```
>>> Rel(x, 1, ">")
x > 1
>>> Relational(x, 1, ">")
x > 1
```

```
>>> StrictGreaterThan(x, 1)
x > 1
>>> GreaterThan(x, 1)
x >= 1
>>> LessThan(x, 1)
x <= 1
>>> StrictLessThan(x, 1)
x < 1
```

Notes

There are a couple of "gotchas" to be aware of when using Python's operators.

The first is that what your write is not always what you get:

```
>>> 1 < x
x > 1
```

Due to the order that Python parses a statement, it may not immediately find two objects comparable. When `1 < x` is evaluated, Python recognizes that the number 1 is a native number and that x is *not*. Because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, `x > 1` and that is the form that gets evaluated, hence returned.

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways:

(1) "sympify" the literal before comparison

```
>>> S(1) < x
1 < x
```

(2) use one of the wrappers or less succinct methods described above

```
>>> Lt(1, x)
1 < x
>>> Relational(1, x, "<")
1 < x
```

The second gotcha involves writing equality tests between relationals when one or both sides of the test involve a literal relational:

```
>>> e = x < 1; e
x < 1
>>> e == e  # neither side is a literal
True
>>> e == x < 1  # expecting True, too
```

(continues on next page)

(continued from previous page)

```
False
>>> e != x < 1  # expecting False
x < 1
>>> x < 1 != x < 1  # expecting False or the same thing as before
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The solution for this case is to wrap literal relationals in parentheses:

```
>>> e == (x < 1)
True
>>> e != (x < 1)
False
>>> (x < 1) != (x < 1)
False
```

The third gotcha involves chained inequalities not involving == or !=. Occasionally, one may be tempted to write:

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R134], there is no way for SymPy to create a chained inequality with that syntax so one must use And:

```
>>> e = And(x < y, y < z)
>>> type( e )
And
>>> e
(x < y) & (y < z)
```

Although this can also be done with the '&' operator, it cannot be done with the 'and' operarator:

```
>>> (x < y) & (y < z)
(x < y) & (y < z)
>>> (x < y) and (y < z)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

multidimensional

vectorize

class sympy.core.multidimensional.**vectorize**(**mdargs*)

Generalizes a function taking scalars to accept multidimensional arguments.

Examples

```
>>> from sympy import vectorize, diff, sin, symbols, Function
>>> x, y, z = symbols('x y z')
>>> f, g, h = list(map(Function, 'fgh'))
```

```
>>> @vectorize(0)
... def vsin(x):
...     return sin(x)
```

```
>>> vsin([1, x, y])
[sin(1), sin(x), sin(y)]
```

```
>>> @vectorize(0, 1)
... def vdiff(f, y):
...     return diff(f, y)
```

```
>>> vdiff([f(x, y, z), g(x, y, z), h(x, y, z)], [x, y, z])
[[Derivative(f(x, y, z), x), Derivative(f(x, y, z), y), Derivative(f(x,␣
↪y, z), z)], [Derivative(g(x, y, z), x), Derivative(g(x, y, z), y),␣
↪Derivative(g(x, y, z), z)], [Derivative(h(x, y, z), x), Derivative(h(x,
↪ y, z), y), Derivative(h(x, y, z), z)]]
```

function

Lambda

class sympy.core.function.**Lambda**(*signature*, *expr*)

Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'. A function of several variables is written as Lambda((x, y, ...), expr).

Examples

A simple example:

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> f = Lambda(x, x**2)
>>> f(4)
16
```

For multivariate functions, use:

```
>>> from sympy.abc import y, z, t
>>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
>>> f2(1, 2, 3, 4)
73
```

It is also possible to unpack tuple arguments:

```
>>> f = Lambda(((x, y), z), x + y + z)
>>> f((1, 2), 3)
6
```

A handy shortcut for lots of arguments:

```
>>> p = x, y, z
>>> f = Lambda(p, x + y*z)
>>> f(*p)
x + y*z
```

property bound_symbols
The variables used in the internal representation of the function

property expr
The return value of the function

property is_identity
Return `True` if this `Lambda` is an identity function.

property signature
The expected form of the arguments to be unpacked into variables

property variables
The variables used in the internal representation of the function

WildFunction

class sympy.core.function.**WildFunction**(**args*)
A WildFunction function matches any function (with its arguments).

Examples

```
>>> from sympy import WildFunction, Function, cos
>>> from sympy.abc import x, y
>>> F = WildFunction('F')
>>> f = Function('f')
>>> F.nargs
Naturals0
>>> x.match(F)
>>> F.match(F)
{F_: F_}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set `nargs` to the desired value at instantiation:

```
>>> F = WildFunction('F', nargs=2)
>>> F.nargs
{2}
>>> f(x).match(F)
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a range of arguments, set `nargs` to a tuple containing the desired number of arguments, e.g. if `nargs = (1, 2)` then functions with 1 or 2 arguments will be matched.

```
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
{1, 2}
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
>>> f(x, y, 1).match(F)
```

Derivative

class sympy.core.function.**Derivative**(*expr*, **variables*, ***kwargs*)

Carries out differentiation of the given expression with respect to symbols.

Examples

```
>>> from sympy import Derivative, Function, symbols, Subs
>>> from sympy.abc import x, y
>>> f, g = symbols('f g', cls=Function)
```

```
>>> Derivative(x**2, x, evaluate=True)
2*x
```

Denesting of derivatives retains the ordering of variables:

```
>>> Derivative(Derivative(f(x, y), y), x)
Derivative(f(x, y), y, x)
```

Contiguously identical symbols are merged into a tuple giving the symbol and the count:

```
>>> Derivative(f(x), x, x, y, x)
Derivative(f(x), (x, 2), y, x)
```

If the derivative cannot be performed, and evaluate is True, the order of the variables of differentiation will be made canonical:

```
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Derivatives with respect to undefined functions can be calculated:

```
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
```

Such derivatives will show up when the chain rule is used to evalulate a derivative:

```
>>> f(g(x)).diff(x)
Derivative(f(g(x)), g(x))*Derivative(g(x), x)
```

Substitution is used to represent derivatives of functions with arguments that are not symbols or functions:

```
>>> f(2*x + 3).diff(x) == 2*Subs(f(y).diff(y), y, 2*x + 3)
True
```

Notes

Simplification of high-order derivatives:

Because there can be a significant amount of simplification that can be done when multiple differentiations are performed, results will be automatically simplified in a fairly conservative fashion unless the keyword `simplify` is set to False.

```
>>> from sympy import sqrt, diff, Function, symbols
>>> from sympy.abc import x, y, z
>>> f, g = symbols('f,g', cls=Function)
```

```
>>> e = sqrt((x + 1)**2 + x)
>>> diff(e, (x, 5), simplify=False).count_ops()
136
>>> diff(e, (x, 5)).count_ops()
30
```

Ordering of variables:

If evaluate is set to True and the expression cannot be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked.

