Univariate factoring over various domains

Consider a univariate polynomial `f` with integer coefficients:

```
>>> f = x**4 - 3*x**2 + 1
```

With *sympy.polys* (page 2360) we can obtain factorizations of `f` over different domains, which includes:

- rationals:

```
>>> factor(f)
⎛ 2         ⎞ ⎛ 2         ⎞
⎝x  - x - 1⎠⋅⎝x  + x - 1⎠
```

- finite fields:

```
>>> factor(f, modulus=5)
       2        2
(x - 2) ⋅(x + 2)
```

- algebraic numbers:

```
>>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')

>>> factor(f, extension=alg)
(x - α)⋅(x + α)⋅(x - 1 - α)⋅(x + α + 1)
```

Factoring polynomials into linear factors

Currently SymPy can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the later only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using *solve()* (page 836) function.

Advanced factoring over finite fields

Consider a univariate polynomial `f` with integer coefficients:

```
>>> f = x**11 + x + 1
```

We can factor `f` over a large finite field F_{65537}:

```
>>> factor(f, modulus=65537)
⎛ 2         ⎞ ⎛ 9    8    6    5    3    2     ⎞
⎝x  + x + 1⎠⋅⎝x  - x  + x  - x  + x  - x  + 1⎠
```

and expand the resulting factorization back:

```
>>> expand(_)
 11
x   + x + 1
```

obtaining polynomial `f`. This was done using symmetric polynomial representation over finite fields The same thing can be done using non-symmetric representation:

```
>>> factor(f, modulus=65537, symmetric=False)
⎛ 2         ⎞ ⎛ 9           8    6           5    3           2     ⎞
⎝x  + x + 1⎠⋅⎝x  + 65536⋅x  + x  + 65536⋅x  + x  + 65536⋅x  + 1⎠
```

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

```
>>> trunc(expand(_), 65537)
 11
x   + x + 1
```

Working with expressions as polynomials

Consider a multivariate polynomial `f` in $\mathbb{Z}[x, y, z]$:

```
>>> f = expand((x - 2*y**2 + 3*z**3)**20)
```

We want to compute factorization of `f`. To do this we use `factor` as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding `f`:

```
>>> factor(f, expand=False)
                  20
⎛       2      3⎞
⎝x - 2⋅y  + 3⋅z ⎠
```

The default in *sympy.polys* (page 2360) is to expand all expressions given as arguments to polynomial manipulation functions and *Poly* (page 2378) class. If we know that expanding is unnecessary, then by setting `expand=False` we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:

```
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)

>>> factor(g, expand=False)
                                    20
⎛                2           3    ⎞
⎝-sin(x) + 2·cos (y) - 3·tan (z)⎠
```

Computing reduced Gröbner bases

To compute a reduced Gröbner basis for a set of polynomials use the *groebner()* (page 2377) function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The `lex` ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to *groebner()* (page 2377) is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)

>>> groebner([f, c**2 + s**2 - 1])
            ⎛⎡ 2    2       20      18       16       14      12    10⎤      ␣
↪                    ⎞
GroebnerBasis⎝⎣c  + s  - 1, c   - 5·c   + 10·c   - 10·c   + 5·c   - c  ⎦, s,␣
↪c, domain=ℤ, order=lex⎠
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```
>>> list(map(factor, _))
⎡ 2    2       10        5        5⎤
⎣c  + s  - 1, c  ·(c - 1) ·(c + 1) ⎦
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use *solve()* (page 836) to achieve this in a more systematic way:

```
>>> solve([f, s**2 + c**2 - 1], c, s)
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

Multivariate factoring over algebraic numbers

Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over $\mathbb{Q}(\sqrt{-3})$:

```
>>> factor(x**3 + y**3, extension=sqrt(-3))
       ⎛      ⎛  1   √3·i⎞⎞ ⎛      ⎛  1   √3·i⎞⎞
(x + y)·⎜x + y·⎜- ─ - ────⎟⎟·⎜x + y·⎜- ─ + ────⎟⎟
       ⎝      ⎝  2    2 ⎠⎠ ⎝      ⎝  2    2 ⎠⎠
```

Note: Currently multivariate polynomials over finite fields aren't supported.

Partial fraction decomposition

Consider a univariate rational function `f` with integer coefficients:

```
>>> f = (x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2)
```

To decompose `f` into partial fractions use *apart()* (page 2443) function:

```
>>> apart(f)
  3       2         2
----- - ----- + ---------
x + 2   x + 1           2
                (x + 1)
```

To return from partial fractions to the rational function use a composition of *together()* (page 2442) and *cancel()* (page 2376):

```
>>> cancel(together(_))
     2
    x  + 2·x + 3
-------------------
 3      2
x  + 4·x  + 5·x + 2
```

Literature

Polynomials Manipulation Module Reference

Polynomial manipulation algorithms and algebraic objects.

See *Polynomial Manipulation* (page 2341) for an index of documentation for the polys module and *Basic functionality of the module* (page 2342) for an introductory explanation.

Basic polynomial manipulation functions

sympy.polys.polytools.**poly**(*expr*, **gens*, ***args*)

Efficiently transform an expression into a polynomial.

Examples

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

```
>>> poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

sympy.polys.polytools.**poly_from_expr**(*expr*, **gens*, ***args*)

Construct a polynomial from an expression.

sympy.polys.polytools.**parallel_poly_from_expr**(*exprs*, **gens*, ***args*)

Construct polynomials from expressions.

sympy.polys.polytools.**degree**(*f*, *gen=0*)

Return the degree of `f` in the given variable.

The degree of 0 is negative infinity.

Examples

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

```
>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-oo
```

See also:

sympy.polys.polytools.Poly.total_degree (page 2421), *degree_list* (page 2361)

sympy.polys.polytools.**degree_list**(*f*, **gens*, ***args*)

Return a list of degrees of `f` in all variables.

Examples

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

```
>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

sympy.polys.polytools.**LC**(*f*, **gens*, ***args*)

Return the leading coefficient of `f`.

Examples

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

```
>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

sympy.polys.polytools.**LM**(*f*, **gens*, ***args*)

Return the leading monomial of `f`.

Examples

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

```
>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

sympy.polys.polytools.**LT**(*f*, **gens*, ***args*)

Return the leading term of `f`.

Examples

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

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

sympy.polys.polytools.**pdiv**(*f*, *g*, **gens*, ***args*)

Compute polynomial pseudo-division of `f` and `g`.

Examples

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

```
>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

sympy.polys.polytools.**prem**(*f*, *g*, **gens*, ***args*)

Compute polynomial pseudo-remainder of `f` and `g`.

Examples

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

```
>>> prem(x**2 + 1, 2*x - 4)
20
```

sympy.polys.polytools.**pquo**(*f*, *g*, **gens*, ***args*)

Compute polynomial pseudo-quotient of `f` and `g`.

Examples

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

```
>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

sympy.polys.polytools.**pexquo**(*f*, *g*, **gens*, ***args*)

Compute polynomial exact pseudo-quotient of `f` and `g`.

Examples

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

```
>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2
```

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

sympy.polys.polytools.**div**(*f*, *g*, **gens*, ***args*)

Compute polynomial division of `f` and `g`.

Examples

```
>>> from sympy import div, ZZ, QQ
>>> from sympy.abc import x
```

```
>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)
```

sympy.polys.polytools.**rem**(*f*, *g*, **gens*, ***args*)

Compute polynomial remainder of `f` and `g`.

Examples

```
>>> from sympy import rem, ZZ, QQ
>>> from sympy.abc import x
```

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

sympy.polys.polytools.**quo**(*f*, *g*, **gens*, ***args*)

Compute polynomial quotient of `f` and `g`.

Examples

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

```
>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

sympy.polys.polytools.**exquo**(*f*, *g*, **gens*, ***args*)

Compute polynomial exact quotient of `f` and `g`.

Examples

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

```
>>> exquo(x**2 - 1, x - 1)
x + 1
```

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

sympy.polys.polytools.**half_gcdex**(*f*, *g*, **gens*, ***args*)

Half extended Euclidean algorithm of `f` and `g`.

Returns `(s, h)` such that `h = gcd(f, g)` and `s*f = h (mod g)`.

Examples

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

```
>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x + 1)
```

sympy.polys.polytools.**gcdex**(*f*, *g*, **gens*, ***args*)

Extended Euclidean algorithm of `f` and `g`.

Returns `(s, t, h)` such that `h = gcd(f, g)` and `s*f + t*g = h`.

Examples

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

```
>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(3/5 - x/5, x**2/5 - 6*x/5 + 2, x + 1)
```

sympy.polys.polytools.**invert**(*f*, *g*, **gens*, ***args*)

Invert `f` modulo `g` when possible.

