Explanation

Given relatively prime univariate polynomials f and g, rewrite their quotient to a normal form defined as follows:

$$\frac{f(n)}{g(n)} = Z \cdot \frac{A(n)C(n+1)}{B(n)C(n)}$$

where Z is an arbitrary constant and A, B, C are monic polynomials in n with the following properties:

1. $\gcd(A(n), B(n+h)) = 1 \forall h \in \mathbb{N}$
2. $\gcd(B(n), C(n+1)) = 1$
3. $\gcd(A(n), C(n)) = 1$

This normal form, or rational factorization in other words, is a crucial step in Gosper's algorithm and in solving of difference equations. It can be also used to decide if two hypergeometric terms are similar or not.

This procedure will return a tuple containing elements of this factorization in the form `(Z*A, B, C)`.

Examples

```
>>> from sympy.concrete.gosper import gosper_normal
>>> from sympy.abc import n
```

```
>>> gosper_normal(4*n+5, 2*(4*n+1)*(2*n+3), n, polys=False)
(1/4, n + 3/2, n + 1/4)
```

sympy.concrete.gosper.**gosper_term**(*f*, *n*)

Compute Gosper's hypergeometric term for f.

Explanation

Suppose f is a hypergeometric term such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and f_k does not depend on n. Returns a hypergeometric term g_n such that $g_{n+1} - g_n = f_n$.

Examples

```
>>> from sympy.concrete.gosper import gosper_term
>>> from sympy import factorial
>>> from sympy.abc import n
```

```
>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)/(n + 1/4)
```

sympy.concrete.gosper.**gosper_sum**(*f*, *k*)

Gosper's hypergeometric summation algorithm.

Explanation

Given a hypergeometric term f such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and $f(n)$ does not depend on n, returns $g_n - g(0)$ where $g_{n+1} - g_n = f_n$, or None if s_n cannot be expressed in closed form as a sum of hypergeometric terms.

Examples

```
>>> from sympy.concrete.gosper import gosper_sum
>>> from sympy import factorial
>>> from sympy.abc import n, k
```

```
>>> f = (4*k + 1)*factorial(k)/factorial(2*k + 1)
>>> gosper_sum(f, (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
>>> _.subs(n, 2) == sum(f.subs(k, i) for i in [0, 1, 2])
True
>>> gosper_sum(f, (k, 3, n))
(-60*factorial(n) + factorial(2*n + 1))/(60*factorial(2*n + 1))
>>> _.subs(n, 5) == sum(f.subs(k, i) for i in [3, 4, 5])
True
```

References

[R98]

Core

sympify

sympify

sympy.core.sympify.**sympify**(*a*, *locals=None*, *convert_xor=True*, *strict=False*, *rational=False*, *evaluate=None*)

Converts an arbitrary expression to a type that can be used inside SymPy.

Parameters

a :

- any object defined in SymPy
- standard numeric Python types: `int`, `long`, `float`, `Decimal`

- strings (like "0.09", "2e-19" or 'sin(x)')
- booleans, including None (will leave None unchanged)
- dicts, lists, sets or tuples containing any of the above

convert_xor : bool, optional

If true, treats ^ as exponentiation. If False, treats ^ as XOR itself. Used only when input is a string.

locals : any object defined in SymPy, optional

In order to have strings be recognized it can be imported into a namespace dictionary and passed as locals.

strict : bool, optional

If the option strict is set to True, only the types for which an explicit conversion has been defined are converted. In the other cases, a SympifyError is raised.

rational : bool, optional

If True, converts floats into *Rational* (page 985). If False, it lets floats remain as it is. Used only when input is a string.

evaluate : bool, optional

If False, then arithmetic and operators will be converted into their SymPy equivalents. If True the expression will be evaluated and the result will be returned.

Explanation

It will convert Python ints into instances of *Integer* (page 987), floats into instances of *Float* (page 982), etc. It is also able to coerce symbolic expressions which inherit from *Basic* (page 927). This can be useful in cooperation with SAGE.

Warning: Note that this function uses eval, and thus shouldn't be used on unsanitized input.

If the argument is already a type that SymPy understands, it will do nothing but return that value. This can be used at the beginning of a function to ensure you are working with the correct type.