Derivative wrt non-Symbols:

For the most part, one may not differentiate wrt non-symbols. For example, we do not allow differentiation wrt $x*y$ because there are multiple ways of structurally defining where x*y appears in an expression: a very strict definition would make (x*y*z).diff(x*y) == 0. Derivatives wrt defined functions (like cos(x)) are not allowed, either:

```
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't calculate derivative wrt x*y.
```

To make it easier to work with variational calculus, however, derivatives wrt AppliedUndef and Derivatives are allowed. For example, in the Euler-Lagrange method one may write F(t, u, v) where u = f(t) and v = f'(t). These variables can be written explicitly as functions of time:

```
>>> from sympy.abc import t
>>> F = Function('F')
>>> U = f(t)
>>> V = U.diff(t)
```

The derivative wrt f(t) can be obtained directly:

```
>>> direct = F(t, U, V).diff(U)
```

When differentiation wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed and the same answer is obtained:

```
>>> indirect = F(t, U, V).subs(U, x).diff(x).subs(x, U)
>>> assert direct == indirect
```

The implication of this non-symbol replacement is that all functions are treated as independent of other functions and the symbols are independent of the functions that contain them:

```
>>> x.diff(f(x))
0
>>> g(x).diff(f(x))
0
```

It also means that derivatives are assumed to depend only on the variables of differentiation, not on anything contained within the expression being differentiated:

```
>>> F = f(x)
>>> Fx = F.diff(x)
>>> Fx.diff(F)  # derivative depends on x, not F
0
>>> Fxx = Fx.diff(x)
>>> Fxx.diff(Fx)  # derivative depends on x, not Fx
0
```

The last example can be made explicit by showing the replacement of Fx in Fxx with y:

```
>>> Fxx.subs(Fx, y)
Derivative(y, x)
```

Since that in itself will evaluate to zero, differentiating wrt Fx will also be zero:

```
>>> _.doit()
0
```

Replacing undefined functions with concrete expressions

One must be careful to replace undefined functions with expressions that contain variables consistent with the function definition and the variables of differentiation or else insconsistent result will be obtained. Consider the following example:

```
>>> eq = f(x)*g(y)
>>> eq.subs(f(x), x*y).diff(x, y).doit()
y*Derivative(g(y), y) + g(y)
>>> eq.diff(x, y).subs(f(x), x*y).doit()
y*Derivative(g(y), y)
```

The results differ because $f(x)$ was replaced with an expression that involved both variables of differentiation. In the abstract case, differentiation of $f(x)$ by y is 0; in the concrete case, the presence of y made that derivative nonvanishing and produced the extra $g(y)$ term.

Defining differentiation for an object

An object must define ._eval_derivative(symbol) method that returns the differentiation result. This function only needs to consider the non-trivial case where expr contains symbol and it should call the diff() method internally (not _eval_derivative); Derivative should be the only one to call _eval_derivative.

Any class can allow derivatives to be taken with respect to itself (while indicating its scalar nature). See the docstring of Expr._diff_wrt.

See also:

_sort_variable_count (page 1045)

property _diff_wrt

An expression may be differentiated wrt a Derivative if it is in elementary form.

Examples

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

```
>>> Derivative(f(x), x)._diff_wrt
True
>>> Derivative(cos(x), x)._diff_wrt
False
>>> Derivative(x + 1, x)._diff_wrt
False
```

A Derivative might be an unevaluated form of what will not be a valid variable of differentiation if evaluated. For example,

```
>>> Derivative(f(f(x)), x).doit()
Derivative(f(x), x)*Derivative(f(f(x)), f(x))
```

Such an expression will present the same ambiguities as arise when dealing with any other product, like 2*x, so _diff_wrt is False:

```
>>> Derivative(f(f(x)), x)._diff_wrt
False
```

classmethod _sort_variable_count(*vc*)

Sort (variable, count) pairs into canonical order while retaining order of variables that do not commute during differentiation:

- symbols and functions commute with each other
- derivatives commute with each other
- a derivative does not commute with anything it contains
- any other object is not allowed to commute if it has free symbols in common with another object

Examples

```
>>> from sympy import Derivative, Function, symbols
>>> vsort = Derivative._sort_variable_count
>>> x, y, z = symbols('x y z')
>>> f, g, h = symbols('f g h', cls=Function)
```

Contiguous items are collapsed into one pair:

```
>>> vsort([(x, 1), (x, 1)])
[(x, 2)]
>>> vsort([(y, 1), (f(x), 1), (y, 1), (f(x), 1)])
[(y, 2), (f(x), 2)]
```

Ordering is canonical.

```
>>> def vsort0(*v):
...     # docstring helper to
...     # change vi -> (vi, 0), sort, and return vi vals
...     return [i[0] for i in vsort([(i, 0) for i in v])]
```

```
>>> vsort0(y, x)
[x, y]
>>> vsort0(g(y), g(x), f(y))
[f(y), g(x), g(y)]
```

Symbols are sorted as far to the left as possible but never move to the left of a derivative having the same symbol in its variables; the same applies to AppliedUndef which are always sorted after Symbols:

```
>>> dfx = f(x).diff(x)
>>> assert vsort0(dfx, y) == [y, dfx]
>>> assert vsort0(dfx, x) == [dfx, x]
```

as_finite_difference(*points=1, x0=None, wrt=None*)

Expresses a Derivative instance as a finite difference.

Parameters

points : sequence or coefficient, optional

If sequence: discrete values (length >= order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around `x0`. Default: 1 (step-size 1)

x0 : number or Symbol, optional

the value of the independent variable (`wrt`) at which the derivative is to be approximated. Default: same as `wrt`.

wrt : Symbol, optional

"with respect to" the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the derivative is ordinary. Default: `None`.

