

---

# **SymPy Documentation**

**Release 0.7.6-git**

**SymPy Development Team**

April 28, 2015



<b>1 Installation</b>	<b>1</b>
1.1 Source . . . . .	1
1.2 Run SymPy . . . . .	2
1.3 Questions . . . . .	2
<b>2 Tutorial</b>	<b>3</b>
2.1 Preliminaries . . . . .	3
2.2 Introduction . . . . .	4
2.3 Gotchas and Pitfalls . . . . .	7
2.4 Basic Operations . . . . .	15
2.5 Printing . . . . .	18
2.6 Simplification . . . . .	24
2.7 Calculus . . . . .	37
2.8 Solvers . . . . .	42
2.9 Matrices . . . . .	45
2.10 Advanced Expression Manipulation . . . . .	51
<b>3 Modules Reference</b>	<b>59</b>
3.1 SymPy Core . . . . .	59
3.2 Combinatorics Module . . . . .	160
3.3 Number Theory . . . . .	244
3.4 Basic Cryptography Module . . . . .	271
3.5 Concrete Mathematics . . . . .	288
3.6 Numerical evaluation . . . . .	303
3.7 Numeric Computation . . . . .	309
3.8 Functions Module . . . . .	311
3.9 Geometry Module . . . . .	429
3.10 Symbolic Integrals . . . . .	522
3.11 Numeric Integrals . . . . .	557
3.12 Logic Module . . . . .	563
3.13 Matrices . . . . .	573
3.14 Polynomials Manipulation Module . . . . .	652
3.15 Printing System . . . . .	850
3.16 Plotting Module . . . . .	872
3.17 Assumptions module . . . . .	880
3.18 Term rewriting . . . . .	895
3.19 Series Expansions . . . . .	897
3.20 Sets . . . . .	903

---

3.21	Simplify . . . . .	916
3.22	Details on the Hypergeometric Function Expansion Module . . . . .	938
3.23	Stats . . . . .	948
3.24	ODE . . . . .	984
3.25	PDE . . . . .	1036
3.26	Solvers . . . . .	1045
3.27	Diophantine . . . . .	1060
3.28	Inequality Solvers . . . . .	1078
3.29	Solveset . . . . .	1081
3.30	Tensor Module . . . . .	1086
3.31	Utilities . . . . .	1110
3.32	Parsing input . . . . .	1161
3.33	Calculus . . . . .	1165
3.34	Category Theory Module . . . . .	1171
3.35	Differential Geometry Module . . . . .	1187
3.36	Contributions to docs . . . . .	1201
<b>4</b>	<b>Special Topics</b>	<b>1203</b>
4.1	Introduction . . . . .	1203
4.2	Finite Difference Approximations to Derivatives . . . . .	1203
<b>5</b>	<b>User’s Guide</b>	<b>1209</b>
5.1	Introduction . . . . .	1209
5.2	Learning SymPy . . . . .	1209
5.3	SymPy’s Architecture . . . . .	1211
5.4	Contributing . . . . .	1217
<b>6</b>	<b>About</b>	<b>1219</b>
6.1	SymPy Development Team . . . . .	1219
6.2	Brief History . . . . .	1230
6.3	Financial and Infrastructure Support . . . . .	1231
6.4	License . . . . .	1232
	<b>Bibliography</b>	<b>1233</b>

---

## Installation

---

The SymPy CAS can be installed on virtually any computer with Python 2.7 or above. SymPy use `setuptools` and does require `mpmath` Python library to be installed first (`setuptools` should handle this dependency automatically). The current recommended method of installation is directly from the source files.

SymPy officially supports Python 2.7, 3.4, and PyPy3.

### 1.1 Source

If you are a developer or like to get the latest updates as they come, be sure to install from git. To download the repository, execute the following from the command line:

```
$ git clone git://github.com/skirkichev/omg.git
```

From your favorite command line terminal, change directory into that folder and execute the following:

```
$ python setup.py install
```

Alternatively, if you don't want to install the package onto your computer, you may run SymPy with the “`isympy`” console (which automatically imports SymPy packages and defines common symbols) by executing within that folder:

```
$ ./bin/isympy
```

You may now run SymPy statements directly within the Python shell:

```
>>> 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)
>>> diff(x**2/2, x)
x
```

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

```
$ git pull origin master
```

You can see old SymPy's history (from Hg and SVN repos) in the branch `sympy-svn-history`. To see this history as part of master's, simply do:

```
$ git fetch origin 'refs/replace/*:refs/replace/*'
```

## 1.2 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 [Tutorial](#) (page 3).

## 1.3 Questions

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

---

**Tutorial**

---

## 2.1 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.

### 2.1.1 Installation

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

### 2.1.2 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.

### 2.1.3 About This 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.

## 2.2 Introduction

### 2.2.1 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)
```

## 2.2.2 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 8)).

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 + 2*y` 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 + 2*y`. 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 `expr - x`, we did not get `x + 2*y - x`, but rather just `2*y`. The `x` and the `-x` automatically canceled one another. This is similar to how `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)
```

## 2.2.3 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 L<sup>A</sup>T<sub>E</sub>X), 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 \ \pi  
-----  
 2
```

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)
- \ 2 , \ 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))
      -t          t   t
y(t) = C2e     + C1 + -e
                  2
```

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

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

```
3   \ 17      \ 17   3
- + -----: 1, - ----- + -: 1
2      2         2      2
```

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

```
>>> besselj(nu, z).rewrite(jn)
-----  
 \ 2 \ z jn(\nu - 1/2, z)
```

```
\ ---  
\\ \pi  
Print  $\int_0^\pi \cos^2(x) dx$  using LATEX.  
>>> latex(Integral(cos(x)**2, (x, 0, pi)))  
\int_{0}^{\pi} \cos^2\left(x\right) dx
```

## 2.2.4 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.

## 2.3 Gotchas and Pitfalls

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.

For example, 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 `*`. Also, to raise something to a power, use `**`, not `^` (logical exclusive or in Python) as many computer algebra systems use. Parentheses `()` change operator precedence as you would normally expect.

### 2.3.1 Variables and 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 visa 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'
>>> 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
```

If you use `ismpy`, 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` module.

```
>>> 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 95) and `symbols()` (page 96).

Lastly, it is recommended that you not use `I` (page 108), `E` (page 108), `S` (page 74), `N()` (page 150), `O` (page 898), or `Q` (page 880) for variable or symbol names, as those are used for the imaginary unit (`i`), the base of the natural logarithm (`e`), the `sympify()` (page 59) function (see *Symbolic Expressions* (page 11) below), numeric evaluation (`N()` (page 150) is equivalent to `evalf()` (page 303)), 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` (page 893)), respectively. You can use the mnemonic QCOSINE 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 `isymfony`, you can also press the TAB key to get a list of all built-in names and to autocomplete.

### 2.3.2 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 symbolically. 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 symbolic 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 24) 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
```

### 2.3.3 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

`sympy.core.sympify.sympify()` (page 59) function, or just `S` (page 59), 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.  
6.200000000000000  
>>> 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 59) one of the numbers, or use `Rational` (page 101).

```
>>> x**(1/2) # evaluates to x**0 or x**0.5  
x**0.5  
>>> x**(S(1)/2) # sympify 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 101) (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)  
>>> 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 `isymmpy`, 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 101) 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)
100.0000000000000
>>> 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 can not 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 71) 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 71) 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.

## 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 317), `acos()` (page 314) and so on instead of the usual `arcsin` and `arccos`. Use the methods described in the section [Variables and Symbols](#) (page 8) above to see the names of all SymPy functions.

## 2.4 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 51) section.

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

### 2.4.1 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 51) 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
```

## 2.4.2 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.

## 2.4.3 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)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068
```

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
```

## 2.4.4 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]
```

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. Not only accurate for small x.
...     """
...     return x
>>> f = lambdify(x, expr, {"sin":mysin})
>>> f(0.1)
0.1
```

## 2.5 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.

### 2.5.1 Printers

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

- `str`
- `repr`

- 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.

## 2.5.2 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 L<sup>A</sup>T<sub>E</sub>X is installed, it will enable a printer that uses L<sup>A</sup>T<sub>E</sub>X.

The screenshot shows an IPython notebook window with the title "IPython". It displays the following session:

```
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.
%guiref   -> 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 L<sup>A</sup>T<sub>E</sub>X 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 L<sup>A</sup>T<sub>E</sub>X.

```
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)

```

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

```
/ 
 |
 | / \ 1
 |   - dx
 |
 /
```

```
>>> █
```

To explicitly not use L<sup>A</sup>T<sub>E</sub>X, pass `use_latex=False` to `init_printing()` or `init_session()`. To explicitly not use Unicode, pass `use_unicode=False`.

### 2.5.3 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)
```

repr

The repr 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 51) 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 `repr` 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)
      /
      |   / 1
      |   / - dx
      |   \/
      |
      /
```

`pprint()` prints the output to the screen. If you want the string form, use `pretty()`.

```
>>> pretty(Integral(sqrt(1/x), x), use_unicode=False)
' / \n | \n | ___ \n | / 1 \n | / - dx\n | \\\n | x \n | \n'
>>> print(pretty(Integral(sqrt(1/x), x), use_unicode=False))
/
| \n |
| ___ \n |
```

---

<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()`.

```
|   / 1
|   / - dx
| \v x
|
/
```

## Unicode Pretty Printer

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.

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

    ---  
    1  
    - dx  
    \ x
```

## L<sup>A</sup>T<sub>E</sub>X

To get the L<sup>A</sup>T<sub>E</sub>X form of an expression, use `latex()`.

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

The `latex()` function has many options to change the formatting of different things. See [its documentation](#) (page 861) for more details.

## 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>
```

`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 51) section for some examples of the output of this printer.

## 2.6 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)
```

### 2.6.1 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  + 2x + 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.

## 2.6.2 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 + 2x + 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 + 2y)
```

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 + 2y, 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
sin (x) + 2sin(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
x - x z + 2x + xy + x - 3
>>> collected_expr = collect(expr, x)
>>> collected_expr
 3      2
x + x (-z + 2) + 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)
-z + 2
```

## 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
3x
--- - 2
2      1
----- + -
x - 4   x
>>> cancel(expr)
2
3x - 2x - 8
-----
2
2x - 8x

>>> expr = (x*y**2 - 2*x*y*z + x*z**2 + y**2 - 2*y*z + z**2)/(x**2 - 1)
>>> expr
2      2      2
xy - 2xyz + xz + y - 2yz + z
-----
2
x - 1
>>> cancel(expr)
2      2
y - 2yz + z
-----
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)
      2
(y - z)
-----
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
      3      2
4x  + 21x  + 10x + 12
-----
      4      3      2
x  + 5x  + 5x  + 4x
>>> apart(expr)
      2x - 1      1      3
----- - ----- + -
      2            x + 4   x
x  + x + 1
```

## 2.6.3 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(4x)    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(2x)
>>> 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))
2tan(x)
-----
2
- tan (x) + 1
```

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)
```

## 2.6.4 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', extended_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
(xy)
```

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

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

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
      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
      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
(tz)
```

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)
ab
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)
ab
z
```

## 2.6.5 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', extended_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  $\log\left(\frac{x}{y}\right) = \log\left(x \cdot \frac{1}{y}\right) = \log(x) + \log\left(y^{-1}\right) = \log(x) - \log(y)$ , 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))
2log(x)
>>> expand_log(log(x**n))
nlog(x)
>>> expand_log(log(z*t))
log(tz)
```

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

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

## logcombine

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

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

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

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

## 2.6.6 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 313) 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)
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)
Γ(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)
      - 1, 2 |
      -           | z
      2 1   3   |
```

## 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(sin)
      2
      2sin (x)
-----
      sin(2x)
>>> factorial(x).rewrite(gamma)
Γ(x + 1)
```

For some tips on applying more targeted rewriting, see the *Advanced Expression Manipulation* (page 51) section.

## expand\_func

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

```
>>> expand_func(gamma(x + 3))
x(x + 1)(x + 2)Γ(x)
```

## hyperexpand

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

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

`hyperexpand()` also works on the more general Meijer G-function (see [its documentation](#) (page 407) for more information).

```
>>> expr = meijerg([[1],[1]], [[1],[]], -z)
>>> expr
-1, 1 1 1 |
|           | -z
-2, 1 1   |
>>> hyperexpand(expr)
1
-
z
e
```

## combsimp

To simplify combinatorial expressions, use `combsimp()`.

```
>>> combsimp(factorial(n)/factorial(n - 3))
n(n - 2)(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
-----
k + 1
```

`combsimp()` also simplifies expressions with `gamma`.

```
>>> combsimp(gamma(x)*gamma(1 - x))
π
-----
sin(πx)
```

### 2.6.7 Example: Continued Fractions

Let's use SymPy to explore continued fractions. A continued fraction is an expression of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

where  $a_0, \dots, a_n$  are integers, and  $a_1, \dots, a_n$  are positive. A continued fraction can also be infinite, but infinite objects are more difficult to represent in computers, so we will only examine the finite case here.

A continued fraction of the above form is often represented as a list  $[a_0; a_1, \dots, a_n]$ . Let's write a simple function that converts such a list to its continued fraction form. The easiest way to construct a continued fraction from a list is to work backwards. Note that despite the apparent symmetry of the definition, the first element,  $a_0$ , must usually be handled differently from the rest.

```
>>> def list_to_frac(l):
...     expr = Integer(0)
...     for i in reversed(l[1:]):
...         expr += i
...         expr = 1/expr
...     return l[0] + expr
```

```
>>> list_to_frac([x, y, z])
      1
x + -----
      1
y + -
      z
```

We use `Integer(0)` in `list_to_frac` so that the result will always be a SymPy object, even if we only pass in Python ints.

```
>>> list_to_frac([1, 2, 3, 4])
43
--
30
```

Every finite continued fraction is a rational number, but we are interested in symbolics here, so let's create a symbolic continued fraction. The `symbols()` function that we have been using has a shortcut to create numbered symbols. `symbols('a0:5')` will create the symbols `a0, a1, ..., a5`.

```
>>> syms = symbols('a0:5')
>>> syms
(a, a1, a2, a, a)
>>> a0, a1, a2, a3, a4 = syms
>>> frac = list_to_frac(syms)
>>> frac
      1
a + -----
      1
a1 + -----
      1
a2 + -----
      1
a + --
      a
```

This form is useful for understanding continued fractions, but lets put it into standard rational function form using `cancel()`.

```
>>> frac = cancel(frac)
>>> frac
aa1a2aa + aa1a2 + aa1a + aaa + a + a2aa + a2 + a
-----
a1a2aa + a1a2 + a1a + aa + 1
```

Now suppose we were given `frac` in the above canceled form. In fact, we might be given the fraction in any form, but we can always put it into the above canonical form with `cancel()`. Suppose that we knew that it could be rewritten as a continued fraction. How could we do this with SymPy? A continued fraction is recursively  $c + \frac{1}{f}$ , where  $c$  is an integer and  $f$  is a (smaller) continued fraction. If we could write the expression in this form, we could pull out each  $c$  recursively and add it to a list. We could then get a continued fraction with our `list_to_frac()` function.

The key observation here is that we can convert an expression to the form  $c + \frac{1}{f}$  by doing a partial fraction decomposition with respect to  $c$ . This is because  $f$  does not contain  $c$ . This means we need to use the `apart()` function. We use `apart()` to pull the term out, then subtract it from the expression, and take the reciprocal to get the  $f$  part.

```
>>> l = []
>>> frac = apart(frac, a0)
>>> frac
```

```
a2aa + a2 + a
a + -----
                   a1a2aa + a1a2 + a1a + aa + 1
>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac
a1a2aa + a1a2 + a1a + aa + 1
-----
a2aa + a2 + a
```

Now we repeat this process

```
>>> frac = apart(frac, a1)
```

```
>>> frac
```

```
      aa + 1
```

```
a1 + -----
      a2aa + a2 + a
```

```
>>> l.append(a1)
```

```
>>> frac = 1/(frac - a1)
```

```
>>> frac = apart(frac, a2)
```

```
>>> frac
```

```
      a
```

```
a2 + -----
      aa + 1
```

```
>>> l.append(a2)
```

```
>>> frac = 1/(frac - a2)
```

```
>>> frac = apart(frac, a3)
```

```
>>> frac
```

```
      1
```

```
a + --
```

```
      a
```

```
>>> l.append(a3)
```

```
>>> frac = 1/(frac - a3)
```

```
>>> frac = apart(frac, a4)
```

```
>>> frac
```

```
a
```

```
>>> l.append(a4)
```

```
>>> list_to_frac(l)
```

```
      1
```

```
a + -----
      1
```

```
      a1 + -----
          1
```

```
      a2 + -----
          1
```

```
      a + --
```

```
      a
```

Of course, this exercise seems pointless, because we already know that our `frac` is `list_to_frac([a0, a1, a2, a3, a4])`. So try the following exercise. Take a list of symbols and randomize them, and create the canceled continued fraction, and see if you can reproduce the original list. For example

```
>>> import random
>>> l = list(symbols('a0:5'))
>>> random.shuffle(l)
>>> orig_frac = frac = cancel(list_to_frac(l))
>>> del l
```

See if you can think of a way to figure out what symbol to pass to `apart()` at each stage (hint: think of

what happens to  $a_0$  in the formula  $a_0 + \frac{1}{a_1 + \dots}$  when it is canceled).

## 2.7 Calculus

This section covers how to do basic calculus tasks such as derivatives, integrals, limits, and series expansions in SymPy. If you are not familiar with the math of any part of this section, you may safely skip it.

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

### 2.7.1 Derivatives

To take derivatives, use the `diff` function.

```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
      2
      x
2xe
```

`diff` can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example, both of the following find the third derivative of  $x^4$ .

```
>>> diff(x**4, x, x, x)
24x
>>> diff(x**4, x, 3)
24x
```

You can also take derivatives with respect to many variables at once. Just pass each derivative in order, using the same syntax as for single variable derivatives. For example, each of the following will compute  $\frac{\partial^7}{\partial x \partial y^2 \partial z^4} e^{xyz}$ .

```
>>> expr = exp(x*y*z)
>>> diff(expr, x, y, y, z, z, z, z)
      3 2 3 3 3      2 2 2
x y x y z + 14x y z + 52xyz + 48e
xyz
>>> diff(expr, x, y, 2, z, 4)
      3 2 3 3 3      2 2 2
x y x y z + 14x y z + 52xyz + 48e
xyz
>>> diff(expr, x, y, y, z, 4)
      3 2 3 3 3      2 2 2
x y x y z + 14x y z + 52xyz + 48e
xyz
```

`diff` can also be called as a method. The two ways of calling `diff` are exactly the same, and are provided only for convenience.

```
>>> expr.diff(x, y, y, z, 4)
      3 2 3 3 3      2 2 2
x y x y z + 14x y z + 52xyz + 48e
xyz
```

To create an unevaluated derivative, use the `Derivative` class. It has the same syntax as `diff`.

```
>>> deriv = Derivative(expr, x, y, y, z, 4)
>>> deriv
    7
xyz
-----
e
  4   2
z  y  x
```

To evaluate an unevaluated derivative, use the `doit` method.

```
>>> deriv.doit()
  3   2   3   3      2   2   2
x  y  x  y  z + 14x  y  z + 52xyz + 48e
```

These unevaluated objects are useful for delaying the evaluation of the derivative, or for printing purposes. They are also used when SymPy does not know how to compute the derivative of an expression (for example, if it contains an undefined function, which are described in the *Solving Differential Equations* (page 44) section).

## 2.7.2 Integrals

To compute an integral, use the `integrate` function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, that is, an antiderivative, or primitive, just pass the variable after the expression.

```
>>> integrate(cos(x), x)
sin(x)
```

Note that SymPy does not include the constant of integration. If you want it, you can add one yourself, or rephrase your problem as a differential equation and use `dsolve` to solve it, which does add the constant (see *Solving Differential Equations* (page 44)).

### Quick Tip

$\infty$  in SymPy is `oo` (that's the lowercase letter "oh" twice). This is because `oo` looks like  $\infty$ , and is easy to type.

To compute a definite integral, pass the argument (`integration_variable`, `lower_limit`, `upper_limit`). For example, to compute

$$\int_0^{\infty} e^{-x} dx,$$

we would do

```
>>> integrate(exp(-x), (x, 0, oo))
1
```

As with indefinite integrals, you can pass multiple limit tuples to perform a multiple integral. For example, to compute

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} dx dy,$$

do

```
>>> integrate(exp(-x**2 - y**2), (x, -oo, oo), (y, -oo, oo))
 $\pi$ 
```

If `integrate` is unable to compute an integral, it returns an unevaluated `Integral` object.

```
>>> expr = integrate(x**x, x)
>>> print(expr)
Integral(x**x, x)
>>> expr

 $x$ 
 $x \, dx$ 
```

As with `Derivative`, you can create an unevaluated integral using `Integral`. To later evaluate this integral, call `doit`.

```
>>> expr = Integral(log(x)**2, x)
>>> expr

 $2$ 
 $\log(x) \, dx$ 

>>> expr.doit()
 $2$ 
 $x \log(x) - 2x \log(x) + 2x$ 
```

`integrate` uses powerful algorithms that are always improving to compute both definite and indefinite integrals, including heuristic pattern matching type algorithms, a partial implementation of the [Risch algorithm](#), and an algorithm using [Meijer G-functions](#) that is useful for computing integrals in terms of special functions, especially definite integrals. Here is a sampling of some of the power of `integrate`.

```
>>> integ = Integral((x**4 + x**2*exp(x) - x**2 - 2*x*exp(x) - 2*x -
...     exp(x))*exp(x)/((x - 1)**2*(x + 1)**2*(exp(x) + 1)), x)
>>> integ


$$\frac{x^4 + x^2 e^x - x^2 - 2x e^x - 2x^2 e^x}{(x - 1)^2 (x + 1)^2 e^x + 1} \, dx$$


>>> integ.doit()

$$\log(e^x + 1) + \frac{x^2}{x^2 - 1}$$


>>> integ = Integral(sin(x**2), x)
>>> integ

 $\sin(x^2) \, dx$ 

>>> integ.doit()

$$\frac{3\sqrt{2}\sqrt{\pi}}{2} \text{fresnels}\left(\frac{\sqrt{2}x}{\sqrt{\pi}}\right) \Gamma(3/4)$$

```

```
-----  
          ---  
          \ \pi  
-----  
          8\Gamma(7/4)  
  
>>> integ = Integral(x**y*exp(-x), (x, 0, oo))  
>>> integ  
∞  
  
y -x  
x e dx  
  
0  
>>> integ.doit()  
Γ(y + 1)   for -re(y) < 1  
  
∞  
  
y -x  
x e dx   otherwise  
  
0
```

This last example returned a `Piecewise` expression because the integral does not converge unless  $\Re(y) > 1$ .

### 2.7.3 Limits

SymPy can compute symbolic limits with the `limit` function. The syntax to compute

$$\lim_{x \rightarrow x_0} f(x)$$

is `limit(f(x), x, x0)`.

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

`limit` should be used instead of `subs` whenever the point of evaluation is a singularity. Even though SymPy has objects to represent  $\infty$ , using them for evaluation is not reliable because they do not keep track of things like rate of growth. Also, things like  $\infty - \infty$  and  $\frac{\infty}{\infty}$  return `nan` (not-a-number). For example

```
>>> expr = x**2/exp(x)  
>>> expr.subs(x, oo)  
nan  
>>> limit(expr, x, oo)  
0
```

Like `Derivative` and `Integral`, `limit` has an unevaluated counterpart, `Limit`. To evaluate it, use `doit`.

```
>>> expr = Limit((cos(x) - 1)/x, x, 0)  
>>> expr  
      cos(x) - 1  
      lim -----  
      x->0      x  
>>> expr.doit()  
0
```

To evaluate a limit at one side only, pass '+' or '-' as a third argument to `limit`. For example, to compute

$$\lim_{x \rightarrow 0^+} \frac{1}{x},$$

do

```
>>> limit(1/x, x, 0, '+')
infinity
```

As opposed to

```
>>> limit(1/x, x, 0, '-')
-infinity
```

## 2.7.4 Series Expansion

SymPy can compute asymptotic series expansions of functions around a point. To compute the expansion of  $f(x)$  around the point  $x = x_0$  terms of order  $x^n$ , use `f(x).series(x, x0, n)`. `x0` and `n` can be omitted, in which case the defaults `x0=0` and `n=6` will be used.

```
>>> expr = exp(sin(x))
>>> expr.series(x, 0, 4)
      2
      x      4
1 + x + -- + 0x
      2
```

The  $O(x^4)$  term at the end represents the Landau order term at  $x = 0$  (not to be confused with big O notation used in computer science, which generally represents the Landau order term at  $x = \infty$ ). It means that all  $x$  terms with power greater than or equal to  $x^4$  are omitted. Order terms can be created and manipulated outside of `series`. They automatically absorb higher order terms.

```
>>> x + x**3 + x**6 + O(x**4)
      3      4
x + x + 0x
>>> x*O(1)
O(x)
```

If you do not want the order term, use the `removeO` method.

```
>>> expr.series(x, 0, 4).removeO()
      2
      x
-- + x + 1
      2
```

The `O` notation supports arbitrary limit points (other than 0):

```
>>> exp(x - 6).series(x, x0=6)
      2      3      4      5
      (x - 6)    (x - 6)    (x - 6)    (x - 6)
-5 + ----- + ----- + ----- + ----- + x + O(x - 6) ; x -> 6
      2      6      24      120
```

## 2.7.5 Finite differences

So far we have looked at expressions with analytical derivatives and primitive functions respectively. But what if we want to have an expression to estimate a derivative of a curve for which we lack a closed form

representation, or for which we don't know the functional values for yet. One approach would be to use a finite difference approach.

You can use the `as_finite_diff` method of on any `Derivative` instance to generate approximations to derivatives of arbitrary order:

```
>>> f = Function('f')
>>> dfdx = f(x).diff(x)
>>> as_finite_diff(dfdx)
-f(x - 1/2) + f(x + 1/2)
```

here the first order derivative was approximated around `x` using a minimum number of points (2 for 1st order derivative) evaluated equidistantly using a step-size of 1. We can use arbitrary steps (possibly containing symbolic expressions):

```
>>> f = Function('f')
>>> d2fdx2 = f(x).diff(x, 2)
>>> h = Symbol('h')
>>> as_finite_diff(d2fdx2, [-3*h, -h, 2*h])
f(-3h)   f(-h)   2f(2h)
----- - ----- + -----
      2           2           2
      5h         3h        15h
```

If you are just interested in evaluating the weights, you can do so manually:

```
>>> finite_diff_weights(2, [-3, -1, 2], 0)[-1][-1]
[1/5, -1/3, 2/15]
```

note that we only need the last element in the last sublist returned from `finite_diff_weights`. The reason for this is that `finite_diff_weights` also generates weights for lower derivatives and using fewer points (see the documentation of `finite_diff_weights` for more details).

if using `finite_diff_weights` directly looks complicated and the `as_finite_diff` function operating on `Derivative` instances is not flexible enough, you can use `apply_finite_diff` which takes order, `x_list`, `y_list` and `x0` as parameters:

```
>>> x_list = [-3, 1, 2]
>>> y_list = symbols('a b c')
>>> apply_finite_diff(1, x_list, y_list, 0)
3a   b   2c
----- - - - + -
 20    4     5
```

## 2.8 Solvers

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

### 2.8.1 A Note about Equations

Recall from the [gotchas](#) (page 10) section of this tutorial that symbolic equations in SymPy are not represented by `=` or `==`, but by `Eq`.

---

```
>>> Eq(x, y)
x = y
```

However, there is an even easier way. In SymPy, any expression is not in an `Eq` is automatically assumed to equal 0 by the solving functions. Since  $a = b$  if and only if  $a - b = 0$ , this means that instead of using `x == y`, you can just use `x - y`. For example

```
>>> solve(Eq(x**2, 1), x)
[-1, 1]
>>> solve(Eq(x**2 - 1, 0), x)
[-1, 1]
>>> solve(x**2 - 1, x)
[-1, 1]
```

This is particularly useful if the equation you wish to solve is already equal to 0. Instead of typing `solve(Eq(expr, 0), x)`, you can just use `solve(expr, x)`.

## 2.8.2 Solving Equations Algebraically

The main function for solving algebraic equations, as we saw above, is `solve`. The syntax is `solve(equations, variables)`, where, as we saw above, `equations` may be in the form of `Eq` instances or expressions that are assumed to be equal to zero.

When solving a single equation, the output of `solve` is a list of the solutions.

```
>>> solve(x**2 - x, x)
[0, 1]
```

If no solutions are found, an empty list is returned, or `NotImplementedError` is raised.

```
>>> solve(exp(x), x)
[]
```

---

**Note:** If `solve` returns `[]` or raises `NotImplementedError`, it doesn't mean that the equation has no solutions. It just means that it couldn't find any. Often this means that the solutions cannot be represented symbolically. For example, the equation  $x = \cos(x)$  has a solution, but it cannot be represented symbolically using standard functions.

```
>>> solve(x - cos(x), x)
Traceback (most recent call last):
...
NotImplementedError: multiple generators [x, exp(I*x)]
No algorithms are implemented to solve equation exp(I*x)
```

In fact, `solve` makes *no guarantees whatsoever* about the completeness of the solutions it finds. Much of `solve` is heuristics, which may find some solutions to an equation or system of equations, but not all of them.

---

`solve` can also solve systems of equations. Pass a list of equations and a list of variables to solve for.

```
>>> solve([x - y + 2, x + y - 3], [x, y])
{x: 1/2, y: 5/2}
>>> solve([x*y - 7, x + y - 6], [x, y])
- \sqrt{2} + 3, \sqrt{2} + 3, \sqrt{2} + 3, - \sqrt{2} + 3
```

---

**Note:** The type of the output of `solve` when solving systems of equations varies depending on the type of the input. If you want a consistent interface, pass `dict=True`.

```
>>> solve([x - y + 2, x + y - 3], [x, y], dict=True)
[{x: 1/2, y: 5/2}]
>>> solve([x*y - 7, x + y - 6], [x, y], dict=True)
x: - \sqrt{2} + 3, y: \sqrt{2} + 3, x: \sqrt{2} + 3, y: - \sqrt{2} + 3
```

---

`solve` reports each solution only once. To get the solutions of a polynomial including multiplicity use `roots`.

```
>>> solve(x**3 - 6*x**2 + 9*x, x)
[0, 3]
>>> roots(x**3 - 6*x**2 + 9*x, x)
{0: 1, 3: 2}
```

The output `{0: 1, 3: 2}` of `roots` means that 0 is a root of multiplicity 1 and 3 is a root of multiplicity 2.

### 2.8.3 Solving Differential Equations

To solve differential equations, use `dsolve`. First, create an undefined function by passing `cls=Function` to the `symbols` function.

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

`f` and `g` are now undefined functions. We can call `f(x)`, and it will represent an unknown function.

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

Derivatives of `f(x)` are unevaluated.

```
>>> f(x).diff(x)
d
--(f(x))
dx
```

(see the *Derivatives* (page 37) section for more on derivatives).

To represent the differential equation  $f''(x) - 2f'(x) + f(x) = \sin(x)$ , we would thus use

```
>>> diffeq = Eq(f(x).diff(x, 2) - 2*f(x).diff(x) + f(x), sin(x))
>>> diffeq

$$\frac{d^2}{dx^2}(f(x)) - 2\frac{d}{dx}(f(x)) + f(x) = \sin(x)$$

```

To solve the ODE, pass it and the function to solve for to `dsolve`.

```
>>> dsolve(diffeq, f(x))

$$f(x) = \frac{(C_1 + C_2x)e^{-2x}}{2} + \frac{\cos(x)}{e^{-2x}}$$

```

`dsolve` returns an instance of `Eq`. This is because in general, solutions to differential equations cannot be solved explicitly for the function.

```
>>> dsolve(f(x).diff(x)*(1 - sin(f(x))), f(x))
f(x) + cos(f(x)) = C1
```

The arbitrary constants in the solutions from `dsolve` are symbols of the form `C1`, `C2`, `C3`, and so on.

## 2.9 Matrices

```
>>> from sympy import *
>>> init_printing(use_unicode=True)
```

To make a matrix in SymPy, use the `Matrix` object. A matrix is constructed by providing a list of row vectors that make up the matrix. For example, to construct the matrix

$$\begin{bmatrix} 1 & -1 \\ 3 & 4 \\ 0 & 2 \end{bmatrix}$$

use

```
>>> Matrix([[1, -1], [3, 4], [0, 2]])
1  -1
3   4
0   2
```

To make it easy to make column vectors, a list of elements is considered to be a column vector.

```
>>> Matrix([1, 2, 3])
1
2
3
```

Matrices are manipulated just like any other object in SymPy or Python.

```
>>> M = Matrix([[1, 2, 3], [3, 2, 1]])
>>> N = Matrix([0, 1, 1])
>>> M*N
5
3
```

One important thing to note about SymPy matrices is that, unlike every other object in SymPy, they are mutable. This means that they can be modified in place, as we will see below. The downside to this is that `Matrix` cannot be used in places that require immutability, such as inside other SymPy expressions or as keys to dictionaries. If you need an immutable version of `Matrix`, use `ImmutableMatrix`.

### 2.9.1 Basic Operations

#### Shape

Here are some basic operations on `Matrix`. To get the shape of a matrix use `shape`

```
>>> M = Matrix([[1, 2, 3], [-2, 0, 4]])
>>> M
1 2 3
-2 0 4
>>> M.shape
(2, 3)
```

## Accessing Rows and Columns

To get an individual row or column of a matrix, use `row` or `col`. For example, `M.row(0)` will get the first row. `M.col(-1)` will get the last column.

```
>>> M.row(0)
[1 2 3]
>>> M.col(-1)
3
4
```

## Deleting and Inserting Rows and Columns

To delete a row or column, use `row_del` or `col_del`. These operations will modify the Matrix **in place**.

```
>>> M.col_del(0)
>>> M
2 3
0 4
>>> M.row_del(1)
>>> M
[2 3]
```

To insert rows or columns, use `row_insert` or `col_insert`. These operations **do not** operate in place.

```
>>> M
[2 3]
>>> M = M.row_insert(1, Matrix([[0, 4]]))
>>> M
2 3
0 4
>>> M = M.col_insert(0, Matrix([1, -2]))
>>> M
1 2 3
-2 0 4
```

Unless explicitly stated, the methods mentioned below do not operate in place. In general, a method that does not operate in place will return a new `Matrix` and a method that does operate in place will return `None`.

## 2.9.2 Basic Methods

As noted above, simple operations like addition and multiplication are done just by using `+`, `*`, and `**`. To find the inverse of a matrix, just raise it to the `-1` power.

```
>>> M = Matrix([[1, 3], [-2, 3]])
>>> N = Matrix([[0, 3], [0, 7]])
>>> M + N
1  6
-2 10
>>> M*N
0 24

0 15
>>> 3*M
3  9

-6 9
>>> M**2
-5 12

-8 3
>>> M**-1
1/3 -1/3

2/9 1/9
>>> N**-1
Traceback (most recent call last):
...
ValueError: Matrix det == 0; not invertible.
```

To take the transpose of a Matrix, use `T`.

```
>>> M = Matrix([[1, 2, 3], [4, 5, 6]])
>>> M
1 2 3
4 5 6
>>> M.T
1 4
2 5
3 6
```

## 2.9.3 Matrix Constructors

Several constructors exist for creating common matrices. To create an identity matrix, use `eye`. `eye(n)` will create an  $n \times n$  identity matrix.

```
>>> eye(3)
1 0 0
0 1 0
0 0 1
```

```
>>> eye(4)
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

To create a matrix of all zeros, use `zeros`. `zeros(n, m)` creates an  $n \times m$  matrix of 0s.

```
>>> zeros(2, 3)
0 0 0
0 0 0
```

Similarly, `ones` creates a matrix of ones.

```
>>> ones(3, 2)
1 1
1 1
1 1
```

To create diagonal matrices, use `diag`. The arguments to `diag` can be either numbers or matrices. A number is interpreted as a  $1 \times 1$  matrix. The matrices are stacked diagonally. The remaining elements are filled with 0s.

```
>>> diag(1, 2, 3)
1 0 0
0 2 0
0 0 3
>>> diag(-1, ones(2, 2), Matrix([5, 7, 5]))
-1 0 0 0
0 1 1 0
0 1 1 0
0 0 0 5
0 0 0 7
0 0 0 5
```

## 2.9.4 Advanced Methods

### Determinant

To compute the determinant of a matrix, use `det`.

```
>>> M = Matrix([[1, 0, 1], [2, -1, 3], [4, 3, 2]])
>>> M
1 0 1
```

```
2 -1 3  
4 3 2  
>>> M.det()  
-1
```

## RREF

To put a matrix into reduced row echelon form, use `rref`. `rref` returns a tuple of two elements. The first is the reduced row echelon form, and the second is a list of indices of the pivot columns.

```
>>> M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])  
>>> M  
1 0 1 3  
2 3 4 7  
-1 -3 -3 -4  
>>> M.rref()  
1 0 1 3 , [0, 1]  
  
0 1 2/3 1/3  
  
0 0 0 0
```

---

**Note:** The first element of the tuple returned by `rref` is of type `Matrix`. The second is of type `list`.

---

## Nullspace

To find the nullspace of a matrix, use `nullspace`. `nullspace` returns a list of column vectors that span the nullspace of the matrix.

```
>>> M = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])  
>>> M  
1 2 3 0 0  
4 10 0 0 1  
>>> M.nullspace()  
-15, 0, 1  
  
6 0 -1/2  
  
1 0 0  
  
0 1 0  
  
0 0 1
```

## Eigenvalues, Eigenvectors, and Diagonalization

To find the eigenvalues of a matrix, use `eigens`. `eigens` returns a dictionary of eigenvalue:algebraic multiplicity pairs (similar to the output of `roots` (page 44)).

```
>>> M = Matrix([[3, -2, 4, -2], [5, 3, -3, -2], [5, -2, 2, -2], [5, -2, -3, 3]])  
>>> M  
3  -2   4   -2  
5   3   -3   -2  
5   -2   2   -2  
5   -2   -3   3  
>>> M.eigenvals()  
{-2: 1, 3: 1, 5: 2}
```

This means that  $M$  has eigenvalues -2, 3, and 5, and that the eigenvalues -2 and 3 have algebraic multiplicity 1 and that the eigenvalue 5 has algebraic multiplicity 2.

To find the eigenvectors of a matrix, use `eigenvecs`. `eigenvecs` returns a list of tuples of the form (`eigenvalue:algebraic multiplicity, [eigenvectors]`).

```
>>> M.eigenvecs()  
-2, 1, 0, 3, 1, 1, 5, 2, 1, 0  
  
1       1       1   -1  
  
1       1       1   0  
  
1       1       0   1
```

This shows us that, for example, the eigenvalue 5 also has geometric multiplicity 2, because it has two eigenvectors. Because the algebraic and geometric multiplicities are the same for all the eigenvalues,  $M$  is diagonalizable.

To diagonalize a matrix, use `diagonalize`. `diagonalize` returns a tuple  $(P, D)$ , where  $D$  is diagonal and  $M = PDP^{-1}$ .

```
>>> P, D = M.diagonalize()  
>>> P  
0   1   1   0  
  
1   1   1   -1  
  
1   1   1   0  
  
1   1   0   1  
>>> D  
-2   0   0   0  
  
0   3   0   0  
  
0   0   5   0  
  
0   0   0   5  
>>> P*D*P**-1  
3   -2   4   -2  
  
5   3   -3   -2  
5   -2   2   -2  
5   -2   -3   3
```

```
>>> P*D*D**-1 == M
True
```

### Quick Tip

`lambda` is a reserved keyword in Python, so to create a Symbol called  $\lambda$ , while using the same names for SymPy Symbols and Python variables, use `lamda` (without the `b`). It will still pretty print as  $\lambda$ .

Note that since `eigenvects` also includes the eigenvalues, you should use it instead of `eigenvals` if you also want the eigenvectors. However, as computing the eigenvectors may often be costly, `eigenvals` should be preferred if you only wish to find the eigenvalues.

If all you want is the characteristic polynomial, use `charpoly`. This is more efficient than `eigenvals`, because sometimes symbolic roots can be expensive to calculate.

```
>>> lamda = symbols('lamda')
>>> p = M.charpoly(lamda)
>>> factor(p)

$$(\lambda - 5)^2 (\lambda - 3) (\lambda + 2)$$

```

## 2.10 Advanced Expression Manipulation

In this section, we discuss some ways that we can perform advanced manipulation of expressions.

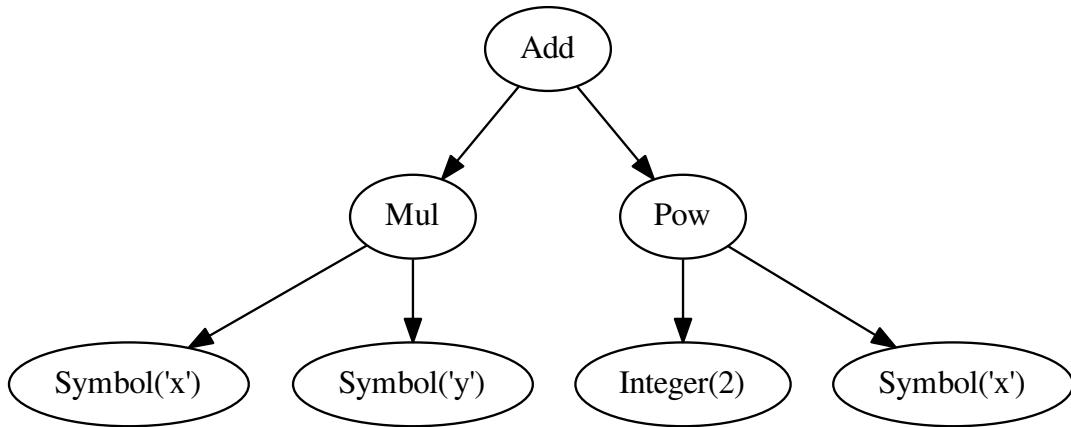
### 2.10.1 Understanding Expression Trees

Before we can do this, we need to understand how expressions are represented in SymPy. A mathematical expression is represented as a tree. Let us take the expression  $x^2 + xy$ , i.e., `x**2 + x*y`. We can see what this expression looks like internally by using `srepr`

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

>>> expr = x**2 + x*y
>>> srepr(expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Symbol('x'), Symbol('y')))"
```

The easiest way to tear this apart is to look at a diagram of the expression tree:



---

**Note:** The above diagram was made using [Graphviz](#) and the `dotprint` (page 871) function.

---

First, let's look at the leaves of this tree. Symbols are instances of the class `Symbol`. While we have been doing

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

we could have also done

```
>>> x = Symbol('x')
```

Either way, we get a `Symbol` with the name "x"<sup>2</sup>. For the number in the expression, 2, we got `Integer(2)`. `Integer` is the SymPy class for integers. It is similar to the Python built-in type `int`, except that `Integer` plays nicely with other SymPy types.

When we write `x**2`, this creates a `Pow` object. `Pow` is short for "power".

```
>>> srepr(x**2)
"Pow(Symbol('x'), Integer(2))"
```

We could have created the same object by calling `Pow(x, 2)`

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

Note that in the `srepr` output, we see `Integer(2)`, the SymPy version of integers, even though technically, we input 2, a Python int. In general, whenever you combine a SymPy object with a non-SymPy object via some function or operation, the non-SymPy object will be converted into a SymPy object. The function that does this is `sympify`<sup>3</sup>.

```
>>> type(2)
<... 'int'>
```

---

<sup>2</sup> We have been using `symbols` instead of `Symbol` because it automatically splits apart strings into multiple `Symbol`s. `symbols('x y z')` returns a tuple of three `Symbol`s. `Symbol('x y z')` returns a single `Symbol` called `x y z`.

<sup>3</sup> Technically, it is an internal function called `_sympify`, which differs from `sympify` in that it does not convert strings. `x + '2'` is not allowed.

```
>>> type(sympify(2))
<class 'sympy.core.numbers.Integer'>
```

We have seen that  $x^{**}2$  is represented as  $\text{Pow}(x, 2)$ . What about  $x*y$ ? As we might expect, this is the multiplication of  $x$  and  $y$ . The SymPy class for multiplication is  $\text{Mul}$ .

```
>>> srepr(x*y)
"Mul(Symbol('x'), Symbol('y'))"
```

Thus, we could have created the same object by writing  $\text{Mul}(x, y)$ .

```
>>> Mul(x, y)
x*y
```

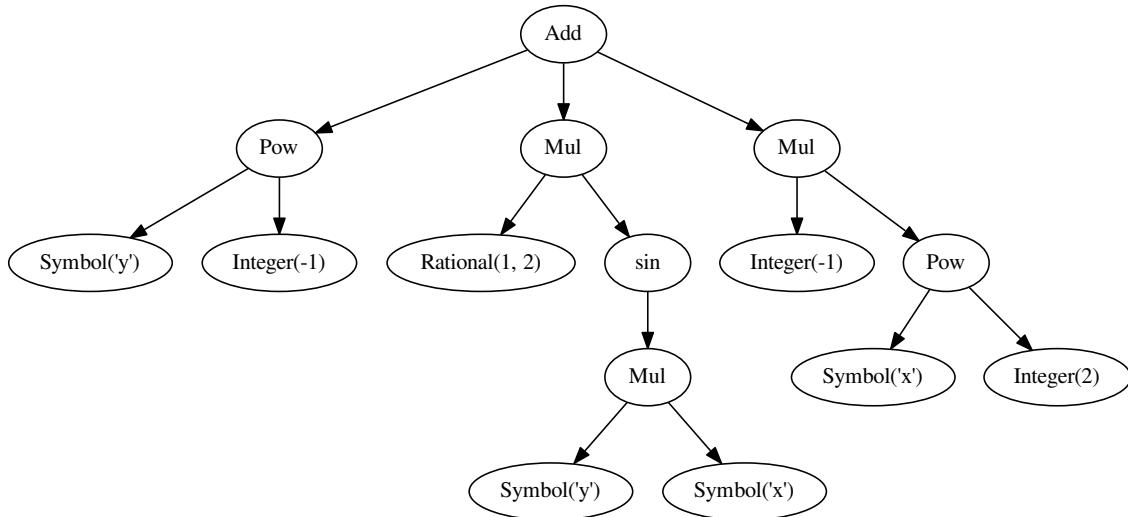
Now we get to our final expression,  $x^{**}2 + x*y$ . This is the addition of our last two objects,  $\text{Pow}(x, 2)$ , and  $\text{Mul}(x, y)$ . The SymPy class for addition is  $\text{Add}$ , so, as you might expect, to create this object, we use  $\text{Add}(\text{Pow}(x, 2), \text{Mul}(x, y))$ .

```
>>> Add(Pow(x, 2), Mul(x, y))
x**2 + x*y
```

SymPy expression trees can have many branches, and can be quite deep or quite broad. Here is a more complicated example

```
>>> expr = sin(x*y)/2 - x**2 + 1/y
>>> srepr(expr)
"Add(Mul(Integer(-1), Pow(Symbol('x'), Integer(2))), Mul(Rational(1, 2),
sin(Mul(Symbol('x'), Symbol('y')))), Pow(Symbol('y'), Integer(-1)))"
```

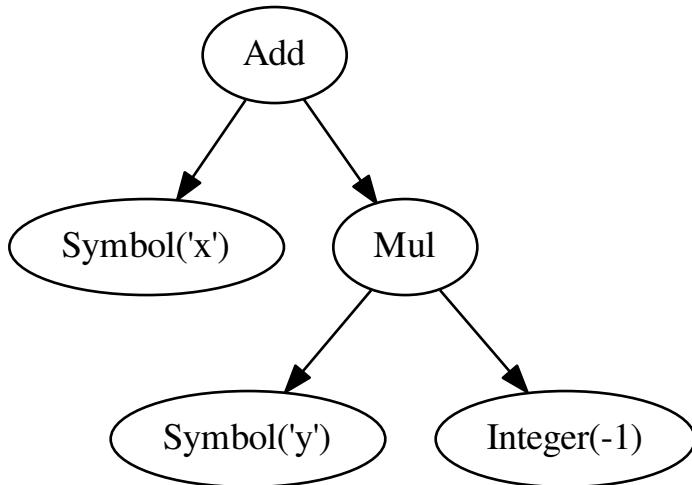
Here is a diagram



This expression reveals some interesting things about SymPy expression trees. Let's go through them one by one.

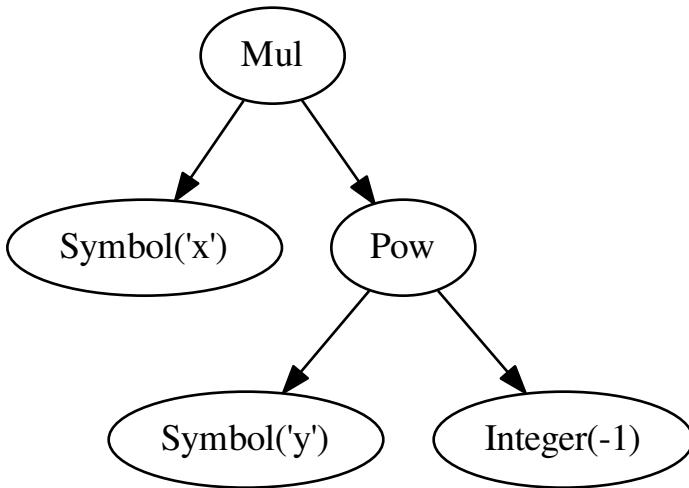
Let's first look at the term  $x^{**}2$ . As we expected, we see  $\text{Pow}(x, 2)$ . One level up, we see we have  $\text{Mul}(-1, \text{Pow}(x, 2))$ . There is no subtraction class in SymPy.  $x - y$  is represented as  $x + -y$ , or, more completely,  $x + -1*y$ , i.e.,  $\text{Add}(x, \text{Mul}(-1, y))$ .

```
>>> expr = x - y
>>> srepr(x - y)
"Add(Symbol('x'), Mul(Integer(-1), Symbol('y')))"
```



Next, look at  $1/y$ . We might expect to see something like `Div(1, y)`, but similar to subtraction, there is no class in SymPy for division. Rather, division is represented by a power of -1. Hence, we have `Pow(y, -1)`. What if we had divided something other than 1 by  $y$ , like  $x/y$ ? Let's see.

```
>>> expr = x/y
>>> srepr(expr)
"Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))"
```



We see that  $x/y$  is represented as  $x*y**-1$ , i.e., `Mul(x, Pow(y, -1))`.

Finally, let's look at the `sin(x*y)/2` term. Following the pattern of the previous example, we might expect to see `Mul(sin(x*y), Pow(Integer(2), -1))`. But instead, we have `Mul(Rational(1, 2), sin(x*y))`. Rational numbers are always combined into a single term in a multiplication, so that when we divide by 2, it is represented as multiplying by 1/2.

Finally, one last note. You may have noticed that the order we entered our expression and the order that it came out from `srepr` or in the graph were different. You may have also noticed this phenomenon earlier in the tutorial. For example

```
>>> 1 + x
x + 1
```

This because in SymPy, the arguments of the commutative operations `Add` and `Mul` are stored in an arbitrary (but consistent!) order, which is independent of the order inputted (if you're worried about noncommutative multiplication, don't be. In SymPy, you can create noncommutative Symbols using `Symbol('A', commutative=False)`, and the order of multiplication for noncommutative Symbols is kept the same as the input). Furthermore, as we shall see in the next section, the printing order and the order in which things are stored internally need not be the same either.

### Quick Tip

The way an expression is represented internally and the way it is printed are often not the same.

In general, an important thing to keep in mind when working with SymPy expression trees is this: the internal representation of an expression and the way it is printed need not be the same. The same is true for the input form. If some expression manipulation algorithm is not working in the way you expected it to, chances are, the internal representation of the object is different from what you thought it was.

## 2.10.2 Recursing through an Expression Tree

Now that you know how expression trees work in SymPy, let's look at how to dig our way through an expression tree. Every object in SymPy has two very important attributes, `func`, and `args`.

### func

`func` is the head of the object. For example, `(x*y).func` is `Mul`. Usually it is the same as the class of the object (though there are exceptions to this rule).

Two notes about `func`. First, the class of an object need not be the same as the one used to create it. For example

```
>>> expr = Add(x, x)
>>> expr.func
<class 'sympy.core.mul.Mul'>
```

We created `Add(x, x)`, so we might expect `expr.func` to be `Add`, but instead we got `Mul`. Why is that? Let's take a closer look at `expr`.

```
>>> expr
2*x
```

`Add(x, x)`, i.e.,  $x + x$ , was automatically converted into `Mul(2, x)`, i.e., `2*x`, which is a `Mul`. SymPy classes make heavy use of the `__new__` class constructor, which, unlike `__init__`, allows a different class to be returned from the constructor.

Second, some classes are special-cased, usually for efficiency reasons<sup>4</sup>.

```
>>> Integer(2).func
<class 'sympy.core.numbers.Integer'>
>>> Integer(0).func
<class 'sympy.core.numbers.Zero'>
>>> Integer(-1).func
<class 'sympy.core.numbers.NegativeOne'>
```

For the most part, these issues will not bother us. The special classes `Zero`, `One`, `NegativeOne`, and so on are subclasses of `Integer`, so as long as you use `isinstance`, it will not be an issue.

### args

`args` are the top-level arguments of the object. `(x*y).args` would be `(x, y)`. Let's look at some examples

```
>>> expr = 3*y**2*x
>>> expr.func
<class 'sympy.core.mul.Mul'>
>>> expr.args
(3, x, y**2)
```

From this, we can see that `expr == Mul(3, y**2, x)`. In fact, we can see that we can completely reconstruct `expr` from its `func` and its `args`.

<sup>4</sup> Classes like `One` and `Zero` are singletonized, meaning that only one object is ever created, no matter how many times the class is called. This is done for space efficiency, as these classes are very common. For example, `Zero` might occur very often in a sparse matrix represented densely. As we have seen, `NegativeOne` occurs any time we have  $-x$  or  $1/x$ . It is also done for speed efficiency because singletonized objects can be compared by `is`. The unique objects for each singletonized class can be accessed from the `S` object.

```
>>> expr.func(*expr.args)
3*x*y**2
>>> expr == expr.func(*expr.args)
True
```

Note that although we entered  $3*y**2*x$ , the `args` are  $(3, x, y**2)$ . In a `Mul`, the Rational coefficient will come first in the `args`, but other than that, the order of everything else follows no special pattern. To be sure, though, there is an order.

```
>>> expr = y**2*3*x
>>> expr.args
(3, x, y**2)
```

`Mul`'s `args` are sorted, so that the same `Mul` will have the same `args`. But the sorting is based on some criteria designed to make the sorting unique and efficient that has no mathematical significance.

The `srepr` form of our `expr` is `Mul(3, x, Pow(y, 2))`. What if we want to get at the `args` of `Pow(y, 2)`. Notice that the  $y^{**2}$  is in the third slot of `expr.args`, i.e., `expr.args[2]`.

```
>>> expr.args[2]
y**2
```

So to get the `args` of this, we call `expr.args[2].args`.

```
>>> expr.args[2].args
(y, 2)
```

Now what if we try to go deeper. What are the args of `y`. Or `2`. Let's see.

```
>>> y.args
()
>>> Integer(2).args
()
```

They both have empty `args`. In SymPy, empty `args` signal that we have hit a leaf of the expression tree.

So there are two possibilities for a SymPy expression. Either it has empty `args`, in which case it is a leaf node in any expression tree, or it has `args`, in which case, it is a branch node of any expression tree. When it has `args`, it can be completely rebuilt from its `func` and its `args`. This is expressed in the key invariant.

### Key Invariant

Every well-formed SymPy expression must either have empty `args` or satisfy `expr == expr.func(*expr.args)`.

(Recall that in Python if `a` is a tuple, then `f(*a)` means to call `f` with arguments from the elements of `a`, e.g., `f(*(1, 2, 3))` is the same as `f(1, 2, 3)`.)

This key invariant allows us to write simple algorithms that walk expression trees, change them, and rebuild them into new expressions.

### Walking the Tree

With this knowledge, let's look at how we can recurse through an expression tree. The nested nature of `args` is a perfect fit for recursive functions. The base case will be empty `args`. Let's write a simple function that goes through an expression and prints all the `args` at each level.

```
>>> def pre(expr):
...     print(expr)
...     for arg in expr.args:
...         pre(arg)
```

See how nice it is that () signals leaves in the expression tree. We don't even have to write a base case for our recursion; it is handled automatically by the for loop.

Let's test our function.

```
>>> expr = x*y + 1
>>> pre(expr)
x*y + 1
1
x*y
x
y
```

Can you guess why we called our function `pre`? We just wrote a pre-order traversal function for our expression tree. See if you can write a post-order traversal function.

Such traversals are so common in SymPy that the generator functions `preorder_traversal` and `postorder_traversal` are provided to make such traversals easy. We could have also written our algorithm as

```
>>> for arg in preorder_traversal(expr):
...     print(arg)
x*y + 1
1
x*y
x
y
```

---

## Modules Reference

---

Because every feature of SymPy must have a test case, when you are not sure how to use something, just look into the `tests/` directories, find that feature and read the tests for it, that will tell you everything you need to know.

Most of the things are already documented though in this document, that is automatically generated using SymPy's docstrings.

Click the “modules” (`modindex`) link in the top right corner to easily access any SymPy module, or use this contents:

### 3.1 SymPy Core

#### 3.1.1 `sympify`

##### `sympify`

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

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

For example, it will convert Python ints into instance of `sympy.Rational`, floats into instances of `sympy.Float`, etc. It is also able to coerce symbolic expressions which inherit from `Basic`. This can be useful in cooperation with SAGE.

**It currently accepts as arguments:**

- any object defined in `sympy`
- standard numeric python types: `int`, `long`, `float`, `Decimal`
- strings (like “0.09” or “2e-19”)
- booleans, including `None` (will leave `None` unchanged)
- lists, sets or tuples containing any of the above

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

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

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

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

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

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

## Notes

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

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

### 3.1.2 assumptions

This module contains the machinery handling assumptions.

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

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

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

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

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

Here follows a list of possible assumption names:

**commutative** object commutes with any other object with respect to multiplication operation.

**complex** object can have only values from the set of complex numbers.

**imaginary** object value is a number that can be written as a real number multiplied by the imaginary unit I. See [R29] (page 1233).

**real** object can have only values from the set of real numbers <sup>1</sup>.

**extended\_real** object can have only values on the extended real number line <sup>2</sup>.

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

**odd, even** object can have only values from the set of odd (even) integers [R28] (page 1233).

**prime** object is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [R32] (page 1233).

**composite** object is a positive integer that has at least one positive divisor other than 1 or the number itself. See [R30] (page 1233).

**zero, nonzero** object is zero (not zero).

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

**algebraic** object can have only values from the set of algebraic numbers <sup>3</sup>.

**transcendental** object can have only values from the set of transcendental numbers <sup>4</sup>.

**irrational** object value cannot be represented exactly by Rational, see [R31] (page 1233).

**finite, infinite** object absolute value is bounded (is value is arbitrarily large). See [R33] (page 1233), [R34] (page 1233), [R35] (page 1233).

**negative, nonnegative** object can have only negative (only nonnegative) values [R27] (page 1233).

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

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

## Examples

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

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Real\\_number](http://en.wikipedia.org/wiki/Real_number)

<sup>2</sup> [http://en.wikipedia.org/wiki/Extended\\_real\\_number\\_line](http://en.wikipedia.org/wiki/Extended_real_number_line)

<sup>3</sup> [http://en.wikipedia.org/wiki/Algebraic\\_number](http://en.wikipedia.org/wiki/Algebraic_number)

<sup>4</sup> [http://en.wikipedia.org/wiki/Transcendental\\_number](http://en.wikipedia.org/wiki/Transcendental_number)

## See Also

See also:

`sympy.core.numbers.ImaginaryUnit` (page 108)    `sympy.core.numbers.Zero` (page 104)  
`sympy.core.numbers.One` (page 104)

## Notes

Assumption values are stored in `obj._assumptions` dictionary or are returned by getter methods (with property decorators) or are attributes of objects/classes.

## References

### 3.1.3 cache

#### cacheit

`sympy.core.cache.cacheit(func)`

### 3.1.4 basic

#### Basic

`class sympy.core.basic.Basic`

Base class for all objects in SymPy.

Conventions:

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

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

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

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

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

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

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

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

#### args

Returns a tuple of arguments of ‘self’.

## Notes

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

## Examples

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

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

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

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

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

### as\_content\_primitive(*radical=False*)

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

See docstring of Expr.as\_content\_primitive

### as\_poly(\*gens, \*\*args)

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

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

>>> print((x**2 + x*y).as_poly())
Poly(x**2 + x*y, x, y, domain='ZZ')

>>> print((x**2 + x*y).as_poly(x, y))
Poly(x**2 + x*y, x, y, domain='ZZ')

>>> print((x**2 + sin(y)).as_poly(x, y))
None
```

### assumptions0

Return object *type* assumptions.

For example:

```
Symbol('x', extended_real=True) Symbol('x', integer=True)
```

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

## Examples

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

atoms(*types)
```

Returns the atoms that form the current object.

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

## Examples

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

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
set([1, 2])

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
set([1, 2, pi])

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
set([1, 2, I, pi])
```

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

The type can be given implicitly, too:

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

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

```
>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
set([1])

>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
set([1, 2])
```

Finally, arguments to atoms() can select more than atomic atoms: any sympy type (loaded in core/\_init\_.py) can be listed as an argument and those types of “atoms” as found in scanning the arguments of the expression recursively:

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

>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
set([I*pi, 2*sin(y + I*pi)])
```

#### canonical\_variables

Return a dictionary mapping any variable defined in `self.variables` as underscore-suffixed numbers corresponding to their position in `self.variables`. Enough underscores are added to ensure that there will be no clash with existing free symbols.

#### Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: 0_}
```

#### classmethod class\_key()

Nice order of classes.

#### compare(*other*)

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

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

#### Examples

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

#### count(*query*)

Count the number of matching subexpressions.

#### count\_ops(*visual=None*)

wrapper for `count_ops` that returns the operation count.

#### doit(\*\*hints)

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

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

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

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

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

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

### Examples

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

>>> u = Dummy('u')

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

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

find(query, group=False)
Find all subexpressions matching a query.
```

### free\_symbols

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

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

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

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

Create a new object from an iterable.

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

### Examples

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

### func

The top-level function in an expression.

The following should hold for all objects:

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

### Examples

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

`has(*args, **kwargs)`

Test whether any subexpression matches any of the patterns.

### Examples

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

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

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

`is_comparable`

Return `True` if self can be computed to a real number with precision, else `False`.

### Examples

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

`match(pattern, old=False)`

Pattern matching.

Wild symbols match all.

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

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

### Examples

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

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

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

`matches(expr, repl_dict={}, old=False)`

Helper method for `match()` that looks for a match between Wild symbols in `self` and expressions in `expr`.

### Examples

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

`rcall(*args)`

Apply on the argument recursively through the expression tree.

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

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

however you can use

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

```
replace(query, value, map=False, simultaneous=True, exact=False)
```

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

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

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to False. In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is True, then the match will only succeed if non-zero values are received for each Wild that appears in the match pattern.

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

**See also:**

`subs` ([page 71](#)) substitution of subexpressions as defined by the objects themselves.

`xreplace` ([page 73](#)) exact node replacement in expr tree; also capable of using matching rules

## Examples

Initial setup

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

### 1.1. type -> type `obj.replace(type, newtype)`

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

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

### 1.2. type -> func `obj.replace(type, func)`

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

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

### 2.1. pattern -> expr `obj.replace(pattern(wild), expr(wild))`

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

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

When the default value of `False` is used with patterns that have more than one Wild symbol, non-intuitive results may be obtained:

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

For this reason, the `exact` option can be used to make the replacement only when the match gives non-zero values for all Wild symbols:

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

## 2.2. pattern -> func obj.replace(pattern(wild), lambda wild: expr(wild))

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

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

## 3.1. func -> func obj.replace(filter, func)

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

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

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

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

```
rewrite(*args, **hints)
```

Rewrite functions in terms of other functions.

Rewrites expression containing applications of functions of one kind in terms of functions of different kind. For example you can rewrite trigonometric functions as complex exponentials or combinatorial functions as gamma function.

As a pattern this function accepts a list of functions to to rewrite (instances of `DefinedFunction` class). As rule you can use string or a destination function instance (in this case `rewrite()` will use the `str()` function).

There is also the possibility to pass hints on how to rewrite the given expressions. For now there is only one such hint defined called ‘deep’. When ‘deep’ is set to `False` it will forbid functions to rewrite their contents.

### Examples

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

Unspecified pattern:

```
>>> sin(x).rewrite(exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a single function:

```
>>> sin(x).rewrite(sin, exp)
-I*(exp(I*x) - exp(-I*x))/2
```

Pattern as a list of functions:

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

`sort_key(*args, **kwargs)`

Return a sort key.

### Examples

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

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

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

`subs(*args, **kwargs)`

Substitutes old for new in an expression after sympifying args.

`args` is either:

- two arguments, e.g. `foo.subs(old, new)`
- one iterable argument, e.g. `foo.subs(iterable)`. The iterable may be
  - o an iterable container with (old, new) pairs. In this case the replacements are processed in the order given with successive patterns possibly affecting replacements already made.

o a dict or set whose key/value items correspond to old/new pairs. In this case the old/new pairs will be sorted by op count and in case of a tie, by number of args and the default\_sort\_key. The resulting sorted list is then processed as an iterable container (see previous).

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

**See also:**

`replace` (page 68) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

`xreplace` (page 73) exact node replacement in expr tree; also capable of using matching rules

`sympy.core.evalf.EvalfMixin.evalf` (page 149) calculates the given formula to a desired level of precision

### Examples

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

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

To replace only the  $x^{**}2$  but not the  $x^{**}4$ , use `xreplace`:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.33333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.33333333333333314830
```

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

### xreplace(rule)

Replace occurrences of objects within the expression.

**Parameters rule** : dict-like

Expresses a replacement rule

**Returns xreplace** : the result of the replacement

**See also:**

[replace](#) ([page 68](#)) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

[subs](#) ([page 71](#)) substitution of subexpressions as defined by the objects themselves.

### Examples

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

```
>>> (1 + x*y).xreplace({x:pi, y:2})
1 + 2*pi
```

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

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

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

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

Trying to replace x with an expression raises an error:

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

## Atom

```
class sympy.core.basic.Atom
```

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

### Examples

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

## 3.1.5 core

### 3.1.6 singleton

#### S

```
sympy.core.singleton.S
```

### 3.1.7 expr

### 3.1.8 Expr

```
class sympy.core.expr.Expr
```

Base class for algebraic expressions.

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

**See also:**

`sympy.core.basic.Basic` (page 62)

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

See the apart function in `sympy.polys`

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

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

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

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

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

The arg is always treated as a Mul:

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

`as_coeff_Add()`

Efficiently extract the coefficient of a summation.

`as_coeff_Mul(rational=False)`

Efficiently extract the coefficient of a product.

`as_coeff_add(*deps)`

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

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

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

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

- if you know self is an Add and want only the head, use `self.args[0]`;
- if you don't want to process the arguments of the tail but need the tail then use `self.as_two_terms()` which gives the head and tail.

•if you want to split self into an independent and dependent parts use `self.as_independent(*deps)`

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

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

`as_coeff_mul(*deps, **kwargs)`

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

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

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

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

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

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

`as_coefficient(expr)`

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

**See also:**

[coeff \(page 83\)](#) return sum of terms have a given factor

[as\\_coeff\\_Add \(page 75\)](#) separate the additive constant from an expression

[as\\_coeff\\_Mul \(page 75\)](#) separate the multiplicative constant from an expression

[as\\_independent \(page 79\)](#) separate x-dependent terms/factors from others

`sympy.polys.polytools.Poly.coeff_monomial` (page 684) efficiently find the single coefficient of a monomial in Poly

`sympy.polys.polytools.Poly.nth` (page 703) like `coeff_monomial` but powers of monomial terms are used

### Examples

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

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

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

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

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

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

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

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

### as\_coefficients\_dict()

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

### Examples

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

```
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

#### as\_content\_primitive(*radical=False*)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need no be in canonical form and should try to preserve the underlying structure if possible (i.e. `expand_mul` should not be applied to self).

#### Examples

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

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

The `as_content_primitive` function is recursive and retains structure:

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

Integer powers will have Rationals extracted from the base:

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

Terms may end up joining once their `as_content_primitives` are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((5*(x*(1 + y)) + 2.0*x*(3 + 3*y))**2).as_content_primitive()
(1, 121.0*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

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

#### as\_expr(\*gens)

Convert a polynomial to a SymPy expression.

#### Examples

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

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

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

as_independent(*deps, **hint)
```

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

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

The only non-naive thing that is done here is to respect noncommutative ordering of variables.

The returned tuple (i, d) has the following interpretation:

- i will have no variable that appears in deps
- d will be 1 or else have terms that contain variables that are in deps
- if self is an Add then self = i + d
- if self is a Mul then self = i\*d
- if self is anything else, either tuple (self, S.One) or (S.One, self) is returned.

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

## Examples

– self is an Add

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

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

– self is a Mul

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

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

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

– self is anything else:

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

– force self to be treated as an Add:

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

– force self to be treated as a Mul:

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

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

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

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

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

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

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

See also: `.separatevars()`, `.expand(log=True)`, `.as_two_terms()`,  
`.as_coeff_add()`,  
`.as_coeff_mul()`

`as_leading_term(*args, **kwargs)`

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

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

## Examples

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

`as_numer_denom()`

expression -> a/b -> a, b

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

See also:

`normal` (page 91) return a/b instead of a, b

`as_ordered_factors(order=None)`

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

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

Transform an expression to an ordered list of terms.

## Examples

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

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

**as\_powers\_dict()**

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

**as\_real\_imag(deep=True, \*\*hints)**

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

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

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

>>> x, y = symbols('x,y', extended_real=True)

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

>>> from sympy.abc import z, w

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

**as\_terms()**

Transform an expression to a list of terms.

**aseries(x, n=6, bound=0, hir=False)**

Returns asymptotic expansion for “self”. See [R38] (page 1233)

This is equivalent to `self.series(x, oo, n)`

Use the `hir` parameter to produce hierarchical series. It stops the recursion at an early level and may provide nicer and more useful results.

If the most rapidly varying subexpression of a given expression `f` is `f` itself, the algorithm tries to find a normalised representation of the mrv set and rewrites `f` using this normalised representation. Use the `bound` parameter to give limit on rewriting coefficients in its normalised form.

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

## Notes

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

## References

[R36] (page 1233), [R37] (page 1233), [R38] (page 1233)

## Examples

```
>>> from sympy import sin, exp
>>> from sympy.abc import x, y
>>> e = sin(1/x + exp(-x)) - sin(1/x)
>>> e.aseries(x)
(1/(24*x**4) - 1/(2*x**2) + 1 + O(x**(-6), (x, oo)))*exp(-x)
>>> e.aseries(x, n=3, hir=True)
-exp(-2*x)*sin(1/x)/2 + exp(-x)*cos(1/x) + O(exp(-3*x), (x, oo))

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

`cancel(*gens, **args)`

See the `cancel` function in `sympy.polys`

`coeff(x, n=1, right=False)`

Returns the coefficient from the term(s) containing `x**n` or `None`. If `n` is zero then all terms independent of `x` will be returned.

When `x` is noncommutative, the `coeff` to the left (default) or right of `x` can be returned. The keyword ‘`right`’ is ignored when `x` is commutative.

See also:

`as_coefficient` (page 76) separate the expression into a coefficient and factor

`as_coeff_Add` (page 75) separate the additive constant from an expression

`as_coeff_Mul` (page 75) separate the multiplicative constant from an expression

`as_independent` (page 79) separate `x`-dependent terms/factors from others

`sympy.polys.polytools.Poly.coeff_monomial` (page 684) efficiently find the single coefficient of a monomial in `Poly`

`sympy.polys.polytools.Poly.nth` (page 703) like `coeff_monomial` but powers of monomial terms are used

## Examples

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

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

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

You can select terms with no Rational coefficient:

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

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

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

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

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

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

The matching is exact:

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

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

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

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

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

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

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

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

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

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

`collect(sym, func=None, evaluate=True, exact=False, distribute_order_term=True)`

See the collect function in `sympy.simplify`

`combsimp()`

See the combsimp function in `sympy.simplify`

`compute_leading_term(x, logx=None)`

`as_leading_term` is only allowed for results of `.series()` This is a wrapper to compute a series first.

`could_extract_minus_sign()`

Canonical way to choose an element in the set {e, -e} where e is any expression. If the canonical element is e, we have `e.could_extract_minus_sign() == True`, else `e.could_extract_minus_sign() == False`.

For any expression, the set `{e.could_extract_minus_sign(), (-e).could_extract_minus_sign()}` must be `{True, False}`.

```
>>> from sympy.abc import x, y
>>> (x-y).could_extract_minus_sign() != (y-x).could_extract_minus_sign()
True
```

`count_ops(visual=None)`

wrapper for `count_ops` that returns the operation count.

`equals(other, failing_expression=False)`

Return True if `self == other`, False if it doesn't, or None. If `failing_expression` is True then the expression which did not simplify to a 0 will be returned instead of None.

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

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

`expand(*args, **kwargs)`

Expand an expression using hints.

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

`extract_additively(c)`

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

**See also:**

`extract_multiplicatively` (page 86), `coeff` (page 83), `as_coefficient` (page 76)

## Examples

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

Sometimes auto-expansion will return a less simplified result than desired; gcd\_terms might be used in such cases:

```
>>> from sympy import gcd_terms
>>> (4*x*(y + 1) + y).extract_additively(x)
4*x*(y + 1) + x*(4*y + 3) - x*(4*y + 4) + y
>>> gcd_terms(_)
x*(4*y + 3) + y

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

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

If allow\_half is True, also extract exp\_polar(I\*pi):

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

### extract\_multiplicatively(c)

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

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

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

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

>>> ((x*y)**3).extract_monomial(x**4 * y)

>>> (2*x).extract_monomial(2)
x

>>> (2*x).extract_monomial(3)

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

factor(*gens, **args)
See the factor() function in sympy.polys.polytools
```

`getO()`  
Returns the additive  $O(\dots)$  symbol if there is one, else None.

`getn()`  
Returns the order of the expression.  
The order is determined either from the  $O(\dots)$  term. If there is no  $O(\dots)$  term, it returns None.

### Examples

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

integrate(*args, **kwargs)
See the integrate function in sympy.integrals

invert(g)
See the invert function in sympy.polys

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

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

#### See also:

[is\\_rational\\_function](#) (page 90)

### References

- [http://en.wikipedia.org/wiki/Algebraic\\_expression](http://en.wikipedia.org/wiki/Algebraic_expression)

## Examples

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

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

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

`is_constant(*wrt, **flags)`

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

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

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

2) differentiation with respect to variables in ‘`wrt`’ (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression that is zero even though an expression is constant (see added test in `test_expr.py`). If all derivatives are zero then self is constant with respect to the given symbols.

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

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

## Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
```

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

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

#### is\_number

Returns True if ‘self’ has no free symbols. It will be faster than *if not self.free\_symbols*, however, since *is\_number* will fail as soon as it hits a free symbol.

#### Examples

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

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

#### is\_polynomial(\*syms)

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

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

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

## Examples

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

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

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

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

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

See also `.is_rational_function()`

### `is_rational_function(*syms)`

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

This function returns `False` for expressions that are “rational functions” with symbolic exponents. Thus, you should be able to call `.as_numer_denom()` and apply polynomial algorithms to the result for expressions for which this returns `True`.

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

## Examples

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

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

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

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

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

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

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

See also `is_algebraic_expr()`.

### `leadterm(x)`

Returns the leading term  $a \cdot x^b$  as a tuple  $(a, b)$ .

#### Examples

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

### `limit(x, xlim, dir='+')`

Compute limit  $x \rightarrow x_{\text{lim}}$ .

### `lseries(x=None, x0=0, dir='+', logx=None)`

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

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

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

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

See also `nseries()`.

### `normal()`

canonicalize ratio, i.e. return numerator if denominator is 1

### `nseries(x=None, x0=0, n=6, dir='+', logx=None)`

Wrapper to `_eval_nseries` if assumptions allow, else to `series`.

If  $x$  is given,  $x_0$  is 0,  $\text{dir}='+'$ , and  $\text{self}$  has  $x$ , then  $\text{eval_nseries}$  is called. This calculates “ $n$ ” terms in the innermost expressions and then builds up the final series just by “cross-multiplying” everything out.

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

Advantage – it’s fast, because we don’t have to determine how many terms we need to calculate in advance.

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

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

See also `lseries()`.

## Examples

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

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

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

In the following example, the expansion works but gives only an Order term unless the `logx` parameter is used:

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

`nsimplify(constants=[], tolerance=None, full=False)`  
See the `nsimplify` function in `sympy.simplify`

`powsimp(deep=False, combine='all')`  
See the `powsimp` function in `sympy.simplify`

`primitive()`  
Return the positive Rational that can be extracted non-recursively from every term of `self` (i.e.,

self is treated like an Add). This is like the as\_coeff\_Mul() method but primitive always extracts a positive Rational (never a negative or a Float).

### Examples

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

#### radsimp()

See the ratsimp function in sympy.simplify

#### ratsimp()

See the ratsimp function in sympy.simplify

#### refine(*assumption=True*)

See the refine function in sympy.assumptions

#### removeO()

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

#### round(*p=0*)

Return x rounded to the given decimal place.

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

### Notes

Do not confuse the Python builtin function, round, with the SymPy method of the same name. The former always returns a float (or raises an error if applied to a complex value) while the latter returns either a Number or a complex number:

```
>>> isinstance(round(S(123), -2), Number)
False
>>> isinstance(S(123).round(-2), Number)
True
>>> isinstance((3*I).round(), Mul)
True
>>> isinstance((1 + 3*I).round(), Add)
True
```

### Examples

```
>>> from sympy import pi, E, I, S, Add, Mul, Number
>>> S(10.5).round()
11.
>>> pi.round()
3.
>>> pi.round(2)
```

```
3.14
>>> (2*pi + E*I).round()
6. + 3.*I
```

The round method has a chopping effect:

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

### `separate(deep=False, force=False)`

See the separate function in `sympy.simplify`

### `series(x=None, x0=0, n=6, dir='+', logx=None)`

Series expansion of “self” around  $x = x_0$  yielding either terms of the series one by one (the lazy series given when  $n=\text{None}$ ), else all the terms at once when  $n \neq \text{None}$ .

Returns the series expansion of “self” around the point  $x = x_0$  with respect to  $x$  up to  $O((x - x_0)^n, x, x_0)$  (default  $n$  is 6).

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

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

If  $n=\text{None}$  then a generator of the series terms will be returned.

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

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

```
>>> abs(x).series(dir="+")
x
>>> abs(x).series(dir="-")
-x
```

### `simplify(ratio=1.7, measure=None)`

See the simplify function in `sympy.simplify`

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

General method for the taylor term.

This method is slow, because it differentiates n-times. Subclasses can redefine it to make it faster by using the “previous\_terms”.

`together(*args, **kwargs)`

See the together function in `sympy.polys`

`trigsimp(**args)`

See the trigsimp function in `sympy.simplify`

### 3.1.9 AtomicExpr

`class sympy.core.expr.AtomicExpr`

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

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

### 3.1.10 symbol

#### Symbol

`class sympy.core.symbol.Symbol`

**Assumptions:** `commutative = True`

You can override the default assumptions in the constructor:

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

`as_dummy()`

Return a Dummy having the same name and same assumptions as self.

#### Wild

`class sympy.core.symbol.Wild`

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

#### Examples

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

```
>>> (3*x**2).match(b*x)
>>> b.match(a)
{a_: b_}
>>> A = WildFunction('A')
>>> A.match(a)
{a_: A_}
```

## Dummy

`class sympy.core.symbol.Dummy`

Dummy symbols are each unique, identified by an internal count index:

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

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

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

## symbols

`sympy.core.symbol.symbols(names, **args)`

Transform strings into instances of `Symbol` (page 95) class.

`symbols()` (page 96) function returns a sequence of symbols with names taken from `names` argument, which can be a comma or whitespace delimited string, or a sequence of strings:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

>>> x, y, z = symbols('x,y,z', extended_real=True)
>>> x.is_extended_real and y.is_extended_real and z.is_extended_real
True
```

Despite its name, `symbols()` (page 96) can create symbol-like objects like instances of Function or Wild classes. To achieve this, set `cls` keyword argument to the desired type:

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

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

## var

```
sympy.core.symbol.var(names, **args)
Create symbols and inject them into the global namespace.
```

This calls `symbols()` (page 96) with the same arguments and puts the results into the *global* namespace. It's recommended not to use `var()` (page 98) in library code, where `symbols()` (page 96) has to be used:

```
.. rubric:: Examples

>>> from sympy import var

>>> var('x')
x
>>> x
x

>>> var('a,ab,abc')
(a, ab, abc)
>>> abc
abc

>>> var('x,y', extended_real=True)
(x, y)
>>> x.is_extended_real and y.is_extended_real
True
```

See `symbols()` (page 96) documentation for more details on what kinds of arguments can be passed to `var()` (page 98).

### 3.1.11 numbers

#### Number

```
class sympy.core.numbers.Number
Represents any kind of number in sympy.
```

Floating point numbers are represented by the `Float` class. Integer numbers (of any size), together with rational numbers (again, there is no limit on their size) are represented by the `Rational` class.

If you want to represent, for example, `1+sqrt(2)`, then you need to do:

```
Rational(1) + sqrt(Rational(2))

as_coeff_Add()
    Efficiently extract the coefficient of a summation.

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

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

gcd(other)
    Compute GCD of self and other.

lcm(other)
    Compute LCM of self and other.
```

## Float

```
class sympy.core.numbers.Float
    Represent a floating-point number of arbitrary precision.
```

### Notes

Floating point numbers are inexact by their nature unless their value is a binary-exact value.

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

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

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

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

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

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

```
>>> Float(0.3, 20)
0.2999999999999998890
```

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

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

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

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

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

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

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

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

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

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

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

## Examples

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

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

```
>>> Float(1)
1.00000000000000
>>> Float(10**20)
10000000000000000000.
>>> Float('1e20')
10000000000000000000.
```

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

```
>>> Float(1e20)
1.0000000000000e+20
>>> Float(1.23456789123456789)
1.23456789123457
```

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

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

The desired number of digits can also be specified:

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

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

```
>>> Float('123 456 789 . 123 456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

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

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

## Rational

```
class sympy.core.numbers.Rational
    Represents integers and rational numbers (p/q) of any size.
```

See also:

`sympy.core.sympify.sympify` (page 59), `sympy.simplify.simplify.nsimplify` (page 928)

## Examples

```
>>> from sympy import Rational, nsimplify, S, pi
>>> Rational(3)
3
>>> Rational(1, 2)
1/2
```

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

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

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

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

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

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

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

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

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

```
>>> Rational(pi)
Traceback (most recent call last):
...
TypeError: invalid input: pi
as_content_primitive(radical=False)
Return the tuple (R, self/R) where R is the positive Rational extracted from self.
```

## Examples

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

See docstring of Expr.as\_content\_primitive for more examples.

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

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

```
limit_denominator(max_denominator=1000000)
```

Closest Rational to self with denominator at most max\_denominator.

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

## Integer

```
class sympy.core.numbers.Integer
```

## NumberSymbol

```
class sympy.core.numbers.NumberSymbol
```

```
approximation(number_cls)
```

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

## RealNumber

```
sympy.core.numbers.RealNumber
alias of Float (page 99)
```

## igcd

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

Computes positive integer greatest common divisor.

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

## Examples

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

## ilcm

```
sympy.core.numbers.ilcm(*args)
    Computes integer least common multiple.
```

### Examples

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

## seterr

```
sympy.core.numbers.seterr(divide=False)
    Should sympy raise an exception on 0/0 or return a nan?
divide == True .... raise an exception divide == False ... return nan
```

## Zero

```
class sympy.core.numbers.Zero
    The number zero.

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

### References

[R39] (page 1233)

### Examples

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

## One

```
class sympy.core.numbers.One
    The number one.

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

## References

[R40] (page 1233)

## Examples

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

## NegativeOne

class sympy.core.numbers.NegativeOne  
The number negative one.

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

See also:

[One](#) (page 104)

## References

[R41] (page 1233)

## Examples

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

## Half

class sympy.core.numbers.Half  
The rational number 1/2.

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

## References

[R42] (page 1233)

## Examples

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

## NaN

```
class sympy.core.numbers.NaN
    Not a Number.
```

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as  $0/0$  or  $\infty - \infty$  produce NaN. Two exceptions are ' $0**0$ ' and ' $\infty**0$ ', which all produce 1 (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a `TypeError`. This is in contrast with floating point nan where all inequalities are false.

NaN is a singleton, and can be accessed by `S.NaN`, or can be imported as `nan`.

## References

[R43] (page 1233)

## Examples

```
>>> from sympy import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan)      # mathematical equality
False
>>> nan == nan       # structural equality
True
```

## Infinity

```
class sympy.core.numbers.Infinity
    Positive infinite quantity.
```

In real analysis the symbol  $\infty$  denotes an unbounded limit:  $x \rightarrow \infty$  means that  $x$  grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled  $+\infty$  and  $-\infty$  can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `S.Infinity`, or can be imported as `oo`.

**See also:**

`NegativeInfinity` (page 107), `NaN` (page 106)

## References

[R44] (page 1233)

## Examples

```
>>> from sympy import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```

## NegativeInfinity

class sympy.core.numbers.NegativeInfinity  
Negative infinite quantity.

NegativeInfinity is a singleton, and can be accessed by S.NegativeInfinity.

See also:

[Infinity](#) (page 106)

## ComplexInfinity

class sympy.core.numbers.ComplexInfinity  
Complex infinity.

In complex analysis the symbol  $\tilde{\infty}$ , called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by S.ComplexInfinity, or can be imported as zoo.

See also:

[Infinity](#) (page 106)

## Examples

```
>>> from sympy import zoo, oo
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

## Exp1

```
class sympy.core.numbers.Exp1
    The e constant.
```

The transcendental number  $e = 2.718281828\dots$  is the base of the natural logarithm and of the exponential function,  $e = \exp(1)$ . Sometimes called Euler's number or Napier's constant.

Exp1 is a singleton, and can be accessed by `S.Exp1`, or can be imported as `E`.

### References

[\[R45\]](#) (page 1233)

### Examples

```
>>> from sympy import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

## ImaginaryUnit

```
class sympy.core.numbers.ImaginaryUnit
    The imaginary unit,  $i = \sqrt{-1}$ .
```

I is a singleton, and can be accessed by `S.I`, or can be imported as `I`.

### References

[\[R46\]](#) (page 1233)

### Examples

```
>>> from sympy import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

## Pi

```
class sympy.core.numbers.Pi
    The  $\pi$  constant.
```

The transcendental number  $\pi = 3.141592654\dots$  represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

`Pi` is a singleton, and can be accessed by `S.Pi`, or can be imported as `pi`.

## References

[R47] (page 1233)

## Examples

```
>>> from sympy import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
True
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)
```

## EulerGamma

`class sympy.core.numbers.EulerGamma`

The Euler-Mascheroni constant.

$\gamma = 0.5772157\dots$  (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \rightarrow \infty} \left( \sum_{k=1}^n \frac{1}{k} - \ln n \right)$$

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

## References

[R48] (page 1233)

## Examples

```
>>> from sympy import S
>>> S.EulerGamma.is_irrational
True
>>> S.EulerGamma > 0
True
>>> S.EulerGamma > 1
False
```

## Catalan

```
class sympy.core.numbers.Catalan
    Catalan's constant.
```

$K = 0.91596559\dots$  is given by the infinite series

$$K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

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

### References

[R49] (page 1233)

### Examples

```
>>> from sympy import S
>>> S.Catalan.is_irrational
>>> S.Catalan > 0
True
>>> S.Catalan > 1
False
```

## GoldenRatio

```
class sympy.core.numbers.GoldenRatio
    The golden ratio,  $\phi$ .
```

$\phi = \frac{1+\sqrt{5}}{2}$  is algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

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

### References

[R50] (page 1233)

### Examples

```
>>> from sympy import S
>>> S.GoldenRatio > 1
True
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```

### 3.1.12 power

#### Pow

`class sympy.core.power.Pow`

Defines the expression  $x^{**}y$  as “ $x$  raised to a power  $y$ ”

Singleton definitions involving (0, 1, -1, oo, -oo):

expr	value	reason
$z^{**}0$	1	Although arguments over $0^{**}0$ exist, see [2].
$z^{**}1$	$z$	
$(-\infty)^{**}(-1)$	0	
$(-1)^{**}-1$	-1	
$S.\text{Zero}^{**}-1$	$\text{zoo}$	This is not strictly true, as $0^{**}-1$ may be undefined, but is convenient in some contexts where the base is assumed to be positive.
$1^{**}-1$	1	
$\infty^{**}-1$	0	
$0^{**}\infty$	0	Because for all complex numbers $z$ near 0, $z^{**}\infty \rightarrow 0$ .
$0^{**}-\infty$	$\text{zoo}$	This is not strictly true, as $0^{**}\infty$ may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
$1^{**}\infty$	$\text{nan}$	Because there are various cases where $\lim(x(t), t)=1$ , $\lim(y(t), t)=\infty$ (or $-\infty$ ), but $\lim(x(t)^{**}y(t), t) \neq 1$ . See [3].
$1^{**}\text{zoo}$		
$(-1)^{**}\infty (-1)^{**}(-\infty)$	$\text{nan}$	Because of oscillations in the limit.
$\infty^{**}\infty$	$\infty$	
$\infty^{**}-\infty$	0	
$(-\infty)^{**}\infty$	$\text{nan}$	
$(-\infty)^{**}-\infty$		

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

#### See also:

[sympy.core.numbers.Infinity](#) (page 106), [sympy.core.numbers.NegativeInfinity](#) (page 107), [sympy.core.numbers.NaN](#) (page 106)

#### References

[R51] (page 1233), [R52] (page 1234), [R53] (page 1234)

#### as\_base\_exp()

Return base and exp of self.

If base is 1/Integer, then return Integer, -exp. If this extra processing is not needed, the base and exp properties will give the raw arguments

## Examples

```
>>> from sympy import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)

as_content_primitive(radical=False)
Return the tuple (R, self/R) where R is the positive Rational extracted from self.
```

## Examples

```
>>> from sympy import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))

>>> from sympy import expand_power_base, powsimp, Mul
>>> from sympy.abc import x, y

>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**((y/2)))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content_primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
>>> powsimp(Mul(*_))
3**(2*x + 2)

>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)
```

See docstring of Expr.as\_content\_primitive for more examples.

## integer\_nthroot

```
sympy.core.power.integer_nthroot(y, n)
```

Return a tuple containing  $x = \text{floor}(y^{**}(1/n))$  and a boolean indicating whether the result is exact (that is, whether  $x^{**}n == y$ ).

```
>>> from sympy import integer_nthroot
>>> integer_nthroot(16,2)
(4, True)
>>> integer_nthroot(26,2)
(5, False)
```

### 3.1.13 mul

#### Mul

```
class sympy.core.mul.Mul
```

```
as_coeff_Mul(rational=False)
```

Efficiently extract the coefficient of a product.

```
as_content_primitive(radical=False)
```

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

#### Examples

```
>>> from sympy import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(-sqrt(2) + 1))
```

See docstring of Expr.as\_content\_primitive for more examples.

```
as_ordered_factors(order=None)
```

Transform an expression into an ordered list of factors.

#### Examples

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

>>> (2*x*y*sin(x)*cos(x)).as_ordered_factors()
[2, x, y, sin(x), cos(x)]
```

```
as_two_terms(*args, **kwargs)
```

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as\_coeff\_mul() which gives the head and a tuple containing the arguments of the tail when treated as a Mul.
- if you want the coefficient when self is treated as an Add then use self.as\_coeff\_add()[0]

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

```
classmethod flatten(seq)
```

Return commutative, noncommutative and order arguments by combining related terms.

## Notes

- In an expression like `a*b*c`, python process this through sympy as `Mul(Mul(a, b), c)`. This can have undesirable consequences.

– Sometimes terms are not combined as one would like: {c.f. <https://github.com/sympy/sympy/issues/4596>}

```
>>> from sympy import Mul, sqrt
>>> from sympy.abc import x, y, z
>>> 2*(x + 1) # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)*2
2*y*(x + 1)
>>> 2*(x + 1)*y # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y) # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y) # parentheses can control this behavior
2*y*(x + 1)
```

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases. {c.f. <https://github.com/sympy/sympy/issues/5728>}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a,a,a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)
```

– If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of `a`, `b` and `c` were `Mul` (page 113) expression, then `a*b*c` (or building up the product with `*=`) will process all the arguments of `a` and `b` twice: once when `a*b` is computed and again when `c` is multiplied.

Using `Mul(a, b, c)` will process all arguments once.

- The results of `Mul` are cached according to arguments, so `flatten` will only be called once for `Mul(a, b, c)`. If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a `Mul`, `M`, that you wished to divide by `d[i]` and multiply by `n[i]` and you suspect there are many repeats in `n`. It would be better to compute `M*n[i]/d[i]` rather than `M/d[i]*n[i]` since every time `n[i]` is a repeat, the product, `M*n[i]` will be returned without flattening – the cached value will be returned. If you divide by the `d[i]` first (and those are more unique than the `n[i]`) then that will create a new `Mul`, `M/d[i]` the args of which will be traversed again when it is multiplied by `n[i]`.

{c.f. <https://github.com/sympy/sympy/issues/5706>}

This consideration is moot if the cache is turned off.

## prod

```
sympy.core.mul.prod(a, start=1)
```

**Return product of elements of a.** Start with int 1 so if only ints are included then an int result is returned.

### Examples

```
>>> from sympy import prod, S
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([S(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```
>>> prod([1, 2], 3)
6
```

## 3.1.14 add

### Add

```
class sympy.core.add.Add
```

#### as\_coeff\_Add()

Efficiently extract the coefficient of a summation.

#### as\_coeff\_add(\*args, \*\*kwargs)

Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

### Examples

```
>>> from sympy.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

#### as\_coefficients\_dict(a)

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

## Examples

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

### as\_content\_primitive(*radical=False*)

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

## Examples

```
>>> from sympy import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

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

See docstring of Expr.as\_content\_primitive for more examples.

### as\_real\_imag(*deep=True, \*\*hints*)

returns a tuple representing a complex number

## Examples

```
>>> from sympy import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

### as\_two\_terms(\*args, \*\*kwargs)

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as\_coeff\_add() which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use self.as\_coeff\_mul()[0]

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

```
classmethod class_key()
    Nice order of classes

extract_leading_order(*args, **kwargs)
    Returns the leading term and its order.
```

### Examples

```
>>> from sympy.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), 0(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, 0(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, 0(x)),)
```

**classmethod flatten(seq)**

Takes the sequence “seq” of nested Adds and returns a flatten list.

Returns: (commutative\_part, noncommutative\_part, order\_symbols)

Applies associativity, all terms are commutable with respect to addition.

See also:

[sympy.core.mul.Mul.flatten](#) (page 113)

**primitive()**

Return  $(R, \text{self}/R)$  where  $R'$  is the Rational GCD of  $\text{self}'$ .

$R$  is collected only from the leading coefficient of each term.

### Examples

```
>>> from sympy.abc import x, y
>>> (2*x + 4*y).primitive()
(2, x + 2*y)

>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)

>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)
```

No subprocessing of term factors is performed:

```
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```

Recursive subprocessing can be done with the `as_content_primitive()` method:

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

See also: `primitive()` function in `polytools.py`

### 3.1.15 mod

#### Mod

```
class sympy.core.mod.Mod
```

Represents a modulo operation on symbolic expressions.

Receives two arguments, dividend p and divisor q.

The convention used is the same as Python's: the remainder always has the same sign as the divisor.

#### Examples

```
>>> from sympy.abc import x, y
>>> x**2 % y
x**2%y
>>> _.subs({x: 5, y: 6})
1
```

### 3.1.16 relational

#### Rel

```
sympy.core.relational.Rel
```

alias of [Relational](#) (page 119)

#### Eq

```
sympy.core.relational.Eq
```

alias of [Equality](#) (page 120)

#### Ne

```
sympy.core.relational.Ne
```

alias of [Unequality](#) (page 127)

#### Lt

```
sympy.core.relational.Lt
```

alias of [StrictLessThan](#) (page 130)

#### Le

```
sympy.core.relational.Le
```

alias of [LessThan](#) (page 124)

#### Gt

```
sympy.core.relational.Gt
```

alias of [StrictGreaterThan](#) (page 127)

## Ge

```
sympy.core.relation.Ge
alias of GreaterThan (page 121)
```

### Relational

```
class sympy.core.relation.Relational
Base class for all relation types.
```

Subclasses of Relational should generally be instantiated directly, but Relational can be instantiated with a valid *rop* value to dispatch to the appropriate subclass.

**Parameters** *rop* : str or None

Indicates what subclass to instantiate. Valid values can be found in the keys of Relational.ValidRelationalOperator.

### Examples

```
>>> from sympy import Rel
>>> from sympy.abc import x, y
>>> Rel(y, x+x**2, '==')
Eq(y, x**2 + x)
```

**as\_set()**

Rewrites univariate inequality in terms of real sets

### Examples

```
>>> from sympy import Symbol, Eq
>>> x = Symbol('x', extended_real=True)
>>> (x>0).as_set()
(0, oo)
>>> Eq(x, 0).as_set()
{0}
```

**canonical**

Return a canonical form of the relational.

The rules for the canonical form, in order of decreasing priority are:

1. Number on right if left is not a Number;
2. Symbol on the left;
3. Gt/Ge changed to Lt/Le;
4. Lt/Le are unchanged;
5. Eq and Ne get ordered args.

**equals(*other*, failing\_expression=False)**

Return True if the sides of the relationship are mathematically identical and the type of relationship is the same. If failing\_expression is True, return the expression whose truth value was unknown.

**lhs**

The left-hand side of the relation.

**reversed**

Return the relationship with sides (and sign) reversed.

**Examples**

```
>>> from sympy import Eq
>>> from sympy.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversed
Eq(1, x)
>>> x < 1
x < 1
>>> _.reversed
1 > x
```

**rhs**

The right-hand side of the relation.

**Equality**

```
class sympy.core.relation.Equality
An equal relation between two objects.
```

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the `simplify` function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument `evaluate=False` can be used to prevent any evaluation.

**See also:**

`sympy.logic.boolalg.Equivalent` ([page 569](#)) for representing equality between two boolean expressions

**Notes**

This class is not the same as the `==` operator. The `==` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `evalEq` method, it can be used in place of the default algorithm. If `lhs.evalEq(rhs)` or `rhs.evalEq(lhs)` returns anything other than None, that return value will be substituted for the Equality. If None is returned by `evalEq`, an Equality object will be created as usual.

**Examples**

```
>>> from sympy import Eq, simplify, exp, cos
>>> from sympy.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
```

```

False
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
False
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(exp(x), sinh(x) + cosh(x))
>>> simplify(_)
True

```

## GreaterThan

```
class sympy.core.relation.GreaterThan
    Class representations of inequalities.
```

The `*Than` classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

`lhs >= rhs`

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	( $\geq$ )
LessThan	( $\leq$ )
StrictGreaterThan	( $>$ )
StrictLessThan	( $<$ )

All classes take two arguments, `lhs` and `rhs`.

Signature	Example	Math equivalent
<code>GreaterThan(lhs, rhs)</code>		<code>lhs &gt;= rhs</code>
<code>LessThan(lhs, rhs)</code>		<code>lhs &lt;= rhs</code>
<code>StrictGreaterThan(lhs, rhs)</code>		<code>lhs &gt; rhs</code>
<code>StrictLessThan(lhs, rhs)</code>		<code>lhs &lt; rhs</code>

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```

>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relation import Relational

>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'

```

## Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement  $(1 < x)$ , Python will first recognize the number 1 as a native number, and then that  $x$  is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation,  $(x > 1)$ . Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement  $(1 < x)$  will turn silently into  $(x > 1)$ .

```
>>> e1 = x > 1
>>> e2 = x >= 1
>>> e3 = x < 1
>>> e4 = x <= 1
>>> e5 = 1 > x
>>> e6 = 1 >= x
>>> e7 = 1 < x
>>> e8 = 1 <= x
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
x < 1      x <= 1
x > 1      x >= 1
```

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways: (1) “sympify” the literal before comparison, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) > x
>>> e2 = S(1) >= x
>>> e3 = S(1) < x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

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

Due to an implementation detail or decision of Python [R54] (page 1234), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

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

---

```
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R55] (page 1234)):

```
>>> e = (x < y) < z
>>> type(e)
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

## Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge(x, 2)      # Ge is a convenience wrapper
>>> print(e1)
x >= 2

>>> rels = Ge(x, 2), Gt(x, 2), Le(x, 2), Lt(x, 2)
>>> print('%s\n%s\n%s\n%s' % rels)
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ( $\geq$ ,  $>$ ,  $\leq$ ,  $<$ ) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print("e1: %s,     e2: %s" % (e1, e2))
e1: x >= 2,     e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a \*Than class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x > 1
x > 1
x > 1
```

```
>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print("%s\n%s\n%s" % rels)
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x < 1
x < 1
x < 1
```

## LessThan

```
class sympy.core.relational.LessThan
    Class representations of inequalities.
```

The `*Than` classes represent unequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an unequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

`lhs >= rhs`

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	( $\geq$ )
LessThan	( $\leq$ )
StrictGreaterThan	( $>$ )
StrictLessThan	( $<$ )

All classes take two arguments, `lhs` and `rhs`.

Signature	Example	Math equivalent
<code>GreaterThan(lhs, rhs)</code>		<code>lhs &gt;= rhs</code>
<code>LessThan(lhs, rhs)</code>		<code>lhs &lt;= rhs</code>
<code>StrictGreaterThan(lhs, rhs)</code>		<code>lhs &gt; rhs</code>
<code>StrictLessThan(lhs, rhs)</code>		<code>lhs &lt; rhs</code>

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational

>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

## Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement ( $1 < x$ ), Python will first recognize the number 1 as a native number, and then that  $x$  is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, ( $x > 1$ ). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement ( $1 < x$ ) will turn silently into ( $x > 1$ ).

```
>>> e1 = x > 1
>>> e2 = x >= 1
>>> e3 = x < 1
>>> e4 = x <= 1
>>> e5 = 1 > x
>>> e6 = 1 >= x
>>> e7 = 1 < x
>>> e8 = 1 <= x
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
x < 1      x <= 1
x > 1      x >= 1
```

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways: (1) “sympify” the literal before comparison, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) > x
>>> e2 = S(1) >= x
>>> e3 = S(1) < x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

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

Due to an implementation detail or decision of Python [R56] (page 1234), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

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

```
>>> e
And(x < y, y < z)
```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R57] (page 1235)):

```
>>> e = (x < y) < z
>>> type(e)
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

## Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge(x, 2)      # Ge is a convenience wrapper
>>> print(e1)
x >= 2

>>> rels = Ge(x, 2), Gt(x, 2), Le(x, 2), Lt(x, 2)
>>> print('%s\n%s\n%s\n%s' % rels)
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ( $\geq$ ,  $>$ ,  $\leq$ ,  $<$ ) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print("e1: %s,     e2: %s" % (e1, e2))
e1: x >= 2,     e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a \*Than class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x > 1
x > 1
x > 1
```

```
>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print("%s\n%s\n%s" % rels)
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x < 1
x < 1
x < 1
```

## Unequality

```
class sympy.core.relational.Unequality
An unequal relation between two objects.
```

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

See also:

[Equality](#) (page 120)

### Notes

This class is not the same as the != operator. The != operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of Equality. As such, it uses the same algorithms, including any available  $\text{eval}_{\text{Eq}}$  methods.

### Examples

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

## StrictGreaterThan

```
class sympy.core.relational.StrictGreaterThan
Class representations of inequalities.
```

The \*Than classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the GreaterThan class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four \*Than classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	( $\geq$ )
LessThan	( $\leq$ )
StrictGreaterThan	( $>$ )
StrictLessThan	( $<$ )

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal .lhs and .rhs of Relations, \*Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational

>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

## Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement ( $1 < x$ ), Python will first recognize the number 1 as a native number, and then that  $x$  is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, ( $x > 1$ ). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement ( $1 < x$ ) will turn silently into ( $x > 1$ ).

```
>>> e1 = x > 1
>>> e2 = x >= 1
>>> e3 = x < 1
>>> e4 = x <= 1
>>> e5 = 1 > x
>>> e6 = 1 >= x
>>> e7 = 1 < x
>>> e8 = 1 <= x
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
x < 1      x <= 1
x > 1      x >= 1
```

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways: (1) “sympify” the literal before comparison, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) > x
>>> e2 = S(1) >= x
>>> e3 = S(1) < x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print("%s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

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

Due to an implementation detail or decision of Python [R58] (page 1235), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

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

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R59] (page 1235)):

```
>>> e = (x < y) < z
>>> type(e)
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

## Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge(x, 2)      # Ge is a convenience wrapper
>>> print(e1)
x >= 2
```

```
>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print('%s\n%s\n%s\n%s' % rels)
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ( $\geq$ ,  $>$ ,  $\leq$ ,  $<$ ) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print("e1: %s,      e2: %s" % (e1, e2))
e1: x >= 2,      e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x < 1
x < 1
x < 1
```

## StrictLessThan

```
class sympy.core.relational.StrictLessThan
    Class representations of inequalities.
```

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$\text{lhs} \geq \text{rhs}$

In total, there are four `*Than` classes, to represent the four inequalities:

Class Name	Symbol
GreaterThan	( $\geq$ )
LessThan	( $\leq$ )
StrictGreaterThan	( $>$ )
StrictLessThan	( $<$ )

All classes take two arguments, lhs and rhs.

Signature	Example	Math equivalent
GreaterThan(lhs, rhs)		$\text{lhs} \geq \text{rhs}$
LessThan(lhs, rhs)		$\text{lhs} \leq \text{rhs}$
StrictGreaterThan(lhs, rhs)		$\text{lhs} > \text{rhs}$
StrictLessThan(lhs, rhs)		$\text{lhs} < \text{rhs}$

In addition to the normal .lhs and .rhs of Relations, \*Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from sympy import GreaterThan, StrictGreaterThan
>>> from sympy import LessThan, StrictLessThan
>>> from sympy import And, Ge, Gt, Le, Lt, Rel, S
>>> from sympy.abc import x, y, z
>>> from sympy.core.relational import Relational

>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

## Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement ( $1 < x$ ), Python will first recognize the number 1 as a native number, and then that  $x$  is *not* a native number. At this point, because a native Python number does not know how to compare itself with a SymPy object Python will try the reflective operation, ( $x > 1$ ). Unfortunately, there is no way available to SymPy to recognize this has happened, so the statement ( $1 < x$ ) will turn silently into ( $x > 1$ ).

```
>>> e1 = x > 1
>>> e2 = x >= 1
>>> e3 = x < 1
>>> e4 = x <= 1
>>> e5 = 1 > x
>>> e6 = 1 >= x
>>> e7 = 1 < x
>>> e8 = 1 <= x
>>> print("%s      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
x > 1      x >= 1
x < 1      x <= 1
x < 1      x <= 1
x > 1      x >= 1
```

If the order of the statement is important (for visual output to the console, perhaps), one can work around this annoyance in a couple ways: (1) “sympify” the literal before comparison, (2) use one of the wrappers, or (3) use the less succinct methods described above:

```
>>> e1 = S(1) > x
>>> e2 = S(1) >= x
>>> e3 = S(1) < x
>>> e4 = S(1) <= x
>>> e5 = Gt(1, x)
>>> e6 = Ge(1, x)
>>> e7 = Lt(1, x)
>>> e8 = Le(1, x)
>>> print("%s    %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

The other gotcha is with chained inequalities. Occasionally, one may be tempted to write statements like:

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

Due to an implementation detail or decision of Python [R60] (page 1235), there is no way for SymPy to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

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

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in SymPy [R61] (page 1236)):

```
>>> e = (x < y) < z
>>> type(e)
<class 'sympy.core.relational.StrictLessThan'>
>>> e
(x < y) < z
```

Any code that explicitly relies on this latter functionality will not be robust as this behaviour is completely wrong and will be corrected at some point. For the time being (circa Jan 2012), use And to create chained inequalities.

## Examples

One generally does not instantiate these classes directly, but uses various convenience methods:

```
>>> e1 = Ge( x, 2 )      # Ge is a convenience wrapper
>>> print(e1)
x >= 2
```

```
>>> rels = Ge( x, 2 ), Gt( x, 2 ), Le( x, 2 ), Lt( x, 2 )
>>> print('%s\n%s\n%s\n%s' % rels)
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators ( $\geq$ ,  $>$ ,  $\leq$ ,  $<$ ) directly. Their main advantage over the Ge, Gt, Le, and Lt counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```
>>> e2 = x >= 2
>>> print(e2)
x >= 2
>>> print("e1: %s,      e2: %s" % (e1, e2))
e1: x >= 2,      e2: x >= 2
>>> e1 == e2
True
```

However, it is also perfectly valid to instantiate a `*Than` class less succinctly and less conveniently:

```
>>> rels = Rel(x, 1, '>='), Relational(x, 1, '>='), GreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x >= 1
x >= 1
x >= 1

>>> rels = Rel(x, 1, '>'), Relational(x, 1, '>'), StrictGreaterThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x > 1
x > 1
x > 1

>>> rels = Rel(x, 1, '<='), Relational(x, 1, '<='), LessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x <= 1
x <= 1
x <= 1

>>> rels = Rel(x, 1, '<'), Relational(x, 1, '<'), StrictLessThan(x, 1)
>>> print('%s\n%s\n%s' % rels)
x < 1
x < 1
x < 1
```

### 3.1.17 multidimensional

#### vectorize

```
class sympy.core.multidimensional.vectorize(*mdargs)
Generalizes a function taking scalars to accept multidimensional arguments.
```

For example

```
>>> from sympy import diff, sin, symbols, Function
>>> from sympy.core.multidimensional import vectorize
```

```
>>> x, y, z = symbols('x y z')
>>> f, g, h = list(map(Function, 'fgh'))

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

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

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

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

### 3.1.18 function

#### Lambda

class `sympy.core.function.Lambda`

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

A simple example:

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

For multivariate functions, use:

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

A handy shortcut for lots of arguments:

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

#### expr

The return value of the function

#### is\_identity

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

#### variables

The variables used in the internal representation of the function

## WildFunction

```
class sympy.core.function.WildFunction(name, **assumptions)
A WildFunction function matches any function (with its arguments).
```

### Examples

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

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

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

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

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

## Derivative

```
class sympy.core.function.Derivative
```

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

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

Simplification of high-order derivatives:

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

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

Ordering of variables:

If `evaluate` is set to True and the expression can not be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked. This sorting assumes that derivatives wrt Symbols commute, derivatives wrt non-Symbols commute, but Symbol and non-Symbol derivatives don't commute with each other.

Derivative wrt non-Symbols:

This class also allows derivatives wrt non-Symbols that have `_diff_wrt` set to True, such as `Function` and `Derivative`. When a derivative wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed.

Note that this may seem strange, that `Derivative` allows things like `f(g(x)).diff(g(x))`, or even `f(cos(x)).diff(cos(x))`. The motivation for allowing this syntax is to make it easier to work with variational calculus (i.e., the Euler-Lagrange method). The best way to understand this is that the action of derivative with respect to a non-Symbol is defined by the above description: the object is substituted for a Symbol and the derivative is taken with respect to that. This action is only allowed for objects for which this can be done unambiguously, for example `Function` and `Derivative` objects. Note that this leads to what may appear to be mathematically inconsistent results. For example:

```
>>> from sympy import cos, sin, sqrt
>>> from sympy.abc import x
>>> (2*cos(x)).diff(cos(x))
2
>>> (2*sqrt(1 - sin(x)**2)).diff(cos(x))
0
```

This appears wrong because in fact  $2\cos(x)$  and  $2\sqrt{1 - \sin(x)^2}$  are identically equal. However this is the wrong way to think of this. Think of it instead as if we have something like this:

```
>>> from sympy.abc import c, s
>>> def F(u):
...     return 2*u
...
>>> def G(u):
...     return 2*sqrt(1 - u**2)
...
>>> F(cos(x))
2*cos(x)
>>> G(sin(x))
2*sqrt(-sin(x)**2 + 1)
>>> F(c).diff(c)
2
>>> F(c).diff(c)
2
>>> G(s).diff(c)
0
```

---

```
>>> G(sin(x)).diff(cos(x))
0
```

Here, the Symbols `c` and `s` act just like the functions `cos(x)` and `sin(x)`, respectively. Think of `2*cos(x)` as `f(c).subs(c, cos(x))` (or `f(c) at c = cos(x)`) and `2*sqrt(1 - sin(x)**2)` as `g(s).subs(s, sin(x))` (or `g(s) at s = sin(x)`), where `f(u) == 2*u` and `g(u) == 2*sqrt(1 - u**2)`. Here, we define the function first and evaluate it at the function, but we can actually unambiguously do this in reverse in SymPy, because `expr.subs(Function, Symbol)` is well-defined: just structurally replace the function everywhere it appears in the expression.

This is the same notational convenience used in the Euler-Lagrange method when one says `F(t, f(t), f'(t)).diff(f(t))`. What is actually meant is that the expression in question is represented by some `F(t, u, v)` at `u = f(t)` and `v = f'(t)`, and `F(t, f(t), f'(t)).diff(f(t))` simply means `F(t, u, v).diff(u)` at `u = f(t)`.

We do not allow derivatives to be taken with respect to expressions where this is not so well defined. For example, we do not allow `expr.diff(x*y)` because there are multiple ways of structurally defining where `x*y` appears in an expression, some of which may surprise the reader (for example, a very strict definition would have that `(x*y*z).diff(x*y) == 0`).

```
>>> from sympy.abc import x, y, z
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't differentiate wrt the variable: x*y, 1
```

Note that this definition also fits in nicely with the definition of the chain rule. Note how the chain rule in SymPy is defined using unevaluated `Subs` objects:

```
>>> from sympy import symbols, Function
>>> f, g = symbols('f g', cls=Function)
>>> f(2*g(x)).diff(x)
2*Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                               (_xi_1,), (2*g(x),))
>>> f(g(x)).diff(x)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                           (_xi_1,), (g(x),))
```

Finally, note that, to be consistent with variational calculus, and to ensure that the definition of substituting a `Function` for a `Symbol` in an expression is well-defined, derivatives of functions are assumed to not be related to the function. In other words, we have:

```
>>> from sympy import diff
>>> diff(f(x), x).diff(f(x))
0
```

The same is true for derivatives of different orders:

```
>>> diff(f(x), x, 2).diff(diff(f(x), x, 1))
0
>>> diff(f(x), x, 1).diff(diff(f(x), x, 2))
0
```

Note, any class can allow derivatives to be taken with respect to itself. See the docstring of `Expr._diff_wrt`.

## Examples

Some basic examples:

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

>>> Derivative(x**2, x, evaluate=True)
2*x
>>> Derivative(Derivative(f(x,y), x), y)
Derivative(f(x, y), x, y)
>>> Derivative(f(x), x, 3)
Derivative(f(x), x, x, x)
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Now some derivatives wrt functions:

```
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
>>> Derivative(f(g(x)), x, evaluate=True)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
(_xi_1,), (g(x),))

doit_numerically(a, b)
Evaluate the derivative at z numerically.
```

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

## diff

```
sympy.core.function.diff(f, *symbols, **kwargs)
Differentiate f with respect to symbols.
```

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

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

### See also:

[Derivative](#) (page 135)

[sympy.geometry.util.idiff](#) ([page 434](#)) computes the derivative implicitly

## References

[http://reference.wolfram.com/legacy/v5\\_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html](http://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html)

## Examples

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

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

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

>>> diff(sin(x))
cos(x)
>>> diff(sin(x*y))
Traceback (most recent call last):
...
ValueError: specify differentiation variables to differentiate sin(x*y)
```

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

## FunctionClass

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

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

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

### nargs

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

## Examples

```
>>> from sympy.core.function import Function
>>> from sympy.abc import x, y
>>> f = Function('f')
```

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

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

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

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

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

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

## Function

```
class sympy.core.function.Function
    Base class for applied mathematical functions.
```

It also serves as a constructor for undefined function classes.

### Examples

First example shows how to use `Function` as a constructor for undefined function classes:

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

In the following example `Function` is used as a base class for `my_func` that represents a mathematical function *my\_func*. Suppose that it is well known, that *my\_func(0)* is 1 and *my\_func* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *my\_func(x)* is real exactly when *x* is real. Here is an implementation that honours those requirements:

```
>>> from sympy import Function, S, oo, I, sin
>>> class my_func(Function):
...
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...             else:
...                 return None
```

```
...     def _eval_is_extended_real(self):
...         return self.args[0].is_extended_real
...
>>> x = S('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_extended_real
False
```

In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```
>>> class my_func(Function):
...     nargs = (1, 2)
...
>>>
as_base_exp()
    Returns the method as the 2-tuple (base, exponent).
fdiff(argindex=1)
    Returns the first derivative of the function.
is_commutative
    Returns whether the function is commutative.
```

---

**Note:** Not all functions are the same

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

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

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

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function)
set([f(x), cos(x)])
```

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

```
>>> from sympy.core.function import AppliedUndef
>>> e.atoms(AppliedUndef)
set([f(x)])
```

---

## Subs

```
class sympy.core.function.Subs
```

Represents unevaluated substitutions of an expression.

`Subs(expr, x, x0)` receives 3 arguments: an expression, a variable or list of distinct variables and a point or list of evaluation points corresponding to those variables.

`Subs` objects are generally useful to represent unevaluated derivatives calculated at a point.

The variables may be expressions, but they are subjected to the limitations of `subs()`, so it is usually a good practice to use only symbols for variables, since in that case there can be no ambiguity.

There's no automatic expansion - use the method `.doit()` to effect all possible substitutions of the object and also of objects inside the expression.

When evaluating derivatives at a point that is not a symbol, a `Subs` object is returned. One is also able to calculate derivatives of `Subs` objects - in this case the expression is always expanded (for the unevaluated form, use `Derivative()`).

A simple example:

```
>>> from sympy import Subs, Function, sin
>>> from sympy.abc import x, y, z
>>> f = Function('f')
>>> e = Subs(f(x).diff(x), x, y)
>>> e.subs(y, 0)
Subs(Derivative(f(x), x), (x,), (0,))
>>> e.subs(f, sin).doit()
cos(y)
```

An example with several variables:

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

### expr

The expression on which the substitution operates

### point

The values for which the variables are to be substituted

### variables

The variables to be evaluated

## expand

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

Expand an expression using methods given as hints.

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

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

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

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

**See also:**

`expand_log` (page 146), `expand_mul` (page 146), `expand_multinomial` (page 147), `expand_complex` (page 147), `expand_trig` (page 147), `expand_power_base` (page 148), `expand_power_exp` (page 148), `expand_func` (page 146), `sympy.simplify.hyperexpand.hyperexpand` (page 935)

## Notes

- You can shut off unwanted methods:

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

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

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

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

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

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

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

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

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

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

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

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

The parts of a rational expression can be targeted:

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

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

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

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

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

## Examples

```
>>> from sympy import Expr, sympify
>>> class MyClass(Expr):
...     def __new__(cls, *args):
...         args = sympify(args)
...         return Expr.__new__(cls, *args)
...
...     def _eval_expand_double(self, **hints):
...         """
...         Doubles the args of MyClass.
...
...         If there more than four args, doubling is not performed,
...         unless force=True is also used (False by default).
...
...         force = hints.pop('force', False)
...         if not force and len(self.args) > 4:
...             return self
...         return self.func(*self.args + self.args)
...
...     a = MyClass(1, 2, MyClass(3, 4))
>>> a
```

```
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))

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

## PoleError

```
class sympy.core.function.PoleError
```

## count\_ops

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

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

If `visual` is `False` (default) then the sum of the coefficients of the visual expression will be returned.

If `visual` is `True` then the number of each type of operation is shown with the core class types (or their virtual equivalent) multiplied by the number of times they occur.

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

## Examples

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

Although there isn't a `SUB` object, minus signs are interpreted as either negations or subtractions:

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

Here, there are two `Adds` and a `Pow`:

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

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

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

for a total of 5:

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

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

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

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

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

The count\_ops function also handles iterables:

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

## expand\_mul

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

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

### Examples

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

## expand\_log

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

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

### Examples

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

## expand\_func

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

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

## Examples

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

## expand\_trig

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

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

## Examples

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

## expand\_complex

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

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

### See also:

`sympy.core.expr.Expr.as_real_imag` (page 82)

## Examples

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

## expand\_multinomial

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

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

## Examples

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

## expand\_power\_exp

```
sympy.core.function.expand_power_exp(expr, deep=True)
    Wrapper around expand that only uses the power_exp hint.
```

See the expand docstring for more information.

### Examples

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

## expand\_power\_base

```
sympy.core.function.expand_power_base(expr, deep=True, force=False)
    Wrapper around expand that only uses the power_base hint.
```

See the expand docstring for more information.

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

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

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

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

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

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

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

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

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

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

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

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

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

## nfloat

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

Make all `Rationals` in `expr` `Floats` except those in exponents (unless the `exponents` flag is set to `True`).

### Examples

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

## 3.1.19 evalf

`class sympy.core.evalf.EvalfMixin`

Mixin class adding `evalf` capability.

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

Evaluate the given formula to an accuracy of `n` digits. Optional keyword arguments:

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

`maxn=<integer>` Allow a maximum temporary working precision of `maxn` digits (default=100)

`chop=<bool>` Replace tiny real or imaginary parts in subresults by exact zeros (default=False)

`strict=<bool>` Raise `PrecisionExhausted` if any subresult fails to evaluate to full accuracy, given the available `maxprec` (default=False)

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

`verbose=<bool>` Print debug information (default=False)

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

Evaluate the given formula to an accuracy of n digits. Optional keyword arguments:

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

**maxn=<integer>** Allow a maximum temporary working precision of maxn digits (default=100)

**chop=<bool>** Replace tiny real or imaginary parts in subresults by exact zeros (default=False)

**strict=<bool>** Raise PrecisionExhausted if any subresult fails to evaluate to full accuracy, given the available maxprec (default=False)

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

**verbose=<bool>** Print debug information (default=False)

## PrecisionExhausted

```
class sympy.core.evalf.PrecisionExhausted
```

## N

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

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

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

## Examples

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

## 3.1.20 containers

### Tuple

```
class sympy.core.containers.Tuple
```

Wrapper around the builtin tuple object

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

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

```
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)

index(value[, start[, stop]]) → integer – return first index of value.
    Raises ValueError if the value is not present.

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

## Dict

```
class sympy.core.containers.Dict
    Wrapper around the builtin dict object
```

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

```
>>> from sympy.core.containers import Dict

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

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

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

`get(k[, d])` → `D[k]` if `k` in `D`, else `d`. `d` defaults to `None`.

`items()` → list of `D`'s (key, value) pairs, as 2-tuples

`keys()` → list of `D`'s keys

`values()` → list of `D`'s values

### 3.1.21 compatibility

Reimplementations of constructs introduced in later versions of Python than we support. Also some functions that are needed SymPy-wide and are located here for easy import.

```
class sympy.core.compatibility.NotIterable
```

Use this as mixin when creating a class which is not supposed to return true when `iterable()` is called on its instances. I.e. avoid infinite loop when calling e.g. `list()` on the instance

```
sympy.core.compatibility.as_int(n)
    Convert the argument to a builtin integer.
```

The return value is guaranteed to be equal to the input. `ValueError` is raised if the input has a non-integral value.

### Examples

```
>>> from sympy.core.compatibility import as_int
>>> from sympy import sqrt
>>> 3.0
3.0
>>> as_int(3.0) # convert to int and test for equality
3
>>> int(sqrt(10))
3
>>> as_int(sqrt(10))
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

`sympy.core.compatibility.default_sort_key(item, order=None)`

Return a key that can be used for sorting.

The key has the structure:

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

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

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

See also:

`ordered` (page 154), `sympy.core.expr.Expr.as_ordered_factors` (page 81),  
`sympy.core.expr.Expr.as_ordered_terms` (page 81)

### Examples

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

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

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

Here are some examples of the key that is produced:

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

---

```
>>> default_sort_key(2)
((1, 0, 'Number'), (0, (), (), 2)
```

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

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

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

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

`sympy.core.compatibility.exec_(_code_, _globals_=None, _locals_=None)`

Execute code in a namespace.

`sympy.core.compatibility.is_sequence(i, include=None)`

Return a boolean indicating whether i is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set 'include' to that object's type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: iterable

## Examples

```
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

`sympy.core.compatibility.iterable(i, exclude=((<type 'str'>, <type 'unicode'>), <type 'dict'>, <class 'sympy.core.compatibility.NotIterable' at 0x7fb9b3546d0>))`

Return a boolean indicating whether i is SymPy iterable. True also indicates that the iterator is finite, i.e. you e.g. call list(...) on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make exclude=None. To exclude multiple items, pass them as a tuple.

See also: `is_sequence`

### Examples

```
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>

>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

`sympy.core.compatibility.lru_cache(maxsize=100, typed=False)`

Least-recently-used cache decorator.

If `maxsize` is set to None, the LRU features are disabled and the cache can grow without bound.

If `typed` is True, arguments of different types will be cached separately. For example, `f(3.0)` and `f(3)` will be treated as distinct calls with distinct results.

Arguments to the cached function must be hashable.

View the cache statistics named tuple (hits, misses, maxsize, currsize) with `f.cache_info()`. Clear the cache and statistics with `f.cache_clear()`. Access the underlying function with `f._wrapped_`.

See: [http://en.wikipedia.org/wiki/Cache\\_algorithms#Least\\_Recently\\_Used](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used)

`sympy.core.compatibility.ordered(seq, keys=None, default=True, warn=False)`

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

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

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

### Notes

The decorated sort is one of the fastest ways to sort a sequence for which special item comparison is desired: the sequence is decorated, sorted on the basis of the decoration (e.g. making all letters lower

case) and then undecorated. If one wants to break ties for items that have the same decorated value, a second key can be used. But if the second key is expensive to compute then it is inefficient to decorate all items with both keys: only those items having identical first key values need to be decorated. This function applies keys successively only when needed to break ties. By yielding an iterator, use of the tie-breaker is delayed as long as possible.

This function is best used in cases when use of the first key is expected to be a good hashing function; if there are no unique hashes from application of a key then that key should not have been used. The exception, however, is that even if there are many collisions, if the first group is small and one does not need to process all items in the list then time will not be wasted sorting what one was not interested in. For example, if one were looking for the minimum in a list and there were several criteria used to define the sort order, then this function would be good at returning that quickly if the first group of candidates is small relative to the number of items being processed.

### Examples

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

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

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

The default\_sort\_key allows the tie to be broken:

```
>>> list(ordered([y + 2, x + 2, x**2 + y + 3]))
...
[x + 2, y + 2, x**2 + y + 3]
```

Here, sequences are sorted by length, then sum:

```
>>> seq, keys = [[[1, 2, 1], [0, 3, 1], [1, 1, 3], [2], [1]], [
...     lambda x: len(x),
...     lambda x: sum(x)]]
...
>>> list(ordered(seq, keys, default=False, warn=False))
[[1], [2], [1, 2, 1], [0, 3, 1], [1, 1, 3]]
```

If warn is True, an error will be raised if there were not enough keys to break ties:

```
>>> list(ordered(seq, keys, default=False, warn=True))
Traceback (most recent call last):
...
ValueError: not enough keys to break ties
```

`sympy.core.compatibility.with_metaclass(meta, *bases)`  
Create a base class with a metaclass.

For example, if you have the metaclass

```
>>> class Meta(type):
...     pass
```

Use this as the metaclass by doing

```
>>> from sympy.core.compatibility import with_metaclass
>>> class MyClass(with_metaclass(Meta, object)):
...     pass
```

This is equivalent to the Python 2:

```
class MyClass(object):
    __metaclass__ = Meta
```

or Python 3:

```
class MyClass(object, metaclass=Meta):
    pass
```

That is, the first argument is the metaclass, and the remaining arguments are the base classes. Note that if the base class is just `object`, you may omit it.

```
>>> MyClass.__mro__
(<class 'MyClass'>, <... 'object'>)
>>> type(MyClass)
<class 'Meta'>
```

## iterable

```
sympy.core.compatibility.iterable(i, exclude=(<type 'str'>, <type 'unicode'>), <type 'dict'>, <class sympy.core.compatibility.NotIterable at 0x7fb9b3546d0>))
```

Return a boolean indicating whether `i` is SymPy iterable. True also indicates that the iterator is finite, i.e. you e.g. call `list(...)` on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make `exclude=None`. To exclude multiple items, pass them as a tuple.

See also: `is_sequence`

## Examples

```
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>

>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
```

```
True
>>> iterable("no", exclude=str)
False
```

### is\_sequence

```
sympy.core.compatibility.is_sequence(i, include=None)
```

Return a boolean indicating whether *i* is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set ‘include’ to that object’s type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: iterable

### Examples

```
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

### as\_int

```
sympy.core.compatibility.as_int(n)
```

Convert the argument to a builtin integer.

The return value is guaranteed to be equal to the input. ValueError is raised if the input has a non-integral value.

### Examples

```
>>> from sympy.core.compatibility import as_int
>>> from sympy import sqrt
>>> 3.0
3.0
>>> as_int(3.0) # convert to int and test for equality
3
>>> int(sqrt(10))
3
```

```
>>> as_int(sqrt(10))
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

### 3.1.22 exprtools

#### gcd\_terms

`sympy.core.exprtools.gcd_terms(terms, isprimitive=False, clear=True, fraction=True)`

Compute the GCD of `terms` and put them together.

`terms` can be an expression or a non-Basic sequence of expressions which will be handled as though they are terms from a sum.

If `isprimitive` is True the `_gcd_terms` will not run the primitive method on the terms.

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

`fraction`, when True (default), will put the expression over a common denominator.

See also:

`factor_terms` (page 159), `sympy.polys.polytools.terms_gcd` (page 670)

#### Examples

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

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

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

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

## factor\_terms

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

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

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

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

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

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

See also:

`gcd_terms` (page 158), `sympy.polys.polytools.terms_gcd` (page 670)

## Examples

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

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

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

This only applies when there is a single Add that the coefficient multiplies:

```
>>> factor_terms(x*y/2 + y, clear=True)
y*(x + 2)/2
>>> factor_terms(x*y/2 + y, clear=False) == -
True
```

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

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

## 3.2 Combinatorics Module

### 3.2.1 Contents

#### Partitions

```
class sympy.combinatorics.partitions.Partition
```

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

See also:

[sympy.utilities.iterables.partitions](#) (page 1141), [sympy.utilities.iterables.multiset\\_partitions](#) (page 1139)

#### RGS

Returns the “restricted growth string” of the partition.

The RGS is returned as a list of indices, L, where L[i] indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ([c], [a, b]) the RGS is [1, 1, 0]: “a” is in block 1, “b” is in block 1 and “c” is in block 0.

#### Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
{{3}, {4}, {5}, {1, 2}}
>>> _.RGS
(0, 0, 1, 2, 3)
```

**classmethod** `from_rgs(rgs, elements)`

Creates a set partition from a restricted growth string.

The indices given in rgs are assumed to be the index of the element as given in elements *as provided* (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in rgs an error will be raised.

#### Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
{{c}, {a, d}, {b, e}}
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
{{e}, {a, c}, {b, d}}
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
{{2}, {1, 4}, {3, 5}}
```

#### partition

Return partition as a sorted list of lists.

## Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

### rank

Gets the rank of a partition.

## Examples

```
>>> from sympy.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

### sort\_key(*order=None*)

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

## Examples

```
>>> from sympy.utilities.iterables import default_sort_key
>>> from sympy.combinatorics.partitions import Partition
>>> from sympy.abc import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[{{1, 2}}, {{1}, {2}}, {{1, x}}, {{3, 4}}, {{0, 1, 2, 3}}]
```

## class sympy.combinatorics.partitions.IntegerPartition

This class represents an integer partition.

In number theory and combinatorics, a partition of a positive integer,  $n$ , also called an integer partition, is a way of writing  $n$  as a list of positive integers that sum to  $n$ . Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: [4], [3, 1], [2, 2], [2, 1, 1], and [1, 1, 1, 1]; the compositions [1, 2, 1] and [1, 1, 2] are the same as partition [2, 1, 1].

### See also:

[sympy.utilities.iterables.partitions](#) (page 1141), [sympy.utilities.iterables.multiset\\_partitions](#) (page 1139)

## References

[R4] (page 1236)

**as\_dict()**

Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer.

**Examples**

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

**as\_ferrers(char='#')**

Prints the ferrer diagram of a partition.

**Examples**

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

**conjugate**

Computes the conjugate partition of itself.

**Examples**

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1, 1]
```

**next\_lex()**

Return the next partition of the integer, n, in lexical order, wrapping around to [n] if the partition is [1, ..., 1].

**Examples**

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

**prev\_lex()**

Return the previous partition of the integer, n, in lexical order, wrapping around to [1, ..., 1] if the partition is [n].

## Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

`sympy.combinatorics.partitions.random_integer_partition(n, seed=None)`

Generates a random integer partition summing to n as a list of reverse-sorted integers.

## Examples

```
>>> from sympy.combinatorics.partitions import random_integer_partition
```

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

`sympy.combinatorics.partitions.RGS_generalized(m)`

Computes the m + 1 generalized unrestricted growth strings and returns them as rows in matrix.

## Examples

```
>>> from sympy.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([
[ 1,    1,    1,    1,    1,  1],
[ 1,    2,    3,    4,    5,  6,  0],
[ 2,    5,   10,   17,   26,  0,  0],
[ 5,   15,   37,   77,  0,  0,  0],
[ 15,   52,  151,  0,  0,  0,  0],
[ 52,  203,   0,  0,  0,  0,  0],
[203,   0,   0,  0,  0,  0,  0]])
```

`sympy.combinatorics.partitions.RGS_enum(m)`

RGS\_enum computes the total number of restricted growth strings possible for a superset of size m.

## Examples

```
>>> from sympy.combinatorics.partitions import RGS_enum
>>> from sympy.combinatorics.partitions import Partition
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
```

```
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))
>>> s = set()
>>> for i in range(20):
...     s.add(a)
...     a += 1
...
>>> assert len(s) == 15
```

```
sympy.combinatorics.partitions.RGS_unrank(rank, m)
```

Gives the unranked restricted growth string for a given superset size.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_unrank
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

```
sympy.combinatorics.partitions.RGS_rank(rgs)
```

Computes the rank of a restricted growth string.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_rank, RGS_unrank
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

## Permutations

```
class sympy.combinatorics.permutations.Permutation
```

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R9] (page 1236). For example, if one started with elements [x, y, a, b] (in that order) and they were reordered as [x, y, b, a] then the permutation would be [0, 1, 3, 2]. Notice that (in SymPy) the first element is always referred to as 0 and the permutation uses the indices of the elements in the original ordering, not the elements (a, b, etc...) themselves.

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = False
```

### See also:

[Cycle](#) (page 181)

## References

[R8] (page 1236), [R9] (page 1236), [R10] (page 1236), [R11] (page 1236), [R12] (page 1236), [R13] (page 1236), [R14] (page 1236)

### array\_form

Return a copy of the attribute `_array_form` Examples ======

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([[2,0], [3,1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2,0,3,1]]).array_form
[3, 2, 0, 1]
>>> Permutation([2,0,3,1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

### ascents()

Returns the positions of ascents in a permutation, ie, the location where  $p[i] < p[i+1]$

See also:

`descents` (page 167), `inversions` (page 171), `min` (page 175), `max` (page 175)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4,0,1,3,2])
>>> p.ascents()
[1, 2]
```

### atoms()

Returns all the elements of a permutation

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
set([0, 1, 2, 3, 4, 5])
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
set([0, 1, 2, 3, 4, 5])
```

### cardinality

Returns the number of all possible permutations.

See also:

`length` (page 174), `order` (page 176), `rank` (page 177), `size` (page 179)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.cardinality
24

commutator(x)
    Return the commutator of self and x:  $\sim x \sim \text{self} * x * \text{self}$ 

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. fg == gf.
```

## References

<http://en.wikipedia.org/wiki/Commutator>

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 2, 3, 1])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c ==  $\sim x \sim p * x * p$ 
True

>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
...
commutes_with(other)
    Checks if the elements are commuting.
```

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1,4,3,0,2,5])
>>> b = Permutation([0,1,2,3,4,5])
>>> a.commutes_with(b)
True
>>> b = Permutation([2,3,5,4,1,0])
>>> a.commutes_with(b)
False

cycle_structure
    Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.
```

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

### cycles

Returns the number of cycles contained in the permutation (including singletons).

See also:

[sympy.functions.combinatorial.numbers.stirling](#) (page 354)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

### cyclic\_form

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

See also:

[array\\_form](#) (page 165), [full\\_cyclic\\_form](#) (page 168)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

### descents()

Returns the positions of descents in a permutation, ie, the location where  $p[i] > p[i+1]$

See also:

[ascents](#) (page 165), [inversions](#) (page 171), [min](#) (page 175), [max](#) (page 175)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4,0,1,3,2])
>>> p.descents()
[0, 3]
```

**classmethod** `from_inversion_vector(inversion)`  
Calculates the permutation from the inversion vector.

#### Examples

```
>>> from sympy.combinatorics.permutations import Permutation  
>>> Permutation.print_cyclic = False  
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])  
Permutation([3, 2, 1, 0, 4, 5])
```

**classmethod** `from_sequence(i, key=None)`

Return the permutation needed to obtain `i` from the sorted elements of `i`. If custom sorting is desired, a key can be given.

#### Examples

```
>>> from sympy.combinatorics import Permutation  
>>> Permutation.print_cyclic = True  
  
>>> Permutation.from_sequence('SymPy')  
Permutation(4)(0, 1, 3)  
>>> _(sorted("SymPy"))  
['S', 'y', 'm', 'P', 'y']  
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())  
Permutation(4)(0, 2)(1, 3)
```

**full\_cyclic\_form**

Return permutation in cyclic form including singletons.

#### Examples

```
>>> from sympy.combinatorics.permutations import Permutation  
>>> Permutation([0, 2, 1]).full_cyclic_form  
[[0], [1, 2]]
```

**get\_adjacency\_distance(other)**

Computes the adjacency distance between two permutations.

This metric counts the number of times a pair  $i,j$  of jobs is adjacent in both  $p$  and  $p'$ . If  $n_{adj}$  is this quantity then the adjacency distance is  $n - n_{adj} - 1$  [1]

[1] Reeves, Colin R. Landscapes, Operators and Heuristic search, Annals of Operational Research, 86, pp 473-490. (1999)

**See also:**

`get_precedence_matrix` (page 170),    `get_precedence_distance` (page 169),  
`get_adjacency_matrix` (page 169)

#### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

### get\_adjacency\_matrix()

Computes the adjacency matrix of a permutation.

If job  $i$  is adjacent to job  $j$  in a permutation  $p$  then we set  $m[i, j] = 1$  where  $m$  is the adjacency matrix of  $p$ .

**See also:**

[get\\_precedence\\_matrix](#) (page 170), [get\\_precedence\\_distance](#) (page 169),  
[get\\_adjacency\\_distance](#) (page 168)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3,6,1)
>>> p.get_adjacency_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]])
```

### get\_positional\_distance(*other*)

Computes the positional distance between two permutations.

**See also:**

[get\\_precedence\\_distance](#) (page 169), [get\\_adjacency\\_distance](#) (page 168)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

**get\_precedence\_distance(*other*)**

Computes the precedence distance between two permutations.

Suppose  $p$  and  $p'$  represent  $n$  jobs. The precedence metric counts the number of times a job  $j$  is preceded by job  $i$  in both  $p$  and  $p'$ . This metric is commutative.

**See also:**

[get\\_precedence\\_matrix](#) (page 170), [get\\_adjacency\\_matrix](#) (page 169),  
[get\\_adjacency\\_distance](#) (page 168)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

**get\_precedence\_matrix()**

Gets the precedence matrix. This is used for computing the distance between two permutations.

**See also:**

[get\\_precedence\\_distance](#) (page 169), [get\\_adjacency\\_matrix](#) (page 169),  
[get\\_adjacency\\_distance](#) (page 168)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3,6,1)
>>> p
Permutation([2, 5, 3, 1, 4, 0])
>>> p.get_precedence_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 1, 0],
[1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 0, 0]])
```

**index()**

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts  $j$  such that  $p[j]$  is greater than  $p[j+1]$ .

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

```
inversion_vector()
Return the inversion vector of the permutation.
```

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

**See also:**

`from_inversion_vector` (page 167)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the  $i$ -th element cycling through 0.. $i$ .

```
>>> p = Permutation(2)
>>> while p:
...     print('%s %s %s' % (p, p.inversion_vector(), p.rank()))
...     p = p.next_lex()
...
Permutation([0, 1, 2]) [0, 0] 0
Permutation([0, 2, 1]) [0, 1] 1
Permutation([1, 0, 2]) [1, 0] 2
Permutation([1, 2, 0]) [1, 1] 3
Permutation([2, 0, 1]) [2, 0] 4
Permutation([2, 1, 0]) [2, 1] 5
```

`inversions()`  
Computes the number of inversions of a permutation.

An inversion is where  $i > j$  but  $p[i] < p[j]$ .

For small length of  $p$ , it iterates over all  $i$  and  $j$  values and calculates the number of inversions.  
For large length of  $p$ , it uses a variation of merge sort to calculate the number of inversions.

**See also:**

`descents` (page 167), `ascents` (page 165), `min` (page 175), `max` (page 175)

### References

[1] <http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3,4,5])
>>> p.inversions()
0
>>> Permutation([3,2,1,0]).inversions()
6
```

**is\_Empty**

Checks to see if the permutation is a set with zero elements

**See also:**

[is\\_Singleton](#) (page 172)

**Examples**

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

**is\_Identity**

Returns True if the Permutation is an identity permutation.

**See also:**

[order](#) (page 176)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
>>> p.is_Identity
False
```

**is\_Singleton**

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

**See also:**

[is\\_Empty](#) (page 172)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

### is\_even

Checks if a permutation is even.

See also:

[is\\_odd](#) (page 173)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.is_even
True
>>> p = Permutation([3,2,1,0])
>>> p.is_even
True
```

### is\_odd

Checks if a permutation is odd.

See also:

[is\\_even](#) (page 173)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.is_odd
False
>>> p = Permutation([3,2,0,1])
>>> p.is_odd
True
```

### classmethod josephus(*m, n, s=1*)

Return as a permutation the shuffling of range(*n*) using the Josephus scheme in which every *m*-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter *s* stops the selection process when there are *s* items remaining and these are selected by continuing the selection, counting by 1 rather than by *m*.

Consider selecting every 3rd item from 6 until only 2 remain:

choices	chosen
=====	=====
012345	
01 345	2
01 34	25

```
01 4    253
0 4    2531
0      25314
          253140
```

## References

- 1.[http://en.wikipedia.org/wiki/Flavius\\_Josephus](http://en.wikipedia.org/wiki/Flavius_Josephus)
- 2.[http://en.wikipedia.org/wiki/Josephus\\_problem](http://en.wikipedia.org/wiki/Josephus_problem)
- 3.<http://www.wou.edu/~burtonl/josephus.html>

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

### length()

Returns the number of integers moved by a permutation.

#### See also:

[min](#) (page 175), [max](#) (page 175), [support](#) (page 180), [cardinality](#) (page 165), [order](#) (page 176), [rank](#) (page 177), [size](#) (page 179)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

### list(size=None)

Return the permutation as an explicit list, possibly trimming unmoved elements if size is less than the maximum element in the permutation; if this is desired, setting `size=-1` will guarantee such trimming.

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Permutation(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
>>> Permutation(3).list(-1)
[]
```

`max()`  
The maximum element moved by the permutation.

**See also:**

[min](#) (page 175), [descents](#) (page 167), [ascents](#) (page 165), [inversions](#) (page 171)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([1,0,2,3,4])
>>> p.max()
1
```

`min()`  
The minimum element moved by the permutation.

**See also:**

[max](#) (page 175), [descents](#) (page 167), [ascents](#) (page 165), [inversions](#) (page 171)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,4,3,2])
>>> p.min()
2
```

`mul_inv(other)`  
other<sup>\*</sup>~self, self and other have `_array_form`

`next_lex()`

Returns the next permutation in lexicographical order. If self is the last permutation in lexicographical order it returns None. See [4] section 2.4.

**See also:**

[rank](#) (page 177), [unrank\\_lex](#) (page 180)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

`next_nonlex()`

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

See also:

[rank\\_nonlex](#) (page 177), [unrank\\_nonlex](#) (page 181)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6
```

#### next\_trotterjohnson()

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the `generate_bell` function.

See also:

[rank\\_trotterjohnson](#) (page 178), [unrank\\_trotterjohnson](#) (page 181),  
[sympy.utilities.iterables.generate\\_bell](#) (page 1133)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5
```

#### order()

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

See also:

[is\\_Identity](#) (page 172), [cardinality](#) (page 165), [length](#) (page 174), [rank](#) (page 177), [size](#) (page 179)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

**parity()**

Computes the parity of a permutation.

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of  $x$  and  $y$  such that  $x > y$  but  $p[x] < p[y]$ .

**See also:**

[\\_af\\_parity](#) (page 183)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.parity()
0
>>> p = Permutation([3,2,0,1])
>>> p.parity()
1
```

**classmethod random( $n$ )**

Generates a random permutation of length  $n$ .

Uses the underlying Python pseudo-random number generator.

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0, 1]))
True
```

**rank()**

Returns the lexicographic rank of the permutation.

**See also:**

[next\\_lex](#) (page 175), [unrank\\_lex](#) (page 180), [cardinality](#) (page 165), [length](#) (page 174), [order](#) (page 176), [size](#) (page 179)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

**rank\_nonlex( $inv\_perm=None$ )**

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

**See also:**

[next\\_nonlex](#) (page 175), [unrank\\_nonlex](#) (page 181)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.rank_nonlex()
23
```

### rank\_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

#### See also:

[unrank\\_trotterjohnson](#) (page 181), [next\\_trotterjohnson](#) (page 176)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2,3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0,2,1,3])
>>> p.rank_trotterjohnson()
7
```

### static rmul(\*args)

Return product of Permutations [a, b, c, ...] as the Permutation whose ith value is  $a(b(c(i)))$ .

a, b, c, ... can be Permutation objects or tuples.

## Notes

All items in the sequence will be parsed by Permutation as necessary as long as the first item is a Permutation:

```
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a `TypeError`.

## Examples

```
>>> from sympy.combinatorics.permutations import _af_rmul, Permutation
>>> Permutation.print_cyclic = False

>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the `*` operator:

```
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]

static rmul_with_af(*args)
    same as rmul, but the elements of args are Permutation objects which have _array_form

runs()
    Returns the runs of a permutation.

    An ascending sequence in a permutation is called a run [5].
```

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([2,5,7,3,6,0,1,4,8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1,3,2,0])
>>> q.runs()
[[1, 3], [2], [0]]

signature()
    Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

    The signature is calculated as  $(-1)^{\text{<number of inversions>}}$ 

See also:
    inversions (page 171)
```

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([0,1,2])
>>> p.inversions()
0
>>> p.signature()
1
>>> q = Permutation([0,2,1])
>>> q.inversions()
1
>>> q.signature()
-1

size
    Returns the number of elements in the permutation.

See also:
    cardinality (page 165), length (page 174), order (page 176), rank (page 177)
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([[3, 2], [0, 1]]).size
4

support()
Return the elements in permutation, P, for which P[i] != i.
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

`transpositions()`  
Return the permutation decomposed into a list of transpositions.

It is always possible to express a permutation as the product of transpositions, see [1]

### References

1.[http://en.wikipedia.org/wiki/Transposition\\_%28mathematics%29#Properties](http://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t]) == p
True

classmethod unrank_lex(size, rank)
Lexicographic permutation unranking.
```

See also:

`rank` (page 177), `next_lex` (page 175)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
```

```
>>> a
Permutation([0, 2, 4, 1, 3])
```

### classmethod unrank\_nonlex(*n, r*)

This is a linear time unranking algorithm that does not respect lexicographic order [3].

See also:

[next\\_nonlex](#) (page 175), [rank\\_nonlex](#) (page 177)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

### classmethod unrank\_trotterjohnson(*size, rank*)

Trotter Johnson permutation unranking. See [4] section 2.4.

See also:

[rank\\_trotterjohnson](#) (page 178), [next\\_trotterjohnson](#) (page 176)

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

## class `sympy.combinatorics.permutations.Cycle(*args)`

Wrapper around dict which provides the functionality of a disjoint cycle.

A cycle shows the rule to use to move subsets of elements to obtain a permutation. The Cycle class is more flexible than Permutation in that 1) all elements need not be present in order to investigate how multiple cycles act in sequence and 2) it can contain singletons:

```
>>> from sympy.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

```
>>> Cycle(1, 2)(2, 3)
Cycle(1, 3, 2)
```

The identity cycle, Cycle(), can be used to start a product:

```
>>> Cycle()(1, 2)(2,3)
Cycle(1, 3, 2)
```

The array form of a Cycle can be obtained by calling the list method (or passing it to the list function) and all elements from 0 will be shown:

```
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
```

```
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the list method and provide the desired size – but the Cycle cannot be truncated to a size smaller than the largest element that is out of place:

```
>>> b = Cycle(2,4)(1,2)(3,1,4)(1,3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown – as a singleton if necessary:

```
>>> Cycle(1, 4, 10)(4, 5)
Cycle(1, 5, 4, 10)
>>> Cycle(1, 2)(4)(5)(10)
Cycle(1, 2)(10)
```

The array form can be used to instantiate a Permutation so other properties of the permutation can be investigated:

```
>>> Perm(Cycle(1,2)(3,4).list()).transpositions()
[(1, 2), (3, 4)]
```

#### See also:

[Permutation](#) (page 164)

#### Notes

The underlying structure of the Cycle is a dictionary and although the `_iter_` method has been redefined to give the array form of the cycle, the underlying dictionary items are still available with the such methods as `items()`:

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

`list(size=None)`

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting `size=-1` will guarantee such trimming.

#### Examples

```
>>> from sympy.combinatorics.permutations import Cycle
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
```

```
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

`sympy.combinatorics.permutations._af_parity(pi)`

Computes the parity of a permutation in array form.

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of  $x$  and  $y$  such that  $x > y$  but  $p[x] < p[y]$ .

See also:

[Permutation](#) (page 164)

## Examples

```
>>> from sympy.combinatorics.permutations import _af_parity
>>> _af_parity([0,1,2,3])
0
>>> _af_parity([3,2,0,1])
1
```

## Generators

`static generators.symmetric(n)`

Generates the symmetric group of order  $n$ ,  $S_n$ .

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import symmetric
>>> list(symmetric(3))
[Permutation(2), Permutation(1, 2), Permutation(2)(0, 1),
 Permutation(0, 1, 2), Permutation(0, 2, 1), Permutation(0, 2)]
```

`static generators.cyclic(n)`

Generates the cyclic group of order  $n$ ,  $C_n$ .

See also:

[dihedral](#) (page 184)

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import cyclic
>>> list(cyclic(5))
```

```
[Permutation(4), Permutation(0, 1, 2, 3, 4), Permutation(0, 2, 4, 1, 3),
 Permutation(0, 3, 1, 4, 2), Permutation(0, 4, 3, 2, 1)]
```

**static generators.alternating(*n*)**

Generates the alternating group of order *n*,  $A_n$ .

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import alternating
>>> list(alternating(3))
[Permutation(2), Permutation(0, 1, 2), Permutation(0, 2, 1)]
```

**static generators.dihedral(*n*)**

Generates the dihedral group of order  $2n$ ,  $D_n$ .

The result is given as a subgroup of  $S_n$ , except for the special cases  $n=1$  (the group  $S_2$ ) and  $n=2$  (the Klein 4-group) where that's not possible and embeddings in  $S_2$  and  $S_4$  respectively are given.

**See also:**

[cyclic](#) (page 183)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.generators import dihedral
>>> list(dihedral(3))
[Permutation(2), Permutation(0, 2), Permutation(0, 1, 2),
 Permutation(1, 2), Permutation(0, 2, 1), Permutation(2)(0, 1)]
```

## Permutation Groups

**class sympy.combinatorics.perm\_groups.PermutationGroup**

The class defining a Permutation group.

`PermutationGroup([p1, p2, ..., pn])` returns the permutation group generated by the list of permutations. This group can be supplied to `Polyhedron` if one desires to decorate the elements to which the indices of the permutation refer.

**See also:**

[sympy.combinatorics.polyhedron.Polyhedron](#) (page 213), [sympy.combinatorics.permutations.Permutation](#) (page 164)

**References**

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of Computational Group Theory"

[2] Seress, A. "Permutation Group Algorithms"

[3] [http://en.wikipedia.org/wiki/Schreier\\_vector](http://en.wikipedia.org/wiki/Schreier_vector)

- [4] [#Product\\_replacement\\_algorithm](http://en.wikipedia.org/wiki/Nielsen_transformation)
- [5] Frank Celler, Charles R.Leedham-Green, Scott H.Murray, Alice C.Niemeyer, and E.A.O'Brien.  
“Generating Random Elements of a Finite Group”
- [6] [http://en.wikipedia.org/wiki/Block\\_%28permutation\\_group\\_theory%29](http://en.wikipedia.org/wiki/Block_%28permutation_group_theory%29)
- [7] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)
- [8] [http://en.wikipedia.org/wiki/Multiply\\_transitive\\_group#Multiply\\_transitive\\_groups](http://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups)
- [9] [http://en.wikipedia.org/wiki/Center\\_%28group\\_theory%29](http://en.wikipedia.org/wiki/Center_%28group_theory%29)
- [10] [http://en.wikipedia.org/wiki/Centralizer\\_and\\_normalizer](http://en.wikipedia.org/wiki/Centralizer_and_normalizer)
- [11] [http://groupprops.subwiki.org/wiki/Derived\\_subgroup](http://groupprops.subwiki.org/wiki/Derived_subgroup)
- [12] [http://en.wikipedia.org/wiki/Nilpotent\\_group](http://en.wikipedia.org/wiki/Nilpotent_group)
- [13] <http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf>

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.permutations import Cycle
>>> from sympy.combinatorics.polyhedron import Polyhedron
>>> from sympy.combinatorics.perm_groups import PermutationGroup
```

The permutations corresponding to motion of the front, right and bottom face of a 2x2 Rubik's cube are defined:

```
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to PermutationGroup:

```
>>> G = PermutationGroup(F, R, D)
>>> G.order()
3674160
```

The group can be supplied to a Polyhedron in order to track the objects being moved. An example involving the 2x2 Rubik's cube is given there, but here is a simple demonstration:

```
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron(list('ABC'), pgroup=G)
>>> P.corners
(A, B, C)
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

#### `_eq_(other)`

Return True if self and other have the same generators.

#### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G = PermutationGroup([p, p**2])
>>> H = PermutationGroup([p**2, p])
>>> G.generators == H.generators
False
>>> G == H
True
```

#### `_mul_(other)`

Return the direct product of two permutation groups as a permutation group.

This implementation realizes the direct product by shifting the index set for the generators of the second group: so if we have G acting on n1 points and H acting on n2 points, G\*H acts on n1 + n2 points.

#### Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(5)
>>> H = G*G
>>> H
PermutationGroup([
    Permutation(9)(0, 1, 2, 3, 4),
    Permutation(5, 6, 7, 8, 9)])
>>> H.order()
25
```

#### `static __new__(*args, **kwargs)`

The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless `dups` keyword is False.

#### `__weakref__`

list of weak references to the object (if defined)

#### `_random_pr_init(r, n, _random_prec_n=None)`

Initialize random generators for the product replacement algorithm.

The implementation uses a modification of the original product replacement algorithm due to Leedham-Green, as described in [1], pp. 69-71; also, see [2], pp. 27-29 for a detailed theoretical analysis of the original product replacement algorithm, and [4].

The product replacement algorithm is used for producing random, uniformly distributed elements of a group `G` with a set of generators `S`. For the initialization `_random_pr_init`, a list `R` of  $\max\{r, |S|\}$  group generators is created as the attribute `G._random_gens`, repeating elements of `S` if

necessary, and the identity element of  $G$  is appended to  $R$  - we shall refer to this last element as the accumulator. Then the function `random_pr()` is called  $n$  times, randomizing the list  $R$  while preserving the generation of  $G$  by  $R$ . The function `random_pr()` itself takes two random elements  $g$ ,  $h$  among all elements of  $R$  but the accumulator and replaces  $g$  with a randomly chosen element from  $\{gh, g(\bar{h}), hg, (\bar{h})g\}$ . Then the accumulator is multiplied by whatever  $g$  was replaced by. The new value of the accumulator is then returned by `random_pr()`.

The elements returned will eventually (for  $n$  large enough) become uniformly distributed across  $G$  ([5]). For practical purposes however, the values  $n = 50$ ,  $r = 11$  are suggested in [1].

**See also:**

`random_pr` (page 208)

#### Notes

THIS FUNCTION HAS SIDE EFFECTS: it changes the attribute `self._random_gens`

`_union_find_merge(first, second, ranks, parents, not_rep)`

Merges two classes in a union-find data structure.

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. The class merging process uses union by rank as an optimization. ([7])

**See also:**

`minimal_block` (page 204), `_union_find_rep` (page 187)

#### Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, the list of class sizes, `ranks`, and the list of elements that are not representatives, `not_rep`, are changed due to class merging.

#### References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[7] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)

`_union_find_rep(num, parents)`

Find representative of a class in a union-find data structure.

Used in the implementation of Atkinson's algorithm as suggested in [1], pp. 83-87. After the representative of the class to which `num` belongs is found, path compression is performed as an optimization ([7]).

**See also:**

`minimal_block` (page 204), `_union_find_merge` (page 187)

#### Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, is altered due to path compression.

## References

- [1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"
- [7] [http://www.algorithmist.com/index.php/Union\\_Find](http://www.algorithmist.com/index.php/Union_Find)

### base

Return a base from the Schreier-Sims algorithm.

For a permutation group  $G$ , a base is a sequence of points  $B = (b_1, b_2, \dots, b_k)$  such that no element of  $G$  apart from the identity fixes all the points in  $B$ . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of  $B$  is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

#### See also:

`strong_gens` (page 211), `basic_transversals` (page 190), `basic_orbits` (page 189), `basic_stabilizers` (page 189)

## Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

`baseswap(base, strong_gens, pos, randomized=False, transversals=None, basic_orbits=None, strong_gens_distr=None)`

Swap two consecutive base points in base and strong generating set.

If a base for a group  $G$  is given by  $(b_1, b_2, \dots, b_k)$ , this function returns a base  $(b_1, b_2, \dots, b_{i+1}, b_i, \dots, b_k)$ , where  $i$  is given by `pos`, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

#### Parameters base, strong\_gens

The base and strong generating set.

#### pos

The position at which swapping is performed.

#### randomized

A switch between randomized and deterministic version.

#### transversals

The transversals for the basic orbits, if known.

#### basic\_orbits

The basic orbits, if known.

#### strong\_gens\_distr

The strong generators distributed by basic stabilizers, if known.

#### Returns (base, strong\_gens)

`base` is the new base, and `strong_gens` is a generating set relative to it.

See also:

`schreier_sims` (page 208)

#### Notes

The deterministic version of the algorithm is discussed in [1], pp. 102-103; the randomized version is discussed in [1], p.103, and [2], p.98. It is of Las Vegas type. Notice that [1] contains a mistake in the pseudocode and discussion of BASESWAP: on line 3 of the pseudocode,  $\beta_{i+1}^{\{ \langle \rangle \}} \leftarrow \beta_i^{\{ \langle \rangle \}}$  should be replaced by  $\beta_i^{\{ \langle \rangle \}} \leftarrow \beta_{i+1}^{\{ \langle \rangle \}}$ , and the same for the discussion of the algorithm.

#### Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1, randomized=False)
>>> base, gens
([0, 2, 1],
[Permutation(0, 1, 2, 3), Permutation(3)(0, 1), Permutation(1, 3, 2),
 Permutation(2, 3), Permutation(1, 3)])
```

check that `base, gens` is a BSGS

```
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```

#### basic\_orbits

Return the basic orbits relative to a base and strong generating set.

If  $(b_1, b_2, \dots, b_k)$  is a base for a group  $G$ , and  $G^{\{i\}} = G_{\{b_1, b_2, \dots, b_{i-1}\}}$  is the  $i$ -th basic stabilizer (so that  $G^{\{1\}} = G$ ), the  $i$ -th basic orbit relative to this base is the orbit of  $b_i$  under  $G^{\{i\}}$ . See [1], pp. 87-89 for more information.

See also:

`base` (page 188), `strong_gens` (page 211), `basic_transversals` (page 190), `basic_stabilizers` (page 189)

#### Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

#### basic\_stabilizers

Return a chain of stabilizers relative to a base and strong generating set.

The  $i$ -th basic stabilizer  $G^{\{i\}}$  relative to a base  $(b_1, b_2, \dots, b_k)$  is  $G_{\{b_1, b_2, \dots, b_{i-1}\}}$ . For more information, see [1], pp. 87-89.

See also:

[base](#) (page 188), [strong\\_gens](#) (page 211), [basic\\_orbits](#) (page 189), [basic\\_transversals](#) (page 190)

### Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
...
PermutationGroup([
    Permutation(3)(0, 1, 2),
    Permutation(1, 2, 3)])
PermutationGroup([
    Permutation(1, 2, 3)])
```

### basic\_transversals

Return basic transversals relative to a base and strong generating set.

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

See also:

[strong\\_gens](#) (page 211), [base](#) (page 188), [basic\\_orbits](#) (page 189), [basic\\_stabilizers](#) (page 189)

### Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.basic_transversals
[{0: Permutation(3),
  1: Permutation(3)(0, 1, 2),
  2: Permutation(3)(0, 2, 1),
  3: Permutation(0, 3, 1)},
 {1: Permutation(3),
  2: Permutation(1, 2, 3),
  3: Permutation(1, 3, 2)}]
```

### center()

Return the center of a permutation group.

The center for a group  $G$  is defined as  $Z(G) = \{z \in G \mid \forall g \in G, zg = gz\}$ , the set of elements of  $G$  that commute with all elements of  $G$ . It is equal to the centralizer of  $G$  inside  $G$ , and is naturally a subgroup of  $G$  ([9]).

See also:

`centralizer` (page 191)

#### Notes

This is a naive implementation that is a straightforward application of `.centralizer()`

#### Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

`centralizer(other)`

Return the centralizer of a group/set/element.

The centralizer of a set of permutations  $S$  inside a group  $G$  is the set of elements of  $G$  that commute with all elements of  $S$ :

$C_G(S) = \{ g \in G \mid gs = sg \text{ for all } s \in S \}$  ([10])

Usually,  $S$  is a subset of  $G$ , but if  $G$  is a proper subgroup of the full symmetric group, we allow for  $S$  to have elements outside  $G$ .

It is naturally a subgroup of  $G$ ; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.

#### Parameters other

a permutation group/list of permutations/single permutation

#### See also:

`subgroup_search` (page 211)

#### Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group  $G$ .

#### Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

```
commutator(G, H)
```

Return the commutator of two subgroups.

For a permutation group  $K$  and subgroups  $G, H$ , the commutator of  $G$  and  $H$  is defined as the group generated by all the commutators  $[g, h] = hgh^{-1}g^{-1}$  for  $g$  in  $G$  and  $h$  in  $H$ . It is naturally a subgroup of  $K$  ([1], p.27).

**See also:**

[derived\\_subgroup](#) (page 195)

## Notes

The commutator of two subgroups  $H, G$  is equal to the normal closure of the commutators of all the generators, i.e.  $hgh^{-1}g^{-1}$  for  $h$  a generator of  $H$  and  $g$  a generator of  $G$  ([1], p.28)

## Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

```
contains(g, strict=True)
```

Test if permutation  $g$  belong to self,  $G$ .

If  $g$  is an element of  $G$  it can be written as a product of factors drawn from the cosets of  $G$ 's stabilizers. To see if  $g$  is one of the actual generators defining the group use  $G.\text{has}(g)$ .

If `strict` is not True,  $g$  will be resized, if necessary, to match the size of permutations in `self`.

**See also:**

[coset\\_factor](#) (page 193), [sympy.core.basic.Basic.has](#) (page 67)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup

>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0]) # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False
```

If `strict` is False, a permutation will be resized, if necessary:

```
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True
```

To test if a given permutation is present in the group:

```
>>> elem in G.generators
False
>>> G.has(elem)
False
```

```
coset_factor(g, factor_index=False)
Return G's (self's) coset factorization of g
```

If  $g$  is an element of  $G$  then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in  $f$  are those for which the product gives  $g$ :  $g = f[n]*\dots*f[1]*f[0]$  where  $n = \text{len}(B)$  and  $B = G.\text{base}$ .  $f[i]$  is one of the permutations in  $\text{self}.\text{_basic\_orbits}[i]$ .

If  $\text{factor\_index}==\text{True}$ , returns a tuple  $[b[0], \dots, b[n]]$ , where  $b[i]$  belongs to  $\text{self}.\text{_basic\_orbits}[i]$

## Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> Permutation.print_cyclic = True
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
```

Define  $g$ :

```
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of  $G$ :

```
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from  $u$ . See below that a factor from  $u_1$  and  $u_2$  and the Identity permutation have been used:

```
>>> f = G.coset_factor(g)
>>> f[2]*f[1]*f[0] == g
True
>>> f1 = G.coset_factor(g, True); f1
[0, 4, 4]
>>> tr = G.basic_transversals
>>> f[0] == tr[0][f1[0]]
True
```

If  $g$  is not an element of  $G$  then  $[]$  is returned:

```
>>> c = Permutation(5, 6, 7)
>>> G.coset_factor(c)
[]
```

see `util._strip`

`coset_rank(g)`

rank using Schreier-Sims representation

The coset rank of `g` is the ordering number in which it appears in the lexicographic listing according to the coset decomposition

The ordering is the same as in `G.generate(method='coset')`. If `g` does not belong to the group it returns `None`.

**See also:**

`coset_factor` (page 193)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
Permutation(7)(2, 4)(3, 5)
```

`coset_unrank(rank, af=False)`

unrank using Schreier-Sims representation

`coset_unrank` is the inverse operation of `coset_rank` if  $0 \leq \text{rank} < \text{order}$ ; otherwise it returns `None`.

`degree`

Returns the size of the permutations in the group.

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

**See also:**

`order` (page 206)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
```

```
>>> list(G.generate())
[Permutation(2), Permutation(2)(0, 1)]
```

#### derived\_series()

Return the derived series for the group.

The derived series for a group  $G$  is defined as  $G = G_0 > G_1 > G_2 > \dots$  where  $G_i = [G_{i-1}, G_{i-1}]$ , i.e.  $G_i$  is the derived subgroup of  $G_{i-1}$ , for  $i \in \mathbb{N}$ . When we have  $G_k = G_{k-1}$  for some  $k \in \mathbb{N}$ , the series terminates.

**Returns** A list of permutation groups containing the members of the derived series in the order  $G = G_0, G_1, G_2, \dots$ .

**See also:**

[derived\\_subgroup](#) (page 195)

#### Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True
```

#### derived\_subgroup()

Compute the derived subgroup.

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators  $[g, h] = hgh^{-1}g^{-1}$  for  $g, h \in G$ ; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

**See also:**

[derived\\_series](#) (page 195)

#### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))
[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]
```

#### generate(*method='coset'*, *af=False*)

Return iterator to generate the elements of the group

Iteration is done with one of these methods:

```
method='coset'  using the Schreier-Sims coset representation
method='dimino' using the Dimino method
```

If af = True it yields the array form of the permutations

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is not true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group()
False
```

But the group generated by the permutations in the tetrahedron pgroup – even the first two – is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([
    Permutation(0, 1)(2, 3),
    Permutation(3),
    Permutation(1, 2, 3),
    Permutation(1, 3, 2),
    Permutation(0, 3, 1),
    Permutation(0, 2, 3),
    Permutation(0, 3)(1, 2),
    Permutation(0, 1, 3),
    Permutation(3)(0, 2, 1),
    Permutation(0, 3, 2),
    Permutation(3)(0, 1, 2),
    Permutation(0, 2)(1, 3)])
>>> _.is_group()
True
```

generate\_dimino(af=False)

Yield group elements using Dimino's algorithm

If af == True it yields the array form of the permutations

### References

[1] The Implementation of Various Algorithms for Permutation Groups in the Computer Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
```

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]

generate_schreier_sims(af=False)
Yield group elements using the Schreier-Sims representation in coset_rank order
If af = True it yields the array form of the permutations
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 3, 2, 1],
 [0, 1, 3, 2], [0, 2, 3, 1], [0, 3, 1, 2]]
```

### generators

Returns the generators of the group.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[Permutation(1, 2), Permutation(2)(0, 1)]
```

### is\_abelian

Test if the group is Abelian.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.is_abelian
False
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
```

```
>>> G.is_abelian
True

is_alt_sym(eps=0.05, _random_prec=None)
Monte Carlo test for the symmetric/alternating group for degrees >= 8.
```

More specifically, it is one-sided Monte Carlo with the answer True (i.e.,  $G$  is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability  $\text{eps}$ .

#### See also:

[sympy.combinatorics.util.\\_check\\_cycles\\_alt\\_sym](#) (page 233)

#### Notes

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group  $G$  of degree  $n$  contains an element with a cycle of length  $n/2 < p < n-2$  for  $p$  a prime,  $G$  is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately  $\log(2)/\log(n)$  ([1], p.82; [2], pp. 226-227). The helper function `_check_cycles_alt_sym` is used to go over the cycles in a permutation and look for ones satisfying 1).

#### Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

#### is\_group()

Return True if the group meets three criteria: identity is present, the inverse of every element is also an element, and the product of any two elements is also an element. If any of the tests fail, False is returned.

#### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics import PermutationGroup
>>> from sympy.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is not a true group:

```
>>> G = tetrahedron.pgroup
>>> G.is_group()
False
```

But the group generated by the permutations in the tetrahedron pgroup is a proper group:

```
>>> H = PermutationGroup(list(G.generate()))
>>> H.is_group()
True
```

The identity permutation is present:

```
>>> H.has(Permutation(G.degree - 1))
True
```

The product of any two elements from the group is also in the group:

```
>>> from sympy import TableForm
>>> g = list(H)
>>> n = len(g)
>>> m = []
>>> for i in g:
...     m.append([g.index(i*H) for H in g])
...
>>> TableForm(m, headings=[range(n), range(n)], wipe_zeros=False)
| 0  1  2  3  4  5  6  7  8  9  10 11
-----+
0 | 11 0  8  10 6  2  7  4  5  3  9  1
1 | 0  1  2  3  4  5  6  7  8  9  10 11
2 | 6  2  7  4  5  3  9  1  11 0  8  10
3 | 5  3  9  1  11 0  8  10 6  2  7  4
4 | 3  4  0  2  10 6  11 8  9  7  1  5
5 | 4  5  6  7  8  9  10 11 0  1  2  3
6 | 10 6  11 8  9  7  1  5  3  4  0  2
7 | 9  7  1  5  3  4  0  2  10 6  11 8
8 | 7  8  4  6  2  10 3  0  1  11 5  9
9 | 8  9  10 11 0  1  2  3  4  5  6  7
10 | 2  10 3  0  1  11 5  9  7  8  4  6
11 | 1  11 5  9  7  8  4  6  2  10 3  0
>>>
```

The entries in the table give the element in the group corresponding to the product of a given column element and row element:

```
>>> g[3]*g[2] == g[9]
True
```

The inverse of every element is also in the group:

```
>>> TableForm([[g.index(~gi) for gi in g]], headings=[[[]], range(n)],
...           wipe_zeros=False)
0  1  2  3  4  5  6  7  8  9  10 11
-----
11 1  7  3  10 9  6  2  8  5  4  0
```

So we see that  $g[1]$  and  $g[3]$  are equivalent to their inverse while  $g[7] = \sim g[2]$ .

#### `is_nilpotent`

Test if the group is nilpotent.

A group  $G$  is nilpotent if it has a central series of finite length. Alternatively,  $G$  is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

#### See also:

`lower_central_series` (page 202), `is_solvable` (page 201)

## Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> C = CyclicGroup(6)
>>> C.is_nilpotent
True
>>> S = SymmetricGroup(5)
>>> S.is_nilpotent
False
```

### is\_normal(*gr*)

Test if  $G=\text{self}$  is a normal subgroup of  $gr$ .

$G$  is normal in  $gr$  if for each  $g_2$  in  $G$ ,  $g_1$  in  $gr$ ,  $g = g_1 * g_2 * g_1^{-1}$  belongs to  $G$ . It is sufficient to check this for each  $g_1$  in  $gr.\text{generator}$  and  $g_2$  in  $G.\text{generator}$

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

### is\_primitive(*randomized=True*)

Test if a group is primitive.

A permutation group  $G$  acting on a set  $S$  is called primitive if  $S$  contains no nontrivial block under the action of  $G$  (a block is nontrivial if its cardinality is more than 1).

**See also:**

[minimal\\_block](#) (page 204), [random\\_stab](#) (page 208)

## Notes

The algorithm is described in [1], p.83, and uses the function `minimal_block` to search for blocks of the form  $\{0, k\}$  for  $k$  ranging over representatives for the orbits of  $G_0$ , the stabilizer of 0. This algorithm has complexity  $O(n^2)$  where  $n$  is the degree of the group, and will perform badly if  $G_0$  is small.

There are two implementations offered: one finds  $G_0$  deterministically using the function `stabilizer`, and the other (default) produces random elements of  $G_0$  using `random_stab`, hoping that they generate a subgroup of  $G_0$  with not too many more orbits than  $G_0$  (this is suggested in [1], p.83). Behavior is changed by the `randomized` flag.

## Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False

is_solvable
Test if the group is solvable.

G is solvable if its derived series terminates with the trivial group ([1], p.29).

See also:
    is_nilpotent (page 199), derived_series (page 195)
```

### Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True

is_subgroup(G, strict=True)
Return True if all elements of self belong to G.

If strict is False then if self's degree is smaller than G's, the elements will be resized to have the same degree.
```

### Examples

```
>>> from sympy.combinatorics import Permutation, PermutationGroup
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
...     CyclicGroup)
```

Testing is strict by default: the degree of each group must be the same:

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set `strict` to False:

```
>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
```

```
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False

is_transitive(strict=True)
Test if the group is transitive.

A group is transitive if it has a single orbit.

If strict is False the group is transitive if it has a single orbit of length different from 1.
```

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False

is_trivial
Test if the group is the trivial group.

This is true if the group contains only the identity permutation.
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True

lower_central_series()
Return the lower central series for the group.

The lower central series for a group G is the series  $G = G_0 > G_1 > G_2 > \dots$  where  $G_k = [G, G_{k-1}]$ , i.e. every term after the first is equal to the commutator of G and the previous term in G1 ([1], p.29).

Returns A list of permutation groups in the order

G = G_0, G_1, G_2, \dots
```

See also:

[commutator](#) (page 191), [derived\\_series](#) (page 195)

### Examples

```
>>> from sympy.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

`make_perm(n, seed=None)`

Multiply `n` randomly selected permutations from `pgroup` together, starting with the identity permutation. If `n` is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: `make_perm((A, B, C))` will give `CBA(I)` where `I` is the identity permutation.

`seed` is used to set the seed for the random selection of permutations from `pgroup`. If this is a list of integers, the corresponding permutations from `pgroup` will be selected in the order give. This is mainly used for testing purposes.

See also:

[random](#) (page 208)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm(1, [0])
Permutation(0, 1)(2, 3)
>>> G.make_perm(3, [0, 1, 0])
Permutation(0, 2, 3, 1)
>>> G.make_perm([0, 1, 0])
Permutation(0, 2, 3, 1)
```

`max_div`

Maximum proper divisor of the degree of a permutation group.

See also:

[minimal\\_block](#) (page 204), [\\_union\\_find\\_merge](#) (page 187)

### Notes

Obviously, this is the degree divided by its minimal proper divisor (larger than 1, if one exists). As it is guaranteed to be prime, the `sieve` from `sympy.nttheory` is used. This function is also used as an optimization tool for the functions `minimal_block` and `_union_find_merge`.

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2

minimal_block(points)
```

For a transitive group, finds the block system generated by `points`.

If a group  $G$  acts on a set  $S$ , a nonempty subset  $B$  of  $S$  is called a block under the action of  $G$  if for all  $g$  in  $G$  we have  $gB = B$  ( $g$  fixes  $B$ ) or  $gB$  and  $B$  have no common points ( $g$  moves  $B$  entirely). ([1], p.23; [6]).

The distinct translates  $gB$  of a block  $B$  for  $g$  in  $G$  partition the set  $S$  and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides  $|S|$  ([1], p.23). A  $G$ -congruence is an equivalence relation  $\sim$  on the set  $S$  such that  $a \sim b$  implies  $g(a) \sim g(b)$  for all  $g$  in  $G$ . For a transitive group, the equivalence classes of a  $G$ -congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the  $G$ -congruence generated by the pairs  $(p_0, p_1), (p_0, p_2), \dots, (p_0, p_{\{k-1\}})$  which is the same as finding the maximal block system (i.e., the one with minimum block size) such that  $p_0, \dots, p_{\{k-1\}}$  are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set  $S$  using a union-find data structure. The running time is just above  $O(|points||S|)$ . ([1], pp. 83-87; [7]).

### See also:

`_union_find_rep` (page 187), `_union_find_merge` (page 187), `is_transitive` (page 202), `is_primitive` (page 200)

## Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.minimal_block([0, 5])
[0, 6, 2, 8, 4, 0, 6, 2, 8, 4]
>>> D.minimal_block([0, 1])
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

`normal_closure(other, k=10)`

Return the normal closure of a subgroup/set of permutations.

If  $S$  is a subset of a group  $G$ , the normal closure of  $A$  in  $G$  is defined as the intersection of all normal subgroups of  $G$  that contain  $A$  ([1], p.14). Alternatively, it is the group generated by the conjugates  $x^{-1}yx$  for  $x$  a generator of  $G$  and  $y$  a generator of the subgroup  $\langle S \rangle$  generated by  $S$  (for some chosen generating set for  $\langle S \rangle$ ) ([1], p.73).

### Parameters other

a subgroup/list of permutations/single permutation

`k`

an implementation-specific parameter that determines the number of conjugates that are adjoined to `other` at once

See also:

[commutator](#) (page 191), [derived\\_subgroup](#) (page 195), [random\\_pr](#) (page 208)

#### Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

#### Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

`orbit(alpha, action='tuples')`

Compute the orbit of alpha  $\{g(\alpha) \mid g \in G\}$  as a set.

The time complexity of the algorithm used here is  $O(|Orb| * r)$  where  $|Orb|$  is the size of the orbit and  $r$  is the number of generators of the group. For a more detailed analysis, see [1], p.78, [2], pp. 19-21. Here alpha can be a single point, or a list of points.

If alpha is a single point, the ordinary orbit is computed. If alpha is a list of points, there are three available options:

‘union’ - computes the union of the orbits of the points in the list  
‘tuples’ - computes the orbit of the list interpreted as an ordered tuple under the group action ( i.e.,  $g((1,2,3)) = (g(1), g(2), g(3))$  )  
‘sets’ - computes the orbit of the list interpreted as a sets

See also:

[orbit\\_transversal](#) (page 206)

#### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
set([0, 1, 2])
>>> G.orbit([0, 4], 'union')
set([0, 1, 2, 3, 4, 5, 6])
```

`orbit_rep(alpha, beta, schreier_vector=None)`

Return a group element which sends `alpha` to `beta`.

If `beta` is not in the orbit of `alpha`, the function returns `False`. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

See also:

`schreier_vector` (page 210)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
Permutation(0, 4, 1, 2, 3)
```

`orbit_transversal(alpha, pairs=False)`

Computes a transversal for the orbit of `alpha` as a set.

For a permutation group `G`, a transversal for the orbit  $\text{Orb} = \{g(\alpha) \mid g \in G\}$  is a set  $\{g_\beta \mid g_\beta(\alpha) = \beta\}$  for  $\beta \in \text{Orb}$ . Note that there may be more than one possible transversal. If `pairs` is set to `True`, it returns the list of pairs  $(\beta, g_\beta)$ . For a proof of correctness, see [1], p.79

See also:

`orbit` (page 205)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[Permutation(5),
 Permutation(0, 1, 2, 3, 4, 5),
 Permutation(0, 5)(1, 4)(2, 3),
 Permutation(0, 2, 4)(1, 3, 5),
 Permutation(5)(0, 4)(1, 3),
 Permutation(0, 3)(1, 4)(2, 5)]
```

`orbits(rep=False)`

Return the orbits of self, ordered according to lowest element in each orbit.

### Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[set([0, 2, 3, 4, 6]), set([1, 5])]
```

**order()**

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

**See also:**

`degree` (page 194)

**Examples**

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup

>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[Permutation(2), Permutation(2)(0, 1)]
```

```
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

**pointwise\_stabilizer(*points*, *incremental=True*)**

Return the pointwise stabilizer for a set of points.

For a permutation group  $G$  and a set of points  $\{p_1, p_2, \dots, p_k\}$ , the pointwise stabilizer of  $p_1, p_2, \dots, p_k$  is defined as  $G_{\{p_1, \dots, p_k\}} = \{g \in G \mid g(p_i) = p_i \text{ for all } i \in \{1, 2, \dots, k\}\}$  ([1], p20). It is a subgroup of ' $G$ '.

**See also:**

`stabilizer` (page 210), `schreier_sims_incremental` (page 208)

**Notes**

When `incremental == True`, rather than the obvious implementation using successive calls to `.stabilizer()`, this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

**Examples**

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
```

```
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True

random(af=False)
    Return a random group element

random_pr(gen_count=11, iterations=50, random_prec=None)
    Return a random group element using product replacement.
```

For the details of the product replacement algorithm, see `_random_pr_init`. In `random_pr` the actual ‘product replacement’ is performed. Notice that if the attribute `_random_gens` is empty, it needs to be initialized by `_random_pr_init`.

**See also:**

[\\_random\\_pr\\_init](#) (page 186)

`random_stab(alpha, schreier_vector=None, random_prec=None)`  
Random element from the stabilizer of `alpha`.

The Schreier vector for `alpha` is an optional argument used for speeding up repeated calls. The algorithm is described in [1], p.81

**See also:**

[random\\_pr](#) (page 208), [orbit\\_rep](#) (page 205)

`schreier_sims()`  
Schreier-Sims algorithm.

It computes the generators of the chain of stabilizers  $G > G_{\{b_1\}} > \dots > G_{\{b_1, \dots, b_r\}} > 1$  in which  $G_{\{b_1, \dots, b_i\}}$  stabilizes  $b_1, \dots, b_i$ , and the corresponding `s` cosets. An element of the group can be written as the product  $h_1 * \dots * h_s$ .

We use the incremental Schreier-Sims algorithm.

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[{:0: Permutation(2)(0, 1), :1: Permutation(2), :2: Permutation(1, 2)},
{:0: Permutation(2), :2: Permutation(0, 2)}]
```

`schreier_sims_incremental(base=None, gens=None)`  
Extend a sequence of points and generating set to a base and strong generating set.

**Parameters** `base`

The sequence of points to be extended to a base. Optional parameter with default value `[]`.

`gens`

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value `self.generators`.

**Returns** (`base, strong_gens`)

`base` is the base obtained, and `strong_gens` is the strong generating set relative to it. The original parameters `base`, `gens` remain unchanged.

See also:

[schreier\\_sims](#) (page 208), [schreier\\_sims\\_random](#) (page 209)

#### Notes

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, `base` and `gens` are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators `gens`, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

#### Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(7)
>>> base = [2, 3]
>>> seq = [2, 3]
>>> base, strong_gens = A.schreier_sims_incremental(base=seq)
>>> _verify_bsgs(A, base, strong_gens)
True
>>> base[:2]
[2, 3]
```

```
schreier_sims_random(base=None, gens=None, consec_succ=10, _random_prec=None)
Randomized Schreier-Sims algorithm.
```

The randomized Schreier-Sims algorithm takes the sequence `base` and the generating set `gens`, and extends `base` to a base, and `gens` to a strong generating set relative to that base with probability of a wrong answer at most  $2^{-\{\text{consec\_succ}\}}$ , provided the random generators are sufficiently random.

#### Parameters base

The sequence to be extended to a base.

#### gens

The generating set to be extended to a strong generating set.

#### consec\_succ

The parameter defining the probability of a wrong answer.

#### \_random\_prec

An internal parameter used for testing purposes.

#### Returns (base, strong\_gens)

`base` is the base and `strong_gens` is the strong generating set relative to it.

See also:

[schreier\\_sims](#) (page 208)

## Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits `orbs` and the permutation groups `stabs` to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to “sift” random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function `_strip` is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend `stabs`, `base`, `gens` and `orbs` accordingly. The halting condition is for `consec_succ` consecutive successful sifts to pass. This makes sure that the current `base` and `gens` form a BSGS with probability at least  $1 - 1/\text{consec\_succ}$ .

## Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

`schreier_vector(alpha)`

Computes the schreier vector for `alpha`.

The Schreier vector efficiently stores information about the orbit of `alpha`. It can later be used to quickly obtain elements of the group that send `alpha` to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use “None” instead of 0 to signify that an element doesn’t belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.

**See also:**

`orbit` (page 205)

## Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

`stabilizer(alpha)`

Return the stabilizer subgroup of `alpha`.

The stabilizer of  $\alpha$  is the group  $G_\alpha = \{g \in G \mid g(\alpha) = \alpha\}$ . For a proof of correctness, see [1], p.79.

**See also:**

`orbit` (page 205)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([
    Permutation(5)(0, 4)(1, 3),
    Permutation(5)])
```

### strong\_gens

Return a strong generating set from the Schreier-Sims algorithm.

A generating set  $S = \{g_1, g_2, \dots, g_t\}$  for a permutation group  $G$  is a strong generating set relative to the sequence of points (referred to as a “base”)  $(b_1, b_2, \dots, b_k)$  if, for  $1 \leq i \leq k$  we have that the intersection of the pointwise stabilizer  $G^{\{(i+1)\}} := G - \{b_1, b_2, \dots, b_i\}$  with  $S$  generates the pointwise stabilizer  $G^{\{(i+1)\}}$ . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

See also:

[base](#) (page 188), [basic\\_transversals](#) (page 190), [basic\\_orbits](#) (page 189), [basic\\_stabilizers](#) (page 189)

## Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> D.strong_gens
[Permutation(0, 1, 2, 3), Permutation(0, 3)(1, 2), Permutation(1, 3)]
>>> D.base
[0, 1]
```

### subgroup\_search(prop, base=None, strong\_gens=None, tests=None, init\_subgroup=None)

Find the subgroup of all elements satisfying the property `prop`.

This is done by a depth-first search with respect to base images that uses several tests to prune the search tree.

#### Parameters prop

The property to be used. Has to be callable on group elements and always return `True` or `False`. It is assumed that all group elements satisfying `prop` indeed form a subgroup.

#### base

A base for the supergroup.

#### strong\_gens

A strong generating set for the supergroup.

#### tests

A list of callables of length equal to the length of `base`. These are used to rule out group elements by partial base images, so that `tests[1](g)` returns `False` if

the element  $g$  is known not to satisfy `prop` base on where  $g$  sends the first  $1 + 1$  base points.

### init\_subgroup

if a subgroup of the sought group is known in advance, it can be passed to the function as this parameter.

#### Returns `res`

The subgroup of all elements satisfying `prop`. The generating set for this group is guaranteed to be a strong generating set relative to the base `base`.

#### Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentiones that this can be done by using `.baseswap(...)`, however the current imlementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

#### Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup,  
... AlternatingGroup  
>>> from sympy.combinatorics.perm_groups import PermutationGroup  
>>> from sympy.combinatorics.testutil import _verify_bsgs  
>>> S = SymmetricGroup(7)  
>>> prop_even = lambda x: x.is_even  
>>> base, strong_gens = S.schreier_sims_incremental()  
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_gens)  
>>> G.is_subgroup(AlternatingGroup(7))  
True  
>>> _verify_bsgs(G, base, G.generators)  
True
```

#### transitivity\_degree

Compute the degree of transitivity of the group.

A permutation group  $G$  acting on  $\Omega = \{0, 1, \dots, n-1\}$  is  $k$ -fold transitive, if, for any  $k$  points  $(a_1, a_2, \dots, a_k) \in \Omega$  and any  $k$  points  $(b_1, b_2, \dots, b_k) \in \Omega$  there exists  $g \in G$  such that  $g(a_1)=b_1, g(a_2)=b_2, \dots, g(a_k)=b_k$ . The degree of transitivity of  $G$  is the maximum  $k$  such that  $G$  is  $k$ -fold transitive. ([8])

#### See also:

`is_transitive` (page 202), `orbit` (page 205)

## Examples

```
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

## Polyhedron

```
class sympy.combinatorics.polyhedron.Polyhedron
    Represents the polyhedral symmetry group (PSG).
```

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

## References

<http://mathworld.wolfram.com/PolyhedralGroup.html>

### array\_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

#### See also:

`corners` (page 213), `cyclic_form` (page 214)

## Examples

```
>>> from sympy.combinatorics import Permutation, Cycle
>>> from sympy.combinatorics.polyhedron import tetrahedron
>>> tetrahedron.array_form
[0, 1, 2, 3]

>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
```

### corners

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

#### See also:

`array_form` (page 213), `cyclic_form` (page 214)

### Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

#### cyclic\_form

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

[corners](#) (page 213), [array\\_form](#) (page 213)

#### edges

Given the faces of the polyhedra we can get the edges.

### Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

#### faces

Get the faces of the Polyhedron.

#### pgroup

Get the permutations of the Polyhedron.

#### reset()

Return corners to their original positions.

### Examples

```
>>> from sympy.combinatorics.polyhedron import tetrahedron as T
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```

#### rotate(*perm*)

Apply a permutation to the polyhedron *in place*. The permutation may be given as a Permutation instance or an integer indicating which permutation from pgroup of the Polyhedron should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

## Notes

When a Permutation is applied, no check is done to see if that is a valid permutation for the Polyhedron. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the `pgroup` of the Polyhedron. On the other hand, allowing arbitrary rotations (applications of permutations) gives a way to follow named elements rather than indices since Polyhedron allows vertices to be named while Permutation works only with indices.

## Examples

```
>>> from sympy.combinatorics import Polyhedron, Permutation
>>> from sympy.combinatorics.polyhedron import cube
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```
>>> cube.reset()
>>> cube.rotate(Permutation([[1,2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

Polyhedron can be used to follow elements of set that are identified by letters instead of integers:

```
>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False
```

### size

Get the number of corners of the Polyhedron.

### vertices

Get the corners of the Polyhedron.

The method `vertices` is an alias for `corners`.

### See also:

`array_form` (page 213), `cyclic_form` (page 214)

## Examples

```
>>> from sympy.combinatorics import Polyhedron
>>> from sympy.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
```

```
>>> p.corners == p.vertices == (a, b, c, d)
True
```

## Prufer Sequences

```
class sympy.combinatorics.prufer.Prufer
```

The Prufer correspondence is an algorithm that describes the bijection between labeled trees and the Prufer code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of  $n - 2$ .

Prufer sequences were first used by Heinz Prufer to give a proof of Cayley's formula.

### References

[R15] (page 1236)

**static edges(\*runs)**

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

### Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5]) # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6]) # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

**next(delta=1)**

Generates the Prufer sequence that is delta beyond the current one.

**See also:**

[prufer\\_rank](#) (page 217), [rank](#) (page 218), [prev](#) (page 217), [size](#) (page 218)

### Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1) # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

### nodes

Returns the number of nodes in the tree.

## Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

`prev(delta=1)`  
Generates the Prufer sequence that is -delta before the current one.

### See also:

[prufer\\_rank](#) (page 217), [rank](#) (page 218), [next](#) (page 216), [size](#) (page 218)

## Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

`prufer_rank()`  
Computes the rank of a Prufer sequence.

### See also:

[rank](#) (page 218), [next](#) (page 216), [prev](#) (page 217), [size](#) (page 218)

## Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

`prufer_repr`  
Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

### See also:

[to\\_prufer](#) (page 218)

## Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

### rank

Returns the rank of the Prufer sequence.

See also:

[prufer\\_rank](#) (page 217), [next](#) (page 216), [prev](#) (page 217), [size](#) (page 218)

### Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

### size

Return the number of possible trees of this Prufer object.

See also:

[prufer\\_rank](#) (page 217), [rank](#) (page 218), [next](#) (page 216), [prev](#) (page 217)

### Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

### static to\_prufer(tree, n)

Return the Prufer sequence for a tree given as a list of edges where  $n$  is the number of nodes in the tree.

See also:

[prufer\\_repr](#) (page 217) returns Prufer sequence of a Prufer object.

### Examples

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_repr
[0, 0]
>>> Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

**static to\_tree(*prufer*)**

Return the tree (as a list of edges) of the given Prufer sequence.

See also:

`tree_repr` (page 219) returns tree representation of a Prufer object.

**References**

- <http://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html>

**Examples**

```
>>> from sympy.combinatorics.prufer import Prufer
>>> a = Prufer([0, 2], 4)
>>> a.tree_repr
[[0, 1], [0, 2], [2, 3]]
>>> Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

**tree\_repr**

Returns the tree representation of the Prufer object.

See also:

`to_tree` (page 218)

**Examples**

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

**classmethod unrank(*rank, n*)**

Finds the unranked Prufer sequence.

**Examples**

```
>>> from sympy.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

**Subsets****class sympy.combinatorics.subsets.Subset**

Represents a basic subset object.

We generate subsets using essentially two techniques, binary enumeration and lexicographic enumeration. The `Subset` class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

**classmethod** `bitlist_from_subset(subset, superset)`

Gets the bitlist corresponding to a subset.

See also:

`subset_from_bitlist` (page 224)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.bitlist_from_subset(['c','d'], ['a','b','c','d'])
'0011'
```

`cardinality`

Returns the number of all possible subsets.

See also:

`subset` (page 224), `superset` (page 224), `size` (page 223), `superset_size` (page 225)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.cardinality
16
```

`iterate_binary(k)`

This is a helper function. It iterates over the binary subsets by k steps. This variable can be both positive or negative.

See also:

`next_binary` (page 221), `prev_binary` (page 222)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a','b','c'], ['a','b','c','d'])
>>> a.iterate_binary(2).subset
[]
```

**iterate\_graycode(*k*)**

Helper function used for prev\_gray and next\_gray. It performs k step overs to get the respective Gray codes.

See also:

[next\\_gray](#) (page 221), [prev\\_gray](#) (page 222)

**Examples**

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1,2,3], [1,2,3,4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

**next\_binary()**

Generates the next binary ordered subset.

See also:

[prev\\_binary](#) (page 222), [iterate\\_binary](#) (page 220)

**Examples**

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a','b','c','d'], ['a','b','c','d'])
>>> a.next_binary().subset
[]
```

**next\_gray()**

Generates the next Gray code ordered subset.

See also:

[iterate\\_graycode](#) (page 220), [prev\\_gray](#) (page 222)

**Examples**

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([1,2,3], [1,2,3,4])
>>> a.next_gray().subset
[1, 3]
```

**next\_lexicographic()**

Generates the next lexicographically ordered subset.

See also:

[prev\\_lexicographic](#) (page 222)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a','b','c','d'])
>>> a.next_lexicographic().subset
[]
```

### prev\_binary()

Generates the previous binary ordered subset.

See also:

[next\\_binary](#) (page 221), [iterate\\_binary](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a','b','c','d'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.prev_binary().subset
['c']
```

### prev\_gray()

Generates the previous Gray code ordered subset.

See also:

[iterate\\_graycode](#) (page 220), [next\\_gray](#) (page 221)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([2,3,4], [1,2,3,4,5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

### prev\_lexicographic()

Generates the previous lexicographically ordered subset.

See also:

[next\\_lexicographic](#) (page 221)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a','b','c','d'])
>>> a.prev_lexicographic().subset
['d']
>>> a = Subset(['c','d'], ['a','b','c','d'])
```

```
>>> a.prev_lexicographic().subset
['c']
```

### rank\_binary

Computes the binary ordered rank.

See also:

[iterate\\_binary](#) (page 220), [unrank\\_binary](#) (page 225)

### Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

### rank\_gray

Computes the Gray code ranking of the subset.

See also:

[iterate\\_graycode](#) (page 220), [unrank\\_gray](#) (page 225)

### Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_gray
2
>>> a = Subset([2,4,5], [1,2,3,4,5,6])
>>> a.rank_gray
27
```

### rank\_lexicographic

Computes the lexicographic ranking of the subset.

### Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_lexicographic
14
>>> a = Subset([2,4,5], [1,2,3,4,5,6])
>>> a.rank_lexicographic
43
```

### size

Gets the size of the subset.

See also:

[subset](#) (page 224), [superset](#) (page 224), [superset\\_size](#) (page 225), [cardinality](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.size
2
```

### subset

Gets the subset represented by the current instance.

#### See also:

[superset](#) (page 224), [size](#) (page 223), [superset\\_size](#) (page 225), [cardinality](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.subset
['c', 'd']
```

### classmethod subset\_from\_bitlist(*super\_set*, *bitlist*)

Gets the subset defined by the bitlist.

#### See also:

[bitlist\\_from\\_subset](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.subset_from_bitlist(['a','b','c','d'], '0011').subset
['c', 'd']
```

### classmethod subset\_indices(*subset*, *superset*)

Return indices of subset in superset in a list; the list is empty if all elements of subset are not in superset.

## Examples

```
>>> from sympy.combinatorics import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
>>> Subset.subset_indices([], superset)
[]
```

### superset

Gets the superset of the subset.

#### See also:

[subset](#) (page 224), [size](#) (page 223), [superset\\_size](#) (page 225), [cardinality](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.superset
['a', 'b', 'c', 'd']
```

### superset\_size

Returns the size of the superset.

#### See also:

[subset](#) (page 224), [superset](#) (page 224), [size](#) (page 223), [cardinality](#) (page 220)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> a = Subset(['c','d'], ['a','b','c','d'])
>>> a.superset_size
4
```

### classmethod unrank\_binary(*rank, superset*)

Gets the binary ordered subset of the specified rank.

#### See also:

[iterate\\_binary](#) (page 220), [rank\\_binary](#) (page 223)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_binary(4, ['a','b','c','d']).subset
['b']
```

### classmethod unrank\_gray(*rank, superset*)

Gets the Gray code ordered subset of the specified rank.

#### See also:

[iterate\\_graycode](#) (page 220), [rank\\_gray](#) (page 223)

## Examples

```
>>> from sympy.combinatorics.subsets import Subset
>>> Subset.unrank_gray(4, ['a','b','c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a','b','c']).subset
[]
```

### static subsets.ksubsets(*superset, k*)

Finds the subsets of size *k* in lexicographic order.

This uses the `itertools` generator.

#### See also:

[Subset](#) (page 219)

## Examples

```
>>> from sympy.combinatorics.subsets import ksubsets
>>> list(ksubsets([1,2,3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1,2,3,4,5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

## Gray Code

```
class sympy.combinatorics.graycode.GrayCode
```

A Gray code is essentially a Hamiltonian walk on a n-dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once. The Gray code for a 3d cube is ['000','100','110','010','011','111','101','001'].

A Gray code solves the problem of sequentially generating all possible subsets of n objects in such a way that each subset is obtained from the previous one by either deleting or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

References: [1] Nijenhuis,A. and Wilf,H.S.(1978). Combinatorial Algorithms. Academic Press. [2] Knuth, D. (2011). The Art of Computer Programming, Vol 4 Addison Wesley

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100', '1100', '1101', '1111', '1110', '1010', '1001', '0100', '0110', '0111', '0101', '0011', '0010', '0001', '0000']
```

### current

Returns the currently referenced Gray code as a bit string.

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(3, start='100').current
'100'

generate_gray(**hints)
```

Generates the sequence of bit vectors of a Gray Code.

[1] Knuth, D. (2011). The Art of Computer Programming, Vol 4, Addison Wesley

### See also:

[skip](#) (page 228)

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

n

Returns the dimension of the Gray code.

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

next(*delta*=1)

Returns the Gray code a distance *delta* (default = 1) from the current value in canonical order.

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

rank

Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

References: [1] <http://statweb.stanford.edu/~susan/courses/s208/node12.html>

See also:

[unrank](#) (page 228)

## Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
```

```
>>> GrayCode(3, rank=7).current
'100'

selections
    Returns the number of bit vectors in the Gray code.
```

### Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

```
skip()
    Skips the bit generation.
```

### See also:

[generate\\_gray](#) (page 226)

### Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...     print(i)
...
000
001
011
010
111
101
100
```

```
classmethod unrank(n, rank)
```

Unranks an *n*-bit sized Gray code of rank *k*. This method exists so that a derivative GrayCode class can define its own code of a given rank.

The string here is generated in reverse order to allow for tail-call optimization.

### See also:

[rank](#) (page 227)

### Examples

```
>>> from sympy.combinatorics.graycode import GrayCode
>>> GrayCode(5, rank=3).current
'00010'
>>> GrayCode.unrank(5, 3)
'00010'
```

**static** `graycode.random_bitstring(n)`  
Generates a random bitlist of length n.

#### Examples

```
>>> from sympy.combinatorics.graycode import random_bitstring  
>>> random_bitstring(3)  
100
```

**static** `graycode.gray_to_bin(bin_list)`  
Convert from Gray coding to binary coding.

We assume big endian encoding.

See also:

`bin_to_gray` (page 229)

#### Examples

```
>>> from sympy.combinatorics.graycode import gray_to_bin  
>>> gray_to_bin('100')  
'111'
```

**static** `graycode.bin_to_gray(bin_list)`  
Convert from binary coding to gray coding.

We assume big endian encoding.

See also:

`gray_to_bin` (page 229)

#### Examples

```
>>> from sympy.combinatorics.graycode import bin_to_gray  
>>> bin_to_gray('111')  
'100'
```

**static** `graycode.get_subset_from_bitstring(super_set, bitstring)`  
Gets the subset defined by the bitstring.

See also:

`graycode_subsets` (page 229)

#### Examples

```
>>> from sympy.combinatorics.graycode import get_subset_from_bitstring  
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')  
['c', 'd']  
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')  
['c', 'a']
```

**static** `graycode.graycode_subsets(gray_code_set)`  
Generates the subsets as enumerated by a Gray code.

See also:

`get_subset_from_bitstring` (page 229)

### Examples

```
>>> from sympy.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'],      ['a', 'c'], ['a']]
>>> list(graycode_subsets(['a', 'b', 'c', 'c']))
[[], ['c'], ['c', 'c'], ['c'], ['b', 'c'], ['b', 'c', 'c'],      ['b', 'c'], ['b'],
 ['a', 'b'], ['a', 'b', 'c']]
```

## Named Groups

`sympy.combinatorics.named_groups.SymmetricGroup(n)`  
Generates the symmetric group on n elements as a permutation group.

The generators taken are the n-cycle (0 1 2 ... n-1) and the transposition (0 1) (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

See also:

`CyclicGroup` (page 230), `DihedralGroup` (page 231), `AlternatingGroup` (page 231)

### References

[1] [http://en.wikipedia.org/wiki/Symmetric\\_group#Generators\\_and\\_relations](http://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations)

### Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> G.is_group()
False
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
 [1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
 [2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
 [3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
 [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

`sympy.combinatorics.named_groups.CyclicGroup(n)`

Generates the cyclic group of order n as a permutation group.

The generator taken is the n-cycle (0 1 2 ... n-1) (in cycle notation). After the group is generated, some of its basic properties are set.

See also:

`SymmetricGroup` (page 230), `DihedralGroup` (page 231), `AlternatingGroup` (page 231)

## Examples

```
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> G = CyclicGroup(6)
>>> G.is_group()
False
>>> G.order()
6
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3, 4, 5], [1, 2, 3, 4, 5, 0], [2, 3, 4, 5, 0, 1],
[3, 4, 5, 0, 1, 2], [4, 5, 0, 1, 2, 3], [5, 0, 1, 2, 3, 4]]
```

`sympy.combinatorics.named_groups.DihedralGroup(n)`

Generates the dihedral group  $D_n$  as a permutation group.

The dihedral group  $D_n$  is the group of symmetries of the regular  $n$ -gon. The generators taken are the  $n$ -cycle  $a = (0 \ 1 \ 2 \ \dots \ n-1)$  (a rotation of the  $n$ -gon) and  $b = (0 \ n-1)(1 \ n-2)\dots$  (a reflection of the  $n$ -gon) in cycle notation. It is easy to see that these satisfy  $a^{**n} = b^{**2} = 1$  and  $bab = \sim a$  so they indeed generate  $D_n$  (See [1]). After the group is generated, some of its basic properties are set.

See also:

[SymmetricGroup](#) (page 230), [CyclicGroup](#) (page 230), [AlternatingGroup](#) (page 231)

## References

[1] [http://en.wikipedia.org/wiki/Dihedral\\_group](http://en.wikipedia.org/wiki/Dihedral_group)

## Examples

```
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group()
False
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
[[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
[[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
[[0, 3], [1, 2]]]
```

`sympy.combinatorics.named_groups.AlternatingGroup(n)`

Generates the alternating group on  $n$  elements as a permutation group.

For  $n > 2$ , the generators taken are  $(0 \ 1 \ 2)$ ,  $(0 \ 1 \ 2 \ \dots \ n-1)$  for  $n$  odd and  $(0 \ 1 \ 2)$ ,  $(1 \ 2 \ \dots \ n-1)$  for  $n$  even (See [1], p.31, ex.6.9.). After the group is generated, some of its basic properties are set. The cases  $n = 1, 2$  are handled separately.

See also:

[SymmetricGroup](#) (page 230), [CyclicGroup](#) (page 230), [DihedralGroup](#) (page 231)

## References

[1] Armstrong, M. “Groups and Symmetry”

## Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group()
False
>>> a = list(G.generate_dimino())
>>> len(a)
12
>>> all(perm.is_even for perm in a)
True
```

`sympy.combinatorics.named_groups.AbelianGroup(*cyclic_orders)`

Returns the direct product of cyclic groups with the given orders.

According to the structure theorem for finite abelian groups ([1]), every finite abelian group can be written as the direct product of finitely many cyclic groups. [1] [http://groupprops.subwiki.org/wiki/Structure\\_theorem\\_for\\_finitely\\_generated\\_abelian\\_groups](http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups)

See also:

`sympy.combinatorics.group_constructs.DirectProduct` (page 238)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import AbelianGroup
>>> AbelianGroup(3, 4)
PermutationGroup([
    Permutation(6)(0, 1, 2),
    Permutation(3, 4, 5, 6)])
>>> _.is_group()
False
```

## Utilities

`sympy.combinatorics.util._base_ordering(base, degree)`

Order  $\{0, 1, \dots, n - 1\}$  so that base points come first and in order.

**Parameters “base” - the base**

“degree” - the degree of the associated permutation group

**Returns** A list `base_ordering` such that `base_ordering[point]` is the number of `point` in the ordering.

## Notes

This is used in backtrack searches, when we define a relation  $\ll$  on the underlying set for a permutation group of degree  $n$ ,  $\{0, 1, \dots, n - 1\}$ , so that if  $(b_1, b_2, \dots, b_k)$  is a base we have  $b_i \ll b_j$  whenever  $i < j$  and  $b_i \ll a$  for all  $i \in \{1, 2, \dots, k\}$  and  $a$  is not in the base. The idea is developed and applied to backtracking algorithms in [1], pp.108-132. The points that are not in the base are taken in increasing order.

## References

[R16] (page 1236)

## Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

`sympy.combinatorics.util._check_cycles_alt_sym(perm)`

Checks for cycles of prime length  $p$  with  $n/2 < p < n-2$ .

Here  $n$  is the degree of the permutation. This is a helper function for the function `is_alt_sym` from `sympy.combinatorics.perm_groups`.

**See also:**

`sympy.combinatorics.perm_groups.PermutationGroup.is_alt_sym` (page 198)

## Examples

```
>>> from sympy.combinatorics.util import _check_cycles_alt_sym
>>> from sympy.combinatorics.permutations import Permutation
>>> a = Permutation([[0,1,2,3,4,5,6,7,8,9,10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0,1,2,3,4,5,6], [7,8,9,10]])
>>> _check_cycles_alt_sym(b)
True
```

`sympy.combinatorics.util._distribute_gens_by_base(base, gens)`

Distribute the group elements `gens` by membership in basic stabilizers.

Notice that for a base  $(b_1, b_2, \dots, b_k)$ , the basic stabilizers are defined as  $G^{(i)} = G_{b_1, \dots, b_{i-1}}$  for  $i \in \{1, 2, \dots, k\}$ .

**Parameters** “`base`“ - a sequence of points in ‘{0, 1, ..., n-1}’

“`gens`“ - a list of elements of a permutation group of degree ‘n’.

**Returns** List of length  $k$ , where  $k$  is

the length of `base`. The  $i$ -th entry contains those elements in `gens` which fix the first  $i$  elements of `base` (so that the 0-th entry is equal to `gens` itself). If no element fixes the first  $i$  elements of `base`, the  $i$ -th element is set to a list containing the identity element.

**See also:**

`_strong_gens_from_distr` (page 237), `_orbits_transversals_from_bsgs` (page 235),  
`_handle_precomputed_bsgs` (page 234)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)],
 [Permutation(1, 2)]]
```

sympy.combinatorics.util.\_handle\_precomputed\_bsgs(*base*, *strong\_gens*, *transversals=None*, *basic\_orbits=None*, *strong\_gens\_distr=None*)

Calculate BSGS-related structures from those present.

The base and strong generating set must be provided; if any of the transversals, basic orbits or distributed strong generators are not provided, they will be calculated from the base and strong generating set.

Parameters “*base*“ - the base

“*strong\_gens*“ - the strong generators

“*transversals*“ - basic transversals

“*basic\_orbits*“ - basic orbits

“*strong\_gens\_distr*“ - strong generators distributed by membership in basic stabilizers

Returns (*transversals*, *basic\_orbits*, *strong\_gens\_distr*) where *transversals*

are the basic transversals, *basic\_orbits* are the basic orbits, and

*strong\_gens\_distr* are the strong generators distributed by membership

in basic stabilizers.

See also:

[\\_orbits\\_transversals\\_from\\_bsgs](#) (page 235), [\\_distribute\\_gens\\_by\\_base](#) (page 233)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import DihedralGroup
>>> from sympy.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
... basic_orbits=D.basic_orbits)
({0: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2)},
{1: Permutation(2), 2: Permutation(1, 2)}],
[[0, 1, 2], [1, 2]], [[Permutation(0, 1, 2),
```

```
Permutation(0, 2),
Permutation(1, 2)],
[Permutation(1, 2)])]

sympy.combinatorics.util._orbits_transversals_from_bsgs(base, strong_gens_distr, transver-
sals_only=False)
```

Compute basic orbits and transversals from a base and strong generating set.

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to True, only the transversals are returned.

**Parameters “base“ - the base**

“`strong_gens_distr`“ - strong generators distributed by membership in basic stabilizers  
“`transversals_only`“ - a flag switching between returning only the transversals/ both orbits and transversals

**See also:**

[\\_distribute\\_gens\\_by\\_base](#) (page 233), [\\_handle\\_precomputed\\_bsgs](#) (page 234)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import _orbits_transversals_from_bsgs
>>> from sympy.combinatorics.util import (_orbits_transversals_from_bsgs,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _orbits_transversals_from_bsgs(S.base, strong_gens_distr)
([[0, 1, 2], [1, 2]],
 [{0: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2, 1)},
 {1: Permutation(2), 2: Permutation(1, 2)}])

sympy.combinatorics.util._remove_gens(base, strong_gens, basic_orbits=None,
                                         strong_gens_distr=None)
```

Remove redundant generators from a strong generating set.

**Parameters “base“ - a base**

“`strong_gens`“ - a strong generating set relative to “`base`“  
“`basic_orbits`“ - basic orbits  
“`strong_gens_distr`“ - strong generators distributed by membership in basic stabilizers

**Returns** A strong generating set with respect to `base` which is a subset of `strong_gens`.

## Notes

This procedure is outlined in [1], p.95.

## References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

## Examples

```
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.util import _remove_gens
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(15)
>>> base, strong_gens = S.schreier_sims_incremental()
>>> new_gens = _remove_gens(base, strong_gens)
>>> len(new_gens)
14
>>> _verify_bsgs(S, base, new_gens)
True
```

`sympy.combinatorics.util._strip(g, base, orbits, transversals)`

Attempt to decompose a permutation using a (possibly partial) BSGS structure.

This is done by treating the sequence `base` as an actual base, and the orbits `orbits` and transversals `transversals` as basic orbits and transversals relative to it.

This process is called "sifting". A sift is unsuccessful when a certain orbit element is not found or when after the sift the decomposition doesn't end with the identity element.

The argument `transversals` is a list of dictionaries that provides transversal elements for the orbits `orbits`.

Parameters “g“ - permutation to be decomposed

“base“ - sequence of points

“orbits“ - a list in which the “i“-th entry is an orbit of “base[i]“

under some subgroup of the pointwise stabilizer of ‘

‘base[0], base[1], ..., base[i - 1]“. The groups themselves are implicit

in this function since the only information we need is encoded in the orbits

and transversals

“transversals“ - a list of orbit transversals associated with the orbits

“orbits“.

See also:

`sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims` (page 208),  
`sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims_random` (page 209)

## Notes

The algorithm is described in [1], pp.89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

## References

[1] Holt, D., Eick, B., O'Brien, E. “Handbook of computational group theory”

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
(Permutation(4), 5)
```

`sympy.combinatorics.util._strong_gens_from_distr(strong_gens_distr)`

Retrieve strong generating set from generators of basic stabilizers.

This is just the union of the generators of the first and second basic stabilizers.

**Parameters** “`strong_gens_distr`“ - strong generators distributed by membership  
in basic  
stabilizers

See also:

`_distribute_gens_by_base` (page 233)

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from sympy.combinatorics.named_groups import SymmetricGroup
>>> from sympy.combinatorics.util import (_strong_gens_from_distr,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
[Permutation(0, 1, 2), Permutation(2)(0, 1), Permutation(1, 2)]
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _strong_gens_from_distr(strong_gens_distr)
[Permutation(0, 1, 2), Permutation(2)(0, 1), Permutation(1, 2)]
```

## Group constructors

```
sympy.combinatorics.group_constructs.DirectProduct(*groups)
    Returns the direct product of several groups as a permutation group.
```

This is implemented much like the `__mul__()` (page 186) procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to `DirectProduct(G1, G2, ..., Gn)` is generally expected to be faster than a call to `G1*G2*...*Gn` (and thus the need for this algorithm).

See also:

```
sympy.combinatorics.perm_groups.PermutationGroup.__mul__ (page 186)
```

## Examples

```
>>> from sympy.combinatorics.group_constructs import DirectProduct
>>> from sympy.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

## Test Utilities

```
sympy.combinatorics.testutil._cmp_perm_lists(first, second)
```

Compare two lists of permutations as sets.

This is used for testing purposes. Since the array form of a permutation is currently a list, Permutation is not hashable and cannot be put into a set.

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

```
sympy.combinatorics.testutil._naive_list_centralizer(self, other, af=False)
```

```
sympy.combinatorics.testutil._verify_bsgs(group, base, gens)
```

Verify the correctness of a base and strong generating set.

This is a naive implementation using the definition of a base and a strong generating set relative to it. There are other procedures for verifying a base and strong generating set, but this one will serve for more robust testing.

See also:

```
sympy.combinatorics.perm_groups.PermutationGroup.schreier_sims (page 208)
```

## Examples

```
>>> from sympy.combinatorics.named_groups import AlternatingGroup
>>> from sympy.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

`sympy.combinatorics.testutil._verify_centralizer(group, arg, centr=None)`

Verify the centralizer of a group/set/element inside another group.

This is used for testing `.centralizer()` from `sympy.combinatorics.perm_groups`

See also:

`_naive_list_centralizer` (page 238), `sympy.combinatorics.perm_groups.PermutationGroup.centralizer` (page 191), `_cmp_perm_lists` (page 238)

## Examples

```
>>> from sympy.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

`sympy.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)`

## Tensor Canonicalization

`sympy.combinatorics.tensor_can.canonicalize(g, dummies, msym, *v)`  
canonicalize tensor formed by tensors

**Parameters** `g` : permutation representing the tensor

**dummies** : list representing the dummy indices

it can be a list of dummy indices of the same type or a list of lists of dummy indices, one list for each type of index; the dummy indices must come after the free indices, and put in order contravariant, covariant [d0, -d0, d1, -d1,...]

**msym** : symmetry of the metric(s)

it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type

`v` : list, (`base_i`, `gens_i`, `n_i`, `sym_i`) for tensors of type *i*

`base_i`, `gens_i` : BSGS for tensors of this type.

The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made do get the minimal BSGS; in case of failure, canonicalize\_naive is used, which is much slower.

**n\_i** : number of tensors of type  $i$ .

**sym\_i** : symmetry under exchange of component tensors of type  $i$ .

Both for msym and sym\_i the cases are

- None no symmetry
- 0 commuting
- 1 anticommuting

**Returns** 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

### Examples

one type of index with commuting metric;

$A_{ab}$  and  $B_{ab}$  antisymmetric and commuting

$T = A_{d0d1} * B^{d0}_{\phantom{d0}d2} * B^{d2d1}$

$ord = [d0, -d0, d1, -d1, d2, -d2]$  order of the indices

$g = [1, 3, 0, 5, 4, 2, 6, 7]$

$T_c = 0$

```
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs, canonicalize, bsgs_direct_product
>>> from sympy.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with  $B_{ab}$  anticommuting

$T_c = -A^{d0d1} * B_{d0}{}^{d2} * B_{d1d2}$

$can = [0, 2, 1, 4, 3, 5, 7, 6]$

```
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 1, 4, 3, 5, 7, 6]
```

two types of indices  $[a, b, c, d, e, f]$  and  $[m, n]$ , in this order, both with commuting metric

$f^{abc}$  antisymmetric, commuting

$A_{ma}$  no symmetry, commuting

$T = f^c{}_{da} * f^f{}_{eb} * A_m{}^d * A^{mb} * A_n{}^a * A^{ne}$

$ord = [c, f, a, -a, b, -b, d, -d, e, -e, m, -m, n, -n]$

$g = [0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15]$

The canonical tensor is  $T_c = -f^{cab} * f^{fde} * A^m{}_a * A_{md} * A^n{}_b * A_{ne}$

```

can = [0,2,4, 1,6,8, 10,3, 11,7, 12,5, 13,9, 15,14]

>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base1, gens1, base1, gens1)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0,7,3,1,9,5,11,6,10,4,13,2,12,8,14,15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]

```

`sympy.combinatorics.tensor_can.double_coset_can.rep(dummies, sym, b_S, sgens, S_transversals,  
g)`

Butler-Portugal algorithm for tensor canonicalization with dummy indices

**dummies** list of lists of dummy indices, one list for each type of index; the dummy indices are put in order contravariant, covariant [d0, -d0, d1, -d1, ...].

**sym** list of the symmetries of the index metric for each type.

**possible symmetries of the metrics**

- 0 symmetric
- 1 antisymmetric
- None no symmetry

**b\_S** base of a minimal slot symmetry BSGS.

**sgens** generators of the slot symmetry BSGS.

**S\_transversals** transversals for the slot BSGS.

**g** permutation representing the tensor.

Return 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

A tensor with dummy indices can be represented in a number of equivalent ways which typically grows exponentially with the number of indices. To be able to establish if two tensors with many indices are equal becomes computationally very slow in absence of an efficient algorithm.

The Butler-Portugal algorithm [3] is an efficient algorithm to put tensors in canonical form, solving the above problem.

Portugal observed that a tensor can be represented by a permutation, and that the class of tensors equivalent to it under slot and dummy symmetries is equivalent to the double coset  $D * g * S$  (Note: in this documentation we use the conventions for multiplication of permutations p, q with  $(p * q)(i) = p[q[i]]$  which is opposite to the one used in the Permutation class)

Using the algorithm by Butler to find a representative of the double coset one can find a canonical form for the tensor.

To see this correspondence, let  $g$  be a permutation in array form; a tensor with indices  $ind$  (the indices including both the contravariant and the covariant ones) can be written as

$t = T(ind[g[0], \dots, ind[g[n-1]])$ ,

where  $n = \text{len}(ind)$ ;  $g$  has size  $n + 2$ , the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation  $s$  is a permutation acting on the slots  $t \rightarrow T(ind[(g*s)[0]], \dots, ind[(g*s)[n-1]])$

A dummy symmetry transformation acts on  $ind\ t -> T(ind[(d * g)[0]], \dots, ind[(d * g)[n - 1]])$

Being interested only in the transformations of the tensor under these symmetries, one can represent the tensor by  $g$ , which transforms as

$g -> d * g * s$ , so it belongs to the coset  $D * g * S$ .

Let us explain the conventions by an example.

**Given a tensor  $T^{d3d2d1}_{d1d2d3}$  with the slot symmetries  $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$**

$$T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering  $d1, d2, d3$  then and contravariant index before covariant index; that is the canonical form of the tensor.

The canonical form is  $-T^{d1d2d3}_{d1d2d3}$  obtained using  $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$ .

To convert this problem in the input for this function, use the following labelling of the index names (- for covariant for short)  $d1, -d1, d2, -d2, d3, -d3$

$T^{d3d2d1}_{d1d2d3}$  corresponds to  $g = [4, 2, 0, 1, 3, 5, 6, 7]$  where the last two indices are for the sign

$$sgens = [Permutation(0, 2)(6, 7), Permutation(0, 4)(6, 7)]$$

$$sgens[0] \text{ is the slot symmetry } -(0, 2) \ T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$$

$$sgens[1] \text{ is the slot symmetry } -(0, 4) \ T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$$

The dummy symmetry group  $D$  is generated by the strong base generators  $[(0, 1), (2, 3), (4, 5), (0, 1)(2, 3), (2, 3)(4, 5)]$

The dummy symmetry acts from the left  $d = [1, 0, 2, 3, 4, 5, 6, 7]$  exchange  $d1 -> -d1$   $T^{d3d2d1}_{d1d2d3} == T^{d3d2}_{d1}{}^{d1}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] -> [4, 2, 1, 0, 3, 5, 6, 7] =_a f_{rmul}(d, g)$  which differs from  $_a f_{rmul}(g, d)$ .

The slot symmetry acts from the right  $s = [2, 1, 0, 3, 4, 5, 7, 6]$  exchanges slots 0 and 2 and changes sign  $T^{d3d2d1}_{d1d2d3} == -T^{d1d2d3}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7] -> [0, 2, 4, 1, 3, 5, 7, 6] =_a f_{rmul}(g, s)$

Example in which the tensor is zero, same slot symmetries as above:  $T^{d3}_{d1, d2}{}^{d1}_{d3}{}^{d2}$

$= -T^{d3}_{d1, d3}{}^{d1}_{d2}{}^{d2}$  under slot symmetry  $-(2, 4)$ ;

$= T_{d3d1}{}^{d3d1}_{d2}{}^{d2}$  under slot symmetry  $-(0, 2)$ ;

$= T^{d3}_{d1d3}{}^{d1}_{d2}{}^{d2}$  symmetric metric;

$= 0$  since two of these lines have tensors differ only for the sign.

The double coset  $D * g * S$  consists of permutations  $h = d * g * s$  corresponding to equivalent tensors; if there are two  $h$  which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to  $[d1, -d1, d2, -d2, d3, -d3]$  that is  $rep = min(D * g * S) = min([d * g * s for in D for in S])$

The indices are fixed one by one; first choose the lowest index for slot 0, then the lowest remaining index for slot 1, etc. Doing this one obtains a chain of stabilizers

$S -> S_{b0} -> S_{b0, b1} -> \dots$  and  $D -> D_{p0} -> D_{p0, p1} -> \dots$

where  $[b0, b1, \dots] = range(b)$  is a base of the symmetric group; the strong base  $b_S$  of  $S$  is an ordered sublist of it; therefore it is sufficient to compute once the strong base generators of  $S$  using the Schreier-Sims algorithm; the stabilizers of the strong base generators are the strong base generators of the stabilizer subgroup.

$d_{base} = [p_0, p_1, \dots]$  is not in general in lexicographic order, so that one must recompute the strong base generators each time; however this is trivial, there is no need to use the Schreier-Sims algorithm for D.

The algorithm keeps a TAB of elements  $(s_i, d_i, h_i)$  where  $h_i = d_i * g * s_i$  satisfying  $h_i[j] = p_j$  for  $0 \leq j < i$  starting from  $s_0 = id, d_0 = id, h_0 = g$ .

The equations  $h_0[0] = p_0, h_1[1] = p_1, \dots$  are solved in this order, choosing each time the lowest possible value of  $p_i$

For  $j < i$   $d_i * g * s_i * S_{b_0, \dots, b_{i-1}} * b_j = D_{p_0, \dots, p_{i-1}} * p_j$  so that for dx in  $D_{p_0, \dots, p_{i-1}}$  and sx in  $S_{base[0], \dots, base[i-1]}$  one has  $dx * d_i * g * s_i * sx * b_j = p_j$

Search for dx, sx such that this equation holds for  $j = i$ ; it can be written as  $s_i * sx * b_j = J, dx * d_i * g * J = p_j$   $sx * b_j = s_i * * - 1 * J; sx = trace(s_i * * - 1, S_{b_0, \dots, b_{i-1}})$   $dx * * - 1 * p_j = d_i * g * J; dx = trace(d_i * g * J, D_{p_0, \dots, p_{i-1}})$

$s_{i+1} = s_i * trace(s_i * * - 1 * J, S_{b_0, \dots, b_{i-1}})$   $d_{i+1} = trace(d_i * g * J, D_{p_0, \dots, p_{i-1}}) * * - 1 * d_i$   $h_{i+1} * b_i = d_{i+1} * g * s_{i+1} * b_i = p_i$

$h_n * b_j = p_j$  for all j, so that  $h_n$  is the solution.

Add the found  $(s, d, h)$  to TAB1.

At the end of the iteration sort TAB1 with respect to the  $h$ ; if there are two consecutive  $h$  in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive  $h$  which are equal, keep only one.

Then stabilize the slot generators under  $i$  and the dummy generators under  $p_i$ .

Assign  $TAB = TAB1$  at the end of the iteration step.

At the end  $TAB$  contains a unique  $(s, d, h)$ , since all the slots of the tensor  $h$  have been fixed to have the minimum value according to the symmetries. The algorithm returns  $h$ .

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an  $i$  which does not belong to the slot base for which  $p_i$  is fixed by the dummy symmetry only, while  $i$  is not invariant from the slot stabilizer, so  $p_i$  is not in general the minimal value.

**This algorithm differs slightly from the original algorithm [3]:** the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors  $h$  are eliminated from TAB.

## Examples

```
>>> from sympy.combinatorics.permutations import Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> from sympy.combinatorics.tensor_can import double_coset_can_rep, get_transversals
>>> gens = [Permutation(x) for x in [[2,1,0,3,4,5,7,6], [4,1,2,3,0,5,7,6]]]
>>> base = [0, 2]
>>> g = Permutation([4,2,0,1,3,5,6,7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
[0, 1, 2, 3, 4, 5, 7, 6]

>>> g = Permutation([4,1,3,0,5,2,6,7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
0
```

```
sympy.combinatorics.tensor_can.get_symmetric_group_sgs(n, antisym=False)
Return base, gens of the minimal BSGS for (anti)symmetric tensor
```

n rank of the tensor

```
antisym = False symmetric tensor antisym = True antisymmetric tensor
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.tensor_can import get_symmetric_group_sgs
>>> Permutation.print_cyclic = True
>>> get_symmetric_group_sgs(3)
([0, 1], [Permutation(4)(0, 1), Permutation(4)(1, 2)])
```

`sympy.combinatorics.tensor_can.bsgs_direct_product(base1, gens1, base2, gens2, signed=True)`

direct product of two BSGS

base1 base of the first BSGS.

gens1 strong generating sequence of the first BSGS.

base2, gens2 similarly for the second BSGS.

signed flag for signed permutations.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy.combinatorics.tensor_can import (get_symmetric_group_sgs, bsgs_direct_product)
>>> Permutation.print_cyclic = True
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [Permutation(4)(1, 2)])
```

## 3.3 Number Theory

### 3.3.1 Ntheory Class Reference

`class sympy.ntheory.generate.Sieve`

An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes. When a lookup is requested involving an odd number that has not been sieved, the sieve is automatically extended up to that number.

```
>>> from sympy import sieve
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])

>>> 25 in sieve
False
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])

extend(n)
Grow the sieve to cover all primes <= n (a real number).
```

## Examples

```
>>> from sympy import sieve
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])

>>> sieve.extend(30)
>>> sieve[10] == 29
True

extend_to_no(i)
Extend to include the ith prime number.
i must be an integer.

The list is extended by 50% if it is too short, so it is likely that it will be longer than requested.
```

## Examples

```
>>> from sympy import sieve
>>> from array import array # this line and next for doctest only
>>> sieve._list = array('l', [2, 3, 5, 7, 11, 13])

>>> sieve.extend_to_no(9)
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])

primerange(a, b)
Generate all prime numbers in the range [a, b].
```

## Examples

```
>>> from sympy import sieve
>>> print([i for i in sieve.primerange(7, 18)])
[7, 11, 13, 17]

search(n)
Return the indices i, j of the primes that bound n.
If n is prime then i == j.

Although n can be an expression, if ceiling cannot convert it to an integer then an n error will be
raised.
```

## Examples

```
>>> from sympy import sieve
>>> sieve.search(25)
(9, 10)
>>> sieve.search(23)
(9, 9)
```

### 3.3.2 Ntheory Functions Reference

`sympy.ntheory.generate.prime(nth)`

Return the nth prime, with the primes indexed as  $\text{prime}(1) = 2$ ,  $\text{prime}(2) = 3$ , etc.... The nth prime is approximately  $n \log(n)$  and can never be larger than  $2^{**n}$ .

See also:

`sympy.ntheory.primetest.isprime` ([page 263](#)) Test if n is prime

`primerange` ([page 247](#)) Generate all primes in a given range

`primepi` ([page 246](#)) Return the number of primes less than or equal to n

#### References

- <http://primes.utm.edu/glossary/xpage/BertrandsPostulate.html>

#### Examples

```
>>> from sympy import prime
>>> prime(10)
29
>>> prime(1)
2
```

`sympy.ntheory.generate.primepi(n)`

Return the value of the prime counting function  $\pi(n)$  = the number of prime numbers less than or equal to n.

See also:

`sympy.ntheory.primetest.isprime` ([page 263](#)) Test if n is prime

`primerange` ([page 247](#)) Generate all primes in a given range

`prime` ([page 246](#)) Return the nth prime

#### Examples

```
>>> from sympy import primepi
>>> primepi(25)
9
```

`sympy.ntheory.generate.nextprime(n, ith=1)`

Return the ith prime greater than n.

i must be an integer.

See also:

`prevprime` ([page 247](#)) Return the largest prime smaller than n

`primerange` ([page 247](#)) Generate all primes in a given range

## Notes

Potential primes are located at  $6*j +/- 1$ . This property is used during searching.

```
>>> from sympy import nextprime
>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]
>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

`sympy.ntheory.generate.prevprime(n)`

Return the largest prime smaller than n.

See also:

[nextprime \(page 246\)](#) Return the ith prime greater than n

[primerange \(page 247\)](#) Generates all primes in a given range

## Notes

Potential primes are located at  $6*j +/- 1$ . This property is used during searching.

```
>>> from sympy import prevprime
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

`sympy.ntheory.generate.primerange(a, b)`

Generate a list of all prime numbers in the range [a, b).

If the range exists in the default sieve, the values will be returned from there; otherwise values will be returned but will not modify the sieve.

See also:

[nextprime \(page 246\)](#) Return the ith prime greater than n

[prevprime \(page 247\)](#) Return the largest prime smaller than n

[randprime \(page 248\)](#) Returns a random prime in a given range

[primorial \(page 248\)](#) Returns the product of primes based on condition

[Sieve.primerange \(page 245\)](#) return range from already computed primes or extend the sieve to contain the requested range.

## Notes

Some famous conjectures about the occurrence of primes in a given range are [1]:

• **Twin primes:** though often not, the following will give 2 primes

an infinite number of times: `primerange(6*n - 1, 6*n + 2)`

• **Legendre's:** the following always yields at least one prime `primerange(n**2, (n+1)**2+1)`

• **Bertrand's (proven):** there is always a prime in the range `primerange(n, 2*n)`

- Brocard's:** there are at least four primes in the range `primerange(prime(n)**2, prime(n+1)**2)`

The average gap between primes is  $\log(n)$  [2]; the gap between primes can be arbitrarily large since sequences of composite numbers are arbitrarily large, e.g. the numbers in the sequence  $n! + 2, n! + 3 \dots n! + n$  are all composite.

## References

- 1.[http://en.wikipedia.org/wiki/Prime\\_number](http://en.wikipedia.org/wiki/Prime_number)
- 2.<http://primes.utm.edu/notes/gaps.html>

## Examples

```
>>> from sympy import primerange, sieve
>>> print([i for i in primerange(1, 30)])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, `primerange`, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named `sieve`, can be used:

```
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

`sympy.nttheory.generate.randprime(a, b)`

Return a random prime number in the range  $[a, b]$ .

Bertrand's postulate assures that `randprime(a, 2*a)` will always succeed for  $a > 1$ .

**See also:**

`primerange` ([page 247](#)) Generate all primes in a given range

## References

- [http://en.wikipedia.org/wiki/Bertrand%27s\\_postulate](http://en.wikipedia.org/wiki/Bertrand%27s_postulate)

## Examples

```
>>> from sympy import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

`sympy.nttheory.generate.primorial(n, nth=True)`

Returns the product of the first  $n$  primes (default) or the primes less than or equal to  $n$  (when  $\text{nth=False}$ ).

```
>>> from sympy.nttheory.generate import primorial, randprime, primerange
>>> from sympy import factorint, Mul, primefactors, sqrt
>>> primorial(4) # the first 4 primes are 2, 3, 5, 7
210
```

---

```
>>> primorial(4, nth=False) # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply them together (e.g. the primorial) and then add or subtract 1, the result cannot be divided by any of the original factors, hence either 1 or more new primes must divide this product of primes.

In this case, the number itself is a new prime:

```
>>> factorint(primorial(4) + 1)
{211: 1}
```

In this case two new primes are the factors:

```
>>> factorint(primorial(4) - 1)
{11: 1, 19: 1}
```

Here, some primes smaller and larger than the primes multiplied together are obtained:

```
>>> p = list(primerange(10, 20))
>>> sorted(set(primefactors(Mul(*p) + 1)).difference(set(p)))
[2, 5, 31, 149]
```

**See also:**

[primerange \(page 247\)](#) Generate all primes in a given range

`sympy.ntheory.generate.cycle_length(f, x0, nmax=None, values=False)`

For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is True then the terms of the sequence will be returned instead. The sequence is started with value `x0`.

Note: more than the first lambda + mu terms may be returned and this is the cost of cycle detection with Brent's method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd's method.

```
>>> from sympy.ntheory.generate import cycle_length
```

This will yield successive values of  $i \leftarrow \text{func}(i)$ :

```
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...         i = ii
... 
```

A function is defined:

```
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the mu and lambda terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, `nmax` can be used to exit early (and `mu` will be returned as `None`):

```
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

**Code modified from:** [http://en.wikipedia.org/wiki/Cycle\\_detection](http://en.wikipedia.org/wiki/Cycle_detection).

`sympy.nttheory.factor_.smoothness(n)`

Return the B-smooth and B-power smooth values of `n`.

The smoothness of `n` is the largest prime factor of `n`; the power- smoothness is the largest divisor raised to its multiplicity.

```
>>> from sympy.nttheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

**See also:**

`factorint` (page 255), `smoothness_p` (page 250)

`sympy.nttheory.factor_.smoothness_p(n, m=-1, power=0, visual=None)`

Return a list of `[m, (p, (M, sm(p + m), psm(p + m)))...]` where:

1. $p^{**M}$  is the base-p divisor of `n`
2. $sm(p + m)$  is the smoothness of  $p + m$  ( $m = -1$  by default)
3. $psm(p + m)$  is the power smoothness of  $p + m$

The list is sorted according to smoothness (default) or by power smoothness if `power=1`.

The smoothness of the numbers to the left ( $m = -1$ ) or right ( $m = 1$ ) of a factor govern the results that are obtained from the  $p +/- 1$  type factoring methods.

```
>>> from sympy.nttheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 4)), (19, (1, 5, 5)), (61, (1, 31, 31))])
>>> smoothness_p(10431)
(-1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If `visual=True` then an annotated string will be returned:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```
>>> factorint(17*9)
{3: 2, 17: 1}
>>> smoothness_p(_)
'p**i=3**2 has p-1 B=2, B-pow=2\np**i=17**1 has p-1 B=2, B-pow=16'
>>> smoothness_p(_)
{3: 2, 17: 1}
```

The table of the output logic is:

Input	Visual		
	True	False	other
dict	str	tuple	str
str	str	tuple	dict
tuple	str	tuple	str
n	str	tuple	tuple
mul	str	tuple	tuple

See also:

[factorint](#) (page 255), [smoothness](#) (page 250)

`sympy.nttheory.factor_.trailing(n)`

Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.

### Examples

```
>>> from sympy import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

`sympy.nttheory.factor_.multiplicity(p, n)`

Find the greatest integer m such that  $p^{*m}$  divides n.

### Examples

```
>>> from sympy.nttheory import multiplicity
>>> from sympy.core.numbers import Rational as R
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, R(1, 9))
-2
```

`sympy.nttheory.factor_.perfect_power(n, candidates=None, big=True, factor=True)`

Return (b, e) such that  $n == b^{*e}$  if n is a perfect power; otherwise return False.

By default, the base is recursively decomposed and the exponents collected so the largest possible  $e$  is sought. If `big=False` then the smallest possible  $e$  (thus prime) will be chosen.

If `candidates` for exponents are given, they are assumed to be sorted and the first one that is larger than the computed maximum will signal failure for the routine.

If `factor=True` then simultaneous factorization of  $n$  is attempted since finding a factor indicates the only possible root for  $n$ . This is True by default since only a few small factors will be tested in the course of searching for the perfect power.

## Examples

```
>>> from sympy import perfect_power
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big = False)
(4, 2)
```

```
sympy.nttheory.factor_.pollard_rho(n, s=2, a=1, retries=5, seed=1234, max_steps=None,
F=None)
```

Use Pollard's rho method to try to extract a nontrivial factor of  $n$ . The returned factor may be a composite number. If no factor is found, `None` is returned.

The algorithm generates pseudo-random values of  $x$  with a generator function, replacing  $x$  with  $F(x)$ . If  $F$  is not supplied then the function  $x^{**2} + a$  is used. The first value supplied to  $F(x)$  is `s`. Upon failure (if `retries` is  $> 0$ ) a new `a` and `s` will be supplied; the `a` will be ignored if  $F$  was supplied.

The sequence of numbers generated by such functions generally have a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 – this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from sympy.nttheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x:(2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('loop length = %4i; leader length = %3i' % next(cycle_length(F, s)))
...
loop length = 2489; leader length =  42
loop length =    78; leader length = 120
loop length = 1482; leader length =   99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...
16
13
11
14
4
```

```

11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
[16, 13, 11, 14, 4]

```

Instead of checking the differences of all generated values for a gcd with n, only the kth and  $2^k$ th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

## References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 229-231

## Examples

```

>>> from sympy import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*pow(x,2,n) + 32767) % n
>>> pollard_rho(n, F=F)
257

```

Use the default setting with a bad value of `a` and no retries:

```
>>> pollard_rho(n, a=n-2, retries=0)
```

If retries is  $> 0$  then perhaps the problem will correct itself when new values are generated for `a`:

```
>>> pollard_rho(n, a=n-2, retries=1)
257
```

`sympy.ntheory.factor_.pollard_pm1(n, B=10, a=2, retries=0, seed=1234)`

Use Pollard’s p-1 method to try to extract a nontrivial factor of `n`. Either a divisor (perhaps composite) or `None` is returned.

The value of `a` is the base that is used in the test  $\gcd(a^{**}M - 1, n)$ . The default is 2. If `retries`  $> 0$  then if no factor is found after the first attempt, a new `a` will be generated randomly (using the `seed`) and the process repeated.

Note: the value of `M` is  $\text{lcm}(1..B) = \text{reduce(ilcm, range(2, B + 1))}$ .

A search is made for factors next to even numbers having a power smoothness less than `B`. Choosing a larger `B` increases the likelihood of finding a larger factor but takes longer. Whether a factor of `n` is found or not depends on `a` and the power smoothness of the even number just less than the factor `p` (hence the name p - 1).

Although some discussion of what constitutes a good `a` some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if  $\gcd(a^{**}M - 1, n) = N$  then  $a^{**}M \% q^{**}r$  is 1 for every prime power divisor of `N`. But consider the following:

```
>>> from sympy.nttheory.factor import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
(-1, [(257, (1, 2, 256)), (1009, (1, 7, 16))])
```

So we should (and can) find a root with  $B=16$ :

```
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase  $B$  to 256 we find that it doesn't work:

```
>>> pollard_pm1(n, B=256)
>>>
```

But if the value of  $a$  is changed we find that only multiples of 257 work, e.g.:

```
>>> pollard_pm1(n, B=256, a=257)
1009
```

Checking different  $a$  values shows that all the ones that didn't work had a gcd value not equal to  $n$  but equal to one of the factors:

```
>>> from sympy.core.numbers import ilcm, igcd
>>> from sympy import factorint, Pow
>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> set([igcd(Pow(a, M, n) - 1, n) for a in range(2, 256) if
...       igcd(Pow(a, M, n) - 1, n) != n])
set([1009])
```

But does  $aM \equiv 1 \pmod{n}$  for every divisor of  $n$  give 1?

```
>>> aM = Pow(255, M, n)
>>> [(d, aM%Pow(*d.args)) for d in factorint(n, visual=True).args]
[(257**1, 1), (1009**1, 1)]
```

No, only one of them. So perhaps the principle is that a root will be found for a given value of  $B$  provided that:

- 1.the power smoothness of the  $p - 1$  value next to the root does not exceed  $B$
2. $a^{2^B} \equiv 1 \pmod{n}$  for any of the divisors of  $n$ .

By trying more than one  $a$  it is possible that one of them will yield a factor.

## References

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 236-238
- <http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node81.html>
- <http://www.cs.toronto.edu/~yuvalf/Factorization.pdf>

## Examples

With the default smoothness bound, this number can't be cracked:

```
>>> from sympy.ntheory import pollard_pm1, primefactors
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```
>>> pollard_pm1(21477639576571, B=2000)
4410317
```

Looking at the smoothness of the factors of this number we find:

```
>>> from sympy.utilities import flatten
>>> from sympy.ntheory.factor_ import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B-pow are the same for the  $p - 1$  factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1)
{2: 2, 617: 1, 1787: 1}
>>> factorint(4869863-1)
{2: 1, 2434931: 1}
```

Note that until B reaches the B-pow value of 1787, the number is not cracked;

```
>>> pollard_pm1(21477639576571, B=1786)
>>> pollard_pm1(21477639576571, B=1787)
4410317
```

The B value has to do with the factors of the number next to the divisor, not the divisors themselves. A worst case scenario is that the number next to the factor  $p$  has a large prime divisor or is a perfect power. If these conditions apply then the power-smoothness will be about  $p/2$  or  $p$ . The more realistic is that there will be a large prime factor next to  $p$  requiring a B value on the order of  $p/2$ . Although primes may have been searched for up to this level, the  $p/2$  is a factor of  $p - 1$ , something that we don't know. The modular.math reference below states that 15% of numbers in the range of  $10^{*15}$  to  $15^{*15} + 10^{*4}$  are  $10^{*6}$  power smooth so a B of  $10^{*6}$  will fail 85% of the time in that range. From  $10^{*8}$  to  $10^{*8} + 10^{*3}$  the percentages are nearly reversed...but in that range the simple trial division is quite fast.

```
sympy.ntheory.factor_.factorint(n, limit=None, use_trial=True, use_rho=True, use_pm1=True,
                                 verbose=False, visual=None)
```

Given a positive integer  $n$ , `factorint(n)` returns a dict containing the prime factors of  $n$  as keys and their respective multiplicities as values. For example:

```
>>> from sympy.ntheory import factorint
>>> factorint(2000)    # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537)   # This number is prime
{65537: 1}
```

For input less than 2, `factorint` behaves as follows:

- `factorint(1)` returns the empty factorization, {}
- `factorint(0)` returns {0:1}

- `factorint(-n)` adds  $-1:1$  to the factors and then factors `n`

Partial Factorization:

If `limit (> 3)` is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of rho/p-1 steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```
>>> from sympy.nttheory import isprime
>>> from sympy.core.compatibility import long
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, long(202916782076162456022877024859): 1, 7: 1}
True
>>> isprime(max(f))
False
```

This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

Visual Factorization:

If `visual` is set to `True`, then it will return a visual factorization of the integer. For example:

```
>>> from sympy import pprint
>>> pprint(factorint(4200, visual=True))
 3 1 2 1
2 *3 *5 *7
```

Note that this is achieved by using the `evaluate=False` flag in `Mul` and `Pow`. If you do other manipulations with an expression where `evaluate=False`, it may evaluate. Therefore, you should use the `visual` option only for visualization, and use the normal dictionary returned by `visual=False` if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to `factorint`:

```
>>> from sympy import Mul, Pow
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular))
 2 2 2
2 *3 *7

>>> visual = factorint(1764, visual=True); pprint(visual)
 2 2 2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
```

---

```
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

See also:

[smoothness](#) (page 250), [smoothness\\_p](#) (page 250), [divisors](#) (page 258)

## Notes

Algorithm:

The function switches between multiple algorithms. Trial division quickly finds small factors (of the order 1-5 digits), and finds all large factors if given enough time. The Pollard rho and p-1 algorithms are used to find large factors ahead of time; they will often find factors of the order of 10 digits within a few seconds:

```
>>> factors = factorint(12345678910111213141516)
>>> for base, exp in sorted(factors.items()):
...     print('%s %s' % (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- `use_trial`: Toggle use of trial division
- `use_rho`: Toggle use of Pollard's rho method
- `use_pm1`: Toggle use of Pollard's p-1 method

`factorint` also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

If `verbose` is set to `True`, detailed progress is printed.

`sympy.nttheory.factor_.primefactors(n, limit=None, verbose=False)`

Return a sorted list of n's prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization. Unlike `factorint()`, `primefactors()` does not return -1 or 0.

See also:

[divisors](#) (page 258)

## Examples

```
>>> from sympy.nttheory import primefactors, factorint, isprime
>>> primefactors(6)
[2, 3]
```

```
>>> primefactors(-5)
[5]

>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
>>> primefactors(123456)
[2, 3, 643]

>>> sorted(factorint(10000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(10000000001, limit=300)
[101]
```

#### sympy.nttheory.factor\_.divisors(*n*, generator=False)

Return all divisors of *n* sorted from 1..*n* by default. If generator is True an unordered generator is returned.

The number of divisors of *n* can be quite large if there are many prime factors (counting repeated factors). If only the number of factors is desired use divisor\_count(*n*).

See also:

[primefactors](#) (page 257), [factorint](#) (page 255), [divisor\\_count](#) (page 258)

#### Examples

```
>>> from sympy import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8

>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

This is a slightly modified version of Tim Peters referenced at:  
<http://stackoverflow.com/questions/1010381/python-factorization>

#### sympy.nttheory.factor\_.divisor\_count(*n*, modulus=1)

Return the number of divisors of *n*. If **modulus** is not 1 then only those that are divisible by **modulus** are counted.

See also:

[factorint](#) (page 255), [divisors](#) (page 258), [totient](#) (page 258)

#### References

- <http://www.mayer.dial.pipex.com/mathsf/formulae.htm>

```
>>> from sympy import divisor_count
>>> divisor_count(6)
4
```

```
sympy.ntheory.factor_.totient()
Calculate the Euler totient function phi(n)
```

```
>>> from sympy.ntheory import totient
>>> totient(1)
1
>>> totient(25)
20
```

See also:

[divisor\\_count](#) (page 258)

```
sympy.ntheory.factor_.core(n, t=2)
```

Calculate  $\text{core}(n,t) = \text{core}_t(n)$  of a positive integer n

`core_2(n)` is equal to the squarefree part of n

If n's prime factorization is:

$$n = \prod_{i=1}^{\omega} p_i^{m_i},$$

then

$$\text{core}_t(n) = \prod_{i=1}^{\omega} p_i^{m_i \bmod t}.$$

**Parameters** `t` : `core(n,t)` calculates the t-th power free part of n

`core(n, 2)` is the squarefree part of n `core(n, 3)` is the cubefree part of n

Default for t is 2.

See also:

[factorint](#) (page 255)

## References

[R336] (page 1236)

## Examples

```
>>> from sympy.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
379238
>>> core(15**11, 10)
15
```

```
sympy.ntheory.modular.symmetric_residue(a, m)
```

Return the residual mod m such that it is within half of the modulus.

```
>>> from sympy.nttheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

`sympy.nttheory.modular.crt(m, v, symmetric=False, check=True)`

Chinese Remainder Theorem.

The moduli in `m` are assumed to be pairwise coprime. The output is then an integer `f`, such that  $f = v_i \text{ mod } m_i$  for each pair out of `v` and `m`. If `symmetric` is `False` a positive integer will be returned, else  $|f|$  will be less than or equal to the LCM of the moduli, and thus `f` may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be `None` if there is no solution.

The keyword `check` can be set to `False` if it is known that the moduli are coprime.

As an example consider a set of residues `U = [49, 76, 65]` and a set of moduli `M = [99, 97, 95]`. Then we have:

```
>>> from sympy.nttheory.modular import crt, solve_congruence
>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use `check=False`:

```
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of `gf_crt`'s arguments is reversed relative to `crt`, and that `solve_congruence` takes residue, modulus pairs.

Programmer's note: rather than checking that all pairs of moduli share no GCD (an  $O(n^{**2})$  test) and rather than factoring all moduli and seeing that there is no factor in common, a check that the result gives the indicated residuals is performed – an  $O(n)$  operation.

**See also:**

`solve_congruence` (page 261)

`sympy.polys.galoistools.gf_crt` (page 796) low level `crt` routine used by this routine

`sympy.nttheory.modular.crt1(m)`

First part of Chinese Remainder Theorem, for multiple application.

## Examples

```
>>> from sympy.nttheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`sympy.nttheory.modular.crt2(m, v, mm, e, s, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

## Examples

```
>>> from sympy.nttheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`sympy.nttheory.modular.solve_congruence(*remainder_modulus_pairs, **hint)`

Compute the integer `n` that has the residual `ai` when it is divided by `mi` where the `ai` and `mi` are given as pairs to this function: `((a1, m1), (a2, m2), ...)`. If there is no solution, return `None`. Otherwise return `n` and its modulus.

The `mi` values need not be co-prime. If it is known that the moduli are not co-prime then the hint `check` can be set to `False` (default=`True`) and the check for a quicker solution via `crt()` (valid when the moduli are co-prime) will be skipped.

If the hint `symmetric` is `True` (default is `False`), the value of `n` will be within  $1/2$  of the modulus, possibly negative.

### See also:

`crt` (page 260) high level routine implementing the Chinese Remainder Theorem

## Examples

```
>>> from sympy.nttheory.modular import solve_congruence
```

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:

```
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The symmetric flag will make the result be within 1/2 of the modulus:

```
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

`sympy.nttheory.multinomial.binomial_coefficients(n)`

Return a dictionary containing pairs  $(k_1, k_2) : C_{k_1} n$  where  $C_{k_1} n$  are binomial coefficients and  $n = k_1 + k_2$ .

Examples ======

```
>>> from sympy.nttheory import binomial_coefficients
>>> binomial_coefficients(9)
{(0, 9): 1, (1, 8): 9, (2, 7): 36, (3, 6): 84,
 (4, 5): 126, (5, 4): 126, (6, 3): 84, (7, 2): 36, (8, 1): 9, (9, 0): 1}
```

See also:

`binomial_coefficients_list` (page 262), `multinomial_coefficients` (page 262)

`sympy.nttheory.multinomial.binomial_coefficients_list(n)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

See also:

`binomial_coefficients` (page 262), `multinomial_coefficients` (page 262)

Examples

```
>>> from sympy.nttheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

`sympy.nttheory.multinomial.multinomial_coefficients(m, n)`

Return a dictionary containing pairs  $\{(k_1, k_2, \dots, k_m) : C_{k_1} n\}$  where  $C_{k_1} n$  are multinomial coefficients such that  $n = k_1 + k_2 + \dots + k_m$ .

For example:

```
>>> from sympy.nttheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5) # indirect doctest
{(0, 5): 1, (1, 4): 5, (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
```

The algorithm is based on the following result:

$$\binom{n}{k_1, \dots, k_m} = \frac{k_1 + 1}{n - k_1} \sum_{i=2}^m \binom{n}{k_1 + 1, \dots, k_i - 1, \dots}$$

Code contributed to Sage by Yann Laigle-Chapuy, copied with permission of the author.

See also:

`binomial_coefficients_list` (page 262), `binomial_coefficients` (page 262)

`sympy.nttheory.multinomial.multinomial_coefficients_iterator(m, n, _tuple=<type 'tuple'>)`

multinomial coefficient iterator

This routine has been optimized for  $m$  large with respect to  $n$  by taking advantage of the fact that when the monomial tuples  $t$  are stripped of zeros, their coefficient is the same as that of the monomial tuples from `multinomial_coefficients(n, n)`. Therefore, the latter coefficients are precomputed to save memory and time.

---

```
>>> from sympy.nttheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5,3), multinomial_coefficients(3,3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,2)]
True
```

**Examples**

```
>>> from sympy.nttheory.multinomial import multinomial_coefficients_iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)
```

`sympy.nttheory.partitions_.npartitions(n, verbose=False)`

Calculate the partition function  $P(n)$ , i.e. the number of ways that  $n$  can be written as a sum of positive integers.

$P(n)$  is computed using the Hardy-Ramanujan-Rademacher formula, described e.g. at <http://mathworld.wolfram.com/PartitionFunctionP.html>

The correctness of this implementation has been tested for  $10^{**}n$  up to  $n = 8$ .

**Examples**

```
>>> from sympy.nttheory import npartitions
>>> npartitions(25)
1958
```

`sympy.nttheory.primetest.mr(n, bases)`

Perform a Miller-Rabin strong pseudoprime test on  $n$  using a given list of bases/witnesses.

**References**

- Richard Crandall & Carl Pomerance (2005), “Prime Numbers: A Computational Perspective”, Springer, 2nd edition, 135-138

A list of thresholds and the bases they require are here:  
[http://en.wikipedia.org/wiki/Miller%20Rabin\\_primality\\_test#Deterministic\\_variants\\_of\\_the\\_test](http://en.wikipedia.org/wiki/Miller%20Rabin_primality_test#Deterministic_variants_of_the_test)

**Examples**

```
>>> from sympy.nttheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

`sympy.nttheory.primetest.isprime(n)`

Test if  $n$  is a prime number (True) or not (False). For  $n < 10^{**}16$  the answer is accurate; greater  $n$  values have a small probability of actually being pseudoprimes.

Negative primes (e.g. -2) are not considered prime.

The function first looks for trivial factors, and if none is found, performs a safe Miller-Rabin strong pseudoprime test with bases that are known to prove a number prime. Finally, a general Miller-Rabin

test is done with the first k bases which will report a pseudoprime as a prime with an error of about  $4^{**k}$ . The current value of k is 46 so the error is about  $2 \times 10^{**28}$ .

See also:

[sympy.nttheory.generate.primerange](#) ([page 247](#)) Generates all primes in a given range  
[sympy.nttheory.generate.primepi](#) ([page 246](#)) Return the number of primes less than or equal to n  
[sympy.nttheory.generate.prime](#) ([page 246](#)) Return the nth prime

### Examples

```
>>> from sympy.nttheory import isprime
>>> isprime(13)
True
>>> isprime(15)
False
```

`sympy.nttheory.residue_nttheory.n_order(a, n)`

Returns the order of a modulo n.

The order of a modulo n is the smallest integer k such that  $a^{**k}$  leaves a remainder of 1 with n.

### Examples

```
>>> from sympy.nttheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

`sympy.nttheory.residue_nttheory.is_primitive_root(a, p)`

Returns True if a is a primitive root of p

a is said to be the primitive root of p if  $\gcd(a, p) == 1$  and totient(p) is the smallest positive number s.t.

$a^{**\text{totient}(p)} \cong 1 \pmod{p}$

### Examples

```
>>> from sympy.nttheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

`sympy.nttheory.residue_nttheory.primitive_root(p)`

Returns the smallest primitive root or None

**Parameters** p : positive integer

## References

- [1] W. Stein “Elementary Number Theory” (2011), page 44 [2] P. Hackman “Elementary Number Theory” (2009), Chapter C

## Examples

```
>>> from sympy.ntheory.residue_nttheory import primitive_root
>>> primitive_root(19)
2
```

```
sympy.ntheory.residue_nttheory.sqrt_mod(a, p, all_roots=False)
find a root of  $x^{**2} = a \pmod{p}$ 
```

**Parameters** **a** : integer

**p** : positive integer

**all\_roots** : if True the list of roots is returned or None

## Notes

If there is no root it is returned None; else the returned root is less or equal to  $p // 2$ ; in general is not the smallest one. It is returned  $p // 2$  only if it is the only root.

Use **all\_roots** only when it is expected that all the roots fit in memory; otherwise use **sqrt\_mod\_iter**.

## Examples

```
>>> from sympy.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

```
sympy.ntheory.residue_nttheory.quadratic_residues(p)
Returns the list of quadratic residues.
```

## Examples

```
>>> from sympy.ntheory.residue_nttheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

```
sympy.ntheory.residue_nttheory.nthroot_mod(a, n, p, all_roots=False)
find the solutions to  $x^{**n} = a \pmod{p}$ 
```

**Parameters** **a** : integer

**n** : positive integer

**p** : positive integer

**all\_roots** : if False returns the smallest root, else the list of roots

## Examples

```
>>> from sympy.nttheory.residue_nttheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

`sympy.nttheory.residue_nttheory.is_nthpow_residue(a, n, m)`

Returns True if  $x^{**n} \equiv a \pmod{m}$  has solutions.

## References

16.Hackman “Elementary Number Theory” (2009), page 76

`sympy.nttheory.residue_nttheory.is_quad_residue(a, p)`

Returns True if  $a \pmod{p}$  is in the set of squares mod  $p$ , i.e  $a \% p$  in  $\{i^{**2 \% p} \text{ for } i \text{ in range}(p)\}$ .

If  $p$  is an odd prime, an iterative method is used to make the determination:

```
>>> from sympy.nttheory import is_quad_residue
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
>>> [j for j in range(7) if is_quad_residue(j, 7)]
[0, 1, 2, 4]
```

### See also:

`legendre_symbol` (page 266), `jacobi_symbol` (page 266)

`sympy.nttheory.residue_nttheory.legendre_symbol(a, p)`

### Returns

1. 0 if  $a$  is multiple of  $p$
2. 1 if  $a$  is a quadratic residue of  $p$
3. -1 otherwise

$p$  should be an odd prime by definition

### See also:

`is_quad_residue` (page 266), `jacobi_symbol` (page 266)

## Examples

```
>>> from sympy.nttheory import legendre_symbol
>>> [legendre_symbol(i, 7) for i in range(7)]
[0, 1, 1, -1, 1, -1, -1]
>>> sorted(set([i**2 % 7 for i in range(7)]))
[0, 1, 2, 4]
```

`sympy.nttheory.residue_nttheory.jacobi_symbol(m, n)`

Returns the product of the `legendre_symbol(m, p)` for all the prime factors,  $p$ , of  $n$ .

### Returns

1. 0 if  $m \equiv 0 \pmod{n}$
2. 1 if  $x^{n/2} \equiv m \pmod{n}$  has a solution
3. -1 otherwise

See also:

[is\\_quad\\_residue](#) (page 266), [legendre\\_symbol](#) (page 266)

### Examples

```
>>> from sympy.ntheory import jacobi_symbol, legendre_symbol
>>> from sympy import Mul, S
>>> jacobi_symbol(45, 77)
-1
>>> jacobi_symbol(60, 121)
1
```

The relationship between the `jacobi_symbol` and `legendre_symbol` can be demonstrated as follows:

```
>>> L = legendre_symbol
>>> S(45).factors()
{3: 2, 5: 1}
>>> jacobi_symbol(7, 45) == L(7, 3)**2 * L(7, 5)**1
True
```

`sympy.ntheory.continued_fraction.continued_fraction_convergents(cf)`

Return an iterator over the convergents of a continued fraction (cf).

The parameter should be an iterable returning successive partial quotients of the continued fraction, such as might be returned by `continued_fraction_iterator`. In computing the convergents, the continued fraction need not be strictly in canonical form (all integers, all but the first positive). Rational and negative elements may be present in the expansion.

See also:

[continued\\_fraction\\_iterator](#) (page 268)

### Examples

```
>>> from sympy.core import Rational, pi
>>> from sympy import S
>>> from sympy.ntheory.continued_fraction import continued_fraction_convergents, continued_fraction_iterator
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]

>>> list(continued_fraction_convergents([1, S('1/2'), -7, S('1/4')]))
[1, 3, 19/5, 7]

>>> it = continued_fraction_convergents(continued_fraction_iterator(pi))
>>> for n in range(7):
...     print(next(it))
3
22/7
333/106
355/113
```

103993/33102  
104348/33215  
208341/66317

`sympy.nttheory.continued_fraction.continued_fraction_iterator(x)`  
Return continued fraction expansion of x as iterator.

## References

[R337] (page 1236)

## Examples

```
>>> from sympy.core import Rational, pi
>>> from sympy.nttheory.continued_fraction import continued_fraction_iterator

>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]

>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
1
1
1
```

`sympy.nttheory.continued_fraction.continued_fraction_periodic(p, q, d=0)`  
Find the periodic continued fraction expansion of a quadratic irrational.

Compute the continued fraction expansion of a rational or a quadratic irrational number, i.e.  $\frac{p+\sqrt{d}}{q}$ , where  $p$ ,  $q$  and  $d \geq 0$  are integers.

Returns the continued fraction representation (canonical form) as a list of integers, optionally ending (for quadratic irrationals) with repeating block as the last term of this list.

**Parameters** `p` : int

the rational part of the number's numerator

`q` : int

the denominator of the number

`d` : int, optional

the irrational part (discriminator) of the number's numerator

**See also:**

`continued_fraction_iterator` (page 268), `continued_fraction_reduce` (page 269)

## References

[R338] (page 1236), [R339] (page 1236)

## Examples

```
>>> from sympy.nttheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

`sympy.nttheory.continued_fraction.continued_fraction_reduce(cf)`  
Reduce a continued fraction to a rational or quadratic irrational.

Compute the rational or quadratic irrational number from its terminating or periodic continued fraction expansion. The continued fraction expansion (cf) should be supplied as a terminating iterator supplying the terms of the expansion. For terminating continued fractions, this is equivalent to `list(continued_fraction_convergents(cf))[-1]`, only a little more efficient. If the expansion has a repeating part, a list of the repeating terms should be returned as the last element from the iterator. This is the format returned by `continued_fraction_periodic`.

For quadratic irrationals, returns the largest solution found, which is generally the one sought, if the fraction is in canonical form (all terms positive except possibly the first).

See also:

`continued_fraction_periodic` (page 268)

## Examples

```
>>> from sympy.nttheory.continued_fraction import continued_fraction_reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
1/2 + sqrt(5)/2
>>> from sympy.nttheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```

```
class sympy.nttheory.mobius
    Mbius function maps natural number to {-1, 0, 1}
```

**It is defined as follows:**

1. 1 if  $n = 1$ .
2. 0 if  $n$  has a squared prime factor.
3.  $(-1)^k$  if  $n$  is a square-free positive integer with  $k$  number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Mbius Function model).

**Parameters n :** positive integer

## References

[R340] (page 1236), [R341] (page 1236)

## Examples

```
>>> from sympy.nttheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

`sympy.nttheory.egyptian_fraction.egyptian_fraction(r, algorithm='Greedy')`

Return the list of denominators of an Egyptian fraction expansion [R342] (page 1236) of the said rational  $r$ .

**Parameters r :** Rational

a positive rational number.

**algorithm :** { “Greedy”, “Graham Jewett”, “Takenouchi”, “Golomb” }, optional

Denotes the algorithm to be used (the default is “Greedy”).

## See also:

`sympy.core.numbers.Rational` (page 101)

## Notes

Currently the following algorithms are supported:

1. Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm [R343] (page 1236). At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

(a) Given  $p/q$  in lowest terms, generates an expansion of maximum length  $p$ . Even as the numerators get large, the number of terms is seldom more than a handful.

(b) Uses minimal memory.

(c) The terms can blow up (standard examples of this are  $5/121$  and  $31/311$ ). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

## 2. Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always  $2**(\text{x}/\text{gcd}(\text{x}, \text{y})) - 1$ . See [R344] (page 1236).

## 3. Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R344] (page 1236).

## 4. Golomb's Algorithm

A method given by Golomb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R345] (page 1236).

If the given rational is greater than or equal to 1, a greedy algorithm of summing the harmonic sequence  $1/1 + 1/2 + 1/3 + \dots$  is used, taking all the unit fractions of this sequence until adding one more would be greater than the given number. This list of denominators is prefixed to the result from the requested algorithm used on the remainder. For example, if  $r$  is  $8/3$ , using the Greedy algorithm, we get  $[1, 2, 3, 4, 5, 6, 7, 14, 420]$ , where the beginning of the sequence,  $[1, 2, 3, 4, 5, 6, 7]$  is part of the harmonic sequence summing to  $363/140$ , leaving a remainder of  $31/420$ , which yields  $[14, 420]$  by the Greedy algorithm. The result of `egyptian_fraction(Rational(8, 3), "Golomb")` is  $[1, 2, 3, 4, 5, 6, 7, 14, 574, 2788, 6460, 11590, 33062, 113820]$ , and so on.

## References

[R342] (page 1236), [R343] (page 1236), [R344] (page 1236), [R345] (page 1236)

## Examples

```
>>> from sympy import Rational
>>> from sympy.nttheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction(Rational(3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction(Rational(3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction(Rational(3, 7), "Golomb")
[3, 15, 35]
>>> egyptian_fraction(Rational(11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

## 3.4 Basic Cryptography Module

Included in this module are both block ciphers and stream ciphers.

- Shift cipher
- Affine cipher
- Bifid ciphers
- Vigenere's cipher
- substitution ciphers
- Hill's cipher
- RSA
- Kid RSA
- linear feedback shift registers (a stream cipher)
- ElGamal encryption

```
sympy.crypto.crypto.alphabet_of_cipher(symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')
```

Returns the list of characters in the string input defining the alphabet.

## Notes

First, some basic definitions.

A *substitution cipher* is a method of encryption by which “units” (not necessarily characters) of plaintext are replaced with ciphertext according to a regular system. The “units” may be characters (ie, words of length 1), words of length 2, and so forth.

A *transposition cipher* is a method of encryption by which the positions held by “units” of plaintext are replaced by a permutation of the plaintext. That is, the order of the units is changed using a bijective function on the characters’ positions to perform the encryption.

A *monoalphabetic cipher* uses fixed substitution over the entire message, whereas a *polyalphabetic cipher* uses a number of substitutions at different times in the message.

Each of these ciphers require an alphabet for the messages to be constructed from.

## Examples

```
>>> from sympy.crypto.crypto import alphabet_of_cipher
>>> alphabet_of_cipher()
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
>>> L = [str(i) for i in range(10)] + ['a', 'b', 'c']; L
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']
>>> A = ''.join(L); A
'0123456789abc'
>>> alphabet_of_cipher(A)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c']
>>> alphabet_of_cipher()
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

```
sympy.crypto.crypto.cycle_list(k, n)
```

Returns the cyclic shift of the list range(n) by k.

## Examples

```
>>> from sympy.crypto.crypto import cycle_list, alphabet_of_cipher
>>> L = cycle_list(3,26); L
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 0, 1, 2]
>>> A = alphabet_of_cipher(); A
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
>>> [A[i] for i in L]
['D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'A', 'B', 'C']
```

`sympy.crypto.crypto.encipher_shift(pt, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')`

Performs shift cipher encryption on plaintext `pt`, and returns the ciphertext.

## Notes

The shift cipher is also called the Caesar cipher, after Julius Caesar, who, according to Suetonius, used it with a shift of three to protect messages of military significance. Caesar's nephew Augustus reportedly used a similar cipher, but with a right shift of 1.

ALGORITHM:

INPUT:

- `k`: an integer from 0 to 25 (the secret key)
- `m`: string of upper-case letters (the plaintext message)

OUTPUT:

- `c`: string of upper-case letters (the ciphertext message)

STEPS:

0. Identify the alphabet `A`, ..., `Z` with the integers 0, ..., 25.
1. Compute from the string `m` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by adding (`k mod 26`) to each element in `L1`.
3. Compute from the list `L2` a string `c` of corresponding letters.

## Examples

```
>>> from sympy.crypto.crypto import encipher_shift
>>> pt = "GONAVYBEATARMY"
>>> encipher_shift(pt, 1)
'HPOBWZCFBUBSNZ'
>>> encipher_shift(pt, 0)
'GONAVYBEATARMY'
>>> encipher_shift(pt, -1)
'FNMZXADZSZQLX'
```

`sympy.crypto.crypto.encipher_affine(pt, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')`

Performs the affine cipher encryption on plaintext `pt`, and returns the ciphertext.

Encryption is based on the map  $x \rightarrow ax + b \pmod{26}$ . Decryption is based on the map  $x \rightarrow cx + d \pmod{26}$ , where  $c = a^{-1} \pmod{26}$  and  $d = -a^{-1}c \pmod{26}$ . (In particular, for the map to be invertible, we need  $\gcd(a, 26) = 1$ .)

## Notes

This is a straightforward generalization of the shift cipher.

ALGORITHM:

INPUT:

`a, b`: a pair integers, where  $\gcd(a, 26) = 1$  (the secret key)

`m`: string of upper-case letters (the plaintext message)

OUTPUT:

`c`: string of upper-case letters (the ciphertext message)

STEPS:

0. Identify the alphabet “A”, ..., “Z” with the integers 0, ..., 25.
1. Compute from the string `m` a list `L1` of corresponding integers.
2. Compute from the list `L1` a new list `L2`, given by replacing `x` by  $a*x + b \pmod{26}$ , for each element `x` in `L1`.
3. Compute from the list `L2` a string `c` of corresponding letters.

## Examples

```
>>> from sympy.crypto.crypto import encipher_affine
>>> pt = "GONAVYBEATARMY"
>>> encipher_affine(pt, (1, 1))
'HPOBWZCFBUBSNZ'
>>> encipher_affine(pt, (1, 0))
'GONAVYBEATARMY'
>>> pt = "GONAVYBEATARMY"
>>> encipher_affine(pt, (3, 1))
'TROBMVENBGBALV'
>>> ct = "TROBMVENBGBALV"
>>> encipher_affine(ct, (9, 17))
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_substitution(pt, key, symbols='ABCDEFGHIJKLMNOPQRSTUVWXYZ')`  
Performs the substitution cipher encryption on plaintext `pt`, and returns the ciphertext.

Assumes the `pt` has only letters taken from `symbols`. Assumes `key` is a permutation of the symbols. This function permutes the letters of the plaintext using the permutation given in `key`. The decryption uses the inverse permutation. Note that if the permutation in `key` is order 2 (eg, a transposition) then the encryption permutation and the decryption permutation are the same.

## Examples

```
>>> from sympy.crypto.crypto import alphabet_of_cipher, encipher_substitution
>>> symbols = "ABCDEFGHIJKLMNPQRSTUVWXYZ"
>>> A = alphabet_of_cipher(symbols)
>>> key = "BACDEFGHIJKLMNOPQRSTUVWXYZ"
>>> pt = "go navy! beat army!"
>>> encipher_substitution(pt, key)
'GONBVYAEETBRMY',
>>> ct = 'GONBVYAEETBRMY',
>>> encipher_substitution(ct, key)
'GONAVYBEATARMY'
```

`sympy.crypto.crypto.encipher_vigenere(pt, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')`  
 Performs the Vigenre cipher encryption on plaintext `pt`, and returns the ciphertext.

### Notes

The Vigenre cipher is named after Blaise de Vigenre, a sixteenth century diplomat and cryptographer, by a historical accident. Vigenre actually invented a different and more complicated cipher. The so-called *Vigenre cipher* was actually invented by Giovan Batista Belaso in 1553.

This cipher was used in the 1800's, for example, during the American Civil War. The Confederacy used a brass cipher disk to implement the Vigenre cipher (now on display in the NSA Museum in Fort Meade) [R62] (page 1236).

The Vigenre cipher is a generalization of the shift cipher. Whereas the shift cipher shifts each letter by the same amount (that amount being the key of the shift cipher) the Vigenre cipher shifts a letter by an amount determined by the key (which is a word or phrase known only to the sender and receiver).

For example, if the key was a single letter, such as “C”, then the so-called Vigenere cipher is actually a shift cipher with a shift of 2 (since “C” is the 2nd letter of the alphabet, if you start counting at 0). If the key was a word with two letters, such as “CA”, then the so-called Vigenre cipher will shift letters in even positions by 2 and letters in odd positions are left alone (shifted by 0, since “A” is the 0th letter, if you start counting at 0).

ALGORITHM:

INPUT:

`key`: a string of upper-case letters (the secret key)

`m`: string of upper-case letters (the plaintext message)

OUTPUT:

`c`: string of upper-case letters (the ciphertext message)

STEPS:

0. Identify the alphabet A, ..., Z with the integers 0, ..., 25.
1. Compute from the string `key` a list `L1` of corresponding integers. Let `n1 = len(L1)`.
2. Compute from the string `m` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
3. Break `L2` up sequentially into sublists of size `n1`, and one sublist at the end of size smaller or equal to `n1`.
4. For each of these sublists `L` of `L2`, compute a new list `C` given by `C[i] = L[i] + L1[i] (mod 26)` to the `i`-th element in the sublist, for each `i`.
5. Assemble these lists `C` by concatenation into a new list of length `n2`.

6. Compute from the new list a string `c` of corresponding letters.

Once it is known that the key is, say,  $n$  characters long, frequency analysis can be applied to every  $n$ -th letter of the ciphertext to determine the plaintext. This method is called *Kasiski examination* (although it was first discovered by Babbage).

The cipher Vigenre actually discovered is an “auto-key” cipher described as follows.

#### ALGORITHM:

##### INPUT:

`key`: a string of upper-case letters (the secret key)

`m`: string of upper-case letters (the plaintext message)

##### OUTPUT:

`c`: string of upper-case letters (the ciphertext message)

#### STEPS:

0. Identify the alphabet A, ..., Z with the integers 0, ..., 25.
1. Compute from the string `m` a list `L2` of corresponding integers. Let `n2 = len(L2)`.
2. Let `n1` be the length of the key. Concatenate the string `key` with the first  $n_2 - n_1$  characters of the plaintext message. Compute from this string of length `n2` a list `L1` of corresponding integers. Note `n2 = len(L1)`.
3. Compute a new list `C` given by  $C[i] = L1[i] + L2[i] \pmod{26}$ , for each  $i$ . Note `n2 = len(C)`.
4. Compute from the new list a string `c` of corresponding letters.

#### References

[R62] (page 1236)

#### Examples

```
>>> from sympy.crypto.crypto import encipher_vigenere
>>> key = "encrypt"
>>> pt = "meet me on monday"
>>> encipher_vigenere(pt, key)
'QRGKKTHRZQEBPR'
```

```
sympy.crypto.crypto.decipher_vigenere(ct, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')
Decode using the Vigenre cipher.
```

#### Examples

```
>>> from sympy.crypto.crypto import decipher_vigenere
>>> key = "encrypt"
>>> ct = "QRGK kt HRZQE BPR"
>>> decipher_vigenere(ct, key)
'MEETMEONMONDAY'
```

```
sympy.crypto.crypto.encipher_hill(pt, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')
```

Performs the Hill cipher encryption on plaintext  $pt$ , and returns the ciphertext.

### Notes

The Hill cipher [R63] (page 1236), invented by Lester S. Hill in the 1920's [R64] (page 1236), was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. The following discussion assumes an elementary knowledge of matrices.

First, each letter is first encoded as a number. We assume here that "A"  $\leftrightarrow$  0, "B"  $\leftrightarrow$  1, ..., "Z"  $\leftrightarrow$  25. We denote the integers  $\{0, 1, \dots, 25\}$  by  $Z_{26}$ . Suppose your message  $m$  consists of  $n$  capital letters, with no spaces. This may be regarded an  $n$ -tuple  $M$  of elements of  $Z_{26}$ . A key in the Hill cipher is a  $k \times k$  matrix  $K$ , all of whose entries are in  $Z_{26}$ , such that the matrix  $K$  is invertible (ie, that the linear transformation  $K : Z_{26}^k \rightarrow Z_{26}^k$  is one-to-one).

ALGORITHM:

INPUT:

key: a  $k \times k$  invertible matrix  $K$ , all of whose entries are in  $Z_{26}$

$m$ : string of  $n$  upper-case letters (the plaintext message) (Note: Sage assumes that  $n$  is a multiple of  $k$ .)

OUTPUT:

$c$ : string of upper-case letters (the ciphertext message)

STEPS:

0. Identify the alphabet A, ..., Z with the integers 0, ..., 25.
1. Compute from the string  $m$  a list  $L$  of corresponding integers. Let  $n = \text{len}(L)$ .
2. Break the list  $L$  up into  $t = \text{ceiling}(n/k)$  sublists  $L_1, \dots, L_t$  of size  $k$  (where the last list might be "padded" by 0's to ensure it is size  $k$ ).
3. Compute new list  $C_1, \dots, C_t$  given by  $C[i] = K * L_i$  (arithmetic is done mod 26), for each  $i$ .
4. Concatenate these into a list  $C = C_1 + \dots + C_t$ .
5. Compute from  $C$  a string  $c$  of corresponding letters. This has length  $k*t$ .

### References

[R63] (page 1236), [R64] (page 1236)

### Examples

```
>>> from sympy.crypto.crypto import encipher_hill
>>> from sympy import Matrix
>>> pt = "meet me on monday"
>>> key = Matrix([[1, 2], [3, 5]])
>>> encipher_hill(pt, key)
'UEQDUEODOCTCWQ'
>>> pt = "meet me on tuesday"
>>> encipher_hill(pt, key)
```

```
'UEQDUEODHBOYDJYU'  
>>> pt = "GONAVYBEATARMY"  
>>> key = Matrix([[1, 0, 1], [0, 1, 1], [2, 2, 3]])  
>>> encipher_hill(pt, key)  
'TBBYTKBEKKRLMYU'  
  
sympy.crypto.crypto.decipher_hill(ct, key, symbols='ABCDEFGHIJKLMNPQRSTUVWXYZ')  
Deciphering is the same as enciphering but using the inverse of the key matrix.
```

## Examples

```
>>> from sympy.crypto.crypto import decipher_hill  
>>> from sympy import Matrix  
>>> ct = "UEQDUEODOCTCWQ"  
>>> key = Matrix([[1, 2], [3, 5]])  
>>> decipher_hill(ct, key)  
'MEETMEONMONDAY'  
>>> ct = "UEQDUEODHBOYDJYU"  
>>> decipher_hill(ct, key)  
'MEETMEONTUESDAY'
```

```
sympy.crypto.crypto.encipher_bifid5(pt, key)
```

Performs the Bifid cipher encryption on plaintext `pt`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the  $5 \times 5$  Polybius square.

## Notes

The Bifid cipher was invented around 1901 by Felix Delastelle. It is a *fractional substitution* cipher, where letters are replaced by pairs of symbols from a smaller alphabet. The cipher uses a  $5 \times 5$  square filled with some ordering of the alphabet, except that “i”s and “j”s are identified (this is a so-called Polybius square; there is a  $6 \times 6$  analog if you add back in “j” and also append onto the usual 26 letter alphabet, the digits 0, 1, ..., 9). According to Helen Gaines’ book *Cryptanalysis*, this type of cipher was used in the field by the German Army during World War I.

ALGORITHM: (5x5 case)

INPUT:

`pt`: plaintext string (no “j”s)

`key`: short string for key (no repetitions, no “j”s)

OUTPUT:

ciphertext (using Bifid5 cipher in all caps, no spaces, no “J”s)

**STEPS:**

1. Create the  $5 \times 5$  Polybius square `S` associated to the `k` as follows:
  - (a) starting top left, moving left-to-right, top-to-bottom, place the letters of the key into a  $5 \times 5$  matrix,
  - (b) when finished, add the letters of the alphabet not in the key until the  $5 \times 5$  square is filled
2. Create a list `P` of pairs of numbers which are the coordinates in the Polybius square of the letters in `pt`.

3. Let  $L_1$  be the list of all first coordinates of  $P$  (length of  $L_1 = n$ ), let  $L_2$  be the list of all second coordinates of  $P$  (so the length of  $L_2$  is also  $n$ ).
4. Let  $L$  be the concatenation of  $L_1$  and  $L_2$  (length  $L = 2n$ ), except that consecutive numbers are paired ( $L[2*i]$ ,  $L[2*i + 1]$ ). You can regard  $L$  as a list of pairs of length  $n$ .
5. Let  $C$  be the list of all letters which are of the form  $S[i, j]$ , for all  $(i, j)$  in  $L$ . As a string, this is the ciphertext  $ct$ .

### Examples

```
>>> from sympy.crypto.crypto import encipher_bifid5
>>> pt = "meet me on monday"
>>> key = "encrypt"
>>> encipher_bifid5(pt, key)
'LNLQNPPNPGADK'
>>> pt = "meet me on friday"
>>> encipher_bifid5(pt, key)
'LNLFGPPNPGRSK'
```

`sympy.crypto.crypto.decipher_bifid5(ct, key)`

Performs the Bifid cipher decryption on ciphertext  $ct$ , and returns the plaintext.

This is the version of the Bifid cipher that uses the  $5 \times 5$  Polybius square.

INPUT:

`ct`: ciphertext string (digits okay)  
`key`: short string for key (no repetitions, digits okay)

OUTPUT:

plaintext from Bifid5 cipher (all caps, no spaces, no “J”s)

### Examples

```
>>> from sympy.crypto.crypto import encipher_bifid5, decipher_bifid5
>>> key = "encrypt"
>>> pt = "meet me on monday"
>>> encipher_bifid5(pt, key)
'LNLQNPPNPGADK'
>>> ct = 'LNLLQNPPNPGADK'
>>> decipher_bifid5(ct, key)
'MEETMEONMONDAY'
```

`sympy.crypto.crypto.bifid5_square(key)`

5x5 Polybius square.

Produce the Polybius square for the  $5 \times 5$  Bifid cipher.

### Examples

```
>>> from sympy.crypto.crypto import bifid5_square
>>> bifid5_square("gold bug")
Matrix([
```

```
[G, O, L, D, B],  
[U, A, C, E, F],  
[H, I, K, M, N],  
[P, Q, R, S, T],  
[V, W, X, Y, Z]])
```

`sympy.crypto.crypto.encipher_bifid6(pt, key)`

Performs the Bifid cipher encryption on plaintext `pt`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the  $6 \times 6$  Polybius square. Assumes alphabet of symbols is “A”, ..., “Z”, “0”, ..., “9”.

INPUT:

`pt`: plaintext string (digits okay)

`key`: short string for key (no repetitions, digits okay)

OUTPUT:

ciphertext from Bifid cipher (all caps, no spaces)

## Examples

```
>>> from sympy.crypto.crypto import encipher_bifid6  
>>> key = "encrypt"  
>>> pt = "meet me on monday at 8am"  
>>> encipher_bifid6(pt, key)  
'HNHOKNTA5MEPEGNQZYG'  
>>> encipher_bifid6(pt, key)  
'HNHOKNTA5MEPEGNQZYG'
```

`sympy.crypto.crypto.decipher_bifid6(ct, key)`

Performs the Bifid cipher decryption on ciphertext `ct`, and returns the plaintext.

This is the version of the Bifid cipher that uses the  $6 \times 6$  Polybius square. Assumes alphabet of symbols is “A”, ..., “Z”, “0”, ..., “9”.

INPUT:

`ct`: ciphertext string (digits okay)

`key`: short string for key (no repetitions, digits okay)

OUTPUT:

plaintext from Bifid cipher (all caps, no spaces)

## Examples

```
>>> from sympy.crypto.crypto import encipher_bifid6, decipher_bifid6  
>>> key = "encrypt"  
>>> pt = "meet me on monday at 8am"  
>>> encipher_bifid6(pt, key)  
'HNHOKNTA5MEPEGNQZYG'  
>>> ct = "HNHOKNTA5MEPEGNQZYG"  
>>> decipher_bifid6(ct, key)  
'MEETMEONMONDAYAT8AM'
```

```
sympy.crypto.crypto.bifid6_square(key)
6x6 Polybius square.
```

Produces the Polybius square for the  $6 \times 6$  Bifid cipher. Assumes alphabet of symbols is “A”, ..., “Z”, “0”, ..., “9”.

### Examples

```
>>> from sympy.crypto.crypto import bifid6_square
>>> key = "encrypt"
>>> bifid6_square(key)
Matrix([
[E, N, C, R, Y, P],
[T, A, B, D, F, G],
[H, I, J, K, L, M],
[O, Q, S, U, V, W],
[X, Z, 0, 1, 2, 3],
[4, 5, 6, 7, 8, 9]])
```

```
sympy.crypto.crypto.encipher_bifid7(pt, key)
```

Performs the Bifid cipher encryption on plaintext `pt`, and returns the ciphertext.

This is the version of the Bifid cipher that uses the  $7 \times 7$  Polybius square. Assumes alphabet of symbols is “A”, ..., “Z”, “0”, ..., “22”. (Also, assumes you have some way of distinguishing “22” from “2”, “2” juxtaposed together for deciphering...)

INPUT:

`pt`: plaintext string (digits okay)  
`key`: short string for key (no repetitions, digits okay)

OUTPUT:

ciphertext from Bifid7 cipher (all caps, no spaces)

### Examples

```
>>> from sympy.crypto.crypto import encipher_bifid7
>>> key = "encrypt"
>>> pt = "meet me on monday at 8am"
>>> encipher_bifid7(pt, key)
'JEJJLNAA3ME19YF3J222R'
```

```
sympy.crypto.crypto.bifid7_square(key)
```

7x7 Polybius square.

Produce the Polybius square for the  $7 \times 7$  Bifid cipher. Assumes alphabet of symbols is “A”, ..., “Z”, “0”, ..., “22”. (Also, assumes you have some way of distinguishing “22” from “2”, “2” juxtaposed together for deciphering...)

### Examples

```
>>> from sympy.crypto.crypto import bifid7_square
>>> bifid7_square("gold bug")
Matrix([
[G,  O,  L,  D,  B,  U,  A],
```

```
[ C,   F,   H,   I,   J,   K],  
[ M,   N,   P,   Q,   R,   S,   T],  
[ V,   W,   X,   Y,   Z,  0,   1],  
[ 2,   3,   4,   5,   6,   7,   8],  
[ 9,  10,  11,  12,  13,  14,  15],  
[16,  17,  18,  19,  20,  21,  22]])
```

`sympy.crypto.crypto.rsa_public_key(p, q, e)`

The RSA *public key* is the pair  $(n, e)$ , where  $n$  is a product of two primes and  $e$  is relatively prime (coprime) to the Euler totient  $\phi(n)$ .

### Examples

```
>>> from sympy.crypto.crypto import rsa_public_key  
>>> p, q, e = 3, 5, 7  
>>> n, e = rsa_public_key(p, q, e)  
>>> n  
15  
>>> e  
7
```

`sympy.crypto.crypto.rsa_private_key(p, q, e)`

The RSA *private key* is the pair  $(n, d)$ , where  $n$  is a product of two primes and  $d$  is the inverse of  $e$  ( $\text{mod } \phi(n)$ ).

### Examples

```
>>> from sympy.crypto.crypto import rsa_private_key  
>>> p, q, e = 3, 5, 7  
>>> rsa_private_key(p, q, e)  
(15, 7)
```

`sympy.crypto.crypto.encipher_rsa(pt, puk)`

In RSA, a message  $m$  is encrypted by computing  $m^e \pmod{n}$ , where `puk` is the public key  $(n, e)$ .

### Examples

```
>>> from sympy.crypto.crypto import encipher_rsa, rsa_public_key  
>>> p, q, e = 3, 5, 7  
>>> puk = rsa_public_key(p, q, e)  
>>> pt = 12  
>>> encipher_rsa(pt, puk)  
3
```

`sympy.crypto.crypto.decipher_rsa(ct, prk)`

In RSA, a ciphertext  $c$  is decrypted by computing  $c^d \pmod{n}$ , where `prk` is the private key  $(n, d)$ .

### Examples

```
>>> from sympy.crypto.crypto import decipher_rsa, rsa_private_key  
>>> p, q, e = 3, 5, 7  
>>> prk = rsa_private_key(p, q, e)
```

```
>>> ct = 3
>>> decipher_rsa(ct, prk)
12

sympy.crypto.crypto.kid_rsa_public_key(a, b, A, B)
Kid RSA is a version of RSA useful to teach grade school children since it does not involve exponentiation.
```

Alice wants to talk to Bob. Bob generates keys as follows. Key generation:

- Select positive integers  $a, b, A, B$  at random.
- Compute  $M = ab - 1$ ,  $e = AM + a$ ,  $d = BM + b$ ,  $n = (ed - 1)/M$ .
- The *public key* is  $(n, e)$ . Bob sends these to Alice.
- The *private key* is  $d$ , which Bob keeps secret.

Encryption: If  $m$  is the plaintext message then the ciphertext is  $c = me \pmod{n}$ .

Decryption: If  $c$  is the ciphertext message then the plaintext is  $m = cd \pmod{n}$ .

### Examples

```
>>> from sympy.crypto.crypto import kid_rsa_public_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_public_key(a, b, A, B)
(369, 58)
```

```
sympy.crypto.crypto.kid_rsa_private_key(a, b, A, B)
```

Compute  $M = ab - 1$ ,  $e = AM + a$ ,  $d = BM + b$ ,  $n = (ed - 1)/M$ . The *private key* is  $d$ , which Bob keeps secret.

### Examples

```
>>> from sympy.crypto.crypto import kid_rsa_private_key
>>> a, b, A, B = 3, 4, 5, 6
>>> kid_rsa_private_key(a, b, A, B)
(369, 70)
```

```
sympy.crypto.crypto.encipher_kid_rsa(pt, puk)
```

Here  $pt$  is the plaintext and  $puk$  is the public key.

### Examples

```
>>> from sympy.crypto.crypto import encipher_kid_rsa, kid_rsa_public_key
>>> pt = 200
>>> a, b, A, B = 3, 4, 5, 6
>>> pk = kid_rsa_public_key(a, b, A, B)
>>> encipher_kid_rsa(pt, pk)
161
```

```
sympy.crypto.crypto.decipher_kid_rsa(ct, prk)
```

Here  $pt$  is the plaintext and  $prk$  is the private key.

## Examples

```
>>> from sympy.crypto.crypto import kid_rsa_public_key, kid_rsa_private_key, decipher_kid_rsa, encipher_kid_rsa
>>> a, b, A, B = 3, 4, 5, 6
>>> d = kid_rsa_private_key(a, b, A, B)
>>> pt = 200
>>> pk = kid_rsa_public_key(a, b, A, B)
>>> prk = kid_rsa_private_key(a, b, A, B)
>>> ct = encipher_kid_rsa(pt, pk)
>>> decipher_kid_rsa(ct, prk)
200
```

`sympy.crypto.crypto.encode_morse(pt)`

Encodes a plaintext into popular Morse Code with letters separated by “|” and words by “||”.

## References

[R65] (page 1236)

## Examples

```
>>> from sympy.crypto.crypto import encode_morse
>>> pt = 'ATTACK THE RIGHT FLANK'
>>> encode_morse(pt)
'.-|-|-.|-.-|-.|-||-|....|.||.-.|..|--.|....|-||..-.|.---|.-|-.|-.-'
```

`sympy.crypto.crypto.decode_morse(mc)`

Decodes a Morse Code with letters separated by “|” and words by “||” into plaintext.

## References

[R66] (page 1236)

## Examples

```
>>> from sympy.crypto.crypto import decode_morse
>>> mc = '--|---|...-|.||.-|...|-'
>>> decode_morse(mc)
'MOVE EAST'
```

`sympy.crypto.crypto.lfsr_sequence(key, fill, n)`

This function creates an lfsr sequence.

INPUT:

**key:** a list of finite field elements,  $[c_0, c_1, \dots, c_k]$ .

**fill:** the list of the initial terms of the lfsr sequence,  $[x_0, x_1, \dots, x_k]$ .

**n:** number of terms of the sequence that the function returns.

OUTPUT:

The lfsr sequence defined by  $x_{n+1} = c_k x_n + \dots + c_0 x_{n-k}$ , for  $n \leq k$ .

## Notes

S. Golomb [G66] (page 1236) gives a list of three statistical properties a sequence of numbers  $a = \{a_n\}_{n=1}^{\infty}$ ,  $a_n \in \{0, 1\}$ , should display to be considered “random”. Define the autocorrelation of  $a$  to be

$$C(k) = C(k, a) = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{n=1}^N (-1)^{a_n + a_{n+k}}.$$

In the case where  $a$  is periodic with period  $P$  then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^P (-1)^{a_n + a_{n+k}}.$$

Assume  $a$  is periodic with period  $P$ .

- balance:

$$\left| \sum_{n=1}^P (-1)^{a_n} \right| \leq 1.$$

- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

(For sequences satisfying these first two properties, it is known that  $\epsilon = -1/P$  must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1’s as there are of 0’s.

## References

[G66] (page 1236)

## Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> lfsr_sequence(key, fill, 10)
[1 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 0 mod 2, 1 mod 2, 1 mod 2, 0 mod 2, 0 mod 2, 1 mod 2]
```

`sympy.crypto.crypto.lfsr_autocorrelation(L, P, k)`

This function computes the lsfr autocorrelation function.

INPUT:

L: is a periodic sequence of elements of  $GF(2)$ . L must have length larger than P.

P: the period of L

k: an integer ( $0 < k < p$ )

OUTPUT:

the k-th value of the autocorrelation of the LFSR L

## Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence, lfsr_autocorrelation
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_autocorrelation(s, 15, 7)
-1/15
>>> lfsr_autocorrelation(s, 15, 0)
1
```

`sympy.crypto.crypto.lfsr_connection_polynomial(s)`

This function computes the lsfr connection polynomial.

INPUT:

`s`: a sequence of elements of even length, with entries in a finite field

OUTPUT:

$C(x)$ : the connection polynomial of a minimal LFSR yielding `s`.

This implements the algorithm in section 3 of J. L. Massey's article [M67] (page 1237).

## References

[M67] (page 1237)

## Examples

```
>>> from sympy.crypto.crypto import lfsr_sequence, lfsr_connection_polynomial
>>> from sympy.polys.domains import FF
>>> F = FF(2)
>>> fill = [F(1), F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**4 + x + 1
>>> fill = [F(1), F(0), F(0), F(1)]
>>> key = [F(1), F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(1), F(0)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x**2 + 1
>>> fill = [F(1), F(0), F(1)]
>>> key = [F(1), F(0), F(1)]
>>> s = lfsr_sequence(key, fill, 20)
>>> lfsr_connection_polynomial(s)
x**3 + x + 1
```

`sympy.crypto.crypto.elgamal_public_key(prk)`

Return three number tuple as public key.

**Parameters** `prk` : Tuple (p, r, e) generated by `elgamal_private_key`  
**Returns** (`p, r, e = r**d mod p`) : d is a random number in private key.

#### Examples

```
>>> from sympy.crypto.crypto import elgamal_public_key
>>> elgamal_public_key((1031, 14, 636))
(1031, 14, 212)
```

`sympy.crypto.crypto.elgamal_private_key(digit=10)`  
Return three number tuple as private key.

Elgamal encryption is based on the mathematical problem called the Discrete Logarithm Problem (DLP). For example,

$$a^b \equiv c \pmod{p}$$

In general, if a and b are known, c is easily calculated. If b is unknown, it is hard to use a and c to get b.

**Parameters** `digit` : Key length in binary

**Returns