(continued from previous page)

```
...         if isint is True:
...             return Integer(1)
...         elif isint is False:
...             return Integer(0)
...         else:
...             return divides(m, n)
```

(Note that this uses the convention that $k \mid 0$ for all k so that we do not need to check if m or n are nonzero. If we used a different convention we would need to check if m.is_zero and n.is_zero before performing the simplification.)

```
>>> n, m, k = symbols('n m k', integer=True)
>>> divides(k, k*n)
divides(k, k*n)
>>> divides(k, k*n).doit()
1
```

Another common way to implement doit() is for it to always return another expression. This effectively treats the function as an “unevaluated” form of another expression.

For example, let’s define a function for fused multiply-add: $\mathrm{FMA}(x, y, z) = xy + z$. It may be useful to express this function as a distinct function, e.g., for the purposes of code generation, but it would also be useful in some contexts to “evaluate” FMA(x, y, z) to x*y + z so that it can properly simplify with other expressions.

```
>>> from sympy import Number
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...     """
...     @classmethod
...     def eval(cls, x, y, z):
...         # Number is the base class of Integer, Rational, and Float
...         if all(isinstance(i, Number) for i in [x, y, z]):
...            return x*y + z
...
...     def doit(self, deep=True, **hints):
...         x, y, z = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             x = x.doit(deep=deep, **hints)
...             y = y.doit(deep=deep, **hints)
...             z = z.doit(deep=deep, **hints)
...         return x*y + z
```

```
>>> x, y, z = symbols('x y z')
>>> FMA(x, y, z)
FMA(x, y, z)
>>> FMA(x, y, z).doit()
x*y + z
```

Most custom functions will not want to define doit() in this way. However, this can provide a happy medium between having a function that always evaluates and a function that never evaluates, producing a function that doesn't evaluate by default but can be evaluated on demand (see the *discussion above* (page 107)).

expand()

The *expand()* (page 1053) function "expands" an expression in various ways. It is actually a wrapper around several sub-expansion hints. Each function corresponds to a hint to the expand() function/method. A specific expand *hint* can be defined in a custom function by defining _eval_expand_hint(self, **hints). See the documentation of *expand()* (page 1053) for details on which hints are defined and the documentation for each specific expand_*hint*() function (e.g., *expand_trig()* (page 1062)) for details on what each hint is designed to do.

The **hints keyword arguments are additional hints that may be passed to the expand function to specify additional behavior (these are separate from the predefined *hints* described in the previous paragraph). Unknown hints should be ignored as they may apply to other functions' custom expand() methods. A common hint to define is force, where force=True would force an expansion that might not be mathematically valid for all the given input assumptions. For example, expand_log(log(x*y), force=True) produces log(x) + log(y) even though this identity is not true for all complex x and y (typically force=False is the default).

Note that expand() automatically takes care of recursively expanding expressions using its own deep flag, so _eval_expand_* methods should not recursively call expand on the arguments of the function.

For our *versin example* (page 105), we can define rudimentary trig expansion by defining an _eval_expand_trig method, which recursively calls expand_trig() on 1 - cos(x):

```
>>> from sympy import expand_trig
>>> y = symbols('y')
>>> class versin(Function):
...     def _eval_expand_trig(self, **hints):
...         x = self.args[0]
...         return expand_trig(1 - cos(x))
>>> versin(x + y).expand(trig=True)
sin(x)*sin(y) - cos(x)*cos(y) + 1
```

A more sophisticated implementation might attempt to rewrite the result of expand_trig(1 - cos(x)) back into versin functions. This is left as an exercise for the reader.

Differentiation

To define differentiation via *diff()* (page 1048), define a method fdiff(self, argindex). fdiff() should return the derivative of the function, without considering the chain rule, with respect to the argindex-th variable. argindex is indexed starting at 1.

That is, f(x1, ..., xi, ..., xn).fdiff(i) should return $\frac{d}{dx_i}f(x_1,\ldots,x_i,\ldots,x_n)$, where x_k are independent of one another. diff() will automatically apply the chain rule using the result of fdiff(). User code should use diff() and not call fdiff() directly.

Note: Function subclasses should define differentiation using fdiff(). Subclasses of *Expr* (page 947) that aren't Function subclasses will need to define _eval_derivative() instead.

It is not recommended to redefine `_eval_derivative()` on a `Function` subclass.

For our versin *example function* (page 105), the derivative is $\sin(x)$.

```
>>> class versin(Function):
...     def fdiff(self, argindex=1):
...         # argindex indexes the args, starting at 1
...         return sin(self.args[0])
```

```
>>> versin(x).diff(x)
sin(x)
>>> versin(x**2).diff(x)
2*x*sin(x**2)
>>> versin(x + y).diff(x)
sin(x + y)
```

As an example of a function that has multiple arguments, consider the *fused multiply-add (FMA) example* (page 117) defined above ($\operatorname{FMA}(x, y, z) = xy + z$).

We have

$$\frac{d}{dx}\operatorname{FMA}(x, y, z) = y,$$

$$\frac{d}{dy}\operatorname{FMA}(x, y, z) = x,$$

$$\frac{d}{dz}\operatorname{FMA}(x, y, z) = 1.$$

So the `fdiff()` method for FMA would look like this:

```
>>> from sympy import Number, symbols
>>> x, y, z = symbols('x y z')
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...     """
...     def fdiff(self, argindex):
...         # argindex indexes the args, starting at 1
...         x, y, z = self.args
...         if argindex == 1:
...             return y
...         elif argindex == 2:
...             return x
...         elif argindex == 3:
...             return 1
```

```
>>> FMA(x, y, z).diff(x)
y
>>> FMA(x, y, z).diff(y)
x
>>> FMA(x, y, z).diff(z)
1
>>> FMA(x**2, x + 1, y).diff(x)
x**2 + 2*x*(x + 1)
```

To leave a derivative unevaluated, raise sympy.core.function.ArgumentIndexError(self, argindex). This is the default behavior if fdiff() is not defined. Here is an example function $f(x, y)$ that is linear in the first argument and has an unevaluated derivative on the second argument.

```
>>> from sympy.core.function import ArgumentIndexError
>>> class f(Function):
...    @classmethod
...    def eval(cls, x, y):
...        pass
...
...    def fdiff(self, argindex):
...        if argindex == 1:
...           return 1
...        raise ArgumentIndexError(self, argindex)
```

```
>>> f(x, y).diff(x)
1
>>> f(x, y).diff(y)
Derivative(f(x, y), y)
```

Printing

You can define how a function prints itself with the varions *printers* (page 2134) such as the *string printer* (page 2178), *pretty printers* (page 2139), and *LaTeX printer* (page 2170), as well as code printers for various languages like *C* (page 2141) and *Fortran* (page 2149).

In most cases, you will not need to define any printing methods. The default behavior is to print functions using their name. However, in some cases we may want to define special printing for a function.

For example, for our *divides example above* (page 109), we may want the LaTeX printer to print a more mathematical expression. Let's make the LaTeX printer represent divides(m, n) as \left [m \middle | n \right], which looks like $[m|n]$ (here $[P]$ is the Iverson bracket, which is 1 if P is true and 0 if P is false).

There are two primary ways to define printing for SymPy objects. One is to define a printer on the printer class. Most classes that are part of the SymPy library should use this method, by defining the printers on the respective classes in sympy.printing. For user code, this may be preferable if you are defining a custom printer, or if you have many custom functions that you want to define printing for. See *Example of Custom Printer* (page 2136) for an example of how to define a printer in this way.

The other way is to define the printing as a method on the function class. To do this, first look up the printmethod attribute on the printer you want to define the printing for. This is the name of the method you should define for that printer. For the LaTeX printer, *LatexPrinter.printmethod* (page 2170) is '_latex'. The print method always takes one argument, printer. printer._print should be used to recursively print any other expressions, including the arguments of the function.

So to define our divides LaTeX printer, we will define the function _latex(self, printer) on the class, like this:

```
>>> from sympy import latex
>>> class divides(Function):
...     def _latex(self, printer):
...         m, n = self.args
...         _m, _n = printer._print(m), printer._print(n)
...         return r'\left [ %s \middle | %s \right ]' % (_m, _n)
```

```
>>> print(latex(divides(m, n)))
\left [ m \middle | n \right ]
```

See *Example of Custom Printing Method* (page 2137) for more details on how to define printer methods and some pitfalls to avoid. Most importantly, you should always use `printer._print()` to recursively print the arguments of the function inside of a custom printer.

