has_only_gens(**gens*)

Return True if Poly(f, *gens) retains ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```

homogeneous_order()

Returns the homogeneous order of f.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of f. If you only want to check if a polynomial is homogeneous, then use *Poly.is_homogeneous()* (page 2400).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

homogenize(*s*)

Returns the homogeneous polynomial of f.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use *Poly.is_homogeneous()* (page 2400). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use *Poly.homogeneous_order()* (page 2397).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> f = Poly(x**5 + 2*x**2*y**2 + 9*x*y**3)
>>> f.homogenize(z)
Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

inject(*front=False*)

Inject ground domain generators into f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)
```

```
>>> f.inject()
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')
>>> f.inject(front=True)
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

integrate(**specs, **args*)

Computes indefinite integral of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```
>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

intervals(*all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)

Compute isolating intervals for roots of f.

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).intervals()
[((-2, -1), 1), ((1, 2), 1)]
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[((-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

References

invert(*g*, *auto=True*)

Invert f modulo g when possible.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

property is_cyclotomic

Returns True if f is a cyclotomic polnomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
```

```
>>> Poly(f).is_cyclotomic
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
```

```
>>> Poly(g).is_cyclotomic
True
```

property is_ground

Returns True if f is an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

property is_homogeneous

Returns True if f is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use *Poly.homogeneous_order()* (page 2397).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

property is_irreducible

Returns True if f has no factors over its domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

property is_linear

Returns True if f is linear in all its variables.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

property is_monic

Returns True if the leading coefficient of f is one.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

property is_monomial

Returns True if f is zero or has only one term.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

property is_multivariate

Returns True if f is a multivariate polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
```

(continues on next page)

(continued from previous page)

```
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

property is_one

Returns True if f is a unit polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

property is_primitive

Returns True if GCD of the coefficients of f is one.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 6*x + 12, x).is_primitive
False
>>> Poly(x**2 + 3*x + 6, x).is_primitive
True
```

property is_quadratic

Returns True if f is quadratic in all its variables.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x*y + 2, x, y).is_quadratic
True
>>> Poly(x*y**2 + 2, x, y).is_quadratic
False
```

property is_sqf

Returns True if f is a square-free polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).is_sqf
False
>>> Poly(x**2 - 1, x).is_sqf
True
```

property is_univariate

Returns True if f is a univariate polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_univariate
True
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
False
>>> Poly(x*y**2 + x*y + 1, x).is_univariate
True
>>> Poly(x**2 + x + 1, x, y).is_univariate
False
```

property is_zero

Returns True if f is a zero polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False
```

l1_norm()

Returns l1 norm of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).l1_norm()
6
```

lcm(*g*)

Returns polynomial LCM of f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

length()

Returns the number of non-zero terms in f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 2*x - 1).length()
3
```

lift()

Convert algebraic coefficients to rationals.

Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

ltrim(*gen*)

Remove dummy generators from f that are to the left of specified gen in the generators as ordered. When gen is an integer, it refers to the generator located at that position within the tuple of generators of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y, z
```

```
>>> Poly(y**2 + y*z**2, x, y, z).ltrim(y)
Poly(y**2 + y*z**2, y, z, domain='ZZ')
>>> Poly(z, x, y, z).ltrim(-1)
Poly(z, z, domain='ZZ')
```

match(**args, **kwargs*)

Match expression from Poly. See Basic.match()

max_norm()

Returns maximum norm of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3
```

monic(*auto=True*)

Divides all coefficients by LC(f).

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')
```

```
>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```

monoms(*order=None*)

Returns all non-zero monomials from f in lex order.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

See also:

all_monoms (page 2381)

mul(*g*)

Multiply two polynomials f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

mul_ground(*coeff*)

Multiply f by a an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')
```

neg()

Negate all coefficients in f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')
```

```
>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')
```

classmethod new(*rep*, **gens*)

Construct *Poly* (page 2378) instance from raw representation.

norm()

Computes the product, `Norm(f)`, of the conjugates of a polynomial `f` defined over a number field K.

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> a, b = sqrt(2), sqrt(3)
```

A polynomial over a quadratic extension. Two conjugates x - a and x + a.