Examples

```
>>> from sympy import symbols, Function, exp, sqrt, Symbol
>>> x, h = symbols('x h')
>>> f = Function('f')
>>> f(x).diff(x).as_finite_difference()
-f(x - 1/2) + f(x + 1/2)
```

The default step size and number of points are 1 and `order + 1` respectively. We can change the step size by passing a symbol as a parameter:

```
>>> f(x).diff(x).as_finite_difference(h)
-f(-h/2 + x)/h + f(h/2 + x)/h
```

We can also specify the discretized values to be used in a sequence:

```
>>> f(x).diff(x).as_finite_difference([x, x+h, x+2*h])
-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)
```

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around `x0`, but we can get an expression estimating the derivative at an offset:

```
>>> e, sq2 = exp(1), sqrt(2)
>>> xl = [x-h, x+h, x+e*h]
>>> f(x).diff(x, 1).as_finite_difference(xl, x+h*sq2)
2*h*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))*f(E*h + x)/...
```

To approximate `Derivative` around `x0` using a non-equidistant spacing step, the algorithm supports assignment of undefined functions to `points`:

```
>>> dx = Function('dx')
>>> f(x).diff(x).as_finite_difference(points=dx(x), x0=x-h)
-f(-h + x - dx(-h + x)/2)/dx(-h + x) + f(-h + x + dx(-h + x)/2)/dx(-h␣
↪+ x)
```

Partial derivatives are also supported:

```
>>> y = Symbol('y')
>>> d2fdxdy=f(x,y).diff(x,y)
>>> d2fdxdy.as_finite_difference(wrt=x)
-Derivative(f(x - 1/2, y), y) + Derivative(f(x + 1/2, y), y)
```

We can apply `as_finite_difference` to `Derivative` instances in compound expressions using `replace`:

```
>>> (1 + 42**f(x).diff(x)).replace(lambda arg: arg.is_Derivative,
...     lambda arg: arg.as_finite_difference())
42**(-f(x - 1/2) + f(x + 1/2)) + 1
```

See also:

sympy.calculus.finite_diff.apply_finite_diff (page 240), *sympy.calculus.finite_diff.differentiate_finite* (page 241), *sympy.calculus.finite_diff.finite_diff_weights* (page 242)

doit_numerically(*z0*)

Evaluate the derivative at z numerically.

When we can represent derivatives at a point, this should be folded into the normal evalf. For now, we need a special method.

diff

sympy.core.function.**diff**(*f*, **symbols*, ***kwargs*)

Differentiate f with respect to symbols.

Explanation

This is just a wrapper to unify .diff() and the Derivative class; its interface is similar to that of integrate(). You can use the same shortcuts for multiple variables as with Derivative. For example, diff(f(x), x, x, x) and diff(f(x), x, 3) both return the third derivative of f(x).

You can pass evaluate=False to get an unevaluated Derivative class. Note that if there are 0 symbols (such as diff(f(x), x, 0), then the result will be the function (the zeroth derivative), even if evaluate=False.

Examples

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

```
>>> diff(sin(x), x)
cos(x)
>>> diff(f(x), x, x, x)
Derivative(f(x), (x, 3))
>>> diff(f(x), x, 3)
Derivative(f(x), (x, 3))
>>> diff(sin(x)*cos(y), x, 2, y, 2)
sin(x)*cos(y)
```

```
>>> type(diff(sin(x), x))
cos
>>> type(diff(sin(x), x, evaluate=False))
<class 'sympy.core.function.Derivative'>
>>> type(diff(sin(x), x, 0))
sin
>>> type(diff(sin(x), x, 0, evaluate=False))
sin
```

```
>>> diff(sin(x))
cos(x)
>>> diff(sin(x*y))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: specify differentiation variables to differentiate sin(x*y)
```

Note that `diff(sin(x))` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

See also:

Derivative (page 1042)

***idiff* (page 2201)**
computes the derivative implicitly

References

[R135]

FunctionClass

class sympy.core.function.**FunctionClass**(**args*, ***kwargs*)

Base class for function classes. FunctionClass is a subclass of type.

Use Function('<function name>' [, signature]) to create undefined function classes.

property nargs

Return a set of the allowed number of arguments for the function.

Examples

```
>>> from sympy import Function
>>> f = Function('f')
```

If the function can take any number of arguments, the set of whole numbers is returned:

```
>>> Function('f').nargs
Naturals0
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```
>>> Function('f', nargs=1).nargs
{1}
>>> Function('f', nargs=(2, 1)).nargs
{1, 2}
```

The undefined function, after application, also has the nargs attribute; the actual number of arguments is always available by checking the `args` attribute:

```
>>> f = Function('f')
>>> f(1).nargs
Naturals0
>>> len(f(1).args)
1
```

Function

class sympy.core.function.**Function**(**args*)

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

See the *Writing Custom Functions* (page 102) guide for details on how to subclass `Function` and what methods can be defined.

Examples

Undefined Functions

To create an undefined function, pass a string of the function name to `Function`.

```
>>> from sympy import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

Assumptions can be passed to `Function` the same as with a *Symbol* (page 976). Alternatively, you can use a `Symbol` with assumptions for the function name and the function will inherit the name and assumptions associated with the `Symbol`:

```
>>> f_real = Function('f', real=True)
>>> f_real(x).is_real
True
>>> f_real_inherit = Function(Symbol('f', real=True))
>>> f_real_inherit(x).is_real
True
```

Note that assumptions on a function are unrelated to the assumptions on the variables it is called on. If you want to add a relationship, subclass `Function` and define custom assumptions handler methods. See the *Assumptions* (page 111) section of the *Writing Custom Functions* (page 102) guide for more details.

Custom Function Subclasses

The *Writing Custom Functions* (page 102) guide has several *Complete Examples* (page 122) of how to subclass `Function` to create a custom function.

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

fdiff(*argindex=1*)

Returns the first derivative of the function.

classmethod is_singular(*a*)

Tests whether the argument is an essential singularity or a branch point, or the functions is non-holomorphic.

Note: Not all functions are the same

SymPy defines many functions (like `cos` and `factorial`). It also allows the user to create generic functions which act as argument holders. Such functions are created just like symbols:

```
>>> from sympy import Function, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)
```

If you want to see which functions appear in an expression you can use the atoms method:

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
{f(x), cos(x)}
```

If you just want the function you defined, not SymPy functions, the thing to search for is AppliedUndef:

```
>>> from sympy.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
{f(x)}
```

Subs

class sympy.core.function.**Subs**(*expr*, *variables*, *point*, ***assumptions*)

Represents unevaluated substitutions of an expression.

`Subs(expr, x, x0)` represents the expression resulting from substituting x with x0 in expr.

Parameters

expr : Expr

An expression.

x : tuple, variable

A variable or list of distinct variables.

x0 : tuple or list of tuples

A point or list of evaluation points corresponding to those variables.

