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesUnaryOperator(series)
```

Bases: `sage.rings.lazy_laurent_series_operator.LazyLaurentSeriesOperator`

Abstract base class for unary operators.

**INPUT:**
- `series` – series upon which the operator operates
CHAPTER
TWELVE

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: sage.structure.unique_representation.UniqueRepresentation, sage.rings.ring.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 sage.rings.puiseux_series_ring_element.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')
sage: A.default_prec()
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 \texttt{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')
sage: A.gen()
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')
sage: A.laurent_series_ring()  # 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')
sage: A.ngens()  # 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
```
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:

```python
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:

```python
sage: p = x^(1/2); p
x^(1/2)
```

```python
sage: p.ramification_index()
2
```

```python
sage: q = x^(1/2) + x**(1/3); q
x^(1/3) + x^(1/2)
```

```python
sage: q.ramification_index()
6
```

Other arithmetic can be performed with Puiseux Series:

```python
sage: p + q
x^(1/3) + 2*x^(1/2)
```

```python
sage: p - q
-x^(1/3)
```

```python
sage: p * q
x^(5/6) + x
```

```python
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:

```python
sage: I*q
Traceback (most recent call last):
(continues on next page)
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

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 \texttt{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)
```

\texttt{add\_bigoh(\texttt{prec})}

Return the truncated series at chosen precision \texttt{prec}.

INPUT:

- \texttt{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)
```

\textbf{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^(1/3))
```

\texttt{change\_ring(\texttt{R})}

Return \texttt{self} over a the new ring \texttt{R}.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
```

(continues on next page)
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*7*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: 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
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) + O(x^5)
sage: q = p.add_bigoh(5); q
x^(-2/3) + 4*x^(8/3) + O(x^5)
precision_absolute()  
Return the precision of self.

EXAMPLES:

```
sage: R.<x> = PuiseuxSeriesRing(ZZ)
sage: p = (x**(1/3) + 2*x**(2/3))**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) + x**(8/3))**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^7/2
```

truncate(r)  
Return the Puiseux series of degree \(< r\).

This is equivalent to self modulo \(x^r\).
EXAMPLES:

```python
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:

```python
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:

```python
sage: R.<x> = PuiseuxSeriesRing(QQ)
sage: p = x^(-7/2) + 3 + 5*x^(1/2) - 7*x**3
sage: p.variable()
'x'
```
CHAPTER
FOURTEEN

TATE ALGEBRAS

Let $K$ be a finite extension of $\mathbb{Q}_p$ for some prime number $p$ and let $(v_1, \ldots, v_n)$ be a tuple of real numbers. The associated Tate algebra consists of series of the form

$$\sum_{i_1, \ldots, i_n \in \mathbb{N}} a_{i_1, \ldots, i_n} x_1^{i_1} \cdots x_n^{i_n}$$

for which the quantity

$$\text{val}(a_{i_1, \ldots, i_n}) - (v_1 i_1 + \cdots + v_n i_n)$$

goes to infinity when the multi-index $(i_1, \ldots, i_n)$ goes to infinity.

These series converge on the closed disc defined by the inequalities $\text{val}(x_i) \geq -v_i$ for all $i \in \{1, \ldots, n\}$. 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 + ...00010*x*y^3 + ...000100*x^2*y^2
sage: g = x^3*y + 2*x*y; g
...00010^x^3*y + ...00010*x*y

and perform all usual arithmetic operations on them:

sage: f + g
...00010*x^3*y + ...00010*x^4*y^4 + ...00010*x*y + ...000100*x^5*y^3 + ...000100*x^2*y^4 + ...0001000*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: var('u')
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()
[...00010*x^3*y + ...00010*y^4 + ...10001*x^2 + ... + O(2^5 * <x, y>),
 ...00001*x*y^4 + ...11001*y^3 + ...00001 + ... + O(2^5 * <x, y>),
 ...00001*y^5 + ...11111*x*y^3 + ...01001*x^2*y + ... + O(2^5 * <x, y>),
(continues on next page)
Construct a Tate algebra over a $p$-adic field.

Given a $p$-adic field $K$, variables $X_1, \ldots, X_k$ and convergence log radii $v_1, \ldots, v_n$ in $R$, the corresponding Tate algebra $KX_1, \ldots, X_k$ consists of power series with coefficients $a_{i_1, \ldots, i_n}$ in $K$ such that

$$\text{val}(a_{i_1, \ldots, i_n}) - (i_1 v_1 + \cdots + i_n v_n)$$

tends to infinity as $i_1, \ldots, i_n$ 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:

```python
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:

```python
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()`:

```python
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
```
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:

```python
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:

```python
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:**

```python
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: sage.rings.ring.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: 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):
(continues on next page)```
ValueError: generator not defined

**gens()**

Return the list of generators of this Tate algebra.

**EXAMPLES:**

```sage
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
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()**

Return True since any Tate algebra is an integral domain.

**EXAMPLES:**

```sage
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
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:
random_element\( (\text{degree}=2, \text{terms}=5, \text{integral}=\text{False}, \text{prec}=\text{None})\)

Return a random element of this Tate algebra.

INPUT:

- \text{degree} – an integer (default: 2), an upper bound on the total degree of the result
- \text{terms} – an integer (default: 5), the maximal number of terms of the result
- \text{integral} – a boolean (default: \text{False}); if \text{True} the result will be in the ring of integers
- \text{prec} – (optional) an integer, the precision of the result

EXAMPLES:

```python
sage: R = Zp(2, prec=10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.random_element()  # random
(...0101100.01)*y + ...1100100010*x^2 + ...0010001111*x*y + ...1110001011 + ... →01000100*y^2
sage: A.random_element(degree=5, terms=3)  # random
(...0101100.01)*x^2*y + (...0101000111)*y^2 + ...01111011*x*y
sage: A.random_element(integral=True)  # random
...0001111111*x + ...1101101101 + ...00010010110*y + ...1110110001 + ... →000001101000*y^2
```

Note that if we are already working on the ring of integers, specifying \text{integral}=\text{False} has no effect:

```python
sage: Ao = A.integer_ring()
sage: f = Ao.random_element(integral=False); f  # random
...1100110111*x^2 + ...1000101011*x*y + ...1101100001 + ... →01000100*y^2
sage: f in Ao
True
```

When the log radii are negative, integral series may have non integral coefficients:

```python
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 + ... →0001001101*y + ...001010001100
```

some_elements()

Return a list of elements in this Tate algebra.

EXAMPLES:

```python
sage: R = Zp(2, 10, print_mode='digits')
sage: A.<x,y> = TateAlgebra(R)
sage: A.some_elements()
(continues on next page)```
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: sage.monoids.monoid.Monoid_class, sage.structure.unique_representation.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 sage.rings.tate_algebra_element.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

(continues on next page)
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:

```python
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:

```python
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:

```python
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:

```python
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()
[...0000000010, ...000000001*x, ...000000001*y, ...0000000010*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')
```
INDICES AND TABLES

- Index
- Module Index
- Search Page
sage.rings.laurent_series_ring, 75
sage.rings.laurent_series_ring_element, 81
sage.rings.lazy_laurent_series, 95
sage.rings.lazy_laurent_series_operator, 105
sage.rings.lazy_laurent_series_ring, 101
sage.rings.multi_power_series_ring, 49
sage.rings.multi_power_series_ring_element, 57
sage.rings.power_series_pari, 43
sage.rings.power_series_poly, 37
sage.rings.power_series_ring, 1
sage.rings.power_series_ring_element, 11
sage.rings.puiseux_series_ring, 109
sage.rings.puiseux_series_ring_element, 113
sage.rings.tate_algebra, 121
INDEX