```
>>> f = Poly(x - a, x, extension=a)
>>> f.norm()
Poly(x**2 - 2, x, domain='QQ')
```

A polynomial over a quartic extension. Four conjugates x - a, x - a, x + a and x + a.

```
>>> f = Poly(x - a, x, extension=(a, b))
>>> f.norm()
Poly(x**4 - 4*x**2 + 4, x, domain='QQ')
```

nroots(*n=15*, *maxsteps=50*, *cleanup=True*)

Compute numerical approximations of roots of `f`.

Parameters

n … the number of digits to calculate

maxsteps … the maximum number of iterations to do

If the accuracy \`n\` cannot be reached in \`maxsteps\`, it will raise an

exception. You need to rerun with higher maxsteps.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

nth(**N*)

Returns the n-th coefficient of `f` where `N` are the exponents of the generators in the term of interest.

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x, y
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*(sqrt(x)), y, sqrt(x), domain='ZZ')
>>> _.nth(1, 1)
4
```

See also:

coeff_monomial (page 2384)

nth_power_roots_poly(*n*)

Construct a polynomial with n-th powers of roots of `f`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**4 - x**2 + 1)
```

```
>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```

property one

Return one polynomial with self's properties.

pdiv(*g*)

Polynomial pseudo-division of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pdiv(Poly(2*x - 4, x))
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

per(*rep, gens=None, remove=None*)

Create a Poly out of the given representation.

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x, y
```

```
>>> from sympy.polys.polyclasses import DMP
```

```
>>> a = Poly(x**2 + 1)
```

```
>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')
```

pexquo(*g*)

Polynomial exact pseudo-quotient of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 1, x).pexquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).pexquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

pow(*n*)

Raise f to a non-negative power n.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

pquo(*g*)

Polynomial pseudo-quotient of `f` by `g`.

See the Caveat note in the function prem(f, g).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')
```

```
>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

prem(*g*)

Polynomial pseudo-remainder of `f` by `g`.

Caveat: The function prem(f, g, x) can be safely used to compute
in Z[x] _only_ subresultant polynomial remainder sequences (prs's).

To safely compute Euclidean and Sturmian prs's in Z[x] employ anyone of the corresponding functions found in the module sympy.polys.subresultants_qq_zz. The functions in the module with suffix _pg compute prs's in Z[x] employing rem(f, g, x), whereas the functions with suffix _amv compute prs's in Z[x] employing rem_z(f, g, x).

The function rem_z(f, g, x) differs from prem(f, g, x) in that to compute the remainder polynomials in Z[x] it premultiplies the divident times the absolute value of the leading coefficient of the divisor raised to the power degree(f, x) - degree(g, x) + 1.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

primitive()

Returns the content and a primitive form of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

quo(*g*, *auto=True*)

Computes polynomial quotient of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

quo_ground(*coeff*)

Quotient of f by a an element of the ground domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

rat_clear_denoms(*g*)

Clear denominators in a rational function f/g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)
```

```
>>> p, q = f.rat_clear_denoms(g)
```

```
>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

real_roots(*multiple=True*, *radicals=True*)

Return a list of real roots with multiplicities.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).real_roots()
[CRootOf(x**3 + x + 1, 0)]
```

refine_root(*s*, *t*, *eps=None*, *steps=None*, *fast=False*, *check_sqf=False*)

Refine an isolating interval of a root to the given precision.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

rem(*g*, *auto=True*)

Computes the polynomial remainder of f by g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

reorder(**gens*, ***args*)

Efficiently apply new order of generators.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + x*y**2, x, y).reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

replace(*x*, *y=None*, ***_ignore*)

Replace `x` with `y` in generators list.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 1, x).replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

resultant(*g*, *includePRS=False*)

Computes the resultant of `f` and `g` via PRS.