Notes

In order to allow expressions to combine before doit is done, a representation of the Subs expression is used internally to make expressions that are superficially different compare the same:

```
>>> a, b = Subs(x, x, 0), Subs(y, y, 0)
>>> a + b
2*Subs(x, x, 0)
```

This can lead to unexpected consequences when using methods like has that are cached:

```
>>> s = Subs(x, x, 0)
>>> s.has(x), s.has(y)
(True, False)
>>> ss = s.subs(x, y)
>>> ss.has(x), ss.has(y)
(True, False)
>>> s, ss
(Subs(x, x, 0), Subs(y, y, 0))
```

Examples

```
>>> from sympy import Subs, Function, sin, cos
>>> from sympy.abc import x, y, z
>>> f = Function('f')
```

Subs are created when a particular substitution cannot be made. The x in the derivative cannot be replaced with 0 because 0 is not a valid variables of differentiation:

```
>>> f(x).diff(x).subs(x, 0)
Subs(Derivative(f(x), x), x, 0)
```

Once f is known, the derivative and evaluation at 0 can be done:

```
>>> _.subs(f, sin).doit() == sin(x).diff(x).subs(x, 0) == cos(0)
True
```

Subs can also be created directly with one or more variables:

```
>>> Subs(f(x)*sin(y) + z, (x, y), (0, 1))
Subs(z + f(x)*sin(y), (x, y), (0, 1))
>>> _.doit()
z + f(0)*sin(1)
```

property bound_symbols

The variables to be evaluated

property expr

The expression on which the substitution operates

property point

The values for which the variables are to be substituted

property variables

The variables to be evaluated

expand

sympy.core.function.**expand**(*e*, *deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)

Expand an expression using methods given as hints.

Explanation

Hints evaluated unless explicitly set to False are: `basic`, `log`, `multinomial`, `mul`, `power_base`, and `power_exp` The following hints are supported but not applied unless set to True: `complex`, `func`, and `trig`. In addition, the following meta-hints are supported by some or all of the other hints: `frac`, `numer`, `denom`, `modulus`, and `force`. `deep` is supported by all hints. Additionally, subclasses of Expr may define their own hints or meta-hints.

The `basic` hint is used for any special rewriting of an object that should be done automatically (along with the other hints like `mul`) when expand is called. This is a catch-all hint to handle any sort of expansion that may not be described by the existing hint names. To use this hint an object should override the `_eval_expand_basic` method. Objects may also define their own expand methods, which are not run by default. See the API section below.

If `deep` is set to `True` (the default), things like arguments of functions are recursively expanded. Use `deep=False` to only expand on the top level.

If the `force` hint is used, assumptions about variables will be ignored in making the expansion.

Hints

These hints are run by default

Mul

Distributes multiplication over addition:

```
>>> from sympy import cos, exp, sin
>>> from sympy.abc import x, y, z
>>> (y*(x + z)).expand(mul=True)
x*y + y*z
```

Multinomial

Expand (x + y + ...)**n where n is a positive integer.

```
>>> ((x + y + z)**2).expand(multinomial=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Power_exp

Expand addition in exponents into multiplied bases.

```
>>> exp(x + y).expand(power_exp=True)
exp(x)*exp(y)
>>> (2**(x + y)).expand(power_exp=True)
2**x*2**y
```

Power_base

Split powers of multiplied bases.

This only happens by default if assumptions allow, or if the `force` meta-hint is used:

```
>>> ((x*y)**z).expand(power_base=True)
(x*y)**z
>>> ((x*y)**z).expand(power_base=True, force=True)
x**z*y**z
>>> ((2*y)**z).expand(power_base=True)
2**z*y**z
```

Note that in some cases where this expansion always holds, SymPy performs it automatically:

```
>>> (x*y)**2
x**2*y**2
```

Log

Pull out power of an argument as a coefficient and split logs products into sums of logs.

Note that these only work if the arguments of the log function have the proper assumptions–the arguments must be positive and the exponents must be real–or else the `force` hint must be True:

```
>>> from sympy import log, symbols
>>> log(x**2*y).expand(log=True)
log(x**2*y)
>>> log(x**2*y).expand(log=True, force=True)
2*log(x) + log(y)
>>> x, y = symbols('x,y', positive=True)
>>> log(x**2*y).expand(log=True)
2*log(x) + log(y)
```

Basic

This hint is intended primarily as a way for custom subclasses to enable expansion by default.

These hints are not run by default:

Complex

Split an expression into real and imaginary parts.

```
>>> x, y = symbols('x,y')
>>> (x + y).expand(complex=True)
re(x) + re(y) + I*im(x) + I*im(y)
>>> cos(x).expand(complex=True)
-I*sin(re(x))*sinh(im(x)) + cos(re(x))*cosh(im(x))
```

Note that this is just a wrapper around `as_real_imag()`. Most objects that wish to redefine `_eval_expand_complex()` should consider redefining `as_real_imag()` instead.

Func

Expand other functions.

```
>>> from sympy import gamma
>>> gamma(x + 1).expand(func=True)
x*gamma(x)
```

Trig

Do trigonometric expansions.

```
>>> cos(x + y).expand(trig=True)
-sin(x)*sin(y) + cos(x)*cos(y)
>>> sin(2*x).expand(trig=True)
2*sin(x)*cos(x)
```

Note that the forms of `sin(n*x)` and `cos(n*x)` in terms of `sin(x)` and `cos(x)` are not unique, due to the identity $\sin^2(x) + \cos^2(x) = 1$. The current implementation uses the form obtained from Chebyshev polynomials, but this may change. See this MathWorld article for more information.

Notes

- You can shut off unwanted methods:

```
>>> (exp(x + y)*(x + y)).expand()
x*exp(x)*exp(y) + y*exp(x)*exp(y)
>>> (exp(x + y)*(x + y)).expand(power_exp=False)
x*exp(x + y) + y*exp(x + y)
>>> (exp(x + y)*(x + y)).expand(mul=False)
(x + y)*exp(x)*exp(y)
```

- Use deep=False to only expand on the top level:

```
>>> exp(x + exp(x + y)).expand()
exp(x)*exp(exp(x)*exp(y))
>>> exp(x + exp(x + y)).expand(deep=False)
exp(x)*exp(exp(x + y))
```

- Hints are applied in an arbitrary, but consistent order (in the current implementation, they are applied in alphabetical order, except multinomial comes before mul, but this may change). Because of this, some hints may prevent expansion by other hints if they are applied first. For example, `mul` may distribute multiplications and prevent `log` and `power_base` from expanding them. Also, if `mul` is applied before `multinomial`, the expression might not be fully distributed. The solution is to use the various ``expand_hint` helper functions or to use `hint=False` to this function to finely control which hints are applied. Here are some examples:

```
>>> from sympy import expand, expand_mul, expand_power_base
>>> x, y, z = symbols('x,y,z', positive=True)

>>> expand(log(x*(y + z)))
log(x) + log(y + z)
```

Here, we see that `log` was applied before `mul`. To get the mul expanded form, either of the following will work:

```
>>> expand_mul(log(x*(y + z)))
log(x*y + x*z)
```

(continues on next page)

(continued from previous page)

```
>>> expand(log(x*(y + z)), log=False)
log(x*y + x*z)
```

A similar thing can happen with the `power_base` hint:

```
>>> expand((x*(y + z))**x)
(x*y + x*z)**x
```

To get the `power_base` expanded form, either of the following will work:

```
>>> expand((x*(y + z))**x, mul=False)
x**x*(y + z)**x
>>> expand_power_base((x*(y + z))**x)
x**x*(y + z)**x

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

The parts of a rational expression can be targeted:

```
>>> expand((x + y)*y/x/(x + 1), frac=True)
(x*y + y**2)/(x**2 + x)
>>> expand((x + y)*y/x/(x + 1), numer=True)
(x*y + y**2)/(x*(x + 1))
>>> expand((x + y)*y/x/(x + 1), denom=True)
y*(x + y)/(x**2 + x)
```

- The `modulus` meta-hint can be used to reduce the coefficients of an expression post-expansion:

```
>>> expand((3*x + 1)**2)
9*x**2 + 6*x + 1
>>> expand((3*x + 1)**2, modulus=5)
4*x**2 + x + 1
```

- Either `expand()` the function or `.expand()` the method can be used. Both are equivalent:

```
>>> expand((x + 1)**2)
x**2 + 2*x + 1
>>> ((x + 1)**2).expand()
x**2 + 2*x + 1
```

Api

Objects can define their own expand hints by defining `_eval_expand_hint()`. The function should take the form:

```
def _eval_expand_hint(self, **hints):
    # Only apply the method to the top-level expression
    ...
```

See also the example below. Objects should define `_eval_expand_hint()` methods only if `hint` applies to that specific object. The generic `_eval_expand_hint()` method defined in Expr will handle the no-op case.

Each hint should be responsible for expanding that hint only. Furthermore, the expansion should be applied to the top-level expression only. `expand()` takes care of the recursion that happens when `deep=True`.

You should only call `_eval_expand_hint()` methods directly if you are 100% sure that the object has the method, as otherwise you are liable to get unexpected `AttributeError``s. Note, again, that you do not need to recursively apply the hint to args of your object: this is handled automatically by ``expand()`. `_eval_expand_hint()` should generally not be used at all outside of an `_eval_expand_hint()` method. If you want to apply a specific expansion from within another method, use the public `expand()` function, method, or `expand_hint()` functions.

In order for expand to work, objects must be rebuildable by their args, i.e., `obj.func(*obj.args) == obj` must hold.

Expand methods are passed `**hints` so that expand hints may use 'metahints'–hints that control how different expand methods are applied. For example, the `force=True` hint described above that causes `expand(log=True)` to ignore assumptions is such a metahint. The `deep` meta-hint is handled exclusively by `expand()` and is not passed to `_eval_expand_hint()` methods.

Note that expansion hints should generally be methods that perform some kind of 'expansion'. For hints that simply rewrite an expression, use the .rewrite() API.

Examples

```
>>> from sympy import Expr, sympify
>>> class MyClass(Expr):
...     def __new__(cls, *args):
...         args = sympify(args)
...         return Expr.__new__(cls, *args)
...
...     def _eval_expand_double(self, *, force=False, **hints):
...         '''
...         Doubles the args of MyClass.
...
...         If there more than four args, doubling is not performed,
...         unless force=True is also used (False by default).
...         '''
...         if not force and len(self.args) > 4:
...             return self
```

(continues on next page)

(continued from previous page)

```
...         return self.func(*(self.args + self.args))
...
>>> a = MyClass(1, 2, MyClass(3, 4))
>>> a
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))
```

```
>>> b = MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True)
MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True, force=True)
MyClass(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

See also:

expand_log (page 1061), *expand_mul* (page 1061), *expand_multinomial* (page 1062), *expand_complex* (page 1062), *expand_trig* (page 1062), *expand_power_base* (page 1063), *expand_power_exp* (page 1063), *expand_func* (page 1061), *sympy.simplify.hyperexpand.hyperexpand* (page 687)

PoleError

class sympy.core.function.**PoleError**

count_ops

sympy.core.function.**count_ops**(*expr*, *visual=False*)

Return a representation (integer or expression) of the operations in expr.

Parameters

expr : Expr

If expr is an iterable, the sum of the op counts of the items will be returned.

visual : bool, optional

If `False` (default) then the sum of the coefficients of the visual expression will be returned. If `True` then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

Examples

```
>>> from sympy.abc import a, b, x, y
>>> from sympy import sin, count_ops
```

Although there is not a SUB object, minus signs are interpreted as either negations or subtractions:

```
>>> (x - y).count_ops(visual=True)
SUB
>>> (-x).count_ops(visual=True)
NEG
```

Here, there are two Adds and a Pow:

```
>>> (1 + a + b**2).count_ops(visual=True)
2*ADD + POW
```

In the following, an Add, Mul, Pow and two functions:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=True)
ADD + MUL + POW + 2*SIN
```

for a total of 5:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=False)
5
```

Note that “what you type” is not always what you get. The expression 1/x/y is translated by sympy into 1/(x*y) so it gives a DIV and MUL rather than two DIVs:

```
>>> (1/x/y).count_ops(visual=True)
DIV + MUL
```

The visual option can be used to demonstrate the difference in operations for expressions in different forms. Here, the Horner representation is compared with the expanded form of a polynomial:

```
>>> eq=x*(1 + x*(2 + x*(3 + x)))
>>> count_ops(eq.expand(), visual=True) - count_ops(eq, visual=True)
-MUL + 3*POW
```

The count_ops function also handles iterables:

```
>>> count_ops([x, sin(x), None, True, x + 2], visual=False)
2
>>> count_ops([x, sin(x), None, True, x + 2], visual=True)
ADD + SIN
>>> count_ops({x: sin(x), x + 2: y + 1}, visual=True)
2*ADD + SIN
```

expand_mul

sympy.core.function.**expand_mul**(*expr*, *deep=True*)

Wrapper around expand that only uses the mul hint. See the expand docstring for more information.

Examples