Other Methods

Several other methods can be defined on custom functions to specify various behaviors.

`inverse()`

The `inverse(self, argindex=1)` method can be defined to specify the inverse of the function. This is used by *`solve()`* (page 836) and *`solveset()`* (page 866). The `argindex` argument is the argument of the function, starting at 1 (similar to the same argument name for the *`fdiff()` method* (page 118)).

`inverse()` should return a function (not an expression) for the inverse. If the inverse is a larger expression than a single function, it can return a `lambda` function.

`inverse()` should only be defined for functions that are one-to-one. In other words, `f(x).inverse()` is the left inverse of `f(x)`. Defining `inverse()` on a function that is not one-to-one may result in `solve()` not giving all possible solutions to an expression containing the function.

Our *example versine function* (page 105) is not one-to-one (because cosine is not), but its inverse arcversin is. We may define it as follows (using the same naming convention as other inverse trig functions in SymPy):

```
>>> class aversin(Function):
...     def inverse(self, argindex=1):
...         return versin
```

This makes `solve()` work on `aversin(x)`:

```
>>> from sympy import solve
>>> solve(aversin(x) - y, x)
[versin(y)]
```

as_real_imag()

The method *as_real_imag()* (page 956) method defines how to split a function into its real and imaginary parts. It is used by various SymPy functions that operate on the real and imaginary parts of an expression separately.

`as_real_imag(self, deep=True, **hints)` should return a 2-tuple containing the real part and imaginary part of the function. That is `expr.as_real_imag()` returns `(re(expr), im(expr))`, where `expr == re(expr)

- im(expr)*I, and re(expr)andim(expr)` are real.

If `deep=True`, it should recursively call `as_real_imag(deep=True, **hints)` on its arguments. As with *doit()* (page 116) and *the _eval_expand_*() methods* (page 118), `**hints` may be any hints to allow the user to specify the behavior of the method. Unknown hints should be ignored and passed through on any recursive calls in case they are meant for other `as_real_imag()` methods.

For our *versin example* (page 105), we can recursively use the `as_real_imag()` that is already defined for `1 - cos(x)`.

```
>>> class versin(Function):
...     def as_real_imag(self, deep=True, **hints):
...         return (1 - cos(self.args[0])).as_real_imag(deep=deep, **hints)
>>> versin(x).as_real_imag()
(-cos(re(x))*cosh(im(x)) + 1, sin(re(x))*sinh(im(x)))
```

Defining `as_real_imag()` also automatically makes *expand_complex()* (page 1062) work.

```
>>> versin(x).expand(complex=True)
I*sin(re(x))*sinh(im(x)) - cos(re(x))*cosh(im(x)) + 1
```

Miscellaneous _eval_* methods

There are many other functions in SymPy whose behavior can be defined on custom functions via a custom `_eval_*` method, analogous to the ones described above. See the documentation of the specific function for details on how to define each method.

3.3.3 Complete Examples

Here are complete examples for the example functions defined in this guide. See the above sections for details on each method.

Versine

The versine (versed sine) function is defined as

$$\operatorname{versin}(x) = 1 - \cos(x).$$

Versine is an example of a simple function defined for all complex numbers. The mathematical definition is simple, which makes it straightforward to define all the above methods on it (in most cases we can just reuse the existing SymPy logic defined on `1 - cos(x)`).

Definition

```
>>> from sympy import Function, cos, expand_trig, Integer, pi, sin
>>> from sympy.core.logic import fuzzy_and, fuzzy_not
>>> class versin(Function):
...     r"""
...     The versine function.
...
...     $\operatorname{versin}(x) = 1 - \cos(x) = 2\sin(x/2)^2.$
...
...     Geometrically, given a standard right triangle with angle x in the
...     unit circle, the versine of x is the positive horizontal distance from
...     the right angle of the triangle to the rightmost point on the unit
...     circle. It was historically used as a more numerically accurate way to
...     compute 1 - cos(x), but it is rarely used today.
...
...     References
...     ==========
...
...     .. [1] https://en.wikipedia.org/wiki/Versine
...     .. [2] https://blogs.scientificamerican.com/roots-of-unity/10-secret-
↪trig-functions-your-math-teachers-never-taught-you/
...     """
...     # Define evaluation on basic inputs.
...     @classmethod
...     def eval(cls, x):
...         # If x is an explicit integer multiple of pi, x/pi will cancel and
...         # be an Integer.
...         n = x/pi
...         if isinstance(n, Integer):
...             return 1 - (-1)**n
...
...     # Define numerical evaluation with evalf().
...     def _eval_evalf(self, prec):
...         return (2*sin(self.args[0]/2)**2)._eval_evalf(prec)
...
...     # Define basic assumptions.
...     def _eval_is_nonnegative(self):
...         # versin(x) is nonnegative if x is real
...         x = self.args[0]
...         if x.is_real is True:
...             return True
```

(continues on next page)

(continued from previous page)

```
...
...     def _eval_is_positive(self):
...         # versin(x) is positive iff x is real and not an even multiple of␣
↪pi
...         x = self.args[0]
...
...         # x.as_independent(pi, as_Add=False) will split x as a Mul of the
...         # form n*pi
...         coeff, pi_ = x.as_independent(pi, as_Add=False)
...         # If pi_ = pi, x = coeff*pi. Otherwise pi_ = 1 and x is not
...         # (structurally) of the form n*pi.
...         if pi_ == pi:
...             return fuzzy_and([x.is_real, fuzzy_not(coeff.is_even)])
...         elif x.is_real is False:
...             return False
...         # else: return None. We do not know for sure whether x is an even
...         # multiple of pi
...
...     # Define the behavior for various simplification and rewriting
...     # functions.
...     def _eval_rewrite(self, rule, args, **hints):
...         if rule == cos:
...             return 1 - cos(*args)
...         elif rule == sin:
...             return 2*sin(x/2)**2
...
...     def _eval_expand_trig(self, **hints):
...         x = self.args[0]
...         return expand_trig(1 - cos(x))
...
...     def as_real_imag(self, deep=True, **hints):
...         # reuse _eval_rewrite(cos) defined above
...         return self.rewrite(cos).as_real_imag(deep=deep, **hints)
...
...     # Define differentiation.
...     def fdiff(self, argindex=1):
...         return sin(self.args[0])
```

Examples

Evaluation:

```
>>> x, y = symbols('x y')
>>> versin(x)
versin(x)
>>> versin(2*pi)
0
>>> versin(1.0)
0.459697694131860
```

Assumptions:

```
>>> n = symbols('n', integer=True)
>>> versin(n).is_real
True
>>> versin((2*n + 1)*pi).is_positive
True
>>> versin(2*n*pi).is_zero
True
>>> print(versin(n*pi).is_positive)
None
>>> r = symbols('r', real=True)
>>> print(versin(r).is_positive)
None
>>> nr = symbols('nr', real=False)
>>> print(versin(nr).is_nonnegative)
None
```

Simplification:

```
>>> a, b = symbols('a b', real=True)
>>> from sympy import I
>>> versin(x).rewrite(cos)
1 - cos(x)
>>> versin(x).rewrite(sin)
2*sin(x/2)**2
>>> versin(2*x).expand(trig=True)
2 - 2*cos(x)**2
>>> versin(a + b*I).expand(complex=True)
I*sin(a)*sinh(b) - cos(a)*cosh(b) + 1
```

Differentiation:

```
>>> versin(x).diff(x)
sin(x)
```

Solving:

(a more general version of aversin would have all the above methods defined as well)

```
>>> class aversin(Function):
...     def inverse(self, argindex=1):
...         return versin
>>> from sympy import solve
>>> solve(aversin(x**2) - y, x)
[-sqrt(versin(y)), sqrt(versin(y))]
```

divides

divides is a function defined by

$$\operatorname{divides}(m, n) = \begin{cases} 1 & \text{for}\: m \mid n \\ 0 & \text{for}\: m\not\mid n \end{cases},$$

that is, `divides(m, n)` is 1 if `m` divides `n` and `0` if `m` does not divide `m`. It is only defined for integer `m` and `n`. For the sake of simplicity, we use the convention that $m \mid 0$ for all integer m.

`divides` is an example of a function that is only defined for certain input values (integers). `divides` also gives an example of defining a custom printer (`_latex()`).