If includePRS=True, it includes the subresultant PRS in the result. Because the PRS is used to calculate the resultant, this is more efficient than calling `subresultants()` (page 2365) separately.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x)
```

```
>>> f.resultant(Poly(x**2 - 1, x))
4
>>> f.resultant(Poly(x**2 - 1, x), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
     Poly(-2, x, domain='ZZ')])
```

retract(*field=None*)

Recalculate the ground domain of a polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')
```

```
>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

revert(*n*)

Compute f**(-1) mod x**n.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(1, x).revert(2)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(1 + x, x).revert(1)
Poly(1, x, domain='ZZ')
```

```
>>> Poly(x**2 - 2, x).revert(2)
Traceback (most recent call last):
...
NotReversible: only units are reversible in a ring
```

```
>>> Poly(1/x, x).revert(1)
Traceback (most recent call last):
...
PolynomialError: 1/x contains an element of the generators set
```

root(*index*, *radicals=True*)

Get an indexed root of a polynomial.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)
```

```
>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3
```

```
>>> Poly(x**5 + x + 1).root(0)
CRootOf(x**3 - x**2 + 1, 0)
```

same_root(*a*, *b*)

Decide whether two roots of this polynomial are equal.

Raises

DomainError

If the domain of the polynomial is not *ZZ* (page 2525), *QQ* (page 2529), *RR* (page 2545), or *CC* (page 2546).

MultivariatePolynomialError

If the polynomial is not univariate.

PolynomialError

If the polynomial is of degree < 2.

Examples

```
>>> from sympy import Poly, cyclotomic_poly, exp, I, pi
>>> f = Poly(cyclotomic_poly(5))
>>> r0 = exp(2*I*pi/5)
>>> indices = [i for i, r in enumerate(f.all_roots()) if f.same_
↪root(r, r0)]
>>> print(indices)
[3]
```

set_domain(*domain*)

Set the ground domain of f.

set_modulus(*modulus*)

Set the modulus of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

shift(*a*)

Efficiently compute Taylor shift f(x + a).

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

slice(*x, m, n=None*)

Take a continuous subsequence of terms of f.

sqf_list(*all=False*)

Returns a list of square-free factors of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16
```

```
>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
     (Poly(x + 2, x, domain='ZZ'), 3)])
```

```
>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
     (Poly(x + 1, x, domain='ZZ'), 2),
     (Poly(x + 2, x, domain='ZZ'), 3)])
```

sqf_list_include(*all=False*)

Returns a list of square-free factors of f.

Examples

```
>>> from sympy import Poly, expand
>>> from sympy.abc import x
```

```
>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4
```

```
>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

```
>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

sqf_norm()

Computes square-free norm of f.

Returns s, f, r, such that g(x) = f(x-sa) and r(x) = Norm(g(x)) is a square-free polynomial over K, where a is the algebraic extension of the ground domain.

Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x
```

```
>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()
```

```
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

sqf_part()

Computes square-free part of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

sqr()

Square a polynomial f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

sturm(*auto=True*)

Computes the Sturm sequence of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

sub(*g*)

Subtract two polynomials f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) - Poly(x - 2, x)
Poly(x**2 - x + 3, x, domain='ZZ')
```

sub_ground(*coeff*)

Subtract an element of the ground domain from f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```

subresultants(*g*)

Computes the subresultant PRS of f and g.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

terms(*order=None*)

Returns all non-zero terms from f in lex order.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[((2, 0), 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

See also:

all_terms (page 2382)

terms_gcd()

Remove GCD of terms from the polynomial f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

termwise(*func, *gens, **args*)

Apply a function to all terms of f.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> def func(k, coeff):
...     k = k[0]
...     return coeff//10**(2-k)
```

```
>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact()

Make the ground domain exact.

Examples

```
>>> from sympy import Poly, RR
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

to_field()

Make the ground domain a field.

Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x, domain=ZZ).to_field()
Poly(x**2 + 1, x, domain='QQ')
```

to_ring()

Make the ground domain a ring.

Examples

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, domain=QQ).to_ring()
Poly(x**2 + 1, x, domain='ZZ')
```

total_degree()

Returns the total degree of `f`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).total_degree()
2
>>> Poly(x + y**5, x, y).total_degree()
5
```

transform(*p, q*)

Efficiently evaluate the functional transformation `q**n * f(p/q)`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).transform(Poly(x + 1, x), Poly(x - 1, x))
Poly(4, x, domain='ZZ')
```

trunc(*p*)

Reduce `f` modulo a constant `p`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

unify(*g*)

Make `f` and `g` belong to the same domain.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import x
```

```
>>> f, g = Poly(x/2 + 1), Poly(2*x + 1)
```

```
>>> f
Poly(1/2*x + 1, x, domain='QQ')
>>> g
Poly(2*x + 1, x, domain='ZZ')
```

```
>>> F, G = f.unify(g)
```

```
>>> F
Poly(1/2*x + 1, x, domain='QQ')
>>> G
Poly(2*x + 1, x, domain='QQ')
```

property unit

Return unit polynomial with `self`'s properties.

property zero

Return zero polynomial with `self`'s properties.

class sympy.polys.polytools.**PurePoly**(*rep, *gens, **args*)

Class for representing pure polynomials.

property free_symbols

Free symbols of a polynomial.

Examples

```
>>> from sympy import PurePoly
>>> from sympy.abc import x, y
```

```
>>> PurePoly(x**2 + 1).free_symbols
set()
>>> PurePoly(x**2 + y).free_symbols
set()
>>> PurePoly(x**2 + y, x).free_symbols
{y}
```

class sympy.polys.polytools.**GroebnerBasis**(*F, *gens, **args*)

Represents a reduced Groebner basis.

contains(*poly*)

Check if `poly` belongs the ideal generated by `self`.

Examples

```
>>> from sympy import groebner
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([x**2 + y**2 - 1, x*y - 2])
```

```
>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

fglm(*order*)

Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import groebner
```

```
>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')
```

```
>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
```

References

[R719]

property is_zero_dimensional

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of `F` is bounded.

References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

reduce(*expr*, *auto=True*)

Reduces a polynomial modulo a Groebner basis.

Given a polynomial `f` and a set of polynomials `G = (g_1, ..., g_n)`, computes a set of quotients `q = (q_1, ..., q_n)` and the remainder `r` such that `f = q_1*f_1 + ... + q_n*f_n + r`, where `r` vanishes or `r` is a completely reduced polynomial with respect to `G`.

Examples

```
>>> from sympy import groebner, expand
>>> from sympy.abc import x, y
```

```
>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])
```

```
>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = _
```

```
>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

Extra polynomial manipulation functions

`sympy.polys.polyfuncs.`**`symmetrize`**(*F*, **gens*, ***args*)

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if $f = f(x_1, x_2, \dots, x_n)$, then $f = f(x_{i_1}, x_{i_2}, \dots, x_{i_n})$, where $(i_1, i_2, \dots, i_n)$ is a permutation of $(1, 2, \dots, n)$ (an element of the group S_n).

Returns a tuple of symmetric polynomials `(f1, f2, ..., fn)` such that `f = f1 + f2 + ... + fn`.

Examples

```
>>> from sympy.polys.polyfuncs import symmetrize
>>> from sympy.abc import x, y
```

```
>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)
```

```
>>> symmetrize(x**2 + y**2, formal=True)
(s1**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])
```

```
>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)
```

```
>>> symmetrize(x**2 - y**2, formal=True)
(s1**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

`sympy.polys.polyfuncs.`**`horner`**(*f*, **gens*, ***args*)

Rewrite a polynomial in Horner form.

Among other applications, evaluation of a polynomial at a point is optimal when it is applied using the Horner scheme ([1]).

Examples

```
>>> from sympy.polys.polyfuncs import horner
>>> from sympy.abc import x, y, a, b, c, d, e
```

```
>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5
```

```
>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))
```

```
>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y
```

```
>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))
```

```
>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```

References

[1] - https://en.wikipedia.org/wiki/Horner_scheme

sympy.polys.polyfuncs.**interpolate**(*data*, *x*)

Construct an interpolating polynomial for the data points evaluated at point x (which can be symbolic or numeric).

Examples

```
>>> from sympy.polys.polyfuncs import interpolate
>>> from sympy.abc import a, b, x
```

A list is interpreted as though it were paired with a range starting from 1:

```
>>> interpolate([1, 4, 9, 16], x)
x**2
```

This can be made explicit by giving a list of coordinates:

```
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
x**2
```

The (x, y) coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```
>>> interpolate([(-1, 2), (1, 2), (2, 5)], x)
x**2 + 1
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
x**2 + 1
```

If the interpolation is going to be used only once then the value of interest can be passed instead of passing a symbol:

```
>>> interpolate([1, 4, 9], 5)
25
```

Symbolic coordinates are also supported:

```
>>> [(i,interpolate((a, b), i)) for i in range(1, 4)]
[(1, a), (2, b), (3, -a + 2*b)]
```

sympy.polys.polyfuncs.**viete**(*f, roots=None, *gens, **args*)

Generate Viete's formulas for f.

Examples

```
>>> from sympy.polys.polyfuncs import viete
>>> from sympy import symbols
```

```
>>> x, a, b, c, r1, r2 = symbols('x,a:c,r1:3')
```

```
>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

Domain constructors

sympy.polys.constructor.**construct_domain**(*obj, **args*)

Construct a minimal domain for a list of expressions.

Parameters

obj: list or dict

The expressions to build a domain for.

****args: keyword arguments**

Options that affect the choice of domain.

Returns

(K, elements): Domain and list of domain elements

The domain K that can represent the expressions and the list or dict of domain elements representing the same expressions as elements of K.

Explanation

Given a list of normal SymPy expressions (of type *Expr* (page 947)) construct_domain will find a minimal *Domain* (page 2504) that can represent those expressions. The expressions will be converted to elements of the domain and both the domain and the domain elements are returned.

Examples

Given a list of *Integer* (page 987) construct_domain will return the domain *ZZ* (page 2525) and a list of integers as elements of *ZZ* (page 2525).

```
>>> from sympy import construct_domain, S
>>> expressions = [S(2), S(3), S(4)]
>>> K, elements = construct_domain(expressions)
>>> K
ZZ
>>> elements
[2, 3, 4]
>>> type(elements[0])
<class 'int'>
>>> type(expressions[0])
<class 'sympy.core.numbers.Integer'>
```

If there are any *Rational* (page 985) then *QQ* (page 2529) is returned instead.

```
>>> construct_domain([S(1)/2, S(3)/4])
(QQ, [1/2, 3/4])
```

If there are symbols then a polynomial ring *K[x]* (page 2547) is returned.

```
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> construct_domain([2*x + 1, S(3)/4])
(QQ[x], [2*x + 1, 3/4])
>>> construct_domain([2*x + 1, y])
(ZZ[x,y], [2*x + 1, y])
```

If any symbols appear with negative powers then a rational function field *K(x)* (page 2548) will be returned.

```
>>> construct_domain([y/x, x/(1 - y)])
(ZZ(x,y), [y/x, -x/(y - 1)])
```

Irrational algebraic numbers will result in the *EX* (page 2549) domain by default. The keyword argument extension=True leads to the construction of an algebraic number field *QQ<a>* (page 2539).

```
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])
(EX, [EX(sqrt(2))])
>>> construct_domain([sqrt(2)], extension=True)
(QQ<sqrt(2)>, [ANP([1, 0], [1, 0, -2], QQ)])
```

See also:

Domain (page 2504), *Expr* (page 947)

Monomials encoded as tuples

class `sympy.polys.monomials.`**`Monomial`**(*monom*, *gens=None*)

Class representing a monomial, i.e. a product of powers.

`as_expr`(**gens*)

Convert a monomial instance to a SymPy expression.

`gcd`(*other*)

Greatest common divisor of monomials.

`lcm`(*other*)

Least common multiple of monomials.

`sympy.polys.monomials.`**`itermonomials`**(*variables*, *max_degrees*, *min_degrees=None*)

`max_degrees` and `min_degrees` are either both integers or both lists. Unless otherwise specified, `min_degrees` is either `0` or `[0, ..., 0]`.

A generator of all monomials `monom` is returned, such that either `min_degree <= total_degree(monom) <= max_degree`, or `min_degrees[i] <= degree_list(monom)[i] <= max_degrees[i]`, for all `i`.