```
>>> from sympy import symbols, expand_mul, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_mul(exp(x+y)*(x+y)*log(x*y**2))
x*exp(x + y)*log(x*y**2) + y*exp(x + y)*log(x*y**2)
```

expand_log

sympy.core.function.**expand_log**(*expr*, *deep=True*, *force=False*, *factor=False*)

Wrapper around expand that only uses the log hint. See the expand docstring for more information.

Examples

```
>>> from sympy import symbols, expand_log, exp, log
>>> x, y = symbols('x,y', positive=True)
>>> expand_log(exp(x+y)*(x+y)*log(x*y**2))
(x + y)*(log(x) + 2*log(y))*exp(x + y)
```

expand_func

sympy.core.function.**expand_func**(*expr*, *deep=True*)

Wrapper around expand that only uses the func hint. See the expand docstring for more information.

Examples

```
>>> from sympy import expand_func, gamma
>>> from sympy.abc import x
>>> expand_func(gamma(x + 2))
x*(x + 1)*gamma(x)
```

expand_trig

sympy.core.function.**expand_trig**(*expr*, *deep=True*)

Wrapper around expand that only uses the trig hint. See the expand docstring for more information.

Examples

```
>>> from sympy import expand_trig, sin
>>> from sympy.abc import x, y
>>> expand_trig(sin(x+y)*(x+y))
(x + y)*(sin(x)*cos(y) + sin(y)*cos(x))
```

expand_complex

sympy.core.function.**expand_complex**(*expr*, *deep=True*)

Wrapper around expand that only uses the complex hint. See the expand docstring for more information.

Examples

```
>>> from sympy import expand_complex, exp, sqrt, I
>>> from sympy.abc import z
>>> expand_complex(exp(z))
I*exp(re(z))*sin(im(z)) + exp(re(z))*cos(im(z))
>>> expand_complex(sqrt(I))
sqrt(2)/2 + sqrt(2)*I/2
```

See also:

sympy.core.expr.Expr.as_real_imag (page 956)

expand_multinomial

sympy.core.function.**expand_multinomial**(*expr*, *deep=True*)

Wrapper around expand that only uses the multinomial hint. See the expand docstring for more information.

Examples

```
>>> from sympy import symbols, expand_multinomial, exp
>>> x, y = symbols('x y', positive=True)
>>> expand_multinomial((x + exp(x + 1))**2)
x**2 + 2*x*exp(x + 1) + exp(2*x + 2)
```

expand_power_exp

sympy.core.function.**expand_power_exp**(*expr*, *deep=True*)

Wrapper around expand that only uses the power_exp hint.

See the expand docstring for more information.

Examples

```
>>> from sympy import expand_power_exp
>>> from sympy.abc import x, y
>>> expand_power_exp(x**(y + 2))
x**2*x**y
```

expand_power_base

sympy.core.function.**expand_power_base**(*expr*, *deep=True*, *force=False*)

Wrapper around expand that only uses the power_base hint.

A wrapper to expand(power_base=True) which separates a power with a base that is a Mul into a product of powers, without performing any other expansions, provided that assumptions about the power's base and exponent allow.

deep=False (default is True) will only apply to the top-level expression.

force=True (default is False) will cause the expansion to ignore assumptions about the base and exponent. When False, the expansion will only happen if the base is non-negative or the exponent is an integer.

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

```
>>> (x*y)**2
x**2*y**2
```

```
>>> (2*x)**y
(2*x)**y
>>> expand_power_base(_)
2**y*x**y
```

```
>>> expand_power_base((x*y)**z)
(x*y)**z
>>> expand_power_base((x*y)**z, force=True)
x**z*y**z
>>> expand_power_base(sin((x*y)**z), deep=False)
sin((x*y)**z)
>>> expand_power_base(sin((x*y)**z), force=True)
sin(x**z*y**z)
```

```
>>> expand_power_base((2*sin(x))**y + (2*cos(x))**y)
2**y*sin(x)**y + 2**y*cos(x)**y
```

```
>>> expand_power_base((2*exp(y))**x)
2**x*exp(y)**x
```

```
>>> expand_power_base((2*cos(x))**y)
2**y*cos(x)**y
```

Notice that sums are left untouched. If this is not the desired behavior, apply full `expand()` to the expression:

```
>>> expand_power_base(((x+y)*z)**2)
z**2*(x + y)**2
>>> (((x+y)*z)**2).expand()
x**2*z**2 + 2*x*y*z**2 + y**2*z**2
```

```
>>> expand_power_base((2*y)**(1+z))
2**(z + 1)*y**(z + 1)
>>> ((2*y)**(1+z)).expand()
2*2**z*y*y**z
```

See also:

expand (page 1053)

nfloat

`sympy.core.function.`**`nfloat`**(*expr, n=15, exponent=False, dkeys=False*)

Make all Rationals in expr Floats except those in exponents (unless the exponents flag is set to True) and those in undefined functions. When processing dictionaries, do not modify the keys unless `dkeys=True`.

Examples

```
>>> from sympy import nfloat, cos, pi, sqrt
>>> from sympy.abc import x, y
>>> nfloat(x**4 + x/2 + cos(pi/3) + 1 + sqrt(y))
x**4 + 0.5*x + sqrt(y) + 1.5
>>> nfloat(x**4 + sqrt(y), exponent=True)
x**4.0 + y**0.5
```

Container types are not modified:

```
>>> type(nfloat((1, 2))) is tuple
True
```

evalf

EvalfMixin

class sympy.core.evalf.**EvalfMixin**

Mixin class adding evalf capability.

evalf(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs : dict, optional

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn : int, optional

Allow a maximum temporary working precision of maxn digits.

chop : bool or number, optional

Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
0.00010000000000000000
>>> N(x, chop=1e-5)
0.00010000000000000000
>>> N(x, chop=1e-4)
0
```

strict : bool, optional

Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad : str, optional

Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose : bool, optional

Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

n(*n=15*, *subs=None*, *maxn=100*, *chop=False*, *strict=False*, *quad=None*, *verbose=False*)

Evaluate the given formula to an accuracy of *n* digits.

Parameters

subs : dict, optional

Substitute numerical values for symbols, e.g. `subs={x:3, y:1+pi}`. The substitutions must be given as a dictionary.

maxn : int, optional

Allow a maximum temporary working precision of maxn digits.

chop : bool or number, optional

Specifies how to replace tiny real or imaginary parts in subresults by exact zeros.