Definition

```
>>> from sympy import Function, Integer
>>> from sympy.core.logic import fuzzy_not
>>> class divides(Function):
...     r"""
...     $$\operatorname{divides}(m, n) = \begin{cases} 1 & \text{for}\: m \
↪mid n \\ 0 & \text{for}\: m\not\mid n  \end{cases}.$$
...
...     That is, ``divides(m, n)`` is ``1`` if ``m`` divides ``n`` and ``0``
...     if ``m`` does not divide ``n`. It is undefined if ``m`` or ``n`` are
...     not integers. For simplicity, the convention is used that
...     ``divides(m, 0) = 1`` for all integers ``m``.
...
...     References
...     ==========
...
...     .. [1] https://en.wikipedia.org/wiki/Divisor#Definition
...     """
...     # Define evaluation on basic inputs, as well as type checking that the
...     # inputs are not nonintegral.
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
...
...     # Define basic assumptions.
...
...     # divides is always either 0 or 1.
...     is_integer = True
...     is_negative = False
...
...     # Whether divides(m, n) is 0 or 1 depends on m and n. Note that this
...     # method only makes sense because we don't automatically evaluate on
```

(continues on next page)

(continued from previous page)

```
...     # such cases, but instead simplify these cases in doit() below.
...     def _eval_is_zero(self):
...         m, n = self.args
...         if m.is_integer and n.is_integer:
...              return fuzzy_not((n/m).is_integer)
...
...     # Define doit() as further evaluation on symbolic arguments using
...     # assumptions.
...     def doit(self, deep=False, **hints):
...         m, n = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...            m, n = m.doit(deep=deep, **hints), n.doit(deep=deep, **hints)
...
...         # divides(m, n) is 1 iff n/m is an integer. Note that m and n are
...         # already assumed to be integers because of the logic in eval().
...         isint = (n/m).is_integer
...         if isint is True:
...             return Integer(1)
...         elif isint is False:
...             return Integer(0)
...         else:
...             return divides(m, n)
...
...     # Define LaTeX printing for use with the latex() function and the
...     # Jupyter notebook.
...     def _latex(self, printer):
...         m, n = self.args
...         _m, _n = printer._print(m), printer._print(n)
...         return r'\left [ %s \middle | %s \right ]' % (_m, _n)
...
```

Examples

Evaluation

```
>>> from sympy import symbols
>>> n, m, k = symbols('n m k', integer=True)
>>> divides(3, 10)
0
>>> divides(3, 12)
1
>>> divides(m, n).is_integer
True
>>> divides(k, 2*k)
divides(k, 2*k)
>>> divides(k, 2*k).is_zero
False
>>> divides(k, 2*k).doit()
1
```

Printing:

```
>>> str(divides(m, n)) # This is using the default str printer
'divides(m, n)'
>>> print(latex(divides(m, n)))
\left [ m \middle | n \right ]
```

Fused Multiply-Add (FMA)

Fused Multiply-Add (FMA) is a multiplication followed by an addition:

$$\mathrm{FMA}(x, y, z) = xy + z.$$

It is often implemented in hardware as a single floating-point operation that has better rounding and performance than the equivalent combination of multiplication and addition operations.

FMA is an example of a custom function that is defined as an unevaluated "shorthand" to another function. This is because the *doit()* (page 116) method is defined to return x*y + z, meaning the FMA function can easily be evaluated to the expression is represents, but the *eval()* (page 106) method does *not* return anything (except when x, y, and z are all explicit numeric values), meaning that it stays unevaluated by default.

Contrast this with the *versine* (page 123) example, which treats versin as a first-class function in its own regard. Even though versin(x) can be expressed in terms of other functions (1 - cos(x)) it does not evaluate on general symbolic inputs in versin.eval(), and versin.doit() is not defined at all.

FMA also represents an example of a continuous function defined on multiple vriables, which demonstrates how argindex works in the *fdiff* (page 118) example.

Finally, FMA shows an example of defining some code printers for C and C++ (using the method names from *C99CodePrinter.printmethod* (page 2141) and *CXX11CodePrinter.printmethod* (page 2144)), since that is a typical use-case for this function.

The mathematical definition of FMA is very simple and it would be easy to define every method on it, but only a handful are shown here. The *versine* (page 123) and *divides* (page 126) examples show how to define the other important methods discussed in this guide.

Note that if you want to actually use fused-multiply add for code generation, there is already a version in SymPy sympy.codegen.cfunctions.fma() which is supported by the existing code printers. The version here is only designed to serve as an example.

Definition

```
>>> from sympy import Number, symbols, Add, Mul
>>> x, y, z = symbols('x y z')
>>> class FMA(Function):
...     """
...     FMA(x, y, z) = x*y + z
...
...     FMA is often defined as a single operation in hardware for better
...     rounding and performance.
```

(continues on next page)

(continued from previous page)

```
...
...     FMA can be evaluated by using the doit() method.
...
...     References
...     ==========
...
...     .. [1] https://en.wikipedia.org/wiki/Multiply%E2%80%93accumulate_
↪operation#Fused_multiply%E2%80%93add
...     """
...     # Define automatic evaluation on explicit numbers
...     @classmethod
...     def eval(cls, x, y, z):
...         # Number is the base class of Integer, Rational, and Float
...         if all(isinstance(i, Number) for i in [x, y, z]):
...            return x*y + z
...
...     # Define numerical evaluation with evalf().
...     def _eval_evalf(self, prec):
...         return self.doit(deep=False)._eval_evalf(prec)
...
...     # Define full evaluation to Add and Mul in doit(). This effectively
...     # treats FMA(x, y, z) as just a shorthand for x*y + z that is useful
...     # to have as a separate expression in some contexts and which can be
...     # evaluated to its expanded form in other contexts.
...     def doit(self, deep=True, **hints):
...         x, y, z = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             x = x.doit(deep=deep, **hints)
...             y = y.doit(deep=deep, **hints)
...             z = z.doit(deep=deep, **hints)
...         return x*y + z
...
...     # Define FMA.rewrite(Add) and FMA.rewrite(Mul).
...     def _eval_rewrite(self, rule, args, **hints):
...         x, y, z = self.args
...         if rule in [Add, Mul]:
...             return self.doit()
...
...     # Define differentiation.
...     def fdiff(self, argindex):
...         # argindex indexes the args, starting at 1
...         x, y, z = self.args
...         if argindex == 1:
...             return y
...         elif argindex == 2:
...             return x
...         elif argindex == 3:
...             return 1
...
...     # Define code printers for ccode() and cxxcode()
```

(continues on next page)

(continued from previous page)

```
...     def _ccode(self, printer):
...         x, y, z = self.args
...         _x, _y, _z = printer._print(x), printer._print(y), printer._
↪print(z)
...         return "fma(%s, %s, %s)" % (_x, _y, _z)
...
...     def _cxxcode(self, printer):
...         x, y, z = self.args
...         _x, _y, _z = printer._print(x), printer._print(y), printer._
↪print(z)
...         return "std::fma(%s, %s, %s)" % (_x, _y, _z)
```

Examples

Evaluation:

```
>>> x, y, z = symbols('x y z')
>>> FMA(2, 3, 4)
10
>>> FMA(x, y, z)
FMA(x, y, z)
>>> FMA(x, y, z).doit()
x*y + z
>>> FMA(x, y, z).rewrite(Add)
x*y + z
>>> FMA(2, pi, 1).evalf()
7.28318530717959
```

Differentiation

```
>>> FMA(x, x, y).diff(x)
2*x
>>> FMA(x, y, x).diff(x)
y + 1
```

Code Printers

```
>>> from sympy import ccode, cxxcode
>>> ccode(FMA(x, y, z))
'fma(x, y, z)'
>>> cxxcode(FMA(x, y, z))
'std::fma(x, y, z)'
```

3.3.4 Additional Tips

- SymPy includes dozens of functions. These can serve as useful examples for how to write a custom function, especially if the function is similar to one that is already implemented. Remember that everything in this guide applies equally well to functions that are included with SymPy and user-defined functions. Indeed, this guide is designed to serve as both a developer guide for contributors to SymPy and a guide for end-users of SymPy.
- If you have many custom functions that share common logic, you can use a common base class to contain this shared logic. For an example of this, see the source code for the trigonometric functions in SymPy, which use `TrigonometricFunction`, `InverseTrigonometricFunction`, and `ReciprocalTrigonometricFunction` base classes with some shared logic.
- As with any code, it is a good idea to write extensive tests for your function. The SymPy test suite is a good resource for examples of how to write tests for such functions. All code included in SymPy itself is required to be tested. Functions included in SymPy should also always contain a docstring with references, a mathematical definition, and doctest examples.