Examples

```
>>> from sympy import sympify
```

```
>>> sympify(2).is_integer
True
>>> sympify(2).is_real
True
```

```
>>> sympify(2.0).is_real
True
>>> sympify("2.0").is_real
True
>>> sympify("2e-45").is_real
True
```

If the expression could not be converted, a SympifyError is raised.

```
>>> sympify("x***2")
Traceback (most recent call last):
...
SympifyError: SympifyError: "could not parse 'x***2'"
```

Locals

The sympification happens with access to everything that is loaded by `from sympy import *`; anything used in a string that is not defined by that import will be converted to a symbol. In the following, the `bitcount` function is treated as a symbol and the `O` is interpreted as the *Order* (page 627) object (used with series) and it raises an error when used improperly:

```
>>> s = 'bitcount(42)'
>>> sympify(s)
bitcount(42)
>>> sympify("O(x)")
O(x)
>>> sympify("O + 1")
Traceback (most recent call last):
...
TypeError: unbound method...
```

In order to have `bitcount` be recognized it can be imported into a namespace dictionary and passed as locals:

```
>>> ns = {}
>>> exec('from sympy.core.evalf import bitcount', ns)
>>> sympify(s, locals=ns)
6
```

In order to have the `O` interpreted as a Symbol, identify it as such in the namespace dictionary. This can be done in a variety of ways; all three of the following are possibilities:

```
>>> from sympy import Symbol
>>> ns["O"] = Symbol("O")  # method 1
>>> exec('from sympy.abc import O', ns)  # method 2
>>> ns.update(dict(O=Symbol("O")))  # method 3
>>> sympify("O + 1", locals=ns)
O + 1
```

If you want *all* single-letter and Greek-letter variables to be symbols then you can use the clashing-symbols dictionaries that have been defined there as private variables: `_clash1` (single-letter variables), _clash2 (the multi-letter Greek names) or _clash (both single and multi-letter names that are defined in abc).

```
>>> from sympy.abc import _clash1
>>> set(_clash1)
{'E', 'I', 'N', 'O', 'Q', 'S'}
>>> sympify('I & Q', _clash1)
I & Q
```

Strict

If the option strict is set to True, only the types for which an explicit conversion has been defined are converted. In the other cases, a SympifyError is raised.

```
>>> print(sympify(None))
None
>>> sympify(None, strict=True)
Traceback (most recent call last):
...
SympifyError: SympifyError: None
```

Deprecated since version 1.6: sympify(obj) automatically falls back to str(obj) when all other conversion methods fail, but this is deprecated. strict=True will disable this deprecated behavior. See *The string fallback in sympify()* (page 176).

Evaluation

If the option evaluate is set to False, then arithmetic and operators will be converted into their SymPy equivalents and the evaluate=False option will be added. Nested Add or Mul will be denested first. This is done via an AST transformation that replaces operators with their SymPy equivalents, so if an operand redefines any of those operations, the redefined operators will not be used. If argument a is not a string, the mathematical expression is evaluated before being passed to sympify, so adding evaluate=False will still return the evaluated result of expression.

```
>>> sympify('2**2 / 3 + 5')
19/3
>>> sympify('2**2 / 3 + 5', evaluate=False)
2**2/3 + 5
>>> sympify('4/2+7', evaluate=True)
9
>>> sympify('4/2+7', evaluate=False)
4/2 + 7
>>> sympify(4/2+7, evaluate=False)
9.00000000000000
```

Extending

To extend sympify to convert custom objects (not derived from Basic), just define a _sympy_ method to your class. You can do that even to classes that you do not own by subclassing or adding the method at runtime.

```
>>> from sympy import Matrix
>>> class MyList1(object):
...     def __iter__(self):
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
...     def _sympy_(self): return Matrix(self)
>>> sympify(MyList1())
Matrix([
[1],
[2]])
```

If you do not have control over the class definition you could also use the converter global dictionary. The key is the class and the value is a function that takes a single argument and returns the desired SymPy object, e.g. converter[MyList] = lambda x: Matrix(x).

```
>>> class MyList2(object):   # XXX Do not do this if you control the␣
↪class!
...     def __iter__(self):  #     Use _sympy_!
...         yield 1
...         yield 2
...         return
...     def __getitem__(self, i): return list(self)[i]
>>> from sympy.core.sympify import converter
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([
[1],
[2]])
```

Notes

The keywords rational and convert_xor are only used when the input is a string.

Convert_xor

```
>>> sympify('x^y',convert_xor=True)
x**y
>>> sympify('x^y',convert_xor=False)
x ^ y
```

Rational

```
>>> sympify('0.1',rational=False)
0.1
>>> sympify('0.1',rational=True)
1/10
```

Sometimes autosimplification during sympification results in expressions that are very different in structure than what was entered. Until such autosimplification is no longer done, the kernS function might be of some use. In the example below you can see how an expression reduces to -1 by autosimplification, but does not do so when kernS is used.

```
>>> from sympy.core.sympify import kernS
>>> from sympy.abc import x
>>> -2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
-1
>>> s = '-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1'
>>> sympify(s)
-1
>>> kernS(s)
-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
```

assumptions

This module contains the machinery handling assumptions. Do also consider the guide *Assumptions* (page 71).

All symbolic objects have assumption attributes that can be accessed via .is_<assumption name> attribute.