When `True` the chop value defaults to standard precision.

Otherwise the chop value is used to determine the magnitude of "small" for purposes of chopping.

```
>>> from sympy import N
>>> x = 1e-4
>>> N(x, chop=True)
```

(continues on next page)

(continued from previous page)

```
0.00010000000000000000
>>> N(x, chop=1e-5)
0.00010000000000000000
>>> N(x, chop=1e-4)
0
```

strict : bool, optional

Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available maxprec.

quad : str, optional

Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

verbose : bool, optional

Print debug information.

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

PrecisionExhausted

class sympy.core.evalf.**PrecisionExhausted**

N

sympy.core.evalf.**N**(*x*, *n=15*, ***options*)

Calls x.evalf(n, **options).

Explanations

Both .n() and N() are equivalent to .evalf(); use the one that you like better. See also the docstring of .evalf() for information on the options.

Examples

```
>>> from sympy import Sum, oo, N
>>> from sympy.abc import k
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

containers

Tuple

class sympy.core.containers.**Tuple**(**args*, ***kwargs*)

Wrapper around the builtin tuple object.

Parameters

sympify : bool

If `False`, `sympify` is not called on `args`. This can be used for speedups for very large tuples where the elements are known to already be SymPy objects.

Explanation

The Tuple is a subclass of Basic, so that it works well in the SymPy framework. The wrapped tuple is available as self.args, but you can also access elements or slices with [:] syntax.

Examples

```
>>> from sympy import Tuple, symbols
>>> a, b, c, d = symbols('a b c d')
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)
```

index(*value*, *start=None*, *stop=None*)

Searches and returns the first index of the value.

property kind

The kind of a Tuple instance.

The kind of a Tuple is always of *TupleKind* (page 1069) but parametrised by the number of elements and the kind of each element.

Examples

```
>>> from sympy import Tuple, Matrix
>>> Tuple(1, 2).kind
TupleKind(NumberKind, NumberKind)
>>> Tuple(Matrix([1, 2]), 1).kind
TupleKind(MatrixKind(NumberKind), NumberKind)
>>> Tuple(1, 2).kind.element_kind
(NumberKind, NumberKind)
```

See also:

sympy.matrices.common.MatrixKind (page 1360), *sympy.core.kind.NumberKind* (page 1074)

tuple_count(*value*)

T.count(value) -> integer – return number of occurrences of value

TupleKind

class sympy.core.containers.**TupleKind**(**args*)

TupleKind is a subclass of Kind, which is used to define Kind of `Tuple`.

Parameters of TupleKind will be kinds of all the arguments in Tuples, for example

Parameters

args : tuple(element_kind)

element_kind is kind of element. args is tuple of kinds of element

Examples

```
>>> from sympy import Tuple
>>> Tuple(1, 2).kind
TupleKind(NumberKind, NumberKind)
>>> Tuple(1, 2).kind.element_kind
(NumberKind, NumberKind)
```

See also:

sympy.core.kind.NumberKind (page 1074), *MatrixKind* (page 1360), *sympy.sets.sets.SetKind* (page 1217)

Dict

class sympy.core.containers.**Dict**(**args*)

Wrapper around the builtin dict object

Explanation

The Dict is a subclass of Basic, so that it works well in the SymPy framework. Because it is immutable, it may be included in sets, but its values must all be given at instantiation and cannot be changed afterwards. Otherwise it behaves identically to the Python dict.

Examples

```
>>> from sympy import Dict, Symbol
```

```
>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print('%s %s' % (key, D[key]))
1 one
```

The args are sympified so the 1 and 2 are Integers and the values are Symbols. Queries automatically sympify args so the following work:

```
>>> 1 in D
True
>>> D.has(Symbol('one')) # searches keys and values
True
>>> 'one' in D # not in the keys
False
>>> D[1]
one
```

get(*key*, *default=None*)

Returns the value for key if the key is in the dictionary.

items()
Returns a set-like object providing a view on dict's items.

keys()
Returns the list of the dict's keys.

values()
Returns the list of the dict's values.

exprtools

gcd_terms

sympy.core.exprtools.**gcd_terms**(*terms*, *isprimitive=False*, *clear=True*, *fraction=True*)
Compute the GCD of `terms` and put them together.

Parameters
terms : Expr
Can be an expression or a non-Basic sequence of expressions which will be handled as though they are terms from a sum.

isprimitive : bool, optional
If `isprimitive` is True the _gcd_terms will not run the primitive method on the terms.

clear : bool, optional
It controls the removal of integers from the denominator of an Add expression. When True (default), all numerical denominator will be cleared; when False the denominators will be cleared only if all terms had numerical denominators other than 1.

fraction : bool, optional
When True (default), will put the expression over a common denominator.

Examples

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

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

(continues on next page)

(continued from previous page)

```
>>> gcd_terms(x/2 + 1/x, fraction=False)
(x + 2/x)/2
>>> gcd_terms(x/2 + 1/x, fraction=False, clear=False)
x/2 + 1/x
```

```
>>> gcd_terms(x/2/y + 1/x/y)
(x**2 + 2)/(2*x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False)
(x**2/2 + 1)/(x*y)
>>> gcd_terms(x/2/y + 1/x/y, clear=False, fraction=False)
(x/2 + 1/x)/y
```

The `clear` flag was ignored in this case because the returned expression was a rational expression, not a simple sum.

See also:

factor_terms (page 1072), *sympy.polys.polytools.terms_gcd* (page 2366)

factor_terms

sympy.core.exprtools.**factor_terms**(*expr*, *radical=False*, *clear=False*, *fraction=False*, *sign=True*)

Remove common factors from terms in all arguments without changing the underlying structure of the expr. No expansion or simplification (and no processing of non-commutatives) is performed.

Parameters

radical: bool, optional

If radical=True then a radical common to all terms will be factored out of any Add sub-expressions of the expr.

clear : bool, optional

If clear=False (default) then coefficients will not be separated from a single Add if they can be distributed to leave one or more terms with integer coefficients.

fraction : bool, optional

If fraction=True (default is False) then a common denominator will be constructed for the expression.

sign : bool, optional

If sign=True (default) then even if the only factor in common is a -1, it will be factored out of the expression.

Examples