3.4 Solve Equations

The Python package SymPy can symbolically solve equations, differential equations, linear equations, nonlinear equations, matrix problems, inequalities, Diophantine equations, and evaluate integrals. SymPy can also solve numerically.

Learn how to use SymPy computer algebra system to:

Description	Example	Solution
Solve an equation algebraically (page 132)	$x^2 = y$	$x \in \{-\sqrt{y}, \sqrt{y}\}$
Solve a system of equations algebraically (page 836)	$x^2 + y = 2, x - y = 4$	$\{(x = -3, y = -7), (x = 2, y = 2)\}$
`Solve an equation numerically` (page 848)	$\cos(x) = x$	$x \approx 0.739085133215161$
`Solve an ordinary differential equation algebraically` (page 755)	$y''(x) + 9y(x) = 0$	$y(x) = C_1 \sin(3x) + C_2 \cos(3x)$
Solve a system of linear equations algebraically (page 836)	$x + y = 2, x - y = 0$	$x = 1, y = 1$
Solve a system of nonlinear equations algebraically (page 836)	$x^2 + y^3 = 1, x^3 - y^2 = 0$	$x = 1, y = 0$
`Solve a matrix problem algebraically` (page 1312)	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$	$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
Solve an inequality algebraically (page 750)	$x^2 < 4$	$-2 < x < 2$
Solve a system of inequalities algebraically (page 750)	$x^2 < 4, x > 0$	$0 < x < 2$
Solve (find the roots of) a polynomial algebraically (page 2342)	$x^2 - x = 0$	$x \in \{0, 1\}$
Solve a Diophantine equation (find integer solutions to a polynomial equation) algebraically (page 714)	$x^2 - 4xy + 8y^2 - 3x + 7y - 5 = 0$	$\{(x = 2, y = 1), (x = 5, y = 1)\}$

Note: SymPy has a function called `solve()` (page 836) which is designed to find the roots of an equation or system of equations. SymPy `solve()` (page 836) may or may not be what you need for a particular problem, so we recommend you use the links on this page to learn how to "solve" your problem. And while a common, colloquial expression is, for example, *"solve an integral"* (page 575), in SymPy's terminology it would be *"evaluate an integral."* (page 575)

3.4.1 Solve an equation algebraically

Use SymPy to solve an equation algebraically (symbolically). For example, solving $x^2 = y$ for x yields $x \in \{-\sqrt{y}, \sqrt{y}\}$.

Alternatives to consider:

- SymPy can also *solve many other types of problems including sets of equations* (page 131).
- Some equations cannot be solved algebraically (either at all or by SymPy), so you may have to `solve your equation numerically` (page 848) instead.

There are two high-level functions to solve equations, `solve()` (page 836) and `solveset()` (page 866). Here is a simple example of each:

`solve()` (page 836)

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
```

`solveset()` (page 866)

```
>>> from sympy import solveset
>>> from sympy.abc import x, y
>>> solveset(x**2 - y, x)
{-sqrt(y), sqrt(y)}
```

Here are recommendations on when to use:

- `solve()` (page 836)
  - You want to get explicit symbolic representations of the different values a variable could take that would satisfy the equation.
  - You want to substitute those explicit solution values into other equations or expressions involving the same variable using `subs()` (page 941)
- `solveset()` (page 866)
  - You want to represent the solutions in a mathematically precise way, using *mathematical sets* (page 1185).
  - You want a representation of all the solutions, including if there are infinitely many.
  - You want a consistent input interface.
  - You want to limit the domain of the solutions to any arbitrary set.
  - You do not need to programmatically extract solutions from the solution set: solution sets cannot necessarily be interrogated programmatically.

Guidance

Include the variable to be solved for in the function call

We recommend you include the variable to be solved for as the second argument for either function. While this is optional for equations with a single symbol, it is a good practice because it ensures SymPy will solve for the desired symbol. For example, you may expect the following to solve for x, and SymPy will solve for y:

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, dict=True)
[{y: x**2}]
```

Specifying the variable to solve for ensures that SymPy solves for it:

```
>>> from sympy.abc import x, y
>>> from sympy import solve
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
```

Ensure consistent formatting from `solve()` by using `dict=True`

`solve()` (page 836) produces various output formats depending on the answer, unless you use `dict=True` to ensure the result will be formatted as a dictionary. We recommend using `dict=True`, especially if you want to extract information from the result programmatically.

Solve an equation using `solve()` or `solveset()`

You can solve an equation in several ways. The examples below demonstrate using both *`solve()`* (page 836) and *`solveset()`* (page 866) where applicable. You can choose the function best suited to your equation.

Make your equation into an expression that equals zero

Use the fact that any expression not in an `Eq` (equation) is automatically assumed to equal zero (0) by the solving functions. You can rearrange the equation $x^2 = y$ to $x^2 - y = 0$, and solve that expression. This approach is convenient if you are interactively solving an expression which already equals zero, or an equation that you do not mind rearranging to $expression = 0$.

```
>>> from sympy import solve, solveset
>>> from sympy.abc import x, y
>>> solve(x**2 - y, x, dict=True)
[{x: -sqrt(y)}, {x: sqrt(y)}]
>>> solveset(x**2 - y, x)
{-sqrt(y), sqrt(y)}
```

Put your equation into `Eq` form

Put your equation into `Eq` form, then solve the `Eq`. This approach is convenient if you are interactively solving an equation which you already have in the form of an equation, or which you think of as an equality.

```
>>> from sympy import Eq, solve, solveset
>>> from sympy.abc import x, y
>>> eqn = Eq(x**2, y)
>>> eqn
Eq(x**2, y)
>>> solutions = solve(eqn, x, dict=True)
>>> print(solutions)
[{x: -sqrt(y)}, {x: sqrt(y)}]
>>> solutions_set = solveset(eqn, x)
>>> print(solutions_set)
{-sqrt(y), sqrt(y)}
>>> for solution_set in solutions_set:
...     print(solution_set)
sqrt(y)
-sqrt(y)
```

Restrict the domain of solutions

By default, SymPy will return solutions in the complex domain, which also includes purely real and imaginary values. Here, the first two solutions are real, and the last two are imaginary:

```
>>> from sympy import Symbol, solve, solveset
>>> x = Symbol('x')
>>> solve(x**4 - 256, x, dict=True)
[{x: -4}, {x: 4}, {x: -4*I}, {x: 4*I}]
>>> solveset(x**4 - 256, x)
{-4, 4, -4*I, 4*I}
```

To restrict returned solutions to real numbers, or another domain or range, the different solving functions use different methods.

For *solve()* (page 836), place an assumption on the symbol to be solved for, x

```
>>> from sympy import Symbol, solve
>>> x = Symbol('x', real=True)
>>> solve(x**4 - 256, x, dict=True)
[{x: -4}, {x: 4}]
```

or restrict the solutions with standard Python techniques for filtering a list such as a list comprehension:

```
>>> from sympy import Or, Symbol, solve
>>> x = Symbol('x', real=True)
>>> expr = (x-4)*(x-3)*(x-2)*(x-1)
>>> solution = solve(expr, x)
>>> print(solution)
[1, 2, 3, 4]
>>> solution_outside_2_3 = [v for v in solution if (v.is_real and Or(v<2,v>
↪3))]
>>> print(solution_outside_2_3)
[1, 4]
```

For *solveset()* (page 866), limit the output domain in the function call by setting a domain

```
>>> from sympy import S, solveset
>>> from sympy.abc import x
>>> solveset(x**4 - 256, x, domain=S.Reals)
{-4, 4}
```

or by restricting returned solutions to any arbitrary set, including an interval:

```
>>> from sympy import Interval, pi, sin, solveset
>>> from sympy.abc import x
>>> solveset(sin(x), x, Interval(-pi, pi))
{0, -pi, pi}
```

and if you restrict the solutions to a domain in which there are no solutions, *solveset()* (page 866) will return the empty set, *EmptySet* (page 1185):

```
>>> from sympy import solveset, S
>>> from sympy.abc import x
```

(continues on next page)

(continued from previous page)

```
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet
```

Explicitly represent infinite sets of possible solutions using `solveset()`

`solveset()` (page 866) *can represent infinite sets of possible solutions* (page 859) and express them in standard mathematical notation, for example $\sin(x) = 0$ for $x = n * \pi$ for every integer value of n:

```
>>> from sympy import pprint, sin, solveset
>>> from sympy.abc import x
>>> solution = solveset(sin(x), x)
>>> pprint(solution)
{2*n*pi | n in Integers} U {2*n*pi + pi | n in Integers}
```

However, *`solve()`* (page 836) will return only a finite number of solutions:

```
>>> from sympy import sin, solve
>>> from sympy.calculus.util import periodicity
>>> from sympy.abc import x
>>> f = sin(x)
>>> solve(f, x)
[0, pi]
>>> periodicity(f, x)
2*pi
```

`solve()` (page 836) tries to return just enough solutions so that all (infinitely many) solutions can generated from the returned solutions by adding integer multiples of the *`periodicity()`* (page 249) of the equation, here 2π.