Assumptions determine certain properties of symbolic objects and can have 3 possible values: True, False, None. True is returned if the object has the property and False is returned if it does not or cannot (i.e. does not make sense):

```
>>> from sympy import I
>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False
```

When the property cannot be determined (or when a method is not implemented) None will be returned. For example, a generic symbol, x, may or may not be positive so a value of None is returned for x.is_positive.

By default, all symbolic values are in the largest set in the given context without specifying the property. For example, a symbol that has a property being integer, is also real, complex, etc.

Here follows a list of possible assumption names:

commutative
: object commutes with any other object with respect to multiplication operation. See[12].

complex
: object can have only values from the set of complex numbers. See[13].

imaginary
: object value is a number that can be written as a real number multiplied by the imaginary unit `I`. See [R101]. Please note that `0` is not considered to be an imaginary number, see issue #7649.

real
: object can have only values from the set of real numbers.

extended_real
: object can have only values from the set of real numbers, `oo` and `-oo`.

integer
: object can have only values from the set of integers.

odd
even
: object can have only values from the set of odd (even) integers [R100].

prime
: object is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [R104].

composite
: object is a positive integer that has at least one positive divisor other than 1 or the number itself. See [R102].

zero
: object has the value of 0.

nonzero
: object is a real number that is not zero.

rational
: object can have only values from the set of rationals.

algebraic
: object can have only values from the set of algebraic numbers[11].

transcendental
: object can have only values from the set of transcendental numbers[10].

irrational
: object value cannot be represented exactly by *`Rational`* (page 985), see [R103].

12 https://en.wikipedia.org/wiki/Commutative_property
13 https://en.wikipedia.org/wiki/Complex_number
11 https://en.wikipedia.org/wiki/Algebraic_number
10 https://en.wikipedia.org/wiki/Transcendental_number

finite
infinite
object absolute value is bounded (arbitrarily large). See [R105], [R106], [R107].

negative
nonnegative
object can have only negative (nonnegative) values [R99].

positive
nonpositive
object can have only positive (nonpositive) values.

extended_negative
extended_nonnegative
extended_positive
extended_nonpositive
extended_nonzero
as without the extended part, but also including infinity with corresponding sign, e.g., extended_positive includes `oo`

hermitian
antihermitian
object belongs to the field of Hermitian (antihermitian) operators.

Examples

```
>>> from sympy import Symbol
>>> x = Symbol('x', real=True); x
x
>>> x.is_real
True
>>> x.is_complex
True
```

See Also

See also:

sympy.core.numbers.ImaginaryUnit (page 1000) *sympy.core.numbers.Zero* (page 995) *sympy.core.numbers.One* (page 995) *sympy.core.numbers.Infinity* (page 998) *sympy.core.numbers.NegativeInfinity* (page 998) *sympy.core.numbers.ComplexInfinity* (page 999)

Notes

The fully-resolved assumptions for any SymPy expression can be obtained as follows:

```
>>> from sympy.core.assumptions import assumptions
>>> x = Symbol('x',positive=True)
>>> assumptions(x + I)
{'commutative': True, 'complex': True, 'composite': False, 'even':
False, 'extended_negative': False, 'extended_nonnegative': False,
'extended_nonpositive': False, 'extended_nonzero': False,
'extended_positive': False, 'extended_real': False, 'finite': True,
'imaginary': False, 'infinite': False, 'integer': False, 'irrational':
False, 'negative': False, 'noninteger': False, 'nonnegative': False,
'nonpositive': False, 'nonzero': False, 'odd': False, 'positive':
False, 'prime': False, 'rational': False, 'real': False, 'zero':
False}
```

Developers Notes

The current (and possibly incomplete) values are stored in the `obj._assumptions` dictionary; queries to getter methods (with property decorators) or attributes of objects/classes will return values and update the dictionary.

```
>>> eq = x**2 + I
>>> eq._assumptions
{}
>>> eq.is_finite
True
>>> eq._assumptions
{'finite': True, 'infinite': False}
```

For a *Symbol* (page 976), there are two locations for assumptions that may be of interest. The `assumptions0` attribute gives the full set of assumptions derived from a given set of initial assumptions. The latter assumptions are stored as `Symbol._assumptions.generator`

```
>>> Symbol('x', prime=True, even=True)._assumptions.generator
{'even': True, 'prime': True}
```

The `generator` is not necessarily canonical nor is it filtered in any way: it records the assumptions used to instantiate a Symbol and (for storage purposes) represents a more compact representation of the assumptions needed to recreate the full set in `Symbol.assumptions0`.

References

cache

cacheit

`sympy.core.cache.__cacheit(maxsize)`

caching decorator.

important: the result of cached function must be *immutable*

Examples

```
>>> from sympy import cacheit
>>> @cacheit
... def f(a, b):
...     return a+b
```

```
>>> @cacheit
... def f(a, b): # noqa: F811
...     return [a, b] # <-- WRONG, returns mutable object
```

to force cacheit to check returned results mutability and consistency, set environment variable SYMPY_USE_CACHE to 'debug'

basic

Basic

class `sympy.core.basic.Basic(*args)`

Base class for all SymPy objects.

Notes And Conventions

1) Always use `.args`, when accessing parameters of some instance:

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

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

2) Never use internal methods or variables (the ones prefixed with _):

```
>>> cot(x)._args    # do not use this, use cot(x).args instead
(x,)
```

3) By "SymPy object" we mean something that can be returned by `sympify`. But not all objects one encounters using SymPy are subclasses of Basic. For example, mutable objects are not:

```
>>> from sympy import Basic, Matrix, sympify
>>> A = Matrix([[1, 2], [3, 4]]).as_mutable()
>>> isinstance(A, Basic)
False
```

```
>>> B = sympify(A)
>>> isinstance(B, Basic)
True
```

property args: tuple[*sympy.core.basic.Basic* (page 927), ...]

Returns a tuple of arguments of 'self'.

Examples

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

```
>>> cot(x).args
(x,)
```

```
>>> cot(x).args[0]
x
```

```
>>> (x*y).args
(x, y)
```

```
>>> (x*y).args[1]
y
```

Notes

Never use self._args, always use self.args. Only use _args in __new__ when creating a new function. Do not override .args() from Basic (so that it is easy to change the interface in the future if needed).

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

A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.

See also:

sympy.core.expr.Expr.as_content_primitive (page 951)

as_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True. When applied to a symbol a new symbol having only the same commutativity will be returned.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x
>>> r = Symbol('r', real=True)
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
>>> r.as_dummy()
_r
```

Notes

Any object that has structurally bound variables should have a property, $bound_symbols$ that returns those symbols appearing in the object.

property assumptions0

Return object $type$ assumptions.

For example:

Symbol('x', real=True) Symbol('x', integer=True)

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'extended_negative': False,
 'extended_nonnegative': True, 'extended_nonpositive': False,
 'extended_nonzero': True, 'extended_positive': True, 'extended_real':
 True, 'finite': True, 'hermitian': True, 'imaginary': False,
 'infinite': False, 'negative': False, 'nonnegative': True,
 'nonpositive': False, 'nonzero': True, 'positive': True, 'real':
 True, 'zero': False}
```

atoms(**types*)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and cannot be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and pi. It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from sympy import I, pi, sin
>>> from sympy.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

If one or more types are given, the results will contain only those types of atoms.

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}
```

Note that I (imaginary unit) and zoo (complex infinity) are special types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}
```

Be careful to check your assumptions when using the implicit option since `S(1).is_Integer = True` but `type(S(1))` is `One`, a special type of SymPy atom, while `type(S(2))` is type `Integer` and will find all integers in an expression:

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}
```

Finally, arguments to atoms() can select more than atomic atoms: any SymPy type (loaded in core/__init__.py) can be listed as an argument and those types of "atoms" as found in scanning the arguments of the expression recursively:

```
>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}
```

```
>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

property canonical_variables

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod class_key()

Nice order of classes.

compare(*other*)

Return -1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type from the "other" then their classes are ordered according to the sorted_classes list.

Examples

```
>>> from sympy.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

count(*query*)
Count the number of matching subexpressions.

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

doit(***hints*)
Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

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

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other, symbol=None*)
Compare two expressions and handle dummy symbols.

Examples

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

```
>>> u = Dummy('u')
```

```
>>> (u**2 + 1).dummy_eq(x**2 + 1)
True
>>> (u**2 + 1) == (x**2 + 1)
False
```

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

find(*query*, *group=False*)

Find all subexpressions matching a query.

property free_symbols: set[*sympy.core.basic.Basic* (page 927)]

Return from the atoms of self those which are free symbols.

Not all free symbols are `Symbol`. Eg: IndexedBase('I')[0].free_symbols

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(*args*, ***assumptions*)

Create a new object from an iterable.

This is a convenience function that allows one to create objects from any iterable, without having to convert to a list or tuple first.

Examples

```
>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

property func

The top-level function in an expression.

The following should hold for all objects:

```
>> x == x.func(*x.args)
```

Examples

```
>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import sin
>>> from sympy.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note has is a structural algorithm with no knowledge of mathematics. Consider the following half-open interval:

```
>>> from sympy import Interval
>>> i = Interval.Lopen(0, 5); i
Interval.Lopen(0, 5)
>>> i.args
(0, 5, True, False)
>>> i.has(4)  # there is no "4" in the arguments
False
>>> i.has(0)  # there *is* a "0" in the arguments
True
```

Instead, use `contains` to determine whether a number is in the interval or not:

```
>>> i.contains(4)
True
>>> i.contains(0)
False
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

has_free(**patterns*)

return True if self has object(s) x as a free expression else False.

Examples

```
>>> from sympy import Integral, Function
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> g = Function('g')
>>> expr = Integral(f(x), (f(x), 1, g(y)))
>>> expr.free_symbols
{y}
>>> expr.has_free(g(y))
True
>>> expr.has_free(*(x, f(x)))
False
```

This works for subexpressions and types, too:

```
>>> expr.has_free(g)
True
>>> (x + y + 1).has_free(y + 1)
True
```

property is_comparable

Return True if self can be computed to a real number (or already is a real number) with precision, else False.

Examples

```
>>> from sympy import exp_polar, pi, I
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

A False result does not mean that $self$ cannot be rewritten into a form that would be comparable. For example, the difference computed below is zero but without simplification it does not evaluate to a zero with precision:

```
>>> e = 2**pi*(1 + 2**pi)
>>> dif = e - e.expand()
>>> dif.is_comparable
False
>>> dif.n(2)._prec
1
```

match(*pattern*, *old=False*)

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

```
pattern.xreplace(self.match(pattern)) == self
```

Examples

```
>>> from sympy import Wild, Sum
>>> from sympy.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

Structurally bound symbols are ignored during matching:

```
>>> Sum(x, (x, 1, 2)).match(Sum(y, (y, 1, p)))
{p_: 2}
```

But they can be identified if desired:

```
>>> Sum(x, (x, 1, 2)).match(Sum(q, (q, 1, p)))
{p_: 2, q_: x}
```

The `old` flag will give the old-style pattern matching where expressions and patterns are essentially solved to give the match. Both of the following give None unless `old=True`:

```
>>> (x - 2).match(p - x, old=True)
{p_: 2*x - 2}
>>> (2/x).match(p*x, old=True)
{p_: 2/x**2}
```

matches(*expr*, *repl_dict=None*, *old=False*)
Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

rcall(**args*)

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance, in SymPy the following will not work:

`(x+Lambda(y, 2*y))(z) == x+2*z`,

however, you can use:

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

refine(*assumption=True*)

See the refine function in sympy.assumptions

replace(*query, value, map=False, simultaneous=True, exact=None*)

Replace matching subexpressions of `self` with `value`.

If `map = True` then also return the mapping {old: new} where `old` was a sub-expression found with query and `new` is the replacement value for it. If the expression itself does not match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to False.

In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is None it will be set to True so the match will only succeed if all non-zero values are received for each Wild that appears in the match pattern. Setting this to False accepts a match of 0; while setting it True accepts all matches that have a 0 in them. See example below for cautions.

The list of possible combinations of queries and replacement values is listed below:

Examples

Initial setup

```
>>> from sympy import log, sin, cos, tan, Wild, Mul, Add
>>> from sympy.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type

obj.replace(type, newtype)

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```

1.2. type -> func

obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr

obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a, b = map(Wild, 'ab')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a)
y - 2
>>> (2*x).replace(a*x + b, b - a)
2*x
```

When set to False, the results may be non-intuitive:

```
>>> (2*x).replace(a*x + b, b - a, exact=False)
2/x
```

2.2. pattern -> func

obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func

obj.replace(filter, func)

Replace subexpression `e` with `func(e)` if `filter(e)` is True.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

When matching a single symbol, $exact$ will default to True, but this may or may not be the behavior that is desired:

Here, we want $exact = False$:

```
>>> from sympy import Function
>>> f = Function('f')
>>> e = f(1) + f(0)
>>> q = f(a), lambda a: f(a + 1)
>>> e.replace(*q, exact=False)
f(1) + f(2)
>>> e.replace(*q, exact=True)
f(0) + f(2)
```

But here, the nature of matching makes selecting the right setting tricky:

```
>>> e = x**(1 + y)
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**(1 + y)).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(-x - y + 1)
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=False)
x
>>> (x**y).replace(x**(1 + a), lambda a: x**-a, exact=True)
x**(1 - y)
```

It is probably better to use a different form of the query that describes the target expression more precisely:

```
>>> (1 + x**(1 + y)).replace(
... lambda x: x.is_Pow and x.exp.is_Add and x.exp.args[0] == 1,
... lambda x: x.base**(1 - (x.exp - 1)))
...
x**(1 - y) + 1
```

See also:

subs **(page 941)**
substitution of subexpressions as defined by the objects themselves.

xreplace **(page 943)**
exact node replacement in expr tree; also capable of using matching rules

rewrite(**args, deep=True, **hints*)

Rewrite *self* using a defined rule.

Rewriting transforms an expression to another, which is mathematically equivalent but structurally different. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

This method takes a *pattern* and a *rule* as positional arguments. *pattern* is optional parameter which defines the types of expressions that will be transformed. If it is not passed, all possible expressions will be rewritten. *rule* defines how the expression will be rewritten.

Parameters

args : *rule*, or *pattern* and *rule*.

- *pattern* is a type or an iterable of types.
- *rule* can be any object.

deep : bool, optional.

If True, subexpressions are recursively transformed. Default is True.

Examples

If *pattern* is unspecified, all possible expressions are transformed.

```
>>> from sympy import cos, sin, exp, I
>>> from sympy.abc import x
>>> expr = cos(x) + I*sin(x)
>>> expr.rewrite(exp)
exp(I*x)
```

Pattern can be a type or an iterable of types.

```
>>> expr.rewrite(sin, exp)
exp(I*x)/2 + cos(x) - exp(-I*x)/2
>>> expr.rewrite([cos,], exp)
exp(I*x)/2 + I*sin(x) + exp(-I*x)/2
```

(continues on next page)

(continued from previous page)

```
>>> expr.rewrite([cos, sin], exp)
exp(I*x)
```

Rewriting behavior can be implemented by defining _eval_rewrite() method.

```
>>> from sympy import Expr, sqrt, pi
>>> class MySin(Expr):
...     def _eval_rewrite(self, rule, args, **hints):
...         x, = args
...         if rule == cos:
...             return cos(pi/2 - x, evaluate=False)
...         if rule == sqrt:
...             return sqrt(1 - cos(x)**2)
>>> MySin(MySin(x)).rewrite(cos)
cos(-cos(-x + pi/2) + pi/2)
>>> MySin(x).rewrite(sqrt)
sqrt(1 - cos(x)**2)
```

Defining _eval_rewrite_as_[...]() method is supported for backwards compatibility reason. This may be removed in the future and using it is discouraged.

```
>>> class MySin(Expr):
...     def _eval_rewrite_as_cos(self, *args, **hints):
...         x, = args
...         return cos(pi/2 - x, evaluate=False)
>>> MySin(x).rewrite(cos)
cos(-x + pi/2)
```

simplify(***kwargs*)

See the simplify function in sympy.simplify

sort_key(*order=None*)

Return a sort key.

Examples

```
>>> from sympy import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(**args, **kwargs*)

Substitutes old for new in an expression after sympifying args.

$args$ **is either:**

- two arguments, e.g. foo.subs(old, new)

- **one iterable argument, e.g. foo.subs(iterable). The iterable may be**
  - o **an iterable container with (old, new) pairs. In this case the** replacements are processed in the order given with successive patterns possibly affecting replacements already made.
  - o **a dict or set whose key/value items correspond to old/new pairs.** In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default_sort_key. The resulting sorted list is then processed as an iterable container (see previous).

If the keyword `simultaneous` is True, the subexpressions will not be evaluated until all the substitutions have been made.

Examples

```
>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the x**2 but not the x**4, use xreplace:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to True:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by count_op length, number of arguments and by the default_sort_key to break any ties. All other iterables are left unsorted.

```
>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e
```

```
>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
>>> E = (exp(x), e)
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to evalf as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

See also:

replace **(page 937)**
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

xreplace **(page 943)**
exact node replacement in expr tree; also capable of using matching rules

sympy.core.evalf.EvalfMixin.evalf **(page 1065)**
calculates the given formula to a desired level of precision

xreplace(*rule*)

Replace occurrences of objects within the expression.

Parameters
rule : dict-like

Expresses a replacement rule

Returns
xreplace : the result of the replacement

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
x + exp(y) + 2
```

xreplace does not differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y})
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

See also:

replace **(page 937)**
replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs **(page 941)**
substitution of subexpressions as defined by the objects themselves.

Atom

class sympy.core.basic.**Atom**(**args*)

A parent class for atomic things. An atom is an expression with no subexpressions.

Examples

Symbol, Number, Rational, Integer, … But not: Add, Mul, Pow, …

core

singleton

S

class sympy.core.singleton.**SingletonRegistry**

The registry for the singleton classes (accessible as S).

Explanation

This class serves as two separate things.

The first thing it is is the SingletonRegistry. Several classes in SymPy appear so often that they are singletonized, that is, using some metaprogramming they are made so that they can only be instantiated once (see the *sympy.core.singleton.Singleton* (page 946) class for details). For instance, every time you create Integer(0), this will return the same instance, *sympy.core.numbers.Zero* (page 995). All singleton instances are attributes of the S object, so Integer(0) can also be accessed as S.Zero.

Singletonization offers two advantages: it saves memory, and it allows fast comparison. It saves memory because no matter how many times the singletonized objects appear in expressions in memory, they all point to the same single instance in memory. The fast comparison comes from the fact that you can use is to compare exact instances in Python (usually, you need to use == to compare things). is compares objects by memory address, and is very fast.

Examples

```
>>> from sympy import S, Integer
>>> a = Integer(0)
>>> a is S.Zero
True
```

For the most part, the fact that certain objects are singletonized is an implementation detail that users should not need to worry about. In SymPy library code, is comparison is often used for performance purposes The primary advantage of S for end users is the convenient access to certain instances that are otherwise difficult to type, like S.Half (instead of Rational(1, 2)).

When using is comparison, make sure the argument is sympified. For instance,

```
>>> x = 0
>>> x is S.Zero
False
```

This problem is not an issue when using ==, which is recommended for most use-cases:

```
>>> 0 == S.Zero
True
```

The second thing S is is a shortcut for *sympy.core.sympify.sympify()* (page 918). *sympy.core.sympify.sympify()* (page 918) is the function that converts Python objects such as `int(1)` into SymPy objects such as `Integer(1)`. It also converts the string form of an expression into a SymPy expression, like `sympify("x**2") -> Symbol("x")**2`. `S(1)` is the same thing as `sympify(1)` (basically, `S.__call__` has been defined to call `sympify`).

This is for convenience, since S is a single letter. It's mostly useful for defining rational numbers. Consider an expression like `x + 1/2`. If you enter this directly in Python, it will evaluate the `1/2` and give `0.5`, because both arguments are ints (see also *Two Final Notes: ^ and /* (page 13)). However, in SymPy, you usually want the quotient of two integers to give an exact rational number. The way Python's evaluation works, at least one side of an operator needs to be a SymPy object for the SymPy evaluation to take over. You could write this as `x + Rational(1, 2)`, but this is a lot more typing. A shorter version is `x + S(1)/2`. Since `S(1)` returns `Integer(1)`, the division will return a `Rational` type, since it will call `Integer.__truediv__`, which knows how to return a `Rational`.

class sympy.core.singleton.**Singleton**(**args*, ***kwargs*)

Metaclass for singleton classes.

Explanation

A singleton class has only one instance which is returned every time the class is instantiated. Additionally, this instance can be accessed through the global registry object S as `S.<class_name>`.

Examples

```
>>> from sympy import S, Basic
>>> from sympy.core.singleton import Singleton
>>> class MySingleton(Basic, metaclass=Singleton):
...     pass
>>> Basic() is Basic()
False
>>> MySingleton() is MySingleton()
True
>>> S.MySingleton is MySingleton()
True
```

Notes

Instance creation is delayed until the first time the value is accessed. (SymPy versions before 1.0 would create the instance during class creation time, which would be prone to import cycles.)

This metaclass is a subclass of ManagedProperties because that is the metaclass of many classes that need to be Singletons (Python does not allow subclasses to have a different metaclass than the superclass, except the subclass may use a subclassed metaclass).

expr

Expr

class `sympy.core.expr.`**`Expr`**(**args*)

Base class for algebraic expressions.

Explanation

Everything that requires arithmetic operations to be defined should subclass this class, instead of Basic (which should be used only for argument storage and expression manipulation, i.e. pattern matching, substitutions, etc).

If you want to override the comparisons of expressions: Should use _eval_is_ge for inequality, or _eval_is_eq, with multiple dispatch. _eval_is_ge return true if x >= y, false if x < y, and None if the two types are not comparable or the comparison is indeterminate

See also:

sympy.core.basic.Basic (page 927)

`apart`(*x=None*, ***args*)

See the apart function in sympy.polys

`args_cnc`(*cset=False*, *warn=True*, *split_1=True*)

Return [commutative factors, non-commutative factors] of self.

Explanation

self is treated as a Mul and the ordering of the factors is maintained. If `cset` is True the commutative factors will be returned in a set. If there were repeated factors (as may happen with an unevaluated Mul) then an error will be raised unless it is explicitly suppressed by setting `warn` to False.

Note: -1 is always separated from a Number unless split_1 is False.

Examples

```
>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]
```

The arg is always treated as a Mul:

```
>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_coeff_Add(*rational=False*) → tuple['Number', *sympy.core.expr.Expr* (page 947)]

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational: bool = False*) → tuple['Number', *sympy.core.expr.Expr* (page 947)]

Efficiently extract the coefficient of a product.

as_coeff_add(**deps*) → tuple[*sympy.core.expr.Expr* (page 947), tuple[*sympy.core.expr.Expr* (page 947), ...]]

Return the tuple (c, args) where self is written as an Add, a.

c should be a Rational added to any terms of the Add that are independent of deps.

args should be a tuple of all other terms of a; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is an Add or not but you want to treat self as an Add or if you want to process the individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
```

(continues on next page)