Examples

```
>>> from sympy import invert, S, mod_inverse
>>> from sympy.abc import x
```

```
>>> invert(x**2 - 1, 2*x - 1)
-4/3
```

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

For more efficient inversion of Rationals, use the *mod_inverse* (page 1004) function:

```
>>> mod_inverse(3, 5)
2
>>> (S(2)/5).invert(S(7)/3)
5/2
```

See also:

sympy.core.numbers.mod_inverse (page 1004)

sympy.polys.polytools.**subresultants**(*f*, *g*, **gens*, ***args*)

Compute subresultant PRS of `f` and `g`.

Examples

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

```
>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

sympy.polys.polytools.**resultant**(*f*, *g*, **gens*, *includePRS=False*, ***args*)

Compute resultant of `f` and `g`.

Examples

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

```
>>> resultant(x**2 + 1, x**2 - 1)
4
```

sympy.polys.polytools.**discriminant**(*f*, **gens*, ***args*)

Compute discriminant of `f`.

Examples

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

```
>>> discriminant(x**2 + 2*x + 3)
-8
```

sympy.polys.polytools.**terms_gcd**(*f*, **gens*, ***args*)

Remove GCD of terms from `f`.

If the `deep` flag is True, then the arguments of `f` will have terms_gcd applied to them.

If a fraction is factored out of `f` and `f` is an Add, then an unevaluated Mul will be returned so that automatic simplification does not redistribute it. The hint `clear`, when set to False, can be used to prevent such factoring when all coefficients are not fractions.

Examples

```
>>> from sympy import terms_gcd, cos
>>> from sympy.abc import x, y
>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of polys routines is to expand the expression given to them. terms_gcd follows this behavior:

```
>>> terms_gcd((3+3*x)*(x+x*y))
3*x*(x*y + x + y + 1)
```

If this is not desired then the hint `expand` can be set to False. In this case the expression will be treated as though it were comprised of one or more terms:

```
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a Mul or the arguments of other functions, the `deep` hint can be used:

```
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

```
>>> terms_gcd(x + y/2)
(2*x + y)/2
```

Only the y-term had a coefficient that was a fraction; if one does not want to factor out the 1/2 in cases like this, the flag `clear` can be set to False:

```
>>> terms_gcd(x + y/2, clear=False)
x + y/2
>>> terms_gcd(x*y/2 + y**2, clear=False)
y*(x/2 + y)
```

The `clear` flag is ignored if all coefficients are fractions:

```
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

See also:

sympy.core.exprtools.gcd_terms (page 1071), *sympy.core.exprtools.factor_terms* (page 1072)

sympy.polys.polytools.**cofactors**(*f*, *g*, **gens*, ***args*)

Compute GCD and cofactors of `f` and `g`.

Returns polynomials `(h, cff, cfg)` such that `h = gcd(f, g)`, and `cff = quo(f, h)` and `cfg = quo(g, h)` are, so called, cofactors of `f` and `g`.

Examples

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

```
>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

sympy.polys.polytools.**gcd**(*f*, *g=None*, **gens*, ***args*)

Compute GCD of `f` and `g`.

Examples

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

```
>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

sympy.polys.polytools.**gcd_list**(*seq*, **gens*, ***args*)

Compute GCD of a list of polynomials.

Examples

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

```
>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

sympy.polys.polytools.**lcm**(*f*, *g=None*, **gens*, ***args*)

Compute LCM of `f` and `g`.

Examples

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

```
>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

sympy.polys.polytools.**lcm_list**(*seq*, **gens*, ***args*)

Compute LCM of a list of polynomials.

Examples

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

```
>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2
```

sympy.polys.polytools.**trunc**(*f*, *p*, **gens*, ***args*)

Reduce `f` modulo a constant `p`.

Examples

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

```
>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

sympy.polys.polytools.**monic**(*f*, **gens*, ***args*)

Divide all coefficients of `f` by `LC(f)`.

Examples

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

```
>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

sympy.polys.polytools.**content**(*f*, **gens*, ***args*)

Compute GCD of coefficients of `f`.

Examples

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

```
>>> content(6*x**2 + 8*x + 12)
2
```

sympy.polys.polytools.**primitive**(*f*, **gens*, ***args*)

Compute content and the primitive form of `f`.

Examples

```
>>> from sympy.polys.polytools import primitive
>>> from sympy.abc import x
```

```
>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)
```

```
>>> eq = (2 + 2*x)*x + 2
```

Expansion is performed by default:

```
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set expand to False to shut this off. Note that the extraction will not be recursive; use the as_content_primitive method for recursive, non-destructive Rational extraction.

```
>>> primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)
```

```
>>> eq.as_content_primitive()
(2, x*(x + 1) + 1)
```

sympy.polys.polytools.**compose**(*f*, *g*, **gens*, ***args*)

Compute functional composition `f(g)`.

Examples

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

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

sympy.polys.polytools.**decompose**(*f*, **gens*, ***args*)

Compute functional decomposition of `f`.

Examples

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

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

sympy.polys.polytools.**sturm**(*f*, **gens*, ***args*)

Compute Sturm sequence of `f`.

Examples

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

```
>>> sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]
```

sympy.polys.polytools.**gff_list**(*f*, **gens*, ***args*)

Compute a list of greatest factorial factors of `f`.

Note that the input to ff() and rf() should be Poly instances to use the definitions here.

Examples

```
>>> from sympy import gff_list, ff, Poly
>>> from sympy.abc import x
```

```
>>> f = Poly(x**5 + 2*x**4 - x**3 - 2*x**2, x)
```

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

```
>>> (ff(Poly(x), 1)*ff(Poly(x + 2), 4)) == f
True
```

```
>>> f = Poly(x**12 + 6*x**11 - 11*x**10 - 56*x**9 + 220*x**8 + 208*x**7 -
↪          1401*x**6 + 1090*x**5 + 2715*x**4 - 6720*x**3 - 1092*x**2 +␣
↪5040*x, x)
```

```
>>> gff_list(f)
[(Poly(x**3 + 7, x, domain='ZZ'), 2), (Poly(x**2 + 5*x, x, domain='ZZ'),␣
↪3)]
```

```
>>> ff(Poly(x**3 + 7, x), 2)*ff(Poly(x**2 + 5*x, x), 3) == f
True
```

sympy.polys.polytools.**gff**(*f*, **gens*, ***args*)

Compute greatest factorial factorization of `f`.

sympy.polys.polytools.**sqf_norm**(*f*, **gens*, ***args*)

Compute 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 sqf_norm, sqrt
>>> from sympy.abc import x
```

```
>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

sympy.polys.polytools.**sqf_part**(*f*, **gens*, ***args*)

Compute square-free part of `f`.

Examples

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

```
>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

sympy.polys.polytools.**sqf_list**(*f*, **gens*, ***args*)

Compute a list of square-free factors of `f`.

Examples

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

```
>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

sympy.polys.polytools.**sqf**(*f*, **gens*, ***args*)

Compute square-free factorization of `f`.

Examples

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

```
>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

sympy.polys.polytools.**factor_list**(*f*, **gens*, ***args*)

Compute a list of irreducible factors of `f`.

Examples

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

```
>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

sympy.polys.polytools.**factor**(*f*, **gens*, *deep=False*, ***args*)

Compute the factorization of expression, `f`, into irreducibles. (To factor an integer into primes, use `factorint`.)

There two modes implemented: symbolic and formal. If `f` is not an instance of *Poly* (page 2378) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, *factor()* (page 2373) will traverse the expression tree and factor its components without any prior expansion, unless an instance of *Add* (page 1013) is encountered (in this case formal factorization is used). This way *factor()* (page 2373) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: `extension`, `modulus` or `domain`.

Examples

```
>>> from sympy import factor, sqrt, exp
>>> from sympy.abc import x, y
```

```
>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2
```

```
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)
```

```
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
```

```
>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**10000000*(x**2 + 1))
(x + 2)**20000000*(x**2 + 1)
```

By default, factor deals with an expression as a whole:

```
>>> eq = 2**(x**2 + 2*x + 1)
>>> factor(eq)
2**(x**2 + 2*x + 1)
```

If the deep flag is True then subexpressions will be factored:

```
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

If the fraction flag is False then rational expressions will not be combined. By default it is True.

```
>>> factor(5*x + 3*exp(2 - 7*x), deep=True)
(5*x*exp(7*x) + 3*exp(2))*exp(-7*x)
>>> factor(5*x + 3*exp(2 - 7*x), deep=True, fraction=False)
5*x + 3*exp(2)*exp(-7*x)
```

See also:

sympy.ntheory.factor_.factorint (page 1493)

sympy.polys.polytools.**intervals**(*F*, *all=False*, *eps=None*, *inf=None*, *sup=None*, *strict=False*, *fast=False*, *sqf=False*)

Compute isolating intervals for roots of f.

Examples

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

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

sympy.polys.polytools.**refine_root**(*f*, *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 refine_root
>>> from sympy.abc import x
```

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

sympy.polys.polytools.**count_roots**(*f*, *inf=None*, *sup=None*)

Return the number of roots of f in [inf, sup] interval.

If one of inf or sup is complex, it will return the number of roots in the complex rectangle with corners at inf and sup.

Examples

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

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

sympy.polys.polytools.**real_roots**(*f*, *multiple=True*)

Return a list of real roots with multiplicities of `f`.

Examples

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

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

sympy.polys.polytools.**nroots**(*f*, *n=15*, *maxsteps=50*, *cleanup=True*)

Compute numerical approximations of roots of `f`.

Examples

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

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

sympy.polys.polytools.**ground_roots**(*f*, **gens*, ***args*)

Compute roots of `f` by factorization in the ground domain.

Examples

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

```
>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}
```

sympy.polys.polytools.**nth_power_roots_poly**(*f*, *n*, **gens*, ***args*)

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

Examples

```
>>> from sympy import nth_power_roots_poly, factor, roots
>>> from sympy.abc import x
```

```
>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))
```

```
>>> g
(x**2 - x + 1)**2
```

```
>>> R_f = [ (r**2).expand() for r in roots(f) ]
>>> R_g = roots(g).keys()
```

```
>>> set(R_f) == set(R_g)
True
```

sympy.polys.polytools.**cancel**(*f*, **gens*, *_signsimp=True*, ***args*)

Cancel common factors in a rational function `f`.

Examples

```
>>> from sympy import cancel, sqrt, Symbol, together
>>> from sympy.abc import x
>>> A = Symbol('A', commutative=False)
```

```
>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

Note: due to automatic distribution of Rationals, a sum divided by an integer will appear as a sum. To recover a rational form use *together* on the result:

```
>>> cancel(x/2 + 1)
x/2 + 1
>>> together(_)
(x + 2)/2
```

sympy.polys.polytools.**reduced**(*f*, *G*, **gens*, ***args*)

Reduces a polynomial `f` modulo a set of polynomials `G`.

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*g_1 + ... + q_n*g_n + r`, where `r` vanishes or `r` is a completely reduced polynomial with respect to `G`.

Examples

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

```
>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

sympy.polys.polytools.**groebner**(*F*, **gens*, ***args*)

Computes the reduced Groebner basis for a set of polynomials.

Use the `order` argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of *solve_poly_system()* (page 856).

Examples

Example taken from [1].

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

```
>>> F = [x*y - 2*y, 2*y**2 - x**2]
```

```
>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y, y**3 - 2*y], x, y,
              domain='ZZ', order='lex')
>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grlex')
>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using the `method` flag or with the *sympy.polys.polyconfig.setup()* (page 2643) function.

```
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]
```

```
>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

References

1. [Buchberger01]
2. [Cox97]

[Buchberger01], [Cox97]

sympy.polys.polytools.**is_zero_dimensional**(*F*, **gens*, ***args*)

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

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

Generic class for representing and operating on polynomial expressions.

See *Polynomial Manipulation* (page 2341) for general documentation.

Poly is a subclass of Basic rather than Expr but instances can be converted to Expr with the *as_expr()* (page 2383) method.

Deprecated since version 1.6: Combining Poly with non-Poly objects in binary operations is deprecated. Explicitly convert both objects to either Poly or Expr first. See *Mixing Poly and non-polynomial expressions in binary operations* (page 175).

Examples

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

Create a univariate polynomial:

```
>>> Poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

Create a univariate polynomial with specific domain:

```
>>> from sympy import sqrt
>>> Poly(x**2 + 2*x + sqrt(3), domain='R')
Poly(1.0*x**2 + 2.0*x + 1.73205080756888, x, domain='RR')
```

Create a multivariate polynomial:

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

Create a univariate polynomial, where y is a constant:

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

You can evaluate the above polynomial as a function of y:

```
>>> Poly(y*x**2 + x*y + 1,x).eval(2)
6*y + 1
```

See also:

sympy.core.expr.Expr (page 947)

EC(*order=None*)

Returns the last non-zero coefficient of `f`.

Examples

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

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

EM(*order=None*)

Returns the last non-zero monomial of `f`.

Examples

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

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

ET(*order=None*)

Returns the last non-zero term of `f`.

Examples

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

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

LC(*order=None*)

Returns the leading coefficient of `f`.

Examples

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

```
>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

LM(*order=None*)

Returns the leading monomial of `f`.

The Leading monomial signifies the monomial having the highest power of the principal generator in the expression f.

Examples

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

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

LT(*order=None*)

Returns the leading term of `f`.

The Leading term signifies the term having the highest power of the principal generator in the expression f along with its coefficient.

Examples

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

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

TC()

Returns the trailing coefficient of `f`.

Examples

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

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

abs()

Make all coefficients in `f` positive.

Examples

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

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

add(*g*)

Add two polynomials `f` and `g`.

Examples

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

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

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

add_ground(*coeff*)

Add an element of the ground domain to `f`.

Examples

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

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

all_coeffs()

Returns all coefficients from a univariate polynomial `f`.

Examples

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

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

all_monoms()

Returns all monomials from a univariate polynomial `f`.

Examples

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

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

See also:

all_terms (page 2382)

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

Return a list of real and complex roots with multiplicities.

Examples

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

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

all_terms()

Returns all terms from a univariate polynomial `f`.

Examples

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

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

as_dict(*native=False*, *zero=False*)

Switch to a `dict` representation.

Examples

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

```
>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

as_expr(**gens*)

Convert a Poly instance to an Expr instance.

Examples

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

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

```
>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379
```

as_list(*native=False*)

Switch to a `list` representation.

as_poly(**gens*, ***args*)

Converts `self` to a polynomial or returns `None`.

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

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

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

```
>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

cancel(*g*, *include=False*)

Cancel common factors in a rational function `f/g`.

Examples

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

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

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

clear_denoms(*convert=False*)

Clear denominators, but keep the ground domain.

Examples

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

```
>>> f = Poly(x/2 + S(1)/3, x, domain=QQ)
```

```
>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

coeff_monomial(*monom*)

Returns the coefficient of `monom` in `f` if there, else None.

Examples

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

```
>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)
```

```
>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*exp(8)
```

Note that `Expr.coeff()` behaves differently, collecting terms if possible; the Poly must be converted to an Expr to use that method, however:

```
>>> p.as_expr().coeff(x)
24*y*exp(8) + 23
>>> p.as_expr().coeff(y)
24*x*exp(8)
>>> p.as_expr().coeff(x*y)
24*exp(8)
```

See also:

nth **(page 2408)**
more efficient query using exponents of the monomial's generators

coeffs(*order=None*)

Returns all non-zero coefficients from `f` in lex order.

Examples

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

```
>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

See also:

all_coeffs (page 2381), *coeff_monomial* (page 2384), *nth* (page 2408)

cofactors(*g*)

Returns the GCD of `f` and `g` and their cofactors.

Returns polynomials `(h, cff, cfg)` such that `h = gcd(f, g)`, and `cff = quo(f, h)` and `cfg = quo(g, h)` are, so called, cofactors of `f` and `g`.

Examples

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

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

compose(*g*)

Computes the functional composition of `f` and `g`.

Examples

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

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

content()

Returns the GCD of polynomial coefficients.

Examples

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

```
>>> Poly(6*x**2 + 8*x + 12, x).content()
2
```

count_roots(*inf=None*, *sup=None*)

Return the number of roots of `f` in `[inf, sup]` interval.

Examples

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

```
>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1
```

decompose()

Computes a functional decomposition of `f`.

Examples

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

```
>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]
```

deflate()

Reduce degree of `f` by mapping `x_i**m` to `y_i`.

Examples

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

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

degree(*gen=0*)

Returns degree of `f` in `x_j`.

The degree of 0 is negative infinity.

Examples

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

```
>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
>>> Poly(0, x).degree()
-oo
```

degree_list()

Returns a list of degrees of `f`.

Examples

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

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

diff(**specs*, ***kwargs*)

Computes partial derivative of `f`.

Examples

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

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

```
>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')
```

discriminant()

Computes the discriminant of `f`.

Examples

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

```
>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

dispersion(*g=None*)

Compute the *dispersion* of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\operatorname{dis}(f, g)$ is defined as:

$$\begin{aligned}\operatorname{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} | \gcd(f(x), g(x + a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial $\operatorname{dis}(f) := \operatorname{dis}(f, f)$.

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>
↪')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2␣
↪+ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

dispersionset (page 2389)

References

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

dispersionset(*g=None*)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $\mathrm{J}(f, g)$ is defined as:

$$\begin{aligned}\mathrm{J}(f,g) &:= \{a \in \mathbb{N}_0 | \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 | \deg \gcd(f(x), g(x+a)) \geq 1\}\end{aligned}$$

For a single polynomial one defines $\mathrm{J}(f) := \mathrm{J}(f, f)$.

Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>
↪')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2␣
↪+ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

See also:

dispersion (page 2388)

References

1. [ManWright94]
2. [Koepf98]
3. [Abramov71]
4. [Man93]

[ManWright94], [Koepf98], [Abramov71], [Man93]

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

Polynomial division with remainder of `f` by `g`.

Examples

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

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

property domain

Get the ground domain of a *Poly* (page 2378)

Returns

Domain (page 2504):

Ground domain of the *Poly* (page 2378).

Examples

```
>>> from sympy import Poly, Symbol
>>> x = Symbol('x')
>>> p = Poly(x**2 + x)
>>> p
Poly(x**2 + x, x, domain='ZZ')
>>> p.domain
ZZ
```

eject(**gens*)

Eject selected generators into the ground domain.

Examples

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

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

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

eval(*x*, *a=None*, *auto=True*)

Evaluate `f` at `a` in the given variable.

Examples

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

```
>>> Poly(x**2 + 2*x + 3, x).eval(2)
11
```

```
>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')
```

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

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45
```

```
>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

exclude()

Remove unnecessary generators from `f`.

Examples

```
>>> from sympy import Poly
>>> from sympy.abc import a, b, c, d, x
```

```
>>> Poly(a + x, a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')
```

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

Computes polynomial exact quotient of `f` by `g`.

Examples

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

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

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

exquo_ground(*coeff*)

Exact 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).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

factor_list()

Returns a list of irreducible factors of `f`.

Examples

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

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

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

factor_list_include()

Returns a list of irreducible factors of `f`.

Examples

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

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

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

property free_symbols

Free symbols of a polynomial expression.

Examples

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

```
>>> Poly(x**2 + 1).free_symbols
{x}
>>> Poly(x**2 + y).free_symbols
{x, y}
>>> Poly(x**2 + y, x).free_symbols
{x, y}
>>> Poly(x**2 + y, x, z).free_symbols
{x, y}
```

property free_symbols_in_domain

Free symbols of the domain of `self`.

Examples

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

```
>>> Poly(x**2 + 1).free_symbols_in_domain
set()
>>> Poly(x**2 + y).free_symbols_in_domain
set()
>>> Poly(x**2 + y, x).free_symbols_in_domain
{y}
```

classmethod from_dict(*rep*, **gens*, ***args*)

Construct a polynomial from a `dict`.

classmethod from_expr(*rep*, **gens*, ***args*)

Construct a polynomial from an expression.

classmethod from_list(*rep*, **gens*, ***args*)

Construct a polynomial from a `list`.

classmethod from_poly(*rep*, **gens*, ***args*)

Construct a polynomial from a polynomial.

gcd(*g*)

Returns the polynomial GCD of `f` and `g`.

Examples

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

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

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

Extended Euclidean algorithm of `f` and `g`.

Returns `(s, t, h)` such that `h = gcd(f, g)` and `s*f + t*g = h`.

Examples

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

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))
```

property gen

Return the principal generator.

Examples

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

```
>>> Poly(x**2 + 1, x).gen
x
```

get_domain()

Get the ground domain of `f`.

get_modulus()

Get the modulus of `f`.

Examples

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

```
>>> Poly(x**2 + 1, modulus=2).get_modulus()
2
```

gff_list()

Computes greatest factorial factorization of `f`.

Examples

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

```
>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

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

ground_roots()

Compute roots of `f` by factorization in the ground domain.

Examples

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

```
>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()
{0: 2, 1: 2}
```

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

Half extended Euclidean algorithm of `f` and `g`.

Returns `(s, h)` such that `h = gcd(f, g)` and `s*f = h (mod g)`.

Examples

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

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).half_gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```