



# SymPy Documentation

*Release 1.11rc1*

**SymPy Development Team**

**August 08, 2022**



# CONTENTS

<b>1</b>	<b>Installation</b>	<b>1</b>
1.1	Anaconda . . . . .	1
1.2	Git . . . . .	1
1.3	Other Methods . . . . .	2
1.4	Run SymPy . . . . .	2
1.5	mpmath . . . . .	2
1.6	Questions . . . . .	3
<b>2</b>	<b>Tutorials</b>	<b>5</b>
2.1	Introductory Tutorial . . . . .	5
<b>3</b>	<b>How-to Guides</b>	<b>71</b>
3.1	Assumptions . . . . .	71
3.2	Symbolic and fuzzy booleans . . . . .	95
3.3	Writing Custom Functions . . . . .	102
3.4	Solve Equations . . . . .	131
3.5	Citing SymPy . . . . .	139
<b>4</b>	<b>Explanations</b>	<b>141</b>
4.1	Gotchas and Pitfalls . . . . .	141
4.2	SymPy Special Topics . . . . .	157
4.3	List of active deprecations . . . . .	163
<b>5</b>	<b>API Reference</b>	<b>185</b>
5.1	Basics . . . . .	185
5.2	Code Generation . . . . .	185
5.3	Logic . . . . .	185
5.4	Matrices . . . . .	185
5.5	Number Theory . . . . .	186
5.6	Physics . . . . .	186
5.7	Utilities . . . . .	186
5.8	Topics . . . . .	186
<b>6</b>	<b>Contributing</b>	<b>2983</b>
6.1	Development Environment Setup . . . . .	2983
6.2	Dependencies . . . . .	2985
6.3	Build the Documentation . . . . .	2992
6.4	Debugging . . . . .	2996
6.5	SymPy Docstrings Style Guide . . . . .	2997
6.6	Documentation Style Guide . . . . .	3010
6.7	Making a Contribution . . . . .	3017

6.8 Deprecation Policy . . . . .	3017
<b>Bibliography</b>	<b>3027</b>
<b>Python Module Index</b>	<b>3065</b>
<b>Index</b>	<b>3069</b>



---

## CHAPTER ONE

---

# INSTALLATION

The SymPy CAS can be installed on virtually any computer with Python. SymPy does require `mpmath` Python library to be installed first. The recommended method of installation is through Anaconda, which includes `mpmath`, as well as several other useful libraries. Alternatively, some Linux distributions have SymPy packages available.

SymPy officially supports Python 3.8, 3.9, 3.10, and PyPy.

## 1.1 Anaconda

**Anaconda** is a free Python distribution from Continuum Analytics that includes SymPy, Matplotlib, IPython, NumPy, and many more useful packages for scientific computing. This is recommended because many nice features of SymPy are only enabled when certain libraries are installed. For example, without Matplotlib, only simple text-based plotting is enabled. With the IPython notebook or qtconsole, you can get nicer  $\text{\LaTeX}$  printing by running `init_printing()`.

If you already have Anaconda and want to update SymPy to the latest version, use:

```
conda update sympy
```

## 1.2 Git

If you wish to contribute to SymPy or like to get the latest updates as they come, install SymPy from git. To download the repository, execute the following from the command line:

```
git clone https://github.com/sympy/sympy.git
```

To update to the latest version, go into your repository and execute:

```
git pull origin master
```

If you want to install SymPy, but still want to use the git version, you can run from your repository:

```
python setup.py develop
```

This will cause the installed version to always point to the version in the git directory.

## 1.3 Other Methods

You may also install SymPy using pip or from source. In addition, most Linux and Python distributions have some SymPy version available to install using their package manager. Here is a list of several such Python distributions:

- [Anaconda](#)
- [Enthought Canopy](#)
- [ActivePython](#)
- [Spack](#)

## 1.4 Run SymPy

After installation, it is best to verify that your freshly-installed SymPy works. To do this, start up Python and import the SymPy libraries:

```
$ python
>>> from sympy import *
```

From here, execute some simple SymPy statements like the ones below:

```
>>> x = Symbol('x')
>>> limit(sin(x)/x, x, 0)
1
>>> integrate(1/x, x)
log(x)
```

For a starter guide on using SymPy effectively, refer to the [Introductory Tutorial](#) (page 5).

## 1.5 mpmath

Versions of SymPy prior to 1.0 included [mpmath](#), but it now depends on it as an external dependency. If you installed SymPy with Anaconda, it will already include mpmath. Use:

```
conda install mpmath
```

to ensure that it is installed.

If you do not wish to use Anaconda, you can use `pip install mpmath`.

If you use mpmath via `sympy.mpmath` in your code, you will need to change this to use just mpmath. If you depend on code that does this that you cannot easily change, you can work around it by doing:

```
import sys
import mpmath
sys.modules['sympy.mpmath'] = mpmath
```

before the code that imports `sympy.mpmath`. It is recommended to change code that uses `sympy.mpmath` to use mpmath directly wherever possible.

## 1.6 Questions

If you have a question about installation or SymPy in general, feel free to visit our chat on [Gitter](#). In addition, our [mailing list](#) is an excellent source of community support.

If you think there's a bug or you would like to request a feature, please open an [issue ticket](#).







---

**CHAPTER  
TWO**

---

**TUTORIALS**

Tutorials are the best place to start for anyone new to SymPy or one of SymPy's features.

## 2.1 Introductory Tutorial

If you are new to SymPy, start here.

### 2.1.1 Introductory Tutorial

This tutorial aims to give an introduction to SymPy for someone who has not used the library before. Many features of SymPy will be introduced in this tutorial, but they will not be exhaustive. In fact, virtually every functionality shown in this tutorial will have more options or capabilities than what will be shown. The rest of the SymPy documentation serves as API documentation, which extensively lists every feature and option of each function.

These are the goals of this tutorial:

- To give a guide, suitable for someone who has never used SymPy (but who has used Python and knows the necessary mathematics).
- To be written in a narrative format, which is both easy and fun to follow. It should read like a book.
- To give insightful examples and exercises, to help the reader learn and to make it entertaining to work through.
- To introduce concepts in a logical order.
- To use good practices and idioms, and avoid antipatterns. Functions or methodologies that tend to lead to antipatterns are avoided. Features that are only useful to advanced users are not shown.
- To be consistent. If there are multiple ways to do it, only the best way is shown.
- To avoid unnecessary duplication, it is assumed that previous sections of the tutorial have already been read.

Feedback on this tutorial, or on SymPy in general is always welcome. Just write to our [mailing list](#).

#### **Content**

## Preliminaries

This tutorial assumes that the reader already knows the basics of the Python programming language. If you do not, the [official Python tutorial](#) is excellent.

This tutorial assumes a decent mathematical background. Most examples require knowledge lower than a calculus level, and some require knowledge at a calculus level. Some of the advanced features require more than this. If you come across a section that uses some mathematical function you are not familiar with, you can probably skip over it, or replace it with a similar one that you are more familiar with. Or look up the function on Wikipedia and learn something new. Some important mathematical concepts that are not common knowledge will be introduced as necessary.

## Installation

You will need to install SymPy first. See the [installation guide](#) (page 1).

## Exercises

This tutorial was the basis for a tutorial given at the 2013 SciPy conference in Austin, TX. The website for that tutorial is [here](#). It has links to videos, materials, and IPython notebook exercises. The IPython notebook exercises in particular are recommended to anyone going through this tutorial.

## Introduction

### What is Symbolic Computation?

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

Let's take an example. Say we wanted to use the built-in Python functions to compute square roots. We might do something like this

```
>>> import math
>>> math.sqrt(9)
3.0
```

9 is a perfect square, so we got the exact answer, 3. But suppose we computed the square root of a number that isn't a perfect square

```
>>> math.sqrt(8)
2.82842712475
```

Here we got an approximate result. 2.82842712475 is not the exact square root of 8 (indeed, the actual square root of 8 cannot be represented by a finite decimal, since it is an irrational number). If all we cared about was the decimal form of the square root of 8, we would be done.

But suppose we want to go further. Recall that  $\sqrt{8} = \sqrt{4 \cdot 2} = 2\sqrt{2}$ . We would have a hard time deducing this from the above result. This is where symbolic computation comes in. With

a symbolic computation system like SymPy, square roots of numbers that are not perfect squares are left unevaluated by default

```
>>> import sympy
>>> sympy.sqrt(3)
sqrt(3)
```

Furthermore—and this is where we start to see the real power of symbolic computation—symbolic results can be symbolically simplified.

```
>>> sympy.sqrt(8)
2*sqrt(2)
```

## A More Interesting Example

The above example starts to show how we can manipulate irrational numbers exactly using SymPy. But it is much more powerful than that. Symbolic computation systems (which by the way, are also often called computer algebra systems, or just CASs) such as SymPy are capable of computing symbolic expressions with variables.

As we will see later, in SymPy, variables are defined using symbols. Unlike many symbolic manipulation systems, variables in SymPy must be defined before they are used (the reason for this will be discussed in the [next section](#) (page 10)).

Let us define a symbolic expression, representing the mathematical expression  $x + 2y$ .

```
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> expr = x + 2*y
>>> expr
x + 2*y
```

Note that we wrote  $x + 2y$  just as we would if  $x$  and  $y$  were ordinary Python variables. But in this case, instead of evaluating to something, the expression remains as just  $x + 2y$ . Now let us play around with it:

```
>>> expr + 1
x + 2*y + 1
>>> expr - x
2*y
```

Notice something in the above example. When we typed  $\text{expr} - x$ , we did not get  $x + 2y - x$ , but rather just  $2y$ . The  $x$  and the  $-x$  automatically canceled one another. This is similar to how  $\text{sqrt}(8)$  automatically turned into  $2\sqrt{2}$  above. This isn't always the case in SymPy, however:

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

Here, we might have expected  $x(x + 2y)$  to transform into  $x^2 + 2xy$ , but instead we see that the expression was left alone. This is a common theme in SymPy. Aside from obvious simplifications like  $x - x = 0$  and  $\sqrt{8} = 2\sqrt{2}$ , most simplifications are not performed automatically. This is because we might prefer the factored form  $x(x + 2y)$ , or we might prefer the expanded form  $x^2 + 2xy$ . Both forms are useful in different circumstances. In SymPy, there are functions to go from one form to the other

```
>>> from sympy import expand, factor
>>> expanded_expr = expand(x*expr)
>>> expanded_expr
x**2 + 2*x*y
>>> factor(expanded_expr)
x*(x + 2*y)
```

## The Power of Symbolic Computation

The real power of a symbolic computation system such as SymPy is the ability to do all sorts of computations symbolically. SymPy can simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, and much, much more, and do it all symbolically. It includes modules for plotting, printing (like 2D pretty printed output of math formulas, or  $\text{\LaTeX}$ ), code generation, physics, statistics, combinatorics, number theory, geometry, logic, and more. Here is a small sampling of the sort of symbolic power SymPy is capable of, to whet your appetite.

```
>>> from sympy import *
>>> x, t, z, nu = symbols('x t z nu')
```

This will make all further examples pretty print with unicode characters.

```
>>> init_printing(use_unicode=True)
```

Take the derivative of  $\sin(x)e^x$ .

```
>>> diff(sin(x)*exp(x), x)
      x      x
e  ·sin(x) + e  ·cos(x)
```

Compute  $\int (e^x \sin(x) + e^x \cos(x)) dx$ .

```
>>> integrate(exp(x)*sin(x) + exp(x)*cos(x), x)
      x
e  ·sin(x)
```

Compute  $\int_{-\infty}^{\infty} \sin(x^2) dx$ .

```
>>> integrate(sin(x**2), (x, -oo, oo))
      2
sqrt(2)*sqrt(pi)
```

Find  $\lim_{x \rightarrow 0} \frac{\sin(x)}{x}$ .

```
>>> limit(sin(x)/x, x, 0)
1
```

Solve  $x^2 - 2 = 0$ .

```
>>> solve(x**2 - 2, x)
[-sqrt(2), sqrt(2)]
```

Solve the differential equation  $y'' - y = e^t$ .

```
>>> y = Function('y')
>>> dsolve(Eq(y(t).diff(t, t) - y(t), exp(t)), y(t))
```

$$y(t) = C_2 \cdot e^{-t} + \left( C_1 + \frac{t}{2} \right) \cdot e^t$$

Find the eigenvalues of  $\begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$ .

```
>>> Matrix([[1, 2], [2, 2]]).eigenvals()
```

$$\left\{ -\frac{3}{2} - \frac{\sqrt{17}}{2}: 1, -\frac{3}{2} + \frac{\sqrt{17}}{2}: 1 \right\}$$

Rewrite the Bessel function  $J_\nu(z)$  in terms of the spherical Bessel function  $j_\nu(z)$ .

```
>>> besselj(nu, z).rewrite(jn)
```

$$\frac{\sqrt{2} \cdot \sqrt{z} \cdot j_n\left(\sqrt{z} - \frac{1}{2}, z\right)}{\sqrt{\pi}}$$

Print  $\int_0^\pi \cos^2(x) dx$  using L<sup>A</sup>T<sub>E</sub>X.

```
>>> latex(Integral(cos(x)**2, (x, 0, pi)))
```

$$\int\limits_0^\pi \cos^2\left(x\right) dx$$

## Why SymPy?

There are many computer algebra systems out there. [This](#) Wikipedia article lists many of them. What makes SymPy a better choice than the alternatives?

First off, SymPy is completely free. It is open source, and licensed under the liberal BSD license, so you can modify the source code and even sell it if you want to. This contrasts with popular commercial systems like Maple or Mathematica that cost hundreds of dollars in licenses.

Second, SymPy uses Python. Most computer algebra systems invent their own language. Not SymPy. SymPy is written entirely in Python, and is executed entirely in Python. This means that if you already know Python, it is much easier to get started with SymPy, because you already know the syntax (and if you don't know Python, it is really easy to learn). We already know that Python is a well-designed, battle-tested language. The SymPy developers are confident in their abilities in writing mathematical software, but programming language design is a completely different thing. By reusing an existing language, we are able to focus on those things that matter: the mathematics.

Another computer algebra system, Sage also uses Python as its language. But Sage is large, with a download of over a gigabyte. An advantage of SymPy is that it is lightweight. In addition to being relatively small, it has no dependencies other than Python, so it can be used almost anywhere easily. Furthermore, the goals of Sage and the goals of SymPy are different. Sage aims to be a full featured system for mathematics, and aims to do so by compiling all the major open source mathematical systems together into one. When you call some function in Sage, such as `integrate`, it calls out to one of the open source packages that it includes. In

fact, SymPy is included in Sage. SymPy on the other hand aims to be an independent system, with all the features implemented in SymPy itself.

A final important feature of SymPy is that it can be used as a library. Many computer algebra systems focus on being usable in interactive environments, but if you wish to automate or extend them, it is difficult to do. With SymPy, you can just as easily use it in an interactive Python environment or import it in your own Python application. SymPy also provides APIs to make it easy to extend it with your own custom functions.

## Gotchas

To begin, we should make something about SymPy clear. SymPy is nothing more than a Python library, like NumPy, Django, or even modules in the Python standard library `sys` or `re`. What this means is that SymPy does not add anything to the Python language. Limitations that are inherent in the Python language are also inherent in SymPy. It also means that SymPy tries to use Python idioms whenever possible, making programming with SymPy easy for those already familiar with programming with Python. As a simple example, SymPy uses Python syntax to build expressions. Implicit multiplication (like  $3x$  or  $3 \ x$ ) is not allowed in Python, and thus not allowed in SymPy. To multiply 3 and  $x$ , you must type  $3*x$  with the `*`.

## Symbols

One consequence of this fact is that SymPy can be used in any environment where Python is available. We just import it, like we would any other library:

```
>>> from sympy import *
```

This imports all the functions and classes from SymPy into our interactive Python session. Now, suppose we start to do a computation.

```
>>> x + 1
Traceback (most recent call last):
...
NameError: name 'x' is not defined
```

Oops! What happened here? We tried to use the variable  $x$ , but it tells us that  $x$  is not defined. In Python, variables have no meaning until they are defined. SymPy is no different. Unlike many symbolic manipulation systems you may have used, in SymPy, variables are not defined automatically. To define variables, we must use `symbols`.

```
>>> x = symbols('x')
>>> x + 1
x + 1
```

`symbols` takes a string of variable names separated by spaces or commas, and creates Symbols out of them. We can then assign these to variable names. Later, we will investigate some convenient ways we can work around this issue. For now, let us just define the most common variable names,  $x$ ,  $y$ , and  $z$ , for use through the rest of this section

```
>>> x, y, z = symbols('x y z')
```

As a final note, we note that the name of a Symbol and the name of the variable it is assigned to need not have anything to do with one another.

```
>>> a, b = symbols('b a')
>>> a
b
>>> b
a
```

Here we have done the very confusing thing of assigning a Symbol with the name `a` to the variable `b`, and a Symbol of the name `b` to the variable `a`. Now the Python variable named `a` points to the SymPy Symbol named `b`, and vice versa. How confusing. We could have also done something like

```
>>> crazy = symbols('unrelated')
>>> crazy + 1
unrelated + 1
```

This also shows that Symbols can have names longer than one character if we want.

Usually, the best practice is to assign Symbols to Python variables of the same name, although there are exceptions: Symbol names can contain characters that are not allowed in Python variable names, or may just want to avoid typing long names by assigning Symbols with long names to single letter Python variables.

To avoid confusion, throughout this tutorial, Symbol names and Python variable names will always coincide. Furthermore, the word “Symbol” will refer to a SymPy Symbol and the word “variable” will refer to a Python variable.

Finally, let us be sure we understand the difference between SymPy Symbols and Python variables. Consider the following:

```
x = symbols('x')
expr = x + 1
x = 2
print(expr)
```

What do you think the output of this code will be? If you thought 3, you’re wrong. Let’s see what really happens

```
>>> x = symbols('x')
>>> expr = x + 1
>>> x = 2
>>> print(expr)
x + 1
```

Changing `x` to 2 had no effect on `expr`. This is because `x = 2` changes the Python variable `x` to 2, but has no effect on the SymPy Symbol `x`, which was what we used in creating `expr`. When we created `expr`, the Python variable `x` was a Symbol. After we created it, we changed the Python variable `x` to 2. But `expr` remains the same. This behavior is not unique to SymPy. All Python programs work this way: if a variable is changed, expressions that were already created with that variable do not change automatically. For example

```
>>> x = 'abc'
>>> expr = x + 'def'
>>> expr
'abcdef'
>>> x = 'ABC'
```

(continues on next page)

(continued from previous page)

```
>>> expr
'abcdef'
```

### Quick Tip

To change the value of a Symbol in an expression, use `subs`

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

In this example, if we want to know what `expr` is with the new value of `x`, we need to reevaluate the code that created `expr`, namely, `expr = x + 1`. This can be complicated if several lines created `expr`. One advantage of using a symbolic computation system like SymPy is that we can build a symbolic representation for `expr`, and then substitute `x` with values. The correct way to do this in SymPy is to use `subs`, which will be discussed in more detail later.

```
>>> x = symbols('x')
>>> expr = x + 1
>>> expr.subs(x, 2)
3
```

### Equals signs

Another very important consequence of the fact that SymPy does not extend Python syntax is that `=` does not represent equality in SymPy. Rather it is Python variable assignment. This is hard-coded into the Python language, and SymPy makes no attempts to change that.

You may think, however, that `==`, which is used for equality testing in Python, is used for SymPy as equality. This is not quite correct either. Let us see what happens when we use `==`.

```
>>> x + 1 == 4
False
```

Instead of treating `x + 1 == 4` symbolically, we just got `False`. In SymPy, `==` represents exact structural equality testing. This means that `a == b` means that we are *asking* if  $a = b$ . We always get a `bool` as the result of `==`. There is a separate object, called `Eq`, which can be used to create symbolic equalities

```
>>> Eq(x + 1, 4)
Eq(x + 1, 4)
```

There is one additional caveat about `==` as well. Suppose we want to know if  $(x+1)^2 = x^2+2x+1$ . We might try something like this:

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

We got `False` again. However,  $(x+1)^2$  *does* equal  $x^2+2x+1$ . What is going on here? Did we find a bug in SymPy, or is it just not powerful enough to recognize this basic algebraic fact?



Recall from above that `==` represents *exact* structural equality testing. “Exact” here means that two expressions will compare equal with `==` only if they are exactly equal structurally. Here,  $(x + 1)^2$  and  $x^2 + 2x + 1$  are not the same structurally. One is the power of an addition of two terms, and the other is the addition of three terms.

It turns out that when using SymPy as a library, having `==` test for exact structural equality is far more useful than having it represent symbolic equality, or having it test for mathematical equality. However, as a new user, you will probably care more about the latter two. We have already seen an alternative to representing equalities symbolically, `Eq`. To test if two things are equal, it is best to recall the basic fact that if  $a = b$ , then  $a - b = 0$ . Thus, the best way to check if  $a = b$  is to take  $a - b$  and simplify it, and see if it goes to 0. We will learn [later](#) (page 25) that the function to do this is called `simplify`. This method is not infallible—in fact, it can be [theoretically proven](#) that it is impossible to determine if two symbolic expressions are identically equal in general—but for most common expressions, it works quite well.

```
>>> a = (x + 1)**2
>>> b = x**2 + 2*x + 1
>>> simplify(a - b)
0
>>> c = x**2 - 2*x + 1
>>> simplify(a - c)
4*x
```

There is also a method called `equals` that tests if two expressions are equal by evaluating them numerically at random points.

```
>>> a = cos(x)**2 - sin(x)**2
>>> b = cos(2*x)
>>> a.equals(b)
True
```

## Two Final Notes: `^` and `/`

You may have noticed that we have been using `**` for exponentiation instead of the standard `^`. That’s because SymPy follows Python’s conventions. In Python, `^` represents logical exclusive or. SymPy follows this convention:

```
>>> True ^ False
True
>>> True ^ True
False
>>> Xor(x, y)
x ^ y
```

Finally, a small technical discussion on how SymPy works is in order. When you type something like `x + 1`, the SymPy Symbol `x` is added to the Python int `1`. Python’s operator rules then allow SymPy to tell Python that SymPy objects know how to be added to Python ints, and so `1` is automatically converted to the SymPy Integer object.

This sort of operator magic happens automatically behind the scenes, and you rarely need to even know that it is happening. However, there is one exception. Whenever you combine a SymPy object and a SymPy object, or a SymPy object and a Python object, you get a SymPy object, but whenever you combine two Python objects, SymPy never comes into play, and so you get a Python object.

```
>>> type(Integer(1) + 1)
<class 'sympy.core.numbers.Integer'>
>>> type(1 + 1)
<... 'int'>
```

This is usually not a big deal. Python ints work much the same as SymPy Integers, but there is one important exception: division. In SymPy, the division of two Integers gives a Rational:

```
>>> Integer(1)/Integer(3)
1/3
>>> type(Integer(1)/Integer(3))
<class 'sympy.core.numbers.Rational'>
```

But in Python `/` represents either integer division or floating point division, depending on whether you are in Python 2 or Python 3, and depending on whether or not you have run `from __future__ import division` in Python 2 which is no longer supported from versions above SymPy 1.5.1:

```
>>> from __future__ import division
>>> 1/2
0.5
```

To avoid this, we can construct the rational object explicitly

```
>>> Rational(1, 2)
1/2
```

This problem also comes up whenever we have a larger symbolic expression with `int/int` in it. For example:

```
>>> x + 1/2
x + 0.5
```

This happens because Python first evaluates `1/2` into `0.5`, and then that is cast into a SymPy type when it is added to `x`. Again, we can get around this by explicitly creating a Rational:

```
>>> x + Rational(1, 2)
x + 1/2
```

There are several tips on avoiding this situation in the [Gotchas and Pitfalls](#) (page 141) document.

## Further Reading

For more discussion on the topics covered in this section, see [Gotchas and Pitfalls](#) (page 141).

## SymPy Features

This section discusses the common and advanced SymPy operations and features.

### Content

#### Basic Operations

Here we discuss some of the most basic operations needed for expression manipulation in SymPy. Some more advanced operations will be discussed later in the [advanced expression manipulation](#) (page 61) section.

```
>>> from sympy import *
>>> x, y, z = symbols("x y z")
```

#### Substitution

One of the most common things you might want to do with a mathematical expression is substitution. Substitution replaces all instances of something in an expression with something else. It is done using the `subs` method. For example

```
>>> expr = cos(x) + 1
>>> expr.subs(x, y)
cos(y) + 1
```

Substitution is usually done for one of two reasons:

1. Evaluating an expression at a point. For example, if our expression is  $\cos(x) + 1$  and we want to evaluate it at the point  $x = 0$ , so that we get  $\cos(0) + 1$ , which is 2.

```
>>> expr.subs(x, 0)
2
```

2. Replacing a subexpression with another subexpression. There are two reasons we might want to do this. The first is if we are trying to build an expression that has some symmetry, such as  $x^{x^x}$ . To build this, we might start with  $x^{**}y$ , and replace  $y$  with  $x^{**}y$ . We would then get  $x^{**}(x^{**}y)$ . If we replaced  $y$  in this new expression with  $x^{**}x$ , we would get  $x^{**}(x^{**}(x^{**}x))$ , the desired expression.

```
>>> expr = x**y
>>> expr
x**y
>>> expr = expr.subs(y, x**y)
>>> expr
x**(x**y)
>>> expr = expr.subs(y, x**x)
>>> expr
x**(x**(x**x))
```

The second is if we want to perform a very controlled simplification, or perhaps a simplification that SymPy is otherwise unable to do. For example, say we have  $\sin(2x) + \cos(2x)$ , and we want to replace  $\sin(2x)$  with  $2\sin(x)\cos(x)$ . As we will learn later, the function

`expand_trig` does this. However, this function will also expand  $\cos(2x)$ , which we may not want. While there are ways to perform such precise simplification, and we will learn some of them in the [advanced expression manipulation](#) (page 61) section, an easy way is to just replace  $\sin(2x)$  with  $2\sin(x)\cos(x)$ .

```
>>> expr = sin(2*x) + cos(2*x)
>>> expand_trig(expr)
2*sin(x)*cos(x) + 2*cos(x)**2 - 1
>>> expr.subs(sin(2*x), 2*sin(x)*cos(x))
2*sin(x)*cos(x) + cos(2*x)
```

There are two important things to note about `subs`. First, it returns a new expression. SymPy objects are immutable. That means that `subs` does not modify it in-place. For example

```
>>> expr = cos(x)
>>> expr.subs(x, 0)
1
>>> expr
cos(x)
>>> x
x
```

### Quick Tip

SymPy expressions are immutable. No function will change them in-place.

Here, we see that performing `expr.subs(x, 0)` leaves `expr` unchanged. In fact, since SymPy expressions are immutable, no function will change them in-place. All functions will return new expressions.

To perform multiple substitutions at once, pass a list of (old, new) pairs to `subs`.

```
>>> expr = x**3 + 4*x*y - z
>>> expr.subs([(x, 2), (y, 4), (z, 0)])
40
```

It is often useful to combine this with a list comprehension to do a large set of similar replacements all at once. For example, say we had  $x^4 - 4x^3 + 4x^2 - 2x + 3$  and we wanted to replace all instances of  $x$  that have an even power with  $y$ , to get  $y^4 - 4x^3 + 4y^2 - 2x + 3$ .

```
>>> expr = x**4 - 4*x**3 + 4*x**2 - 2*x + 3
>>> replacements = [(x**i, y**i) for i in range(5) if i % 2 == 0]
>>> expr.subs(replacements)
-4*x**3 - 2*x + y**4 + 4*y**2 + 3
```

## Converting Strings to SymPy Expressions

The `sympify` function (that's `sympify`, not to be confused with `simplify`) can be used to convert strings into SymPy expressions.

For example

```
>>> str_expr = "x**2 + 3*x - 1/2"
>>> expr = sympify(str_expr)
>>> expr
x**2 + 3*x - 1/2
>>> expr.subs(x, 2)
19/2
```

**Warning:** `sympify` uses `eval`. Don't use it on unsanitized input.

## evalf

To evaluate a numerical expression into a floating point number, use `evalf`.

```
>>> expr = sqrt(8)
>>> expr.evalf()
2.82842712474619
```

SymPy can evaluate floating point expressions to arbitrary precision. By default, 15 digits of precision are used, but you can pass any number as the argument to `evalf`. Let's compute the first 100 digits of  $\pi$ .

```
>>> pi.evalf(100)
3.
→ 1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348
```

To numerically evaluate an expression with a Symbol at a point, we might use `subs` followed by `evalf`, but it is more efficient and numerically stable to pass the substitution to `evalf` using the `subs` flag, which takes a dictionary of Symbol: point pairs.

```
>>> expr = cos(2*x)
>>> expr.evalf(subs={x: 2.4})
0.0874989834394464
```

Sometimes there are roundoff errors smaller than the desired precision that remain after an expression is evaluated. Such numbers can be removed at the user's discretion by setting the `chop` flag to `True`.

```
>>> one = cos(1)**2 + sin(1)**2
>>> (one - 1).evalf()
-0.e-124
>>> (one - 1).evalf(chop=True)
0
```

## lambdify

`subs` and `evalf` are good if you want to do simple evaluation, but if you intend to evaluate an expression at many points, there are more efficient ways. For example, if you wanted to evaluate an expression at a thousand points, using SymPy would be far slower than it needs to be, especially if you only care about machine precision. Instead, you should use libraries like NumPy and SciPy.

The easiest way to convert a SymPy expression to an expression that can be numerically evaluated is to use the `lambdify` function. `lambdify` acts like a `lambda` function, except it converts the SymPy names to the names of the given numerical library, usually NumPy. For example

```
>>> import numpy
>>> a = numpy.arange(10)
>>> expr = sin(x)
>>> f = lambdify(x, expr, "numpy")
>>> f(a)
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427
 -0.2794155  0.6569866  0.98935825  0.41211849]
```

**Warning:** `lambdify` uses `eval`. Don't use it on unsanitized input.

You can use other libraries than NumPy. For example, to use the standard library `math` module, use `"math"`.

```
>>> f = lambdify(x, expr, "math")
>>> f(0.1)
0.0998334166468
```

To use `lambdify` with numerical libraries that it does not know about, pass a dictionary of `sympy_name:numerical_function` pairs. For example

```
>>> def mysin(x):
...     """
...     My sine. Note that this is only accurate for small x.
...     """
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

## Printing

As we have already seen, SymPy can pretty print its output using Unicode characters. This is a short introduction to the most common printing options available in SymPy.

## Printers

There are several printers available in SymPy. The most common ones are

- str
- srepr
- ASCII pretty printer
- Unicode pretty printer
- LaTeX
- MathML
- Dot

In addition to these, there are also “printers” that can output SymPy objects to code, such as C, Fortran, Javascript, Theano, and Python. These are not discussed in this tutorial.

## Setting up Pretty Printing

If all you want is the best pretty printing, use the `init_printing()` function. This will automatically enable the best printer available in your environment.

```
>>> from sympy import init_printing
>>> init_printing()
```

If you plan to work in an interactive calculator-type session, the `init_session()` function will automatically import everything in SymPy, create some common Symbols, setup plotting, and run `init_printing()`.

```
>>> from sympy import init_session
>>> init_session()
```

```
Python console for SymPy 0.7.3 (Python 2.7.5-64-bit) (ground types:
↳ gmpy)
```

These commands were executed:

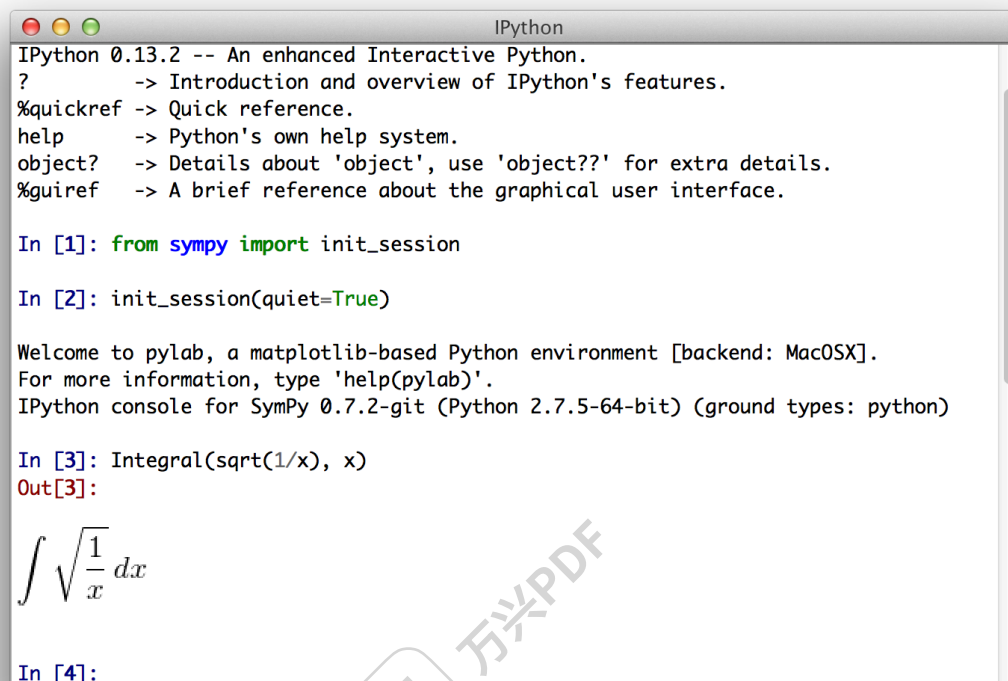
```
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> init_printing() # doctest: +SKIP
```

Documentation can be found at <http://www.sympy.org>

```
>>>
```

In any case, this is what will happen:

- In the IPython QTConsole, if  $\text{\LaTeX}$  is installed, it will enable a printer that uses  $\text{\LaTeX}$ .



```
IPython 0.13.2 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%gui ref -> A brief reference about the graphical user interface.

In [1]: from sympy import init_session

In [2]: init_session(quiet=True)

Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX].
For more information, type 'help(pylab)'.
IPython console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: python)

In [3]: Integral(sqrt(1/x), x)
Out[3]:
```

$$\int \sqrt{\frac{1}{x}} dx$$

```
In [4]:
```

If  $\text{\LaTeX}$  is not installed, but Matplotlib is installed, it will use the Matplotlib rendering engine. If Matplotlib is not installed, it uses the Unicode pretty printer.

- In the IPython notebook, it will use MathJax to render  $\text{\LaTeX}$ .

```
In [1]: from sympy import *
x, y, z = symbols('x y z')
init_printing()
```

```
In [2]: Integral(sqrt(1/x), x)
```

```
Out[2]:
```

$$\int \sqrt{\frac{1}{x}} dx$$



- In an IPython console session, or a regular Python session, it will use the Unicode pretty printer if the terminal supports Unicode.

```
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> 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)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

$$\int \sqrt{\frac{1}{x}} dx$$

>>>
```

- In a terminal that does not support Unicode, the ASCII pretty printer is used.

```
Python 2.7.5 (default, May 16 2013, 18:48:51)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import init_session
>>> init_session()
Python console for SymPy 0.7.2-git (Python 2.7.5-64-bit) (ground types: gmpy)

These commands were executed:
>>> 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)

Documentation can be found at http://www.sympy.org

>>> Integral(sqrt(1/x), x)

$$\int \sqrt{\frac{1}{x}} dx$$

>>> █
```

To explicitly not use  $\LaTeX$ , pass `use_latex=False` to `init_printing()` or `init_session()`.

To explicitly not use Unicode, pass `use_unicode=False`.

## Printing Functions

In addition to automatic printing, you can explicitly use any one of the printers by calling the appropriate function.

**str**

To get a string form of an expression, use `str(expr)`. This is also the form that is produced by `print(expr)`. String forms are designed to be easy to read, but in a form that is correct Python syntax so that it can be copied and pasted. The `str()` form of an expression will usually look exactly the same as the expression as you would enter it.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> str(Integral(sqrt(1/x), x))
'Integral(sqrt(1/x), x)'
>>> print(Integral(sqrt(1/x), x))
Integral(sqrt(1/x), x)
```

## srepr

The `srepr` form of an expression is designed to show the exact form of an expression. It will be discussed more in the *Advanced Expression Manipulation* (page 61) section. To get it, use `srepr()`<sup>1</sup>.

```
>>> srepr(Integral(sqrt(1/x), x))
"Integral(Pow(Pow(Symbol('x'), Integer(-1)), Rational(1, 2)), Tuple(Symbol('x'
↑
)))"
```

The srepr form is mostly useful for understanding how an expression is built internally.

## ASCII Pretty Printer

The ASCII pretty printer is accessed from `pprint()`. If the terminal does not support Unicode, the ASCII printer is used by default. Otherwise, you must pass `use_unicode=False`.

```
>>> pprint(Integral(sqrt(1/x), x), use_unicode=False)
```

$$\int \frac{1}{x} dx$$

---

(continues on next page)

<sup>1</sup> SymPy does not use the Python builtin `repr()` function for repr printing, because in Python `str(list)` calls `repr()` on the elements of the list, and some SymPy functions return lists (such as `solve()`). Since `srepr()` is so verbose, it is unlikely that anyone would want it called by default on the output of `solve()`.

```
>>> pretty(Integral(sqrt(1/x), x), use_unicode=False)

$$\int \frac{1}{\sqrt{x}} dx$$

>>> print(pretty(Integral(sqrt(1/x), x), use_unicode=False))

$$\int \frac{1}{\sqrt{x}} dx$$

```

The Unicode pretty printer is also accessed from `pprint()` and `pretty()`. If the terminal supports Unicode, it is used automatically. If `pprint()` is not able to detect that the terminal supports unicode, you can pass `use_unicode=True` to force it to use Unicode.

$$\int \sqrt{\frac{1}{x}} \, dx$$

```
>>> print(latex(Integral(sqrt(1/x), x)))
\int \sqrt{\frac{1}{x}}\, dx
```

## 2.1. Introductory Tutorial

## MathML

There is also a printer to MathML, called `print_mathml()`. It must be imported from `sympy.printing.mathml`.

```
>>> from sympy.printing.mathml import print_mathml
>>> print_mathml(Integral(sqrt(1/x), x))
<apply>
  <int/>
    <bvar>
      <ci>x</ci>
    </bvar>
    <apply>
      <root/>
        <apply>
          <power/>
            <ci>x</ci>
            <cn>-1</cn>
          </apply>
        </apply>
      </apply>
    </apply>
```

`print_mathml()` prints the output. If you want the string, use the function `mathml()`.

## Dot

The `dotprint()` function in `sympy.printing.dot` prints output to dot format, which can be rendered with Graphviz. See the [Advanced Expression Manipulation](#) (page 61) section for some examples of the output of this printer.

Here is an example of the raw output of the `dotprint()` function

```
>>> from sympy.printing.dot import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{

# Graph style
"ordering"="out"
"rankdir"="TD"

#####
# Nodes #
#####

"Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape"=
->"ellipse"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#####
# Edges #
```

(continues on next page)

(continued from previous page)

```
#####
"Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,)"
"Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,)"
}
```

## Simplification

To make this document easier to read, we are going to enable pretty printing.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> init_printing(use_unicode=True)
```

### simplify

Now let's jump in and do some interesting mathematics. One of the most useful features of a symbolic manipulation system is the ability to simplify mathematical expressions. SymPy has dozens of functions to perform various kinds of simplification. There is also one general function called `simplify()` that attempts to apply all of these functions in an intelligent way to arrive at the simplest form of an expression. Here are some examples

```
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> simplify((x**3 + x**2 - x - 1)/(x**2 + 2*x + 1))
x - 1
>>> simplify(gamma(x)/gamma(x - 2))
(x - 2)*(x - 1)
```

Here, `gamma(x)` is  $\Gamma(x)$ , the [gamma function](#). We see that `simplify()` is capable of handling a large class of expressions.

But `simplify()` has a pitfall. It just applies all the major simplification operations in SymPy, and uses heuristics to determine the simplest result. But “simplest” is not a well-defined term. For example, say we wanted to “simplify”  $x^2 + 2x + 1$  into  $(x + 1)^2$ :

```
>>> simplify(x**2 + 2*x + 1)
2
x + 2*x + 1
```

We did not get what we want. There is a function to perform this simplification, called `factor()`, which will be discussed below.

Another pitfall to `simplify()` is that it can be unnecessarily slow, since it tries many kinds of simplifications before picking the best one. If you already know exactly what kind of simplification you are after, it is better to apply the specific simplification function(s) that apply those simplifications.

Applying specific simplification functions instead of `simplify()` also has the advantage that specific functions have certain guarantees about the form of their output. These will be discussed with each function below. For example, `factor()`, when called on a polynomial

with rational coefficients, is guaranteed to factor the polynomial into irreducible factors. `simplify()` has no guarantees. It is entirely heuristical, and, as we saw above, it may even miss a possible type of simplification that SymPy is capable of doing.

`simplify()` is best when used interactively, when you just want to whittle down an expression to a simpler form. You may then choose to apply specific functions once you see what `simplify()` returns, to get a more precise result. It is also useful when you have no idea what form an expression will take, and you need a catchall function to simplify it.

## Polynomial/Rational Function Simplification

### expand

`expand()` is one of the most common simplification functions in SymPy. Although it has a lot of scopes, for now, we will consider its function in expanding polynomial expressions. For example:

```
>>> expand((x + 1)**2)
2
x  + 2·x + 1
>>> expand((x + 2)*(x - 3))
2
x  - x - 6
```

Given a polynomial, `expand()` will put it into a canonical form of a sum of monomials.

`expand()` may not sound like a simplification function. After all, by its very name, it makes expressions bigger, not smaller. Usually this is the case, but often an expression will become smaller upon calling `expand()` on it due to cancellation.

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

### factor

`factor()` takes a polynomial and factors it into irreducible factors over the rational numbers. For example:

```
>>> factor(x**3 - x**2 + x - 1)
      2
      (x - 1)·(x + 1)
>>> factor(x**2*z + 4*x*y*z + 4*y**2*z)
      2
      z·(x + 2·y)
```

For polynomials, `factor()` is the opposite of `expand()`. `factor()` uses a complete multivariate factorization algorithm over the rational numbers, which means that each of the factors returned by `factor()` is guaranteed to be irreducible.

If you are interested in the factors themselves, `factor_list` returns a more structured output.

```
>>> factor_list(x**2*z + 4*x*y*z + 4*y**2*z)
(1, [(z, 1), (x + 2·y, 2)])
```

Note that the input to `factor` and `expand` need not be polynomials in the strict sense. They will intelligently factor or expand any kind of expression (though note that the factors may not be irreducible if the input is no longer a polynomial over the rationals).

```
>>> expand((cos(x) + sin(x))**2)
      2      2
sin (x) + 2·sin(x)·cos(x) + cos (x)
>>> factor(cos(x)**2 + 2*cos(x)*sin(x) + sin(x)**2)
      2
(sin(x) + cos(x))
```

## collect

`collect()` collects common powers of a term in an expression. For example

```
>>> expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
>>> expr
      3      2      2
x  - x  ·z + 2·x  + x·y + x - 3
>>> collected_expr = collect(expr, x)
>>> collected_expr
      3      2
x  + x  ·(2 - z) + x·(y + 1) - 3
```

`collect()` is particularly useful in conjunction with the `.coeff()` method. `expr.coeff(x, n)` gives the coefficient of  $x^n$  in `expr`:

```
>>> collected_expr.coeff(x, 2)
2 - z
```

## cancel

`cancel()` will take any rational function and put it into the standard canonical form,  $\frac{p}{q}$ , where  $p$  and  $q$  are expanded polynomials with no common factors, and the leading coefficients of  $p$  and  $q$  do not have denominators (i.e., are integers).

```
>>> cancel((x**2 + 2*x + 1)/(x**2 + x))
x + 1
-----
x
```

```
>>> expr = 1/x + (3*x/2 - 2)/(x - 4)
>>> expr
      3·x
----- - 2
      2
      1
----- + -
      x - 4  x
>>> cancel(expr)
      2
```

(continues on next page)

(continued from previous page)

$$\frac{3 \cdot x^2 - 2 \cdot x - 8}{2 \cdot x^2 - 8 \cdot x}$$

```
>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
```

$$\frac{x^2 \cdot y^2 - 2 \cdot x \cdot y \cdot z + x \cdot z^2 + y^2 - 2 \cdot y \cdot z + z^2}{x^2 - 1}$$

```
>>> cancel(expr)
```

$$\frac{y^2 - 2 \cdot y \cdot z + z^2}{x - 1}$$

Note that since `factor()` will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)
```

$$\frac{(y - z)^2}{x - 1}$$

However, if you are only interested in making sure that the expression is in canceled form, `cancel()` is more efficient than `factor()`.

## apart

`apart()` performs a [partial fraction decomposition](#) on a rational function.

```
>>> expr = (4*x**3 + 21*x**2 + 10*x + 12)/(x**4 + 5*x**3 + 5*x**2 + 4*x)
>>> expr
```

$$\frac{4 \cdot x^3 + 21 \cdot x^2 + 10 \cdot x + 12}{x^4 + 5 \cdot x^3 + 5 \cdot x^2 + 4 \cdot x}$$

```
>>> apart(expr)
```

$$\frac{2 \cdot x - 1}{x^2 + x + 1} - \frac{1}{x + 4} + \frac{3}{x}$$



## Trigonometric Simplification

**Note:** SymPy follows Python’s naming conventions for inverse trigonometric functions, which is to append an `a` to the front of the function’s name. For example, the inverse cosine, or arc cosine, is called `acos()`.

```
>>> acos(x)
acos(x)
>>> cos(acos(x))
x
>>> asin(1)
π
—
2
```

### trigsimp

To simplify expressions using trigonometric identities, use `trigsimp()`.

```
>>> trigsimp(sin(x)**2 + cos(x)**2)
1
>>> trigsimp(sin(x)**4 - 2*cos(x)**2*sin(x)**2 + cos(x)**4)
cos(4·x)  1
——— + —
  2      2
>>> trigsimp(sin(x)*tan(x)/sec(x))
  2
sin (x)
```

`trigsimp()` also works with hyperbolic trig functions.

```
>>> trigsimp(cosh(x)**2 + sinh(x)**2)
cosh(2·x)
>>> trigsimp(sinh(x)/tanh(x))
cosh(x)
```

Much like `simplify()`, `trigsimp()` applies various trigonometric identities to the input expression, and then uses a heuristic to return the “best” one.

### expand\_trig

To expand trigonometric functions, that is, apply the sum or double angle identities, use `expand_trig()`.

```
>>> expand_trig(sin(x + y))
sin(x)·cos(y) + sin(y)·cos(x)
>>> expand_trig(tan(2·x))
  2·tan(x)
```

(continues on next page)

(continued from previous page)

$$1 - \tan^2(x)$$

Because `expand_trig()` tends to make trigonometric expressions larger, and `trigsimp()` tends to make them smaller, these identities can be applied in reverse using `trigsimp()`

```
>>> trigsimp(sin(x)*cos(y) + sin(y)*cos(x))
sin(x + y)
```

## Powers

Before we introduce the power simplification functions, a mathematical discussion on the identities held by powers is in order. There are three kinds of identities satisfied by exponents

1.  $x^a x^b = x^{a+b}$
2.  $x^a y^a = (xy)^a$
3.  $(x^a)^b = x^{ab}$

Identity 1 is always true.

Identity 2 is not always true. For example, if  $x = y = -1$  and  $a = \frac{1}{2}$ , then  $x^a y^a = \sqrt{-1}\sqrt{-1} = i \cdot i = -1$ , whereas  $(xy)^a = \sqrt{-1 \cdot -1} = \sqrt{1} = 1$ . However, identity 2 is true at least if  $x$  and  $y$  are nonnegative and  $a$  is real (it may also be true under other conditions as well). A common consequence of the failure of identity 2 is that  $\sqrt{x}\sqrt{y} \neq \sqrt{xy}$ .

Identity 3 is not always true. For example, if  $x = -1$ ,  $a = 2$ , and  $b = \frac{1}{2}$ , then  $(x^a)^b = ((-1)^2)^{1/2} = \sqrt{1} = 1$  and  $x^{ab} = (-1)^{2 \cdot 1/2} = (-1)^1 = -1$ . However, identity 3 is true when  $b$  is an integer (again, it may also hold in other cases as well). Two common consequences of the failure of identity 3 are that  $\sqrt{x^2} \neq x$  and that  $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$ .

To summarize

Identity	Sufficient conditions to hold	Counterexample when conditions are not met	Important consequences
1. $x^a x^b = x^{a+b}$	Always true	None	None
2. $x^a y^a = (xy)^a$	$x, y \geq 0$ and $a \in \mathbb{R}$	$(-1)^{1/2}(-1)^{1/2} \neq (-1 \cdot -1)^{1/2}$	$\sqrt{x}\sqrt{y} \neq \sqrt{xy}$ in general
3. $(x^a)^b = x^{ab}$	$b \in \mathbb{Z}$	$((-1)^2)^{1/2} \neq (-1)^{2 \cdot 1/2}$	$\sqrt{x^2} \neq x$ and $\sqrt{\frac{1}{x}} \neq \frac{1}{\sqrt{x}}$ in general

This is important to remember, because by default, SymPy will not perform simplifications if they are not true in general.

In order to make SymPy perform simplifications involving identities that are only true under certain assumptions, we need to put assumptions on our Symbols. We will undertake a full discussion of the assumptions system later, but for now, all we need to know are the following.

- By default, SymPy Symbols are assumed to be complex (elements of  $\mathbb{C}$ ). That is, a simplification will not be applied to an expression with a given Symbol unless it holds for all complex numbers.
- Symbols can be given different assumptions by passing the assumption to `symbols()`. For the rest of this section, we will be assuming that  $x$  and  $y$  are positive, and that  $a$  and  $b$  are real. We will leave  $z$ ,  $t$ , and  $c$  as arbitrary complex Symbols to demonstrate what happens in that case.

```
>>> x, y = symbols('x y', positive=True)
>>> a, b = symbols('a b', real=True)
>>> z, t, c = symbols('z t c')
```

**Note:** In SymPy, `sqrt(x)` is just a shortcut to `x**Rational(1, 2)`. They are exactly the same object.

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

## powsimp

`powsimp()` applies identities 1 and 2 from above, from left to right.

```
>>> powsimp(x**a*x**b)
a + b
x
>>> powsimp(x**a*y**a)
a
(x·y)
```

Notice that `powsimp()` refuses to do the simplification if it is not valid.

```
>>> powsimp(t**c*z**c)
c c
t · z
```

If you know that you want to apply this simplification, but you don't want to mess with assumptions, you can pass the `force=True` flag. This will force the simplification to take place, regardless of assumptions.

```
>>> powsimp(t**c*z**c, force=True)
c
(t·z)
```

Note that in some instances, in particular, when the exponents are integers or rational numbers, and identity 2 holds, it will be applied automatically.

```
>>> (z*t)**2
2 2
t · z
>>> sqrt(x*y)
√x·√y
```

This means that it will be impossible to undo this identity with `powsimp()`, because even if `powsimp()` were to put the bases together, they would be automatically split apart again.

```
>>> powsimp(z**2*t**2)
      2 2
      t · z
>>> powsimp(sqrt(x)*sqrt(y))
      √x·√y
```

## expand\_power\_exp / expand\_power\_base

`expand_power_exp()` and `expand_power_base()` apply identities 1 and 2 from right to left, respectively.

```
>>> expand_power_exp(x**(a + b))
      a  b
      x · x
```

```
>>> expand_power_base((x*y)**a)
      a  a
      x · y
```

As with `powsimp()`, identity 2 is not applied if it is not valid.

```
>>> expand_power_base((z*t)**c)
      c
      (t·z)
```

And as with `powsimp()`, you can force the expansion to happen without fiddling with assumptions by using `force=True`.

```
>>> expand_power_base((z*t)**c, force=True)
      c  c
      t · z
```

As with identity 2, identity 1 is applied automatically if the power is a number, and hence cannot be undone with `expand_power_exp()`.

```
>>> x**2*x**3
      5
      x
>>> expand_power_exp(x**5)
      5
      x
```

## powdenest

`powdenest()` applies identity 3, from left to right.

```
>>> powdenest((x**a)**b)
a·b
x
```

As before, the identity is not applied if it is not true under the given assumptions.

```
>>> powdenest((z**a)**b)
b
( a )
( z )
```

And as before, this can be manually overridden with `force=True`.

```
>>> powdenest((z**a)**b, force=True)
a·b
z
```

## Exponentials and logarithms

**Note:** In SymPy, as in Python and most programming languages, `log` is the natural logarithm, also known as `ln`. SymPy automatically provides an alias `ln = log` in case you forget this.

```
>>> ln(x)
log(x)
```

Logarithms have similar issues as powers. There are two main identities

1.  $\log(xy) = \log(x) + \log(y)$
2.  $\log(x^n) = n \log(x)$

Neither identity is true for arbitrary complex  $x$  and  $y$ , due to the branch cut in the complex plane for the complex logarithm. However, sufficient conditions for the identities to hold are if  $x$  and  $y$  are positive and  $n$  is real.

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

As before,  $z$  and  $t$  will be Symbols with no additional assumptions.

Note that the identity  $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$  is a special case of identities 1 and 2 by  $\log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log(y^{-1}) = \log(x) - \log(y)$ , and thus it also holds if  $x$  and  $y$  are positive, but may not hold in general.

We also see that  $\log(e^x) = x$  comes from  $\log(e^x) = x \log(e) = x$ , and thus holds when  $x$  is real (and it can be verified that it does not hold in general for arbitrary complex  $x$ , for example,  $\log(e^{x+2\pi i}) = \log(e^x) = x \neq x + 2\pi i$ ).

## expand\_log

To apply identities 1 and 2 from left to right, use `expand_log()`. As always, the identities will not be applied unless they are valid.

```
>>> expand_log(log(x*y))
log(x) + log(y)
>>> expand_log(log(x/y))
log(x) - log(y)
>>> expand_log(log(x**2))
2*log(x)
>>> expand_log(log(x**n))
n*log(x)
>>> expand_log(log(z*t))
log(t*z)
```

As with `powsimp()` and `powdenest()`, `expand_log()` has a `force` option that can be used to ignore assumptions.

```
>>> expand_log(log(z**2))
log(z**2)
>>> expand_log(log(z**2), force=True)
2*log(z)
```

## logcombine

To apply identities 1 and 2 from right to left, use `logcombine()`.

```
>>> logcombine(log(x) + log(y))
log(x*y)
>>> logcombine(n*log(x))
log(x**n)
>>> logcombine(n*log(z))
n*log(z)
```

`logcombine()` also has a `force` option that can be used to ignore assumptions.

```
>>> logcombine(n*log(z), force=True)
log(z**n)
```

## Special Functions

SymPy implements dozens of special functions, ranging from functions in combinatorics to mathematical physics.

An extensive list of the special functions included with SymPy and their documentation is at the [Functions Module](#) (page 382) page.

For the purposes of this tutorial, let's introduce a few special functions in SymPy.

Let's define  $x$ ,  $y$ , and  $z$  as regular, complex Symbols, removing any assumptions we put on them in the previous section. We will also define  $k$ ,  $m$ , and  $n$ .

```
>>> x, y, z = symbols('x y z')
>>> k, m, n = symbols('k m n')
```

The [factorial](#) function is `factorial`. `factorial(n)` represents  $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ .  $n!$  represents the number of permutations of  $n$  distinct items.

```
>>> factorial(n)
n!
```

The [binomial coefficient](#) function is `binomial`. `binomial(n, k)` represents  $\binom{n}{k}$ , the number of ways to choose  $k$  items from a set of  $n$  distinct items. It is also often written as  $nCk$ , and is pronounced “ $n$  choose  $k$ ”.

```
>>> binomial(n, k)

$$\binom{n}{k}$$

```

The factorial function is closely related to the [gamma function](#), `gamma`. `gamma(z)` represents  $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$ , which for positive integer  $z$  is the same as  $(z-1)!$ .

```
>>> gamma(z)
 $\Gamma(z)$ 
```

The [generalized hypergeometric function](#) is `hyper`. `hyper([a_1, ..., a_p], [b_1, ..., b_q], z)` represents  ${}_pF_q \left( \begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right)$ . The most common case is  ${}_2F_1$ , which is often referred to as the [ordinary hypergeometric function](#).

```
>>> hyper([1, 2], [3], z)

$${}_2F_1 \left( \begin{matrix} 1, 2 \\ 3 \end{matrix} \middle| z \right)$$

```

## rewrite

A common way to deal with special functions is to rewrite them in terms of one another. This works for any function in SymPy, not just special functions. To rewrite an expression in terms of a function, use `expr.rewrite(function)`. For example,

```
>>> tan(x).rewrite(cos)
cos\left(x - \frac{\pi}{2}\right)
cos(x)
>>> factorial(x).rewrite(gamma)
\Gamma(x + 1)
```

For some tips on applying more targeted rewriting, see the [Advanced Expression Manipulation](#) (page 61) section.

## expand\_func

To expand special functions in terms of some identities, use `expand_func()`. For example

```
>>> expand_func(gamma(x + 3))
x \cdot (x + 1) \cdot (x + 2) \cdot \Gamma(x)
```

## hyperexpand

To rewrite hyper in terms of more standard functions, use `hyperexpand()`.

```
>>> hyperexpand(hyper([1, 1], [2], z))
-log(1 - z)
z
```

`hyperexpand()` also works on the more general Meijer G-function (see [its documentation](#) (page 522) for more information).

```
>>> expr = meijerg([[1],[1]], [[1],[ ]], -z)
>>> expr
G_{2, 1}^{1, 1}\left(\begin{matrix} 1 & 1 \\ 1 \end{matrix} \middle| -z\right)
>>> hyperexpand(expr)
1
-
z
e
```