(continued from previous page)

```
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(*x*) → tuple[*sympy.core.expr.Expr* (page 947), *sympy.core.expr.Expr* (page 947)]

`c*x**e -> c,e` where x can be any symbolic expression.

as_coeff_mul(**deps*, ***kwargs*) → tuple[*sympy.core.expr.Expr* (page 947), tuple[*sympy.core.expr.Expr* (page 947), ...]]

Return the tuple (c, args) where self is written as a Mul, `m`.

c should be a Rational multiplied by any factors of the Mul that are independent of deps.

args should be a tuple of all other factors of m; args is empty if self is a Number or if self is independent of deps (when given).

This should be used when you do not know if self is a Mul or not but you want to treat self as a Mul or if you want to process the individual arguments of the tail of self as a Mul.

- if you know self is a Mul and want only the head, use self.args[0];
- if you do not want to process the arguments of the tail but need the tail then use self.as_two_terms() which gives the head and tail;
- if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

```
>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_mul()
(3, ())
>>> (3*x*y).as_coeff_mul()
(3, (x, y))
>>> (3*x*y).as_coeff_mul(x)
(3*y, (x,))
>>> (3*y).as_coeff_mul(x)
(3*y, ())
```

as_coefficient(*expr*)

Extracts symbolic coefficient at the given expression. In other words, this functions separates 'self' into the product of 'expr' and 'expr'-free coefficient. If such separation is not possible it will return None.

Examples

```
>>> from sympy import E, pi, sin, I, Poly
>>> from sympy.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0]  # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient `2*x` is desired then the `coeff` method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

See also:

coeff **(page 958)**
return sum of terms have a given 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

as_coefficients_dict(**syms*)

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0.

If symbols syms are provided, any multiplicative terms independent of them will be considered a coefficient and a regular dictionary of syms-dependent generators as keys and their corresponding coefficients as values will be returned.

Examples

```
>>> from sympy.abc import a, x, y
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
>>> (3*a*x).as_coefficients_dict(x)
{x: 3*a}
>>> (3*a*x).as_coefficients_dict(y)
{1: 3*a*x}
```

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

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need not be in canonical form and should try to preserve the underlying structure if possible (i.e. expand_mul should not be applied to self).

Examples

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

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The as_content_primitive function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their as_content_primitives are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((x*(1 + y) + 0.4*x*(3 + 3*y))**2).as_content_primitive()
(1, 4.84*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

If clear=False (default is True) then content will not be removed from an Add if it can be distributed to leave one or more terms with integer coefficients.

```
>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_expr(**gens*)

Convert a polynomial to a SymPy expression.

Examples

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

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

as_independent(**deps*, ***hint*) → tuple[*sympy.core.expr.Expr* (page 947), *sympy.core.expr.Expr* (page 947)]

A mostly naive separation of a Mul or Add into arguments that are not are dependent on deps. To obtain as complete a separation of variables as possible, use a separation method first, e.g.:

- separatevars() to change Mul, Add and Pow (including exp) into Mul
- .expand(mul=True) to change Add or Mul into Add
- .expand(log=True) to change log expr into an Add

The only non-naive thing that is done here is to respect noncommutative ordering of variables and to always return (0, 0) for $self$ of zero regardless of hints.

For nonzero $self$, the returned tuple (i, d) has the following interpretation:

- i will has no variable that appears in deps
- d will either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)
- if self is an Add then self = i + d
- if self is a Mul then self = i*d
- otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=True

Examples

– self is an Add

```
>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

– self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
```

(continues on next page)

(continued from previous page)

```
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

– self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))
```

– force self to be treated as an Add:

```
>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

– force self to be treated as a Mul:

```
>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)
```

Note how the below differs from the above in making the constant on the dep term positive.

```
>>> (y*(-3+x)).as_independent(x)
(y, x - 3)
```

– use .as_independent() for true independence testing instead
of .has(). The former considers only symbols in the free symbols while the latter considers all symbols

```
>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
```

(continues on next page)

(continued from previous page)

```
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))
```

See also:

separatevars (page 664), *expand* (page 962), *sympy.core.add.Add.as_two_terms* (page 1016), *sympy.core.mul.Mul.as_two_terms* (page 1011), *as_coeff_add* (page 948), *as_coeff_mul* (page 949)

as_leading_term(**symbols, logx=None, cdir=0*)

Returns the leading (nonzero) term of the series expansion of self.

The _eval_as_leading_term routines are used to do this, and they must always return a non-zero value.

Examples

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

as_numer_denom()

expression -> a/b -> a, b

This is just a stub that should be defined by an object's class methods to get anything else.

See also:

***normal* (page 971)**
return a/b instead of (a, b)

as_ordered_factors(*order=None*)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(*order=None, data=False*)

Transform an expression to an ordered list of terms.

Examples

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

```
>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_poly(**gens, **args*)

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

Explanation

```
>>> 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
```

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non- commutative factors since the order that they appeared will be lost in the dictionary.

See also:

as_ordered_factors **(page 955)**
An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc **(page 947)**
Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(*deep=True, **hints*)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method cannot be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```