Use the solution result

Substitute solutions from `solve()` into an expression

You can substitute solutions from *`solve()`* (page 836) into an expression.

A common use case is finding the critical points and values for a function f. At the critical points, the *`Derivative`* (page 1042) equals zero (or is undefined). You can then obtain the function values at those critical points by substituting the critical points back into the function using *`subs()`* (page 941). You can also tell if the critical point is a maxima or minima by substituting the values into the expression for the second derivative: a negative value indicates a maximum, and a positive value indicates a minimum.

```
>>> from sympy.abc import x
>>> from sympy import solve, diff
>>> f = x**3 + x**2 - x
>>> derivative = diff(f, x)
>>> critical_points = solve(derivative, x, dict=True)
>>> print(critical_points)
[{x: -1}, {x: 1/3}]
```

(continues on next page)

(continued from previous page)

```
>>> point1, point2 = critical_points
>>> print(f.subs(point1))
1
>>> print(f.subs(point2))
-5/27
>>> curvature = diff(f, x, 2)
>>> print(curvature.subs(point1))
-4
>>> print(curvature.subs(point2))
4
```

`solveset()` solution sets cannot necessarily be interrogated programmatically

If *solveset()* (page 866) returns a finite set (class *FiniteSet* (page 1197)), you can iterate through the solutions:

```
>>> from sympy import solveset
>>> from sympy.abc import x, y
>>> solution_set = solveset(x**2 - y, x)
>>> print(solution_set)
{-sqrt(y), sqrt(y)}
>>> solution_list = list(solution_set)
>>> print(solution_list)
[sqrt(y), -sqrt(y)]
```

However, for more complex results, it may not be possible to list the solutions:

```
>>> from sympy import S, solveset, symbols
>>> x, y = symbols('x, y')
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
Intersection({-sqrt(y), sqrt(y)}, Reals)
>>> list(solution_set)
Traceback (most recent call last):
    ...
TypeError: The computation had not completed because of the undecidable set
membership is found in every candidates.
```

In this case, it is because, if y is negative, its square root would be imaginary rather than real and therefore outside the declared domain of the solution set. By declaring y to be real and positive, SymPy can determine that its square root is real, and thus resolve the intersection between the solutions and the set of real numbers:

```
>>> from sympy import S, Symbol, solveset
>>> x = Symbol('x')
>>> y = Symbol('y', real=True, positive=True)
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
{-sqrt(y), sqrt(y)}
>>> list(solution_set)
[sqrt(y), -sqrt(y)]
```

Alternatively, you can extract the sets from the solution set using *args* (page 928), then create a list from the set containing the symbolic solutions:

```
>>> from sympy import S, solveset, symbols
>>> x, y = symbols('x, y')
>>> solution_set = solveset(x**2 - y, x, domain=S.Reals)
>>> print(solution_set)
Intersection({-sqrt(y), sqrt(y)}, Reals)
>>> solution_set_args = solution_set.args
>>> print(solution_set.args)
(Reals, {-sqrt(y), sqrt(y)})
>>> list(solution_set_args[1])
[sqrt(y), -sqrt(y)]
```

Options that can speed up solve()

Include solutions making any denominator zero by using check=False

Normally, *solve()* (page 836) checks whether any solutions make any denominator zero, and automatically excludes them. If you want to include those solutions, and speed up *solve()* (page 836) (at the risk of obtaining invalid solutions), set check=False:

```
>>> from sympy import Symbol, sin, solve
>>> x = Symbol("x")
>>> solve(sin(x)/x)  # 0 is excluded
[pi]
>>> solve(sin(x)/x, check=False)  # 0 is not excluded
[0, pi]
```

Do not simplify solutions by using simplify=False

Normally, *solve()* (page 836) simplifies all but polynomials of order 3 or greater before returning them and (if check is not False) uses the general *simplify* (page 661) function on the solutions and the expression obtained when they are substituted into the function which should be zero. If you do not want the solutions simplified, and want to speed up *solve()* (page 836), use simplify=False.

```
>>> from sympy import solve
>>> from sympy.abc import x, y
>>> expr = x**2 - (y**5 - 3*y**3 + y**2 - 3)
>>> solve(expr, x, dict=True)
[{x: -sqrt(y**5 - 3*y**3 + y**2 - 3)}, {x: sqrt(y**5 - 3*y**3 + y**2 - 3)}]
>>> solve(expr, x, dict=True, simplify=False)
[{x: -sqrt((y + 1)*(y**2 - 3)*(y**2 - y + 1))}, {x: sqrt((y + 1)*(y**2 -␣
↪3)*(y**2 - y + 1))}]
```

Not all equations can be solved

Equations with no solution

Some equations have no solution, in which case SymPy may return an empty set. For example, the equation $x - 7 = x + 2$ reduces to $-7 = 2$, which has no solution because no value of x will make it true:

```
>>> from sympy import solve, Eq
>>> from sympy.abc import x
>>> eqn = Eq(x - 7, x + 2)
>>> solve(eqn, x)
[]
```

So if SymPy returns an empty list, you may want to check whether there is a mistake in the equation.

Equations which have an analytical solution, and SymPy cannot solve

It is also possible that there is an algebraic solution to your equation, and SymPy has not implemented an appropriate algorithm. If that happens, or SymPy returns an empty set or list when there is a mathematical solution (indicating a bug in SymPy), please post it on the mailing list, or open an issue on SymPy's GitHub page. Until the issue is resolved, you can *solve your equation numerically* (page 848) instead.

3.5 Citing SymPy

To cite SymPy in publications use

```
Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A,
Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP,
Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR,
Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. (2017) SymPy:
symbolic computing in Python. *PeerJ Computer Science* 3:e103
https://doi.org/10.7717/peerj-cs.103
```

A BibTeX entry for LaTeX users is

```
    @article{10.7717/peerj-cs.103,
     title = {SymPy: symbolic computing in Python},
     author = {Meurer, Aaron and Smith, Christopher P. and Paprocki, Mateusz␣
↪and \v{C}ert\'{i}k, Ond\v{r}ej and Kirpichev, Sergey B. and Rocklin,␣
↪Matthew and Kumar, AMiT and Ivanov, Sergiu and Moore, Jason K. and Singh,␣
↪Sartaj and Rathnayake, Thilina and Vig, Sean and Granger, Brian E. and␣
↪Muller, Richard P. and Bonazzi, Francesco and Gupta, Harsh and Vats, Shivam␣
↪and Johansson, Fredrik and Pedregosa, Fabian and Curry, Matthew J. and␣
↪Terrel, Andy R. and Rou\v{c}ka, \v{S}t\v{e}p\'{a}n and Saboo, Ashutosh and␣
↪Fernando, Isuru and Kulal, Sumith and Cimrman, Robert and Scopatz, Anthony},
     year = 2017,
     month = jan,
```

(continues on next page)

(continued from previous page)

```
    keywords = {Python, Computer algebra system, Symbolics},
    abstract = {
               SymPy is an open source computer algebra system written in␣
↪pure Python. It is built with a focus on extensibility and ease of use,␣
↪through both interactive and programmatic applications. These␣
↪characteristics have led SymPy to become a popular symbolic library for the␣
↪scientific Python ecosystem. This paper presents the architecture of SymPy,␣
↪a description of its features, and a discussion of select submodules. The␣
↪supplementary material provide additional examples and further outline␣
↪details of the architecture and features of SymPy.
            },
    volume = 3,
    pages = {e103},
    journal = {PeerJ Computer Science},
    issn = {2376-5992},
    url = {https://doi.org/10.7717/peerj-cs.103},
    doi = {10.7717/peerj-cs.103}
   }
```

SymPy is BSD licensed, so you are free to use it whatever you like, be it academic, commercial, creating forks or derivatives, as long as you copy the BSD statement if you redistribute it (see the LICENSE file for details). That said, although not required by the SymPy license, if it is convenient for you, please cite SymPy when using it in your work and also consider contributing all your changes back, so that we can incorporate it and all of us will benefit in the end.