```
>>> from sympy import factor_terms, Symbol
>>> from sympy.abc import x, y
>>> factor_terms(x + x*(2 + 4*y)**3)
x*(8*(2*y + 1)**3 + 1)
>>> A = Symbol('A', commutative=False)
>>> factor_terms(x*A + x*A + x*y*A)
x*(y*A + 2*A)
```

When `clear` is False, a rational will only be factored out of an Add expression if all terms of the Add have coefficients that are fractions:

```
>>> factor_terms(x/2 + 1, clear=False)
x/2 + 1
>>> factor_terms(x/2 + 1, clear=True)
(x + 2)/2
```

If a -1 is all that can be factored out, to *not* factor it out, the flag `sign` must be False:

```
>>> factor_terms(-x - y)
-(x + y)
>>> factor_terms(-x - y, sign=False)
-x - y
>>> factor_terms(-2*x - 2*y, sign=False)
-2*(x + y)
```

See also:

gcd_terms (page 1071), *sympy.polys.polytools.terms_gcd* (page 2366)

kind

Kind

class sympy.core.kind.**Kind**(**args*)

Base class for kinds.

Kind of the object represents the mathematical classification that the entity falls into. It is expected that functions and classes recognize and filter the argument by its kind.

Kind of every object must be carefully selected so that it shows the intention of design. Expressions may have different kind according to the kind of its arguements. For example, arguements of `Add` must have common kind since addition is group operator, and the resulting `Add()` has the same kind.

For the performance, each kind is as broad as possible and is not based on set theory. For example, `NumberKind` includes not only complex number but expression containing `S.Infinity` or `S.NaN` which are not strictly number.

Kind may have arguments as parameter. For example, `MatrixKind()` may be constructed with one element which represents the kind of its elements.

`Kind` behaves in singleton-like fashion. Same signature will return the same object.

NumberKind

sympy.core.kind.**NumberKind**
alias of NumberKind

UndefinedKind

sympy.core.kind.**UndefinedKind**
alias of UndefinedKind

BooleanKind

sympy.core.kind.**BooleanKind**
alias of BooleanKind

Sorting

default_sort_key

sympy.core.sorting.**default_sort_key**(*item*, *order=None*)

Return a key that can be used for sorting.

The key has the structure:

(class_key, (len(args), args), exponent.sort_key(), coefficient)

This key is supplied by the sort_key routine of Basic objects when `item` is a Basic object or an object (other than a string) that sympifies to a Basic object. Otherwise, this function produces the key.

The `order` argument is passed along to the sort_key routine and is used to determine how the terms *within* an expression are ordered. (See examples below) `order` options are: 'lex', 'grlex', 'grevlex', and reversed values of the same (e.g. 'rev-lex'). The default order value is None (which translates to 'lex').

Examples

```
>>> from sympy import S, I, default_sort_key, sin, cos, sqrt
>>> from sympy.core.function import UndefinedFunction
>>> from sympy.abc import x
```

The following are equivalent ways of getting the key for an object:

```
>>> x.sort_key() == default_sort_key(x)
True
```

Here are some examples of the key that is produced:

```
>>> default_sort_key(UndefinedFunction('f'))
((0, 0, 'UndefinedFunction'), (1, ('f',)), ((1, 0, 'Number'),
    (0, ()), (), 1), 1)
>>> default_sort_key('1')
((0, 0, 'str'), (1, ('1',)), ((1, 0, 'Number'), (0, ()), (), 1), 1)
>>> default_sort_key(S.One)
((1, 0, 'Number'), (0, ()), (), 1)
>>> default_sort_key(2)
((1, 0, 'Number'), (0, ()), (), 2)
```

While sort_key is a method only defined for SymPy objects, default_sort_key will accept anything as an argument so it is more robust as a sorting key. For the following, using key= lambda i: i.sort_key() would fail because 2 does not have a sort_key method; that's why default_sort_key is used. Note, that it also handles sympification of non-string items likes ints:

```
>>> a = [2, I, -I]
>>> sorted(a, key=default_sort_key)
[2, -I, I]
```

The returned key can be used anywhere that a key can be specified for a function, e.g. sort, min, max, etc...:

```
>>> a.sort(key=default_sort_key); a[0]
2
>>> min(a, key=default_sort_key)
2
```


Note

The key returned is useful for getting items into a canonical order that will be the same across platforms. It is not directly useful for sorting lists of expressions:

```
>>> a, b = x, 1/x
```

Since `a` has only 1 term, its value of sort_key is unaffected by `order`:

```
>>> a.sort_key() == a.sort_key('rev-lex')
True
```

If `a` and `b` are combined then the key will differ because there are terms that can be ordered:

```
>>> eq = a + b
>>> eq.sort_key() == eq.sort_key('rev-lex')
False
>>> eq.as_ordered_terms()
[x, 1/x]
>>> eq.as_ordered_terms('rev-lex')
[1/x, x]
```

But since the keys for each of these terms are independent of `order`'s value, they do not sort differently when they appear separately in a list:

```
>>> sorted(eq.args, key=default_sort_key)
[1/x, x]
>>> sorted(eq.args, key=lambda i: default_sort_key(i, order='rev-lex'))
[1/x, x]
```

The order of terms obtained when using these keys is the order that would be obtained if those terms were *factors* in a product.

Although it is useful for quickly putting expressions in canonical order, it does not sort expressions based on their complexity defined by the number of operations, power of variables and others:

```
>>> sorted([sin(x)*cos(x), sin(x)], key=default_sort_key)
[sin(x)*cos(x), sin(x)]
>>> sorted([x, x**2, sqrt(x), x**3], key=default_sort_key)
[sqrt(x), x, x**2, x**3]
```

See also:

ordered (page 1076), *sympy.core.expr.Expr.as_ordered_factors* (page 955), *sympy.core.expr.Expr.as_ordered_terms* (page 955)

ordered

sympy.core.sorting.**ordered**(*seq*, *keys=None*, *default=True*, *warn=False*)

Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.

Two default keys will be applied if 1) keys are not provided or 2) the given keys do not resolve all ties (but only if `default` is True). The two keys are `_nodes` (which places smaller expressions before large) and `default_sort_key` which (if the `sort_key` for an object is defined properly) should resolve any ties.

If `warn` is True then an error will be raised if there were no keys remaining to break ties. This can be used if it was expected that there should be no ties between items that are not identical.

Examples

```
>>> from sympy import ordered, count_ops
>>> from sympy.abc import x, y
```

The count_ops is not sufficient to break ties in this list and the first two items appear in their original order (i.e. the sorting is stable):

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3],
...    count_ops, default=False, warn=False))
...
[y + 2, x + 2, x**2 + y + 3]
```

The default_sort_key allows the tie to be broken: