```
>>> x, y = symbols('x,y', real=True)
```

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_terms()

Transform an expression to a list of terms.

aseries(*x=None*, *n=6*, *bound=0*, *hir=False*)

Asymptotic Series expansion of self. This is equivalent to `self.series(x, oo, n)`.

Parameters

self : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

n : Value

The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

hir : Boolean

Set this parameter to be True to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

bound : Value, Integer

Use the `bound` parameter to give limit on rewriting coefficients in its normalised form.

Returns

Expr

Asymptotic series expansion of the expression.

Examples

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

```
>>> e = sin(1/x + exp(-x)) - sin(1/x)
```

```
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + 0(x**(-6), (x, oo)))*exp(-x)
```

```
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x)
exp(exp(x)/(1 - 1/x))
```

```
>>> e.aseries(x, bound=3)
exp(exp(x)/x**2)*exp(exp(x)/x)*exp(-exp(x) + exp(x)/(1 - 1/x) -␣
↪exp(x)/x - exp(x)/x**2)*exp(exp(x))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).aseries(x, n=8) 1/x

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the mrv and rewrite sub-routines. The overall idea of this algorithm is first to look for the most rapidly varying subexpression w of a given expression f and then expands f in a series in w. Then same thing is recursively done on the leading coefficient till we get constant coefficients.

If the most rapidly varying subexpression of a given expression f is f itself, the algorithm tries to find a normalised representation of the mrv set and rewrites f using this normalised representation.

If the expansion contains an order term, it will be either `O(x ** (-n))` or `O(w ** (-n))` where `w` belongs to the most rapidly varying expression of `self`.

See also:

Expr.aseries **(page 957)**
See the docstring of this function for complete details of this wrapper.

References

[R108], [R109], [R110]

cancel(**gens*, ***args*)
See the cancel function in sympy.polys

coeff(*x*, *n=1*, *right=False*, *_first=True*)
Returns the coefficient from the term(s) containing `x**n`. If `n` is zero then all terms independent of `x` will be returned.

Explanation

When x is noncommutative, the coefficient to the left (default) or right of x can be returned. The keyword 'right' is ignored when x is commutative.

Examples

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

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case expr.as_independent(x)[0] is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
```

(continues on next page)

(continued from previous page)

```
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so 1 + z*(1 + y) is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, factor_terms can be used first:

```
>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

See also:

as_coefficient **(page 949)**
separate the expression into a coefficient and factor

as_coeff_Add **(page 948)**
separate the additive constant from an expression

as_coeff_Mul **(page 948)**
separate the multiplicative constant from an expression

as_independent **(page 952)**
separate x-dependent terms/factors from others

sympy.polys.polytools.Poly.coeff_monomial **(page 2384)**
efficiently find the single coefficient of a monomial in Poly

sympy.polys.polytools.Poly.nth **(page 2408)**
like coeff_monomial but powers of monomial terms are used

collect(*syms*, *func=None*, *evaluate=True*, *exact=False*, *distribute_order_term=True*)
See the collect function in sympy.simplify

combsimp()
See the combsimp function in sympy.simplify

compute_leading_term(*x*, *logx=None*)
as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

conjugate()
Returns the complex conjugate of 'self'.

could_extract_minus_sign()
Return True if self has -1 as a leading factor or has more literal negative signs than positive signs in a sum, otherwise False.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

Though the `y - x` is considered like `-(x - y)`, since it is in a product without a leading factor of -1, the result is false below:

```
>>> (x*(y - x)).could_extract_minus_sign()
False
```

To put something in canonical form wrt to sign, use *signsimp*:

```
>>> from sympy import signsimp
>>> signsimp(x*(y - x))
-x*(x - y)
>>> _.could_extract_minus_sign()
True
```

count_ops(*visual=None*)
wrapper for count_ops that returns the operation count.

equals(*other*, *failing_expression=False*)
Return True if self == other, False if it does not, or None. If failing_expression is True then the expression which did not simplify to a 0 will be returned instead of None.

Explanation

If `self` is a Number (or complex number) that is not zero, then the result is False.

If `self` is a number and has not evaluated to zero, evalf will be used to test whether the expression evaluates to zero. If it does so and the result has significance (i.e. the precision is either -1, for a Rational result, or is greater than 1) then the evalf value will be used to return True or False.

expand(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for more information.

property expr_free_symbols

Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, do not return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(*c*)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

Examples

```
>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

See also:

extract_multiplicatively (page 963), *coeff* (page 958), *as_coefficient* (page 949)

extract_branch_factor(*allow_half=False*)

Try to write self as `exp_polar(2*pi*I*n)*z` in a nice way. Return (z, n).

```
>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_
↪factor()
(y*exp_polar(2*pi*x), -1)
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)
```

If allow_half is True, also extract exp_polar(I*pi):

```
>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(*c*)

Return None if it's not possible to make self in the form c * something in a nice way, i.e. preserving the properties of arguments of self.

Examples

```
>>> from sympy import symbols, Rational
```

```
>>> x, y = symbols('x,y', real=True)
```

```
>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2
```

```
>>> ((x*y)**3).extract_multiplicatively(x**4 * y)
```

```
>>> (2*x).extract_multiplicatively(2)
x
```

```
>>> (2*x).extract_multiplicatively(3)
```

```
>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(**gens*, ***args*)

See the factor() function in sympy.polys.polytools

fourier_series(*limits=None*)

Compute fourier sine/cosine series of self.

See the docstring of the *fourier_series()* (page 964) in sympy.series.fourier for more information.

fps(*x=None*, *x0=0*, *dir=1*, *hyper=True*, *order=4*, *rational=True*, *full=False*)

Compute formal power power series of self.

See the docstring of the *fps()* (page 964) function in sympy.series.formal for more information.

gammasimp()

See the gammasimp function in sympy.simplify

getO()

Returns the additive O(..) symbol if there is one, else None.

getn()

Returns the order of the expression.

Explanation

The order is determined either from the O(...) term. If there is no O(...) term, it returns None.

Examples

```
>>> from sympy import O
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(**args*, ***kwargs*)

See the integrate function in sympy.integrals

invert(*g*, **gens*, ***args*)

Return the multiplicative inverse of `self` mod `g` where `self` (and `g`) may be symbolic expressions).

See also:

sympy.core.numbers.mod_inverse (page 1004), *sympy.polys.polytools.invert* (page 2365)

is_algebraic_expr(**syms*)

This tests whether a given expression is algebraic or not, in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are "algebraic expressions" with symbolic exponents. This is a simple extension to the is_rational_function, including rational exponentiation.

Examples

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be an algebraic expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

See also:

is_rational_function (page 969)

References

[R111]

is_constant(**wrt*, ***flags*)

Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

Explanation

If an expression has no free symbols then it is a constant. If there are free symbols it is possible that the expression is a constant, perhaps (but not necessarily) zero. To test such expressions, a few strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two different values and the values have a precision greater than 1 then self is not constant. If the evaluations agree or could not be obtained with any precision, no decision is made. The numerical testing is done only if `wrt` is different than the free symbols.

2) differentiation with respect to variables in 'wrt' (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression

that is zero even though an expression is constant (see added test in test_expr.py). If all derivatives are zero then self is constant with respect to the given symbols.

3) finding out zeros of denominator expression with free_symbols. It will not be constant if there are zeros. It gives more negative answers for expression that are not constant.

If neither evaluation nor differentiation can prove the expression is constant, None is returned unless two numerical values happened to be the same and the flag `failing_number` is True – in that case the numerical value will be returned.

If flag simplify=False is passed, self will not be simplified; the default is True since self should be simplified before testing.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0␣
↪or 1
True
```

is_meromorphic(*x*, *a*)

This tests whether an expression is meromorphic as a function of the given symbol x at the point a.

This method is intended as a quick test that will return None if no decision can be made without simplification or more detailed analysis.

Examples

```
>>> from sympy import zoo, log, sin, sqrt
>>> from sympy.abc import x
```

```
>>> f = 1/x**2 + 1 - 2*x**3
>>> f.is_meromorphic(x, 0)
True
>>> f.is_meromorphic(x, 1)
True
>>> f.is_meromorphic(x, zoo)
True
```

```
>>> g = x**log(3)
>>> g.is_meromorphic(x, 0)
False
>>> g.is_meromorphic(x, 1)
True
>>> g.is_meromorphic(x, zoo)
False
```

```
>>> h = sin(1/x)*x**2
>>> h.is_meromorphic(x, 0)
False
>>> h.is_meromorphic(x, 1)
True
>>> h.is_meromorphic(x, zoo)
True
```

Multivalued functions are considered meromorphic when their branches are meromorphic. Thus most functions are meromorphic everywhere except at essential singularities and branch points. In particular, they will be meromorphic also on branch cuts except at their endpoints.

```
>>> log(x).is_meromorphic(x, -1)
True
>>> log(x).is_meromorphic(x, 0)
False
>>> sqrt(x).is_meromorphic(x, -1)
True
>>> sqrt(x).is_meromorphic(x, 0)
False
```

property is_number

Returns True if `self` has no free symbols and no undefined functions (AppliedUndef, to be precise). It will be faster than `if not self.free_symbols`, however, since `is_number` will fail as soon as it hits a free symbol or undefined function.

Examples

```
>>> from sympy import Function, Integral, cos, sin, pi
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> x.is_number
False
>>> f(1).is_number
False
>>> (2*x).is_number
False
>>> (2 + Integral(2, x)).is_number
False
>>> (2 + Integral(2, (x, 1, 2))).is_number
True
```

Not all numbers are Numbers in the SymPy sense:

```
>>> pi.is_number, pi.is_Number
(True, False)
```

If something is a number it should evaluate to a number with real and imaginary parts that are Numbers; the result may not be comparable, however, since the real and/or imaginary part of the result may not have precision.

```
>>> cos(1).is_number and cos(1).is_comparable
True
```

```
>>> z = cos(1)**2 + sin(1)**2 - 1
>>> z.is_number
True
>>> z.is_comparable
False
```

See also:

sympy.core.basic.Basic.is_comparable (page 935)

is_polynomial(**syms*)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are “polynomials” with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and Poly(expr, *syms) should work if and only if expr.is_polynomial(*syms) returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do Symbol(‘z’, polynomial=True).

Examples

```
>>> from sympy import Symbol, Function
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
>>> (2**x + 1).is_polynomial(2**x)
True
>>> f = Function('f')
>>> (f(x) + 1).is_polynomial(x)
False
>>> (f(x) + 1).is_polynomial(f(x))
True
>>> (1/f(x) + 1).is_polynomial(f(x))
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a polynomial to become one.

```
>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

See also .is_rational_function()

is_rational_function(**syms*)

Test whether function is a ratio of two polynomials in the given symbols, syms. When syms is not given, all free symbols will be used. The rational function does not have to be in expanded or in any kind of canonical form.

This function returns False for expressions that are “rational functions” with symbolic exponents. Thus, you should be able to call .as_numer_denom() and apply

polynomial algorithms to the result for expressions for which this returns True.

This is not part of the assumptions system. You cannot do Symbol('z', rational_function=True).

Examples

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

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

This function does not attempt any nontrivial simplifications that may result in an expression that does not appear to be a rational function to become one.

```
>>> from sympy import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

See also is_algebraic_expr().

leadterm(*x*, *logx=None*, *cdir=0*)

Returns the leading term a*x**b as a tuple (a, b).

Examples

```
>>> from sympy.abc import x
>>> (1+x+x**2).leadterm(x)
(1, 0)
>>> (1/x**2+x+x**2).leadterm(x)
(1, -2)
```

limit(*x*, *xlim*, *dir='+'*)

Compute limit x->xlim.

lseries(*x=None*, *x0=0*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper for series yielding an iterator of the terms of the series.

Note: an infinite series will yield an infinite iterator. The following, for exaxmple, will never terminate. It will just keep printing terms of the sin(x) series:

```
for term in sin(x).lseries(x):
    print term
```

The advantage of lseries() over nseries() is that many times you are just interested in the next term in the series (i.e. the first term for example), but you do not know how many you should ask for in nseries() using the "n" parameter.

See also nseries().

normal()

expression -> a/b

See also:

as_numer_denom **(page 955)**
return `(a, b)` instead of `a/b`

nseries(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries is called. This calculates "n" terms in the innermost expressions and then builds up the final series just by "cross-multiplying" everything out.

The optional `logx` parameter can be used to replace any log(x) in the returned series with a symbolic value to avoid evaluating log(x) at 0. A symbol to use in place of log(x) should be provided.

Advantage – it's fast, because we do not have to determine how many terms we need to calculate in advance.

Disadvantage – you may end up with less terms than you may have expected, but the O(x**n) term appended will always be correct and so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a wrapper to series which will try harder to return the correct number of terms.

See also lseries().

Examples

```
>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)
```

Handling of the `logx` parameter — in the following example the expansion fails since `sin` does not have an asymptotic expansion at -oo (the limit of log(x) as x approaches 0):

```
>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)
```

In the following example, the expansion works but only returns self unless the `logx` parameter is used:

```
>>> e = x**y
>>> e.nseries(x, 0, 2)
x**y
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(*constants=()*, *tolerance=None*, *full=False*)

See the nsimplify function in sympy.simplify

powsimp(**args*, ***kwargs*)

See the powsimp function in sympy.simplify

primitive()

Return the positive Rational that can be extracted non-recursively from every term of self (i.e., self is treated like an Add). This is like the as_coeff_Mul() method but primitive always extracts a positive Rational (never a negative or a Float).

Examples

```
>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(***kwargs*)

See the radsimp function in sympy.simplify

ratsimp()

See the ratsimp function in sympy.simplify

removeO()

Removes the additive O(..) symbol if there is one

round(*n=None*)

Return x rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Examples

```
>>> from sympy import pi, E, I, S, Number
>>> pi.round()
3
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6 + 3*I
```

The round method has a chopping effect:

```
>>> (2*pi + I/10).round()
6
>>> (pi/10 + 2*I).round()
2*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I
```

Notes

The Python `round` function uses the SymPy `round` method so it will always return a SymPy number (not a Python float or int):

```
>>> isinstance(round(S(123), -2), Number)
True
```

separate(*deep=False*, *force=False*)

See the separate function in sympy.simplify

series(*x=None*, *x0=0*, *n=6*, *dir='+'*, *logx=None*, *cdir=0*)

Series expansion of "self" around `x = x0` yielding either terms of the series one by one (the lazy series given when n=None), else all the terms at once when n != None.

Returns the series expansion of "self" around the point `x = x0` with respect to `x` up to `O((x - x0)**n, x, x0)` (default n is 6).

If `x=None` and `self` is univariate, the univariate symbol will be supplied, otherwise an error will be raised.

Parameters

expr : Expression

The expression whose series is to be expanded.

x : Symbol

It is the variable of the expression to be calculated.

x0 : Value

The value around which x is calculated. Can be any value from `-oo` to `oo`.

n : Value

The value used to represent the order in terms of `x**n`, up to which the series is to be expanded.

dir : String, optional

The series-expansion can be bi-directional. If `dir="+"`, then (x->x0+). If `dir="-", then (x->x0-). For infinite ``x0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

logx : optional

It is used to replace any log(x) in the returned series with a symbolic value rather than evaluating the actual value.

cdir : optional

It stands for complex direction, and indicates the direction from which the expansion needs to be evaluated.

Returns

Expr : Expression

Series expansion of the expression about x0

Raises

TypeError

If "n" and "x0" are infinity objects

PoleError

If "x0" is an infinity object

Examples

```
>>> from sympy import cos, exp, tan
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)
```

If `n=None` then a generator of the series terms will be returned.

```
>>> term=cos(x).series(n=None)
>>> [next(term) for i in range(2)]
[1, -x**2/2]
```

For dir=+ (default) the series is calculated from the right and for dir=- the series from the left. For smooth functions this flag will not alter the results.

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
>>> f = tan(x)
>>> f.series(x, 2, 6, "+")
tan(2) + (1 + tan(2)**2)*(x - 2) + (x - 2)**2*(tan(2)**3 + tan(2)) +
(x - 2)**3*(1/3 + 4*tan(2)**2/3 + tan(2)**4) + (x - 2)**4*(tan(2)**5 +
5*tan(2)**3/3 + 2*tan(2)/3) + (x - 2)**5*(2/15 + 17*tan(2)**2/15 +
2*tan(2)**4 + tan(2)**6) + O((x - 2)**6, (x, 2))
```

```
>>> f.series(x, 2, 3, "-")
tan(2) + (2 - x)*(-tan(2)**2 - 1) + (2 - x)**2*(tan(2)**3 + tan(2))
+ O((x - 2)**3, (x, 2))
```

For rational expressions this method may return original expression without the Order term. >>> (1/x).series(x, n=8) 1/x

taylor_term(*n*, *x*, **previous_terms*)

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the "previous_terms".

together(**args*, ***kwargs*)

See the together function in sympy.polys

trigsimp(***args*)

See the trigsimp function in sympy.simplify

UnevaluatedExpr

class sympy.core.expr.**UnevaluatedExpr**(*arg*, ***kwargs*)

Expression that is not evaluated unless released.

Examples

```
>>> from sympy import UnevaluatedExpr
>>> from sympy.abc import x
>>> x*(1/x)
1
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

AtomicExpr

class sympy.core.expr.**AtomicExpr**(**args*)

A parent class for object which are both atoms and Exprs.

For example: Symbol, Number, Rational, Integer, ... But not: Add, Mul, Pow, ...

symbol

Symbol

class sympy.core.symbol.**Symbol**(*name*, ***assumptions*)

Assumptions:

commutative = True

You can override the default assumptions in the constructor.

Examples

```
>>> from sympy import symbols
>>> A,B = symbols('A,B', commutative = False)
>>> bool(A*B != B*A)
True
>>> bool(A*B*2 == 2*A*B) == True # multiplication by scalars is␣
↪commutative
True
```

Wild

class sympy.core.symbol.**Wild**(*name*, *exclude=()*, *properties=()*, ***assumptions*)

A Wild symbol matches anything, or anything without whatever is explicitly excluded.

Parameters

name : str

Name of the Wild instance.

exclude : iterable, optional

Instances in `exclude` will not be matched.

properties : iterable of functions, optional

Functions, each taking an expressions as input and returns a `bool`. All functions in `properties` need to return `True` in order for the Wild instance to match the expression.

Examples

```
>>> from sympy import Wild, WildFunction, cos, pi
>>> from sympy.abc import x, y, z
>>> a = Wild('a')
>>> x.match(a)
{a_: x}
>>> pi.match(a)
{a_: pi}
>>> (3*x**2).match(a*x)
{a_: 3*x}
>>> cos(x).match(a)
{a_: cos(x)}
>>> b = Wild('b', exclude=[x])
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b_}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

Tips

When using Wild, be sure to use the exclude keyword to make the pattern more precise. Without the exclude pattern, you may get matches that are technically correct, but not what you wanted. For example, using the above without exclude:

```
>>> from sympy import symbols
>>> a, b = symbols('a b', cls=Wild)
>>> (2 + 3*y).match(a*x + b*y)
{a_: 2/x, b_: 3}
```

This is technically correct, because (2/x)*x + 3*y == 2 + 3*y, but you probably wanted it to not match at all. The issue is that you really did not want a and b to include x and y, and the exclude parameter lets you specify exactly this. With the exclude parameter, the pattern will not match.

```
>>> a = Wild('a', exclude=[x, y])
>>> b = Wild('b', exclude=[x, y])
>>> (2 + 3*y).match(a*x + b*y)
```

Exclude also helps remove ambiguity from matches.

```
>>> E = 2*x**3*y*z
>>> a, b = symbols('a b', cls=Wild)
>>> E.match(a*b)
{a_: 2*y*z, b_: x**3}
>>> a = Wild('a', exclude=[x, y])
>>> E.match(a*b)
{a_: z, b_: 2*x**3*y}
>>> a = Wild('a', exclude=[x, y, z])
```

(continues on next page)

(continued from previous page)

```
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Wild also accepts a `properties` parameter:

```
>>> a = Wild('a', properties=[lambda k: k.is_Integer])
>>> E.match(a*b)
{a_: 2, b_: x**3*y*z}
```

Dummy

class sympy.core.symbol.**Dummy**(*name=None*, *dummy_index=None*, ***assumptions*)

Dummy symbols are each unique, even if they have the same name:

Examples

```
>>> from sympy import Dummy
>>> Dummy("x") == Dummy("x")
False
```

If a name is not supplied then a string value of an internal count will be used. This is useful when a temporary variable is needed and the name of the variable used in the expression is not important.

```
>>> Dummy()
_Dummy_10
```

symbols

sympy.core.symbol.**symbols**(*names*, *, *cls=<class 'sympy.core.symbol.Symbol'>*, ***args*) → Any

Transform strings into instances of *Symbol* (page 976) class.

symbols() (page 978) function returns a sequence of symbols with names taken from `names` argument, which can be a comma or whitespace delimited string, or a sequence of strings:

```
>>> from sympy import symbols, Function

>>> x, y, z = symbols('x,y,z')
>>> a, b, c = symbols('a b c')
```

The type of output is dependent on the properties of input arguments:

```
>>> symbols('x')
x
>>> symbols('x,')
(x,)
```

(continues on next page)

(continued from previous page)

```
>>> symbols('x,y')
(x, y)
>>> symbols(('a', 'b', 'c'))
(a, b, c)
>>> symbols(['a', 'b', 'c'])
[a, b, c]
>>> symbols({'a', 'b', 'c'})
{a, b, c}
```

If an iterable container is needed for a single symbol, set the `seq` argument to `True` or terminate the symbol name with a comma:

```
>>> symbols('x', seq=True)
(x,)
```

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

>>> symbols('x5:10')
(x5, x6, x7, x8, x9)
>>> symbols('x5(:2)')
(x50, x51)

>>> symbols('x5:10,y:5')
(x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)

>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

If the character to the right of the colon is a letter, then the single letter to the left (or 'a' if there is none) is taken as the start and all characters in the lexicographic range *through* the letter to the right are used as the range:

```
>>> symbols('x:z')
(x, y, z)
>>> symbols('x:c')  # null range
()
>>> symbols('x(:c)')
(xa, xb, xc)

>>> symbols(':c')
(a, b, c)

>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)
```

(continues on next page)

(continued from previous page)

```
>>> symbols(('a:d', 'x:z'))
((a, b, c, d), (x, y, z))
```

Multiple ranges are supported; contiguous numerical ranges should be separated by parentheses to disambiguate the ending number of one range from the starting number of the next:

```
>>> symbols('x:2(1:3)')
(x01, x02, x11, x12)
>>> symbols(':3:2')  # parsing is from left to right
(00, 01, 10, 11, 20, 21)
```

Only one pair of parentheses surrounding ranges are removed, so to include parentheses around ranges, double them. And to include spaces, commas, or colons, escape them with a backslash:

```
>>> symbols('x((a:b))')
(x(a), x(b))
>>> symbols(r'x(:1\,:2)')  # or r'x((:1)\,(:2))'
(x(0,0), x(0,1))
```

All newly created symbols have assumptions set according to `args`:

```
>>> a = symbols('a', integer=True)
>>> a.is_integer
True

>>> x, y, z = symbols('x,y,z', real=True)
>>> x.is_real and y.is_real and z.is_real
True
```

Despite its name, *symbols()* (page 978) can create symbol-like objects like instances of Function or Wild classes. To achieve this, set `cls` keyword argument to the desired type:

```
>>> symbols('f,g,h', cls=Function)
(f, g, h)

>>> type(_[0])
<class 'sympy.core.function.UndefinedFunction'>
```

var

sympy.core.symbol.**var**(*names*, ***args*)

Create symbols and inject them into the global namespace.

Explanation

This calls *symbols()* (page 978) with the same arguments and puts the results into the *global* namespace. It's recommended not to use *var()* (page 980) in library code, where *symbols()* (page 978) has to be used:

```
.. rubric:: Examples
```

```
>>> from sympy import var
```

```
>>> var('x')
x
>>> x # noqa: F821
x
```

```
>>> var('a,ab,abc')
(a, ab, abc)
>>> abc # noqa: F821
abc
```

```
>>> var('x,y', real=True)
(x, y)
>>> x.is_real and y.is_real # noqa: F821
True
```

See *symbols()* (page 978) documentation for more details on what kinds of arguments can be passed to *var()* (page 980).

numbers

Number

class sympy.core.numbers.**Number**(**obj*)

Represents atomic numbers in SymPy.

Explanation

Floating point numbers are represented by the Float class. Rational numbers (of any size) are represented by the Rational class. Integer numbers (of any size) are represented by the Integer class. Float and Rational are subclasses of Number; Integer is a subclass of Rational.

For example, `2/3` is represented as `Rational(2, 3)` which is a different object from the floating point number obtained with Python division `2/3`. Even for numbers that are exactly represented in binary, there is a difference between how two forms, such as `Rational(1, 2)` and `Float(0.5)`, are used in SymPy. The rational form is to be preferred in symbolic computations.

Other kinds of numbers, such as algebraic numbers `sqrt(2)` or complex numbers `3 + 4*I`, are not instances of Number class as they are not atomic.

See also:

Float (page 982), *Integer* (page 987), *Rational* (page 985)

as_coeff_Add(*rational=False*)
Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)
Efficiently extract the coefficient of a product.

cofactors(*other*)
Compute GCD and cofactors of $self$ and $other$.

gcd(*other*)
Compute GCD of $self$ and $other$.

lcm(*other*)
Compute LCM of $self$ and $other$.

Float

class sympy.core.numbers.**Float**(*num*, *dps=None*, *precision=None*)
Represent a floating-point number of arbitrary precision.

Examples

```
>>> from sympy import Float
>>> Float(3.5)
3.50000000000000
>>> Float(3)
3.00000000000000
```

Creating Floats from strings (and Python `int` and `long` types) will give a minimum precision of 15 digits, but the precision will automatically increase to capture all digits entered.

```
>>> Float(1)
1.00000000000000
>>> Float(10**20)
100000000000000000000.
>>> Float('1e20')
100000000000000000000.
```

However, *floating-point* numbers (Python `float` types) retain only 15 digits of precision:

```
>>> Float(1e20)
1.00000000000000e+20
>>> Float(1.23456789123456789)
1.23456789123457
```

It may be preferable to enter high-precision decimal numbers as strings:

```
>>> Float('1.23456789123456789')
1.23456789123456789
```

The desired number of digits can also be specified:

```
>>> Float('1e-3', 3)
0.00100
>>> Float(100, 4)
100.0
```

Float can automatically count significant figures if a null string is sent for the precision; spaces or underscores are also allowed. (Auto- counting is only allowed for strings, ints and longs).

```
>>> Float('123 456 789.123_456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

If a number is written in scientific notation, only the digits before the exponent are considered significant if a decimal appears, otherwise the "e" signifies only how to move the decimal:

```
>>> Float('60.e2', '')  # 2 digits significant
6.0e+3
>>> Float('60e2', '')  # 4 digits significant
6000.
>>> Float('600e-2', '')  # 3 digits significant
6.00
```

Notes

Floats are inexact by their nature unless their value is a binary-exact value.

```
>>> approx, exact = Float(.1, 1), Float(.125, 1)
```

For calculation purposes, evalf needs to be able to change the precision but this will not increase the accuracy of the inexact value. The following is the most accurate 5-digit approximation of a value of 0.1 that had only 1 digit of precision:

```
>>> approx.evalf(5)
0.099609
```

By contrast, 0.125 is exact in binary (as it is in base 10) and so it can be passed to Float or evalf to obtain an arbitrary precision with matching accuracy:

```
>>> Float(exact, 5)
0.12500
>>> exact.evalf(20)
0.12500000000000000000
```

Trying to make a high-precision Float from a float is not disallowed, but one must keep in mind that the *underlying float* (not the apparent decimal value) is being obtained with high precision. For example, 0.3 does not have a finite binary representation. The closest rational is the fraction 5404319552844595/2**54. So if you try to obtain a Float of 0.3 to 20 digits of precision you will not see the same thing as 0.3 followed by 19 zeros:

```
>>> Float(0.3, 20)
0.29999999999999998890
```

If you want a 20-digit value of the decimal 0.3 (not the floating point approximation of 0.3) you should send the 0.3 as a string. The underlying representation is still binary but a higher precision than Python's float is used:

```
>>> Float('0.3', 20)
0.30000000000000000000
```

Although you can increase the precision of an existing Float using Float it will not increase the accuracy – the underlying value is not changed:

```
>>> def show(f): # binary rep of Float
...     from sympy import Mul, Pow
...     s, m, e, b = f._mpf_
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print('%s at prec=%s' % (v, f._prec))
...
>>> t = Float('0.3', 3)
>>> show(t)
4915/2**14 at prec=13
>>> show(Float(t, 20)) # higher prec, not higher accuracy
4915/2**14 at prec=70
>>> show(Float(t, 2)) # lower prec
307/2**10 at prec=10
```

The same thing happens when evalf is used on a Float:

```
>>> show(t.evalf(20))
4915/2**14 at prec=70
>>> show(t.evalf(2))
307/2**10 at prec=10
```

Finally, Floats can be instantiated with an mpf tuple (n, c, p) to produce the number (-1)**n*c*2**p:

```
>>> n, c, p = 1, 5, 0
>>> (-1)**n*c*2**p
-5
>>> Float((1, 5, 0))
-5.00000000000000
```

An actual mpf tuple also contains the number of bits in c as the last element of the tuple:

```
>>> _._mpf_
(1, 5, 0, 3)
```

This is not needed for instantiation and is not the same thing as the precision. The mpf tuple and the precision are two separate quantities that Float tracks.

In SymPy, a Float is a number that can be computed with arbitrary precision. Although floating point 'inf' and 'nan' are not such numbers, Float can create these numbers:

```
>>> Float('-inf')
-oo
>>> _.is_Float
False
```

Rational

class sympy.core.numbers.**Rational**(*p*, *q=None*, *gcd=None*)

Represents rational numbers (p/q) of any size.

Examples

```
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(1, 2)
1/2
```

Rational is unprejudiced in accepting input. If a float is passed, the underlying value of the binary representation will be returned:

```
>>> Rational(.5)
1/2
>>> Rational(.2)
3602879701896397/18014398509481984
```

If the simpler representation of the float is desired then consider limiting the denominator to the desired value or convert the float to a string (which is roughly equivalent to limiting the denominator to 10**12):

```
>>> Rational(str(.2))
1/5
>>> Rational(.2).limit_denominator(10**12)
1/5
```

An arbitrarily precise Rational is obtained when a string literal is passed:

```
>>> Rational("1.23")
123/100
>>> Rational('1e-2')
1/100
>>> Rational(".1")
1/10
>>> Rational('1e-2/3.2')
1/320
```

The conversion of other types of strings can be handled by the sympify() function, and conversion of floats to expressions or simple fractions can be handled with nsimplify:

```
>>> S('.[3]')  # repeating digits in brackets
1/3
>>> S('3**2/10')  # general expressions
9/10
>>> nsimplify(.3)  # numbers that have a simple form
3/10
```

But if the input does not reduce to a literal Rational, an error will be raised:

```
>>> Rational(pi)
Traceback (most recent call last):
...
TypeError: invalid input: pi
```

Low-level

Access numerator and denominator as .p and .q:

```
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.p
3
>>> r.q
4
```

Note that p and q return integers (not SymPy Integers) so some care is needed when using them in expressions:

```
>>> r.p/r.q
0.75
```

If an unevaluated Rational is desired, `gcd=1` can be passed and this will keep common divisors of the numerator and denominator from being eliminated. It is not possible, however, to leave a negative value in the denominator.

```
>>> Rational(2, 4, gcd=1)
2/4
>>> Rational(2, -4, gcd=1).q
4
```

See also:

sympy.core.sympify.sympify (page 918), *sympy.simplify.simplify.nsimplify* (page 667)

as_coeff_Add(*rational=False*)

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

as_content_primitive(*radical=False*, *clear=True*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self.

Examples

```
>>> from sympy import S
>>> (S(-3)/2).as_content_primitive()
(3/2, -1)
```

See docstring of Expr.as_content_primitive for more examples.

factors(*limit=None*, *use_trial=True*, *use_rho=False*, *use_pm1=False*, *verbose=False*, *visual=False*)

A wrapper to factorint which return factors of self that are smaller than limit (or cheap to compute). Special methods of factoring are disabled by default so that only trial division is used.

limit_denominator(*max_denominator=1000000*)

Closest Rational to self with denominator at most max_denominator.

Examples

```
>>> from sympy import Rational
>>> Rational('3.141592653589793').limit_denominator(10)
22/7
>>> Rational('3.141592653589793').limit_denominator(100)
311/99
```

Integer

class sympy.core.numbers.**Integer**(*i*)

Represents integer numbers of any size.

Examples

```
>>> from sympy import Integer
>>> Integer(3)
3
```

If a float or a rational is passed to Integer, the fractional part will be discarded; the effect is of rounding toward zero.

```
>>> Integer(3.8)
3
>>> Integer(-3.8)
-3
```

A string is acceptable input if it can be parsed as an integer:

```
>>> Integer("9" * 20)
99999999999999999999
```

It is rarely needed to explicitly instantiate an Integer, because Python integers are automatically converted to Integer when they are used in SymPy expressions.

AlgebraicNumber

class sympy.core.numbers.**AlgebraicNumber**(*expr*, *coeffs=None*, *alias=None*, ***args*)

Class for representing algebraic numbers in SymPy.

Symbolically, an instance of this class represents an element $\alpha \in \mathbb{Q}(\theta) \hookrightarrow \mathbb{C}$. That is, the algebraic number α is represented as an element of a particular number field $\mathbb{Q}(\theta)$, with a particular embedding of this field into the complex numbers.

Formally, the primitive element θ is given by two data points: (1) its minimal polynomial (which defines $\mathbb{Q}(\theta)$), and (2) a particular complex number that is a root of this polynomial (which defines the embedding $\mathbb{Q}(\theta) \hookrightarrow \mathbb{C}$). Finally, the algebraic number α which we represent is then given by the coefficients of a polynomial in θ.

static __new__(*cls*, *expr*, *coeffs=None*, *alias=None*, ***args*)

Construct a new algebraic number α belonging to a number field $k = \mathbb{Q}(\theta)$.

There are four instance attributes to be determined:

Attribute	Type/Meaning
`root`	*Expr* (page 947) for θ as a complex number
`minpoly`	*Poly* (page 2378), the minimal polynomial of θ
`rep`	*DMP* (page 2561) giving α as poly in θ
`alias`	*Symbol* (page 976) for θ, or `None`

See Parameters section for how they are determined.

Parameters

expr : *Expr* (page 947), or pair (m, r)

There are three distinct modes of construction, depending on what is passed as *expr*.

(1) *expr* is an *AlgebraicNumber* (page 988): In this case we begin by copying all four instance attributes from *expr*. If *coeffs* were also given, we compose the two coeff polynomials (see below). If an *alias* was given, it overrides.

(2) *expr* is any other type of *Expr* (page 947): Then `root` will equal *expr*. Therefore it must express an algebraic quantity, and we will compute its `minpoly`.

(3) *expr* is an ordered pair (m, r) giving the `minpoly` m, and a `root` r thereof, which together define θ. In this case m may be either a univariate *Poly* (page 2378) or any *Expr* (page 947) which represents the same, while r must be some *Expr* (page 947) representing a complex number that is a root of m, including both explicit expressions in radicals, and instances of *ComplexRootOf* (page 2431) or *AlgebraicNumber* (page 988).

coeffs : list, *ANP* (page 2568), None, optional (default=None)

This defines `rep`, giving the algebraic number α as a polynomial in θ.

If a list, the elements should be integers or rational numbers. If an *ANP* (page 2568), we take its coefficients (using its *to_list()* (page 2569) method). If `None`, then the list of coefficients defaults to `[1, 0]`, meaning that $\alpha = \theta$ is the primitive element of the field.

If *expr* was an *AlgebraicNumber* (page 988), let $g(x)$ be its `rep` polynomial, and let $f(x)$ be the polynomial defined by *coeffs*. Then `self.rep` will represent the composition $(f \circ g)(x)$.

alias : str, *Symbol* (page 976), None, optional (default=None)

This is a way to provide a name for the primitive element. We described several ways in which the *expr* argument can define the value of the primitive element, but none of these methods gave it a name. Here, for example, *alias* could be set as `Symbol('theta')`, in order to make this symbol appear when α is printed, or rendered as a polynomial, using the *as_poly()* (page 991) method.

Examples

Recall that we are constructing an algebraic number as a field element $\alpha \in \mathbb{Q}(\theta)$.

```
>>> from sympy import AlgebraicNumber, sqrt, CRootOf, S
>>> from sympy.abc import x
```

Example (1): $\alpha = \theta = \sqrt{2}$

```
>>> a1 = AlgebraicNumber(sqrt(2))
>>> a1.minpoly_of_element().as_expr(x)
x**2 - 2
>>> a1.evalf(10)
1.414213562
```

Example (2): $\alpha = 3\sqrt{2} - 5$, $\theta = \sqrt{2}$. We can either build on the last example:

```
>>> a2 = AlgebraicNumber(a1, [3, -5])
>>> a2.as_expr()
-5 + 3*sqrt(2)
```

or start from scratch:

```
>>> a2 = AlgebraicNumber(sqrt(2), [3, -5])
>>> a2.as_expr()
-5 + 3*sqrt(2)
```

Example (3): $\alpha = 6\sqrt{2} - 11$, $\theta = \sqrt{2}$. Again we can build on the previous example, and we see that the coeff polys are composed:

```
>>> a3 = AlgebraicNumber(a2, [2, -1])
>>> a3.as_expr()
-11 + 6*sqrt(2)
```

reflecting the fact that $(2x - 1) \circ (3x - 5) = 6x - 11$.

Example (4): $\alpha = \sqrt{2}$, $\theta = \sqrt{2} + \sqrt{3}$. The easiest way is to use the *to_number_field()* (page 2714) function:

```
>>> from sympy import to_number_field
>>> a4 = to_number_field(sqrt(2), sqrt(2) + sqrt(3))
>>> a4.minpoly_of_element().as_expr(x)
x**2 - 2
>>> a4.to_root()
sqrt(2)
>>> a4.primitive_element()
sqrt(2) + sqrt(3)
>>> a4.coeffs()
[1/2, 0, -9/2, 0]
```

but if you already knew the right coefficients, you could construct it directly:

```
>>> a4 = AlgebraicNumber(sqrt(2) + sqrt(3), [S(1)/2, 0, S(-9)/2, 0])
>>> a4.to_root()
sqrt(2)
>>> a4.primitive_element()
sqrt(2) + sqrt(3)
```

Example (5): Construct the Golden Ratio as an element of the 5th cyclotomic field, supposing we already know its coefficients. This time we introduce the alias ζ for the primitive element of the field:

```
>>> from sympy import cyclotomic_poly
>>> from sympy.abc import zeta
>>> a5 = AlgebraicNumber(CRootOf(cyclotomic_poly(5), -1),
...                      [-1, -1, 0, 0], alias=zeta)
>>> a5.as_poly().as_expr()
-zeta**3 - zeta**2
>>> a5.evalf()
1.61803398874989
```

(The index `-1` to `CRootOf` selects the complex root with the largest real and imaginary parts, which in this case is $e^{2i\pi/5}$. See *ComplexRootOf* (page 2431).)

Example (6): Building on the last example, construct the number $2\phi \in \mathbb{Q}(\phi)$, where ϕ is the Golden Ratio:

```
>>> from sympy.abc import phi
>>> a6 = AlgebraicNumber(a5.to_root(), coeffs=[2, 0], alias=phi)
>>> a6.as_poly().as_expr()
2*phi
>>> a6.primitive_element().evalf()
1.61803398874989
```

Note that we needed to use `a5.to_root()`, since passing `a5` as the first argument would have constructed the number 2ϕ as an element of the field $\mathbb{Q}(\zeta)$:

```
>>> a6_wrong = AlgebraicNumber(a5, coeffs=[2, 0])
>>> a6_wrong.as_poly().as_expr()
-2*zeta**3 - 2*zeta**2
>>> a6_wrong.primitive_element().evalf()
0.309016994374947 + 0.951056516295154*I
```

as_expr(*x=None*)

Create a Basic expression from `self`.

as_poly(*x=None*)

Create a Poly instance from `self`.

coeffs()

Returns all SymPy coefficients of an algebraic number.

field_element(*coeffs*)

Form another element of the same number field.

Parameters

coeffs : list, *ANP* (page 2568)

Like the *coeffs* arg to the class *constructor* (page 988), defines the new element as a polynomial in the primitive element.

If a list, the elements should be integers or rational numbers. If an *ANP* (page 2568), we take its coefficients (using its *to_list()* (page 2569) method).

Explanation

If we represent $\alpha \in \mathbb{Q}(\theta)$, form another element $\beta \in \mathbb{Q}(\theta)$ of the same number field.

Examples

```
>>> from sympy import AlgebraicNumber, sqrt
>>> a = AlgebraicNumber(sqrt(5), [-1, 1])
>>> b = a.field_element([3, 2])
>>> print(a)
1 - sqrt(5)
>>> print(b)
2 + 3*sqrt(5)
>>> print(b.primitive_element() == a.primitive_element())
True
```

See also:

AlgebraicNumber.__new__ (page 988)

property is_aliased

Returns `True` if `alias` was set.

property is_primitive_element

Say whether this algebraic number $\alpha \in \mathbb{Q}(\theta)$ is equal to the primitive element θ for its field.

minpoly_of_element()

Compute the minimal polynomial for this algebraic number.

Explanation

Recall that we represent an element $\alpha \in \mathbb{Q}(\theta)$. Our instance attribute `self.minpoly` is the minimal polynomial for our primitive element θ. This method computes the minimal polynomial for α.

native_coeffs()

Returns all native coefficients of an algebraic number.

primitive_element()

Get the primitive element θ for the number field $\mathbb{Q}(\theta)$ to which this algebraic number α belongs.

Returns

AlgebraicNumber

to_algebraic_integer()

Convert `self` to an algebraic integer.

to_primitive_element(*radicals=True*)

Convert `self` to an *AlgebraicNumber* (page 988) instance that is equal to its own primitive element.

Parameters

radicals : boolean, optional (default=True)

If `True`, then we will try to return an *AlgebraicNumber* (page 988) whose `root` is an expression in radicals. If that is not possible (or if *radicals* is `False`), `root` will be a *ComplexRootOf* (page 2431).

Returns

AlgebraicNumber

Explanation

If we represent $\alpha \in \mathbb{Q}(\theta)$, $\alpha \neq \theta$, construct a new *AlgebraicNumber* (page 988) that represents $\alpha \in \mathbb{Q}(\alpha)$.

Examples

```
>>> from sympy import sqrt, to_number_field
>>> from sympy.abc import x
>>> a = to_number_field(sqrt(2), sqrt(2) + sqrt(3))
```

The *AlgebraicNumber* (page 988) `a` represents the number $\sqrt{2}$ in the field $\mathbb{Q}(\sqrt{2}+\sqrt{3})$. Rendering `a` as a polynomial,

```
>>> a.as_poly().as_expr(x)
x**3/2 - 9*x/2
```

reflects the fact that $\sqrt{2} = \theta^3/2 - 9\theta/2$, where $\theta = \sqrt{2} + \sqrt{3}$.

`a` is not equal to its own primitive element. Its minpoly

```
>>> a.minpoly.as_poly().as_expr(x)
x**4 - 10*x**2 + 1
```

is that of θ.

Converting to a primitive element,

```
>>> a_prim = a.to_primitive_element()
>>> a_prim.minpoly.as_poly().as_expr(x)
x**2 - 2
```

we obtain an *AlgebraicNumber* (page 988) whose `minpoly` is that of the number itself.

See also:

is_primitive_element (page 991)

to_root(*radicals=True*, *minpoly=None*)

Convert to an *Expr* (page 947) that is not an *AlgebraicNumber* (page 988), specifically, either a *ComplexRootOf* (page 2431), or, optionally and where possible, an expression in radicals.

Parameters

radicals : boolean, optional (default=True)

If `True`, then we will try to return the root as an expression in radicals. If that is not possible, we will return a *ComplexRootOf* (page 2431).

minpoly : *Poly* (page 2378)

If the minimal polynomial for $self$ has been pre-computed, it can be passed in order to save time.

NumberSymbol

class sympy.core.numbers.**NumberSymbol**

approximation(*number_cls*)

Return an interval with number_cls endpoints that contains the value of NumberSymbol. If not implemented, then return None.

RealNumber

sympy.core.numbers.**RealNumber**

alias of *Float* (page 982)

igcd

sympy.core.numbers.**igcd**(**args*)

Computes nonnegative integer greatest common divisor.

Explanation

The algorithm is based on the well known Euclid's algorithm [R112]. To improve speed, `igcd()` has its own caching mechanism.

Examples

```
>>> from sympy import igcd
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

References

[R112]

ilcm

sympy.core.numbers.**ilcm**(**args*)

Computes integer least common multiple.

Examples

```
>>> from sympy import ilcm
>>> ilcm(5, 10)
10
>>> ilcm(7, 3)
21
>>> ilcm(5, 10, 15)
30
```

seterr

sympy.core.numbers.**seterr**(*divide=False*)

Should SymPy raise an exception on 0/0 or return a nan?

divide == True .... raise an exception divide == False ... return nan

Zero

class sympy.core.numbers.**Zero**

The number zero.

Zero is a singleton, and can be accessed by `S.Zero`

Examples

```
>>> from sympy import S, Integer
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```

References

[R113]

One

class sympy.core.numbers.**One**

The number one.

One is a singleton, and can be accessed by `S.One`.

Examples

```
>>> from sympy import S, Integer
>>> Integer(1) is S.One
True
```

References

[R114]

NegativeOne

class sympy.core.numbers.**NegativeOne**

The number negative one.

NegativeOne is a singleton, and can be accessed by `S.NegativeOne`.

Examples

```
>>> from sympy import S, Integer
>>> Integer(-1) is S.NegativeOne
True
```

See also:

One (page 995)

References

[R115]

Half

class sympy.core.numbers.**Half**

The rational number 1/2.

Half is a singleton, and can be accessed by `S.Half`.

Examples

```
>>> from sympy import S, Rational
>>> Rational(1, 2) is S.Half
True
```

References

[R116]