The SymPy development team members are listed in the AUTHORS file on GitHub.

A list of papers citing SymPy can be found on Zotero.

CHAPTER

FOUR

EXPLANATIONS

Explanations provide in-depth discussions about select SymPy features. These topic guides talk about things like the motivation behind design decisions, technical implementation details, and opinionated recommendations.

Content

4.1 Gotchas and Pitfalls

4.1.1 Introduction

SymPy runs under the Python Programming Language, so there are some things that may behave differently than they do in other, independent computer algebra systems like Maple or Mathematica. These are some of the gotchas and pitfalls that you may encounter when using SymPy. See also the FAQ, the *introductory tutorial* (page 5), the remainder of the SymPy Docs, and the official Python Tutorial.

If you are already familiar with C or Java, you might also want to look at this 4 minute Python tutorial.

Ignore `#doctest: +SKIP` in the examples. That has to do with internal testing of the examples.

4.1.2 Equals Signs (=)

Single Equals Sign

The equals sign (=) is the assignment operator, not equality. If you want to do $x = y$, use `Eq(x, y)` for equality. Alternatively, all expressions are assumed to equal zero, so you can just subtract one side and use `x - y`.

The proper use of the equals sign is to assign expressions to variables.

For example:

```
>>> from sympy.abc import x, y
>>> a = x - y
>>> print(a)
x - y
```

Double Equals Signs

Double equals signs (==) are used to test equality. However, this tests expressions exactly, not symbolically. For example:

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

If you want to test for symbolic equality, one way is to subtract one expression from the other and run it through functions like *expand()* (page 1053), *simplify()* (page 661), and *trigsimp()* (page 679) and see if the equation reduces to 0.

```
>>> from sympy import simplify, cos, sin, expand
>>> simplify((x + 1)**2 - (x**2 + 2*x + 1))
0
>>> eq = sin(2*x) - 2*sin(x)*cos(x)
>>> simplify(eq)
0
>>> expand(eq, trig=True)
0
```

Note: See also Why does SymPy say that two equal expressions are unequal? in the FAQ.

4.1.3 Variables

Variables Assignment does not Create a Relation Between Expressions

When you use = to do assignment, remember that in Python, as in most programming languages, the variable does not change if you change the value you assigned to it. The equations you are typing use the values present at the time of creation to "fill in" values, just like regular Python definitions. They are not altered by changes made afterwards. Consider the following:

```
>>> from sympy import Symbol
>>> a = Symbol('a')  # Symbol, `a`, stored as variable "a"
>>> b = a + 1        # an expression involving `a` stored as variable "b"
>>> print(b)
a + 1
>>> a = 4            # "a" now points to literal integer 4, not Symbol('a')
>>> print(a)
4
>>> print(b)          # "b" is still pointing at the expression involving `a`
a + 1
```

Changing quantity a does not change b; you are not working with a set of simultaneous equations. It might be helpful to remember that the string that gets printed when you print a variable referring to a SymPy object is the string that was given to it when it was created; that string does not have to be the same as the variable that you assign it to.

```
>>> from sympy import var
>>> r, t, d = var('rate time short_life')
>>> d = r*t
>>> print(d)
rate*time
>>> r = 80
>>> t = 2
>>> print(d)        # We haven't changed d, only r and t
rate*time
>>> d = r*t
>>> print(d)        # Now d is using the current values of r and t
160
```

If you need variables that have dependence on each other, you can define functions. Use the `def` operator. Indent the body of the function. See the Python docs for more information on defining functions.

```
>>> c, d = var('c d')
>>> print(c)
c
>>> print(d)
d
>>> def ctimesd():
...     """
...     This function returns whatever c is times whatever d is.
...     """
...     return c*d
...
>>> ctimesd()
c*d
>>> c = 2
>>> print(c)
2
>>> ctimesd()
2*d
```

If you define a circular relationship, you will get a `RuntimeError`.

```
>>> def a():
...     return b()
...
>>> def b():
...     return a()
...
>>> a()
Traceback (most recent call last):
  File "...", line ..., in ...
    compileflags, 1) in test.globs
  File "<...>", line 1, in <module>
    a()
  File "<...>", line 2, in a
    return b()
  File "<...>", line 2, in b
```

(continues on next page)

(continued from previous page)

```
    return a()
  File "<...>", line 2, in a
    return b()
...
RuntimeError: maximum recursion depth exceeded
```

Note: See also Why doesn't changing one variable change another that depends on it? in the FAQ.

Symbols

Symbols are variables, and like all other variables, they need to be assigned before you can use them. For example:

```
>>> import sympy
>>> z**2  # z is not defined yet
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'z' is not defined
>>> sympy.var('z')  # This is the easiest way to define z as a standard symbol
z
>>> z**2
z**2
```

If you use **isympy**, it runs the following commands for you, giving you some default Symbols and Functions.

```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

You can also import common symbol names from *sympy.abc* (page 884).

```
>>> from sympy.abc import w
>>> w
w
>>> import sympy
>>> dir(sympy.abc)
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'Symbol', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
'__builtins__', '__doc__', '__file__', '__name__', '__package__', '_greek',
'_latin', 'a', 'alpha', 'b', 'beta', 'c', 'chi', 'd', 'delta', 'e',
'epsilon', 'eta', 'f', 'g', 'gamma', 'h', 'i', 'iota', 'j', 'k', 'kappa',
'l', 'm', 'mu', 'n', 'nu', 'o', 'omega', 'omicron', 'p', 'phi', 'pi',
'psi', 'q', 'r', 'rho', 's', 'sigma', 't', 'tau', 'theta', 'u', 'upsilon',
'v', 'w', 'x', 'xi', 'y', 'z', 'zeta']
```

If you want control over the assumptions of the variables, use *Symbol* (page 976) and *symbols()* (page 978). See *Keyword Arguments* (page 153) below.

Lastly, it is recommended that you not use *I* (page 1000), *E* (page 2961), *S* (page 1845), *N* (page 1068), C, *O* (page 627), or *Q* (page 191) for variable or symbol names, as those are used for the imaginary unit (i), the base of the natural logarithm (e), the *sympify()* (page 918) function (see *Symbolic Expressions* (page 146) below), numeric evaluation (*N()* (page 1068) is equivalent to *evalf()* (page 1095)), the big O order symbol (as in $O(n \log n)$), and the assumptions object that holds a list of supported ask keys (such as `Q.real`), respectively. You can use the mnemonic `OSINEQ` to remember what Symbols are defined by default in SymPy. Or better yet, always use lowercase letters for Symbol names. Python will not prevent you from overriding default SymPy names or functions, so be careful.

```
>>> cos(pi)  # cos and pi are a built-in sympy names.
-1
>>> pi = 3   # Notice that there is no warning for overriding pi.
>>> cos(pi)
cos(3)
>>> def cos(x):  # No warning for overriding built-in functions either.
...     return 5*x
...
>>> cos(pi)
15
>>> from sympy import cos  # reimport to restore normal behavior
```

To get a full list of all default names in SymPy do:

```
>>> import sympy
>>> dir(sympy)
# A big list of all default sympy names and functions follows.
# Ignore everything that starts and ends with __.
```

If you have IPython installed and use **isympy**, you can also press the TAB key to get a list of all built-in names and to autocomplete. Also, see this page for a trick for getting tab completion in the regular Python console.

Note: See also What is the best way to create symbols? in the FAQ.

Functions

A function like `f(x)` can be created by defining the Function and the variable:

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

If you assign `f(x)` to a Python variable f you will lose your ability to copy and paste that function or to create a function with a different argument: `Function('f')` is callable, but `Function('f')(x)` is not:

```
>>> f1 = Function('f1')
>>> f2 = Function('f2')('x')
>>> f1
f1
>>> f2
f2(x)
>>> f1(1)
f1(1)
>>> f2(1)
Traceback (most recent call last):
...
TypeError: 'f2' object is not callable
>>> f2.subs(x, 1)
f2(1)
```

4.1.4 Symbolic Expressions

Python numbers vs. SymPy Numbers

SymPy uses its own classes for integers, rational numbers, and floating point numbers instead of the default Python `int` and `float` types because it allows for more control. But you have to be careful. If you type an expression that just has numbers in it, it will default to a Python expression. Use the *sympify()* (page 918) function, or just *S* (page 1845), to ensure that something is a SymPy expression.