Case I. `max_degrees` And `min_degrees` Are Both Integers

Given a set of variables V and a min_degree N and a max_degree M generate a set of monomials of degree less than or equal to N and greater than or equal to M. The total number of monomials in commutative variables is huge and is given by the following formula if $M = 0$:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree $N = 50$ and $M = 0$, which is the worst case, in 5 variables, assuming that exponents and all of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of memory! Fortunately most polynomials, that we will encounter, are sparse.

Consider monomials in commutative variables x and y and non-commutative variables a and b:

```
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
```

(continues on next page)

(continued from previous page)

```
>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]

>>> a, b = symbols('a, b', commutative=False)
>>> set(itermonomials([a, b, x], 2))
{1, a, a**2, b, b**2, x, x**2, a*b, b*a, x*a, x*b}

>>> sorted(itermonomials([x, y], 2, 1), key=monomial_key('grlex', [y,␣
↪x]))
[x, y, x**2, x*y, y**2]
```

Case Ii. max_degrees And min_degrees Are Both Lists

If max_degrees = [d_1, ..., d_n] and min_degrees = [e_1, ..., e_n], the number of monomials generated is:

$$(d_1 - e_1 + 1)(d_2 - e_2 + 1) \cdots (d_n - e_n + 1)$$

Let us generate all monomials monom in variables x and y such that [1, 2][i] <= degree_list(monom)[i] <= [2, 4][i], i = 0, 1

```
>>> from sympy import symbols
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], [2, 4], [1, 2]), reverse=True,␣
↪key=monomial_key('lex', [x, y]))
[x**2*y**4, x**2*y**3, x**2*y**2, x*y**4, x*y**3, x*y**2]
```

sympy.polys.monomials.**monomial_count**(V, N)

Computes the number of monomials.

The number of monomials is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

where N is a total degree and V is a set of variables.

Examples

```
>>> from sympy.polys.monomials import itermonomials, monomial_count
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y
```

```
>>> monomial_count(2, 2)
6
```

```
>>> M = list(itermonomials([x, y], 2))
```

```
>>> sorted(M, key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
>>> len(M)
6
```

Orderings of monomials

class `sympy.polys.orderings.`**`MonomialOrder`**

Base class for monomial orderings.

class `sympy.polys.orderings.`**`LexOrder`**

Lexicographic order of monomials.

class `sympy.polys.orderings.`**`GradedLexOrder`**

Graded lexicographic order of monomials.

class `sympy.polys.orderings.`**`ReversedGradedLexOrder`**

Reversed graded lexicographic order of monomials.

Formal manipulation of roots of polynomials

`sympy.polys.rootoftools.`**`rootof`**(*f, x, index=None, radicals=True, expand=True*)

An indexed root of a univariate polynomial.

Returns either a *ComplexRootOf* (page 2431) object or an explicit expression involving radicals.

Parameters

f : Expr

Univariate polynomial.

x : Symbol, optional

Generator for `f`.

index : int or Integer

radicals : bool

Return a radical expression if possible.

expand : bool

Expand `f`.

class `sympy.polys.rootoftools.`**`RootOf`**(*f, x, index=None, radicals=True, expand=True*)

Represents a root of a univariate polynomial.

Base class for roots of different kinds of polynomials. Only complex roots are currently supported.

class sympy.polys.rootoftools.**ComplexRootOf**(*f, x, index=None, radicals=False, expand=True*)

Represents an indexed complex root of a polynomial.

Roots of a univariate polynomial separated into disjoint real or complex intervals and indexed in a fixed order:

- real roots come first and are sorted in increasing order;
- complex roots come next and are sorted primarily by increasing real part, secondarily by increasing imaginary part.

Currently only rational coefficients are allowed. Can be imported as `CRootOf`. To avoid confusion, the generator must be a Symbol.

Examples

```
>>> from sympy import CRootOf, rootof
>>> from sympy.abc import x
```

CRootOf is a way to reference a particular root of a polynomial. If there is a rational root, it will be returned:

```
>>> CRootOf.clear_cache()  # for doctest reproducibility
>>> CRootOf(x**2 - 4, 0)
-2
```

Whether roots involving radicals are returned or not depends on whether the `radicals` flag is true (which is set to True with rootof):

```
>>> CRootOf(x**2 - 3, 0)
CRootOf(x**2 - 3, 0)
>>> CRootOf(x**2 - 3, 0, radicals=True)
-sqrt(3)
>>> rootof(x**2 - 3, 0)
-sqrt(3)
```

The following cannot be expressed in terms of radicals:

```
>>> r = rootof(4*x**5 + 16*x**3 + 12*x**2 + 7, 0); r
CRootOf(4*x**5 + 16*x**3 + 12*x**2 + 7, 0)
```

The root bounds can be seen, however, and they are used by the evaluation methods to get numerical approximations for the root.

```
>>> interval = r._get_interval(); interval
(-1, 0)
>>> r.evalf(2)
-0.98
```

The evalf method refines the width of the root bounds until it guarantees that any decimal approximation within those bounds will satisfy the desired precision. It then stores the refined interval so subsequent requests at or below the requested precision will not have to recompute the root bounds and will return very quickly.

Before evaluation above, the interval was

```
>>> interval
(-1, 0)
```

After evaluation it is now

```
>>> r._get_interval()
(-165/169, -206/211)
```

To reset all intervals for a given polynomial, the *_reset()* (page 2435) method can be called from any CRootOf instance of the polynomial:

```
>>> r._reset()
>>> r._get_interval()
(-1, 0)
```

The *eval_approx()* (page 2435) method will also find the root to a given precision but the interval is not modified unless the search for the root fails to converge within the root bounds. And the secant method is used to find the root. (The `evalf` method uses bisection and will always update the interval.)

```
>>> r.eval_approx(2)
-0.98
```

The interval needed to be slightly updated to find that root:

```
>>> r._get_interval()
(-1, -1/2)
```

The `evalf_rational` will compute a rational approximation of the root to the desired accuracy or precision.

```
>>> r.eval_rational(n=2)
-69629/71318
```

```
>>> t = CRootOf(x**3 + 10*x + 1, 1)
>>> t.eval_rational(1e-1)
15/256 - 805*I/256
>>> t.eval_rational(1e-1, 1e-4)
3275/65536 - 414645*I/131072
>>> t.eval_rational(1e-4, 1e-4)
6545/131072 - 414645*I/131072
>>> t.eval_rational(n=2)
104755/2097152 - 6634255*I/2097152
```

Notes

Although a PurePoly can be constructed from a non-symbol generator RootOf instances of non-symbols are disallowed to avoid confusion over what root is being represented.

```
>>> from sympy import exp, PurePoly
>>> PurePoly(x) == PurePoly(exp(x))
True
>>> CRootOf(x - 1, 0)
1
>>> CRootOf(exp(x) - 1, 0)  # would correspond to x == 0
Traceback (most recent call last):
...
sympy.polys.polyerrors.PolynomialError: generator must be a Symbol
```

See also:

eval_approx (page 2435), *eval_rational* (page 2436)

classmethod _all_roots(*poly*, *use_cache=True*)
Get real and complex roots of a composite polynomial.

classmethod _complexes_index(*complexes*, *index*)
Map initial complex root index to an index in a factor where the root belongs.

classmethod _complexes_sorted(*complexes*)
Make complex isolating intervals disjoint and sort roots.

classmethod _count_roots(*roots*)
Count the number of real or complex roots with multiplicities.

_ensure_complexes_init()
Ensure that our poly has entries in the complexes cache.

_ensure_reals_init()
Ensure that our poly has entries in the reals cache.

_eval_evalf(*prec*, ***kwargs*)
Evaluate this complex root to the given precision.

_eval_is_imaginary()
Return True if the root is imaginary.

_eval_is_real()
Return True if the root is real.

classmethod _get_complexes(*factors*, *use_cache=True*)
Compute complex root isolating intervals for a list of factors.

classmethod _get_complexes_sqf(*currentfactor*, *use_cache=True*)
Get complex root isolating intervals for a square-free factor.

_get_interval()
Internal function for retrieving isolation interval from cache.

classmethod _get_reals(*factors*, *use_cache=True*)
Compute real root isolating intervals for a list of factors.

classmethod _get_reals_sqf(*currentfactor*, *use_cache=True*)
: Get real root isolating intervals for a square-free factor.

classmethod _get_roots(*method*, *poly*, *radicals*)
: Return postprocessed roots of specified kind.

classmethod _indexed_root(*poly*, *index*, *lazy=False*)
: Get a root of a composite polynomial by index.

classmethod _new(*poly*, *index*)
: Construct new `CRootOf` object from raw data.

classmethod _postprocess_root(*root*, *radicals*)
: Return the root if it is trivial or a `CRootOf` object.

classmethod _preprocess_roots(*poly*)
: Take heroic measures to make `poly` compatible with `CRootOf`.

classmethod _real_roots(*poly*)
: Get real roots of a composite polynomial.

classmethod _reals_index(*reals*, *index*)
: Map initial real root index to an index in a factor where the root belongs.

classmethod _reals_sorted(*reals*)
: Make real isolating intervals disjoint and sort roots.

classmethod _refine_complexes(*complexes*)
: return complexes such that no bounding rectangles of non-conjugate roots would intersect. In addition, assure that neither ay nor by is 0 to guarantee that non-real roots are distinct from real roots in terms of the y-bounds.

_reset()
: Reset all intervals

classmethod _roots_trivial(*poly*, *radicals*)
: Compute roots in linear, quadratic and binomial cases.

_set_interval(*interval*)
: Internal function for updating isolation interval in cache.

classmethod all_roots(*poly*, *radicals=True*)
: Get real and complex roots of a polynomial.

classmethod clear_cache()
: Reset cache for reals and complexes.

The intervals used to approximate a root instance are updated as needed. When a request is made to see the intervals, the most current values are shown. $clear_cache$ will reset all CRootOf instances back to their original state.

See also:

_reset (page 2435)

eval_approx(*n*)
: Evaluate this complex root to the given precision.

This uses secant method and root bounds are used to both generate an initial guess and to check that the root returned is valid. If ever the method converges outside the root bounds, the bounds will be made smaller and updated.

eval_rational(*dx=None*, *dy=None*, *n=15*)

Return a Rational approximation of `self` that has real and imaginary component approximations that are within `dx` and `dy` of the true values, respectively. Alternatively, `n` digits of precision can be specified.

The interval is refined with bisection and is sure to converge. The root bounds are updated when the refinement is complete so recalculation at the same or lesser precision will not have to repeat the refinement and should be much faster.

The following example first obtains Rational approximation to 1e-8 accuracy for all roots of the 4-th order Legendre polynomial. Since the roots are all less than 1, this will ensure the decimal representation of the approximation will be correct (including rounding) to 6 digits:

```
>>> from sympy import legendre_poly, Symbol
>>> x = Symbol("x")
>>> p = legendre_poly(4, x, polys=True)
>>> r = p.real_roots()[-1]
>>> r.eval_rational(10**-8).n(6)
0.861136
```

It is not necessary to a two-step calculation, however: the decimal representation can be computed directly:

```
>>> r.evalf(17)
0.86113631159405258
```

classmethod real_roots(*poly*, *radicals=True*)

Get real roots of a polynomial.

class sympy.polys.rootoftools.**RootSum**(*expr*, *func=None*, *x=None*, *auto=True*, *quadratic=False*)

Represents a sum of all roots of a univariate polynomial.

classmethod new(*poly*, *func*, *auto=True*)

Construct new `RootSum` instance.

Symbolic root-finding algorithms

sympy.polys.polyroots.**roots**(*f*, **gens*, *auto=True*, *cubics=True*, *trig=False*, *quartics=True*, *quintics=False*, *multiple=False*, *filter=None*, *predicate=None*, *strict=False*, ***flags*)

Computes symbolic roots of a univariate polynomial.

Given a univariate polynomial f with symbolic coefficients (or a list of the polynomial's coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use RootOf class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set `cubics=False` or `quartics=False` respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis [1]) the `trig` flag can be set to True to have the solutions returned in terms of cosine and inverse cosine functions.