A

absolute_e() (sage.rings.tate_algebra.TateAlgebra_generic method), 125
add_bigoh() (sage.rings.laurent_series_ring_element.LaurentSeries method), 82
add_bigoh() (sage.rings.multi_power_series_ring_element.MPowerSeries method), 62
add_bigoh() (sage.rings.power_series_ring_element.PowerSeries method), 13
add_bigoh() (sage.rings.puiseux_series_ring_element.PuiseuxSeries method), 115
algebra_of_series()
(sage.rings.tate_algebra.TateTermMonoid method), 129
apply_to_coefficients()
(sage.rings.lazy_laurent_series.LazyLaurentSeries method), 97
approximate_series()
(sage.rings.lazy_laurent_series.LazyLaurentSeries method), 97
change_ring()
(sage.rings.power_series_ring_element.PuiseuxSeries method), 109
change_ring()
(sage.rings.lazy_laurent_series.LazyLaurentSeries method), 98
change_ring()
(sage.rings.multi_power_series_ring.MPowerSeriesRing_generic method), 53
change_ring()
(sage.rings.power_series_ring.PowerSeriesRing_generic method), 5
change_ring()
(sage.rings.power_series_ring_element.PowerSeries method), 13
change_ring()
(sage.rings.puiseux_series_ring.PuiseuxSeriesRing method), 115
change_var()
(sage.rings.power_series_ring.PowerSeriesRing_generic method), 6
characteristic()
(sage.rings.laurent_series_ring.LaurentSeriesRing method), 77
characteristic()
(sage.rings.multi_power_series_ring.MPowerSeriesRing_generic method), 53
characteristic()
(sage.rings.power_series_ring.PowerSeriesRing_generic method), 6
characteristic()
(sage.rings.tate_algebra.TateAlgebra_generic method), 125
coefficient()
(sage.rings.lazy_laurent_series.LazyLaurentSeries method), 98
coefficients()
(sage.rings.laurent_series_ring_element.LaurentSeries method), 83
coefficients()
(sage.rings.multi_power_series_ring_element.MPowerSeries method), 62
coefficients()
(sage.rings.power_series_ring_element.PowerSeries method), 14
coefficients()
(sage.rings.puiseux_series_ring_element.PuiseuxSeries method), 116
coefficients()
(sage.rings.puiseux_series_ring_element.PuiseuxSeriesRing method), 14
common_prec()
(sage.rings.laurent_series_ring_element.LaurentSeries method), 116
common_prec()
(sage.rings.power_series_ring_element.PowerSeries method), 83
common_prec()
(sage.rings.power_series_ring_element.PuiseuxSeries method), 14
common_prec()
(sage.rings.puiseux_series_ring_element.PuiseuxSeriesRing method), 116
common_valuation()
(sage.rings.laurent_series_ring_element.LaurentSeries method), 84
constant_coefficient()
(sage.rings.multi_power_series_ring_element.MPowerSeries method), 83

B

base_extend() (sage.rings.laurent_series_ring.LaurentSeriesRing method), 76
base_extend() (sage.rings.power_series_ring.PowerSeriesRing_generic method), 5
base_extend() (sage.rings.power_series_ring_element.PowerSeries method), 13
base_extend() (sage.rings.puiseux_series_ring.PuiseuxSeriesRing method), 109
base_ring()
(sage.rings.power_series_ring_element.PowerSeries method), 13
base_ring()
(sage.rings.tate_algebra.TateTermMonoid method), 130
bigoh()
(sage.rings.multi_power_series_ring_element.MPowerSeriesRing_generic method), 53

C

change_ring()
(sage.rings.laurent_series_ring.LaurentSeriesRing method), 76
change_ring()
(sage.rings.laurent_series_ring_element.LaurentSeries method), 83
change_ring()
(sage.rings.multi_power_series_ring_element.MPowerSeries method), 53

137
construction() (sage.rings.laurent_series_ring.LaurentSeriesRing method), 77
construction() (sage.rings.multi_power_series_ring.MPowerSeriesRing_generic method), 53
construction() (sage.rings.power_series_ring.PowerSeriesRing_generic method), 6
create_key() (sage.rings.laurent_series_ring_element.LaurentSeriesRingElement method), 129
cos() (sage.rings.power_series_ring_element.PowerSeries method), 14
cosh() (sage.rings.power_series_ring_element.PowerSeries method), 15
debug() (sage.rings.tate_algebra.TateAlgebraFactory method), 124
default_prec() (sage.rings.laurent_series_ring.LaurentSeriesRing method), 77
default_prec() (sage.rings.puiseux_series_ring.PuiseuxSeriesRing method), 110
degree() (sage.rings.laurent_series_ring_element.LaurentSeriesRingElement method), 84
degree() (sage.rings.multi_power_series_ring_element.MPowerSeriesRing_generic method), 63
degree() (sage.rings.power_series_poly.PowerSeries_poly method), 37
degree() (sage.rings.power_series_ring_element.PowerSeriesRing method), 16
degree() (sage.rings.puiseux_series_ring_element.PuiseuxSeriesRing method), 116
derivative() (sage.rings.laurent_series_ring_element.LaurentSeriesRingElement method), 85
derivative() (sage.rings.multi_power_series_ring_element.MPowerSeriesRing_generic method), 63
derivative() (sage.rings.power_series_ring_element.PowerSeries method), 16
dict() (sage.rings.multi_power_series_ring_element.MPowerSeriesRing_generic method), 63
dict() (sage.rings.power_series_pari.PowerSeries_pari method), 44
dict() (sage.rings.power_series_poly.PowerSeries_poly method), 38
dict() (sage.rings.power_series_poly_element.PowerSeries_poly method), 64
dict() (sage.rings.power_series_ring_element.PowerSeriesRing method), 50
element() (sage.rings.puiseux_series_ring_element.PuiseuxSeriesRingElement method), 78
exp() (sage.rings.power_series_ring_element.MPowerSeriesRing method), 64
exp() (sage.rings.power_series_ring_element.PowerSeries method), 65
exp() (sage.rings.multi_power_series_ring_element.MPowerSeriesRing method), 85
exp() (sage.rings.power_series_ring_element.PowerSeries method), 18
exp() (sage.rings.power_series_ring_element.PuiseuxSeriesRingElement method), 110
fraction_field() (sage.rings.laurent_series_ring.LaurentSeriesRing method), 77
fraction_field() (sage.rings.power_series_ring.PowerSeriesRing method), 5
fraction_field() (sage.rings.power_series_ring_over_field.Element method), 10
fraction_field() (sage.rings.puiseux_series_ring.PuiseuxSeriesRing method), 110
gen() (sage.rings.laurent_series_ring.LaurentSeriesRing method), 78
gen() (sage.rings.lazy_laurent_series_ring.LazyLaurentSeriesRing method), 102
gen() (sage.rings.multi_power_series_ring_element.MPowerSeriesRingGeneric method), 54
gen() (sage.rings.power_series_ring_element.PowerSeriesRingGeneric method), 6
ngen() (sage.rings.puiseux_series_ring_element.PuiseuxSeriesRingElement method), 6
ngen() (sage.rings.tate_algebra.TateAlgebraGeneric method), 125
ngen() (sage.rings.tate_algebra.TateTermMonoid method), 130
ngen() (sage.rings.lazy_laurent_series_ring.LazyLaurentSeriesRing method), 102
integer_ring() (sage.rings.tate_algebra.TateAlgebraGeneric method), 126

method), 78

laurent_series() (sage.rings.multi_power_series_ring_element.MultivariatePowerSeries method), 67

laurent_series() (sage.rings.power_series_ring_element.UnivariatePowerSeries method), 21

laurent_series() (sage.rings.puiseux_series_ring_element.PuiseuxSeries method), 118

laurent_series_ring() (sage.rings.multi_power_series_ring_element.MultivariatePowerSeriesRing_generic method), 55

laurent_series_ring() (sage.rings.power_series_ring_element.UnivariatePowerSeriesRing_generic method), 7

laurent_series_ring() (sage.rings.puiseux_series_ring_element.PuiseuxSeriesRing method), 111