```
>>> 6.2  # Python float. Notice the floating point accuracy problems.
6.2000000000000002
>>> type(6.2)  # <type 'float'> in Python 2.x,  <class 'float'> in Py3k
<... 'float'>
>>> S(6.2)  # SymPy Float has no such problems because of arbitrary precision.
6.20000000000000
>>> type(S(6.2))
<class 'sympy.core.numbers.Float'>
```

If you include numbers in a SymPy expression, they will be sympified automatically, but there is one gotcha you should be aware of. If you do `<number>/<number>` inside of a SymPy expression, Python will evaluate the two numbers before SymPy has a chance to get to them. The solution is to *sympify()* (page 918) one of the numbers, or use *Rational* (page 985).

```
>>> x**(1/2)  # evaluates to x**0 or x**0.5
x**0.5
>>> x**(S(1)/2)  # sympyify one of the ints
sqrt(x)
>>> x**Rational(1, 2)  # use the Rational class
sqrt(x)
```

With a power of `1/2` you can also use `sqrt` shorthand:

```
>>> sqrt(x) == x**Rational(1, 2)
True
```

If the two integers are not directly separated by a division sign then you don't have to worry about this problem:

```
>>> x**(2*x/3)
x**(2*x/3)
```

Note: A common mistake is copying an expression that is printed and reusing it. If the expression has a *Rational* (page 985) (i.e., `<number>/<number>`) in it, you will not get the same result, obtaining the Python result for the division rather than a SymPy Rational.

```
>>> x = Symbol('x')
>>> print(solve(7*x -22, x))
[22/7]
>>> 22/7  # If we just copy and paste we get int 3 or a float
3.142857142857143
>>> # One solution is to just assign the expression to a variable
>>> # if we need to use it again.
>>> a = solve(7*x - 22, x)[0]
>>> a
22/7
```

The other solution is to put quotes around the expression and run it through S() (i.e., sympify it):

```
>>> S("22/7")
22/7
```

Also, if you do not use **isympy**, you could use `from __future__ import division` to prevent the `/` sign from performing integer division.

```
>>> from __future__ import division
>>> 1/2   # With division imported it evaluates to a python float
0.5
>>> 1//2  # You can still achieve integer division with //
0
```

But be careful: you will now receive floats where you might have desired a Rational:

```
>>> x**(1/2)
x**0.5
```

Rational (page 985) only works for number/number and is only meant for rational numbers. If you want a fraction with symbols or expressions in it, just use `/`. If you do number/expression or expression/number, then the number will automatically be converted into a SymPy Number. You only need to be careful with number/number.

```
>>> Rational(2, x)
Traceback (most recent call last):
...
TypeError: invalid input: x
>>> 2/x
2/x
```

Evaluating Expressions with Floats and Rationals

SymPy keeps track of the precision of `Float` objects. The default precision is 15 digits. When an expression involving a `Float` is evaluated, the result will be expressed to 15 digits of precision but those digits (depending on the numbers involved with the calculation) may not all be significant.

The first issue to keep in mind is how the `Float` is created: it is created with a value and a precision. The precision indicates how precise of a value to use when that `Float` (or an expression it appears in) is evaluated.

The values can be given as strings, integers, floats, or rationals.

- strings and integers are interpreted as exact

```
>>> Float(100)
100.000000000000
>>> Float('100', 5)
100.00
```

- to have the precision match the number of digits, the null string can be used for the precision

```
>>> Float(100, '')
100.
>>> Float('12.34')
12.3400000000000
>>> Float('12.34', '')
12.34
```

```
>>> s, r = [Float(j, 3) for j in ('0.25', Rational(1, 7))]
>>> for f in [s, r]:
...     print(f)
0.250
0.143
```

Next, notice that each of those values looks correct to 3 digits. But if we try to evaluate them to 20 digits, a difference will become apparent:

The 0.25 (with precision of 3) represents a number that has a non-repeating binary decimal; 1/7 is repeating in binary and decimal – it cannot be represented accurately too far past those first 3 digits (the correct decimal is a repeating 142857):

```
>>> s.n(20)
0.25000000000000000000
>>> r.n(20)
0.14285278320312500000
```

It is important to realize that although a Float is being displayed in decimal at arbitrary precision, it is actually stored in binary. Once the Float is created, its binary information is set at the given precision. The accuracy of that value cannot be subsequently changed; so 1/7, at a precision of 3 digits, can be padded with binary zeros, but these will not make it a more accurate value of 1/7.

If inexact, low-precision numbers are involved in a calculation with higher precision values, the evalf engine will increase the precision of the low precision values and inexact results will be obtained. This is feature of calculations with limited precision:

```
>>> Float('0.1', 10) + Float('0.1', 3)
0.2000061035
```

Although the `evalf` engine tried to maintain 10 digits of precision (since that was the highest precision represented) the 3-digit precision used limits the accuracy to about 4 digits – not all the digits you see are significant. evalf doesn't try to keep track of the number of significant digits.

That very simple expression involving the addition of two numbers with different precisions will hopefully be instructive in helping you understand why more complicated expressions (like trig expressions that may not be simplified) will not evaluate to an exact zero even though, with the right simplification, they should be zero. Consider this unsimplified trig identity, multiplied by a big number:

```
>>> big = 12345678901234567890
>>> big_trig_identity = big*cos(x)**2 + big*sin(x)**2 - big*1
>>> abs(big_trig_identity.subs(x, .1).n(2)) > 1000
True
```

When the cos and sin terms were evaluated to 15 digits of precision and multiplied by the big number, they gave a large number that was only precise to 15 digits (approximately) and when the 20 digit big number was subtracted the result was not zero.

There are three things that will help you obtain more precise numerical values for expressions:

1) Pass the desired substitutions with the call to evaluate. By doing the subs first, the `Float` values cannot be updated as necessary. By passing the desired substitutions with the call to evalf the ability to re-evaluate as necessary is gained and the results are impressively better:

```
>>> big_trig_identity.n(2, {x: 0.1})
-0.e-91
```

2) Use Rationals, not Floats. During the evaluation process, the Rational can be computed to an arbitrary precision while the Float, once created – at a default of 15 digits – cannot. Compare the value of `-1.4e+3` above with the nearly zero value obtained when replacing x with a Rational representing 1/10 – before the call to evaluate:

```
>>> big_trig_identity.subs(x, S('1/10')).n(2)
0.e-91
```

3) Try to simplify the expression. In this case, SymPy will recognize the trig identity and simplify it to zero so you don't even have to evaluate it numerically:

```
>>> big_trig_identity.simplify()
0
```

Immutability of Expressions

Expressions in SymPy are immutable, and cannot be modified by an in-place operation. This means that a function will always return an object, and the original expression will not be modified. The following example snippet demonstrates how this works:

```
def main():
    var('x y a b')
    expr = 3*x + 4*y
    print('original =', expr)
    expr_modified = expr.subs({x: a, y: b})
    print('modified =', expr_modified)

if __name__ == "__main__":
    main()
```

The output shows that the *subs()* (page 941) function has replaced variable x with variable a, and variable y with variable b:

```
original = 3*x + 4*y
modified = 3*a + 4*b
```

The *subs()* (page 941) function does not modify the original expression expr. Rather, a modified copy of the expression is returned. This returned object is stored in the variable expr_modified. Note that unlike C/C++ and other high-level languages, Python does not require you to declare a variable before it is used.

Mathematical Operators

SymPy uses the same default operators as Python. Most of these, like */+-, are standard. Aside from integer division discussed in *Python numbers vs. SymPy Numbers* (page 146) above, you should also be aware that implied multiplication is not allowed. You need to use * whenever you wish to multiply something. Also, to raise something to a power, use **, not ^ as many computer algebra systems use. Parentheses () change operator precedence as you would normally expect.

In **isympy**, with the **ipython** shell:

```
>>> 2x
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> 2*x
2*x
>>> (x + 1)^2  # This is not power.  Use ** instead.
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ^: 'Add' and 'int'
>>> (x + 1)**2
(x + 1)**2
>>> pprint(3 - x**(2*x)/(x + 1))
    2*x
   x
```

(continues on next page)

(continued from previous page)

```
- ----- + 3
  x + 1
```

Inverse Trig Functions

SymPy uses different names for some functions than most computer algebra systems. In particular, the inverse trig functions use the python names of *asin* (page 395), *acos* (page 396) and so on instead of the usual `arcsin` and `arccos`. Use the methods described in *Symbols* (page 144) above to see the names of all SymPy functions.

Sqrt is not a Function

There is no `sqrt` function in the same way that there is an exponential function (`exp`). `sqrt(x)` is used to represent `Pow(x, S(1)/2)` so if you want to know if an expression has any square roots in it, `expr.has(sqrt)` will not work. You must look for `Pow` with an exponent of one half (or negative one half if it is in a denominator, e.g.

```
>>> (y + sqrt(x)).find(Wild('w')**S.Half)
{sqrt(x)}
>>> (y + 1/sqrt(x)).find(Wild('w')**-S.Half)
{1/sqrt(x)}
```

If you are interested in any power of the `sqrt` then the following pattern would be appropriate

```
>>> sq = lambda s: s.is_Pow and s.exp.is_Rational and s.exp.q == 2
>>> (y + sqrt(x)**3).find(sq)
{x**(3/2)}
```

4.1.5 Special Symbols

The symbols `[]`, `{}`, `=`, and `()` have special meanings in Python, and thus in SymPy. See the Python docs linked to above for additional information.

Lists

Square brackets `[]` denote a list. A list is a container that holds any number of different objects. A list can contain anything, including items of different types. Lists are mutable, which means that you can change the elements of a list after it has been created. You access the items of a list also using square brackets, placing them after the list or list variable. Items are numbered using the space before the item.

Note: List indexes begin at 0.

Example:

```
>>> a = [x, 1]  # A simple list of two items
>>> a
[x, 1]
>>> a[0]  # This is the first item
x
>>> a[0] = 2  # You can change values of lists after they have been created
>>> print(a)
[2, 1]
>>> print(solve(x**2 + 2*x - 1, x)) # Some functions return lists
[-1 + sqrt(2), -sqrt(2) - 1]
```

Note: See the Python docs for more information on lists and the square bracket notation for accessing elements of a list.

Dictionaries

Curly brackets `{}` denote a dictionary, or a dict for short. A dictionary is an unordered list of non-duplicate keys and values. The syntax is `{key: value}`. You can access values of keys using square bracket notation.

```
>>> d = {'a': 1, 'b': 2}  # A dictionary.
>>> d
{'a': 1, 'b': 2}
>>> d['a']  # How to access items in a dict
1
>>> roots((x - 1)**2*(x - 2), x)  # Some functions return dicts
{1: 2, 2: 1}
>>> # Some SymPy functions return dictionaries.  For example,
>>> # roots returns a dictionary of root:multiplicity items.
>>> roots((x - 5)**2*(x + 3), x)
{-3: 1, 5: 2}
>>> # This means that the root -3 occurs once and the root 5 occurs twice.
```

Note: See the Python docs for more information on dictionaries.

Tuples

Parentheses `()`, aside from changing operator precedence and their use in function calls, (like `cos(x)`), are also used for tuples. A `tuple` is identical to a *list* (page 151), except that it is not mutable. That means that you cannot change their values after they have been created. In general, you will not need tuples in SymPy, but sometimes it can be more convenient to type parentheses instead of square brackets.

```
>>> t = (1, 2, x)  # Tuples are like lists
>>> t
(1, 2, x)
>>> t[0]
```

(continues on next page)

(continued from previous page)

```
1
>>> t[0] = 4  # Except you cannot change them after they have been␣
↪created
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Single element tuples, unlike lists, must have a comma in them:

```
>>> (x,)
(x,)
```

Without the comma, a single expression without a comma is not a tuple:

```
>>> (x)
x
```

integrate takes a sequence as the second argument if you want to integrate with limits (and a tuple or list will work):

```
>>> integrate(x**2, (x, 0, 1))
1/3
>>> integrate(x**2, [x, 0, 1])
1/3
```

Note: See the Python docs for more information on tuples.

Keyword Arguments

Aside from the usage described *above* (page 141), equals signs (=) are also used to give named arguments to functions. Any function that has `key=value` in its parameters list (see below on how to find this out), then `key` is set to `value` by default. You can change the value of the key by supplying your own value using the equals sign in the function call. Also, functions that have `**` followed by a name in the parameters list (usually `**kwargs` or `**assumptions`) allow you to add any number of `key=value` pairs that you want, and they will all be evaluated according to the function.

`sqrt(x**2)` doesn't auto simplify to x because x is assumed to be complex by default, and, for example, `sqrt((-1)**2) == sqrt(1) == 1 != -1`:

```
>>> sqrt(x**2)
sqrt(x**2)
```

Giving assumptions to Symbols is an example of using the keyword argument:

```
>>> x = Symbol('x', positive=True)
```

The square root will now simplify since it knows that `x >= 0`:

```
>>> sqrt(x**2)
x
```

powsimp has a default argument of `combine='all'`:

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m))
     m + n
(x*y)
```

Setting combine to the default value is the same as not setting it.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='all'))
     m + n
(x*y)
```

The non-default options are `'exp'`, which combines exponents…

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='exp'))
 m + n  m + n
x      *y
```

…and 'base', which combines bases.

```
>>> pprint(powsimp(x**n*x**m*y**n*y**m, combine='base'))
     m      n
(x*y) *(x*y)
```

Note: See the Python docs for more information on function parameters.

4.1.6 Getting help from within SymPy

help()

Although all docs are available at docs.sympy.org or on the SymPy Wiki, you can also get info on functions from within the Python interpreter that runs SymPy. The easiest way to do this is to do `help(function)`, or `function?` if you are using **`ipython`**:

```
In [1]: help(powsimp)  # help() works everywhere

In [2]: # But in ipython, you can also use ?, which is better because it
In [3]: # it gives you more information
In [4]: powsimp?
```

These will give you the function parameters and docstring for *`powsimp()`* (page 680). The output will look something like this:

`sympy.simplify.simplify.`**`powsimp`**(*expr, deep=False, combine='all', force=False, measure=<function count_ops>*)

reduces expression by combining powers with similar bases and exponents.

Explanation

If deep is True then powsimp() will also simplify arguments of functions. By default deep is set to False.

If force is True then bases will be combined without checking for assumptions, e.g. sqrt(x)*sqrt(y) -> sqrt(x*y) which is not true if x and y are both negative.

You can make powsimp() only combine bases or only combine exponents by changing combine='base' or combine='exp'. By default, combine='all', which does both. combine='base' will only combine:

```
 a   a          a                          2x      x
x * y  =>  (x*y)   as well as things like 2   =>  4
```

and combine='exp' will only combine

```
 a   b      (a + b)
x * x  =>  x
```

combine='exp' will strictly only combine exponents in the way that used to be automatic. Also use deep=True if you need the old behavior.

When combine='all', 'exp' is evaluated first. Consider the first example below for when there could be an ambiguity relating to this. This is done so things like the second example can be completely combined. If you want 'base' combined first, do something like powsimp(powsimp(expr, combine='base'), combine='exp').

Examples

```
>>> from sympy import powsimp, exp, log, symbols
>>> from sympy.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(exp(x)*exp(y))
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from sympy import sqrt
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

source()

Another useful option is the *source()* (page 2116) function. This will print the source code of a function, including any docstring that it may have. You can also do `function??` in **ipython**. For example, from SymPy 0.6.5:

```
>>> source(simplify)  # simplify() is actually only 2 lines of code.
In file: ./sympy/simplify/simplify.py
def simplify(expr):
    """Naively simplifies the given expression.
       ...
       Simplification is not a well defined term and the exact strategies
       this function tries can change in the future versions of SymPy. If
       your algorithm relies on "simplification" (whatever it is), try to
       determine what you need exactly  -  is it powsimp()? radsimp()?
       together()?, logcombine()?, or something else? And use this particular
       function directly, because those are well defined and thus your
↪algorithm
       will be robust.
       ...
    """
    expr = Poly.cancel(powsimp(expr))
    return powsimp(together(expr.expand()), combine='exp', deep=True)
```