LaurentSeries (class in sage.rings.laurent_series_ring_element), 81

LaurentSeriesRing (class in sage.rings.laurent_series_ring), 75

LazyLaurentSeries (class in sage.rings.laurent_series_ring), 96

LazyLaurentSeriesBinaryOperator (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 106

LazyLaurentSeriesOperator (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 106

LazyLaurentSeriesOperator_add (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 106

LazyLaurentSeriesOperator_apply (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 106

LazyLaurentSeriesOperator_change_ring (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_constant (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_div (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_gen (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_inv (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_list (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_mul (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_neg (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 107

LazyLaurentSeriesOperator_polynomial (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 108

LazyLaurentSeriesOperator_scale (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 108

LazyLaurentSeriesOperator_sub (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 108

LazyLaurentSeriesOperator_truncate (class in sage.rings.laurent_series_ring_element.MPowerSeriesOperator), 108

LazyLaurentSeriesRing (class in sage.rings.laurent_series_ring_element.MPowerSeriesRing_generic), 101

LazyLaurentSeriesUnaryOperator (class in sage.rings.laurent_series_ring_element.MPowerSeriesRing_generic), 108

lift_to_precision() (sage.rings.laurent_series_ring_element.LaurentSeries method), 87

lift_to_precision() (sage.rings.power_series_ring_element.PowerSeries method), 21

list() (sage.rings.laurent_series_ring_element.LaurentSeries method), 88

list() (sage.rings.multi_power_series_ring_element.MPowerSeries method), 67

list() (sage.rings.power_series_pari.PowerSeries method), 44

list() (sage.rings.power_series_poly.PowerSeries method), 38

list() (sage.rings.power_series_ring_element.PowerSeries method), 22

list() (sage.rings.puiseux_series_ring_element.PuiseuxSeries method), 118

log() (sage.rings.multi_power_series_ring_element.MPowerSeries method), 67

log() (sage.rings.power_series_ring_element.PowerSeries method), 22

log_radii() (sage.rings.tate_algebra.TateAlgebra_generic method), 126

log_radii() (sage.rings.tate_algebra.TateTermMonoid method), 131

M

make_element_from_parent_v0() (in module sage.rings.power_series_ring_element), 36

make_powerseries_poly_v0() (in module sage.rings.power_series_poly), 42

make_powerseries_poly_v0() (in module sage.rings.power_series_ring_element.Powerseries method), 36

map_coefficients() (sage.rings.power_series_ring_element.PowerSeries method), 23

M0 (class in sage.rings.multi_power_series_ring_element), 59

module
Index
Index
valuation_zero_part()
  (sage.rings.power_series_ring_element.PowerSeries
   method), 35

variable() (sage.rings.laurent_series_ring_element.LaurentSeries
  method), 93

variable() (sage.rings.multi_power_series_ring_element.MPowerSeries
  method), 72

variable() (sage.rings.power_series_ring_element.PowerSeries
  method), 35

variable() (sage.rings.puiseux_series_ring_element.PuiseuxSeries
  method), 120

variable_names() (sage.rings.tate_algebra.TateAlgebra_generic
  method), 129

variable_names() (sage.rings.tate_algebra.TateTermMonoid
  method), 132

variable_names_recursive()
  (sage.rings.power_series_ring.PowerSeriesRing_generic
   method), 9

variables() (sage.rings.multi_power_series_ring_element.MPowerSeries
  method), 72

verschiebung() (sage.rings.laurent_series_ring_element.LaurentSeries
  method), 94

Z

zero() (sage.rings.lazy_laurent_series_ring.LazyLaurentSeriesRing
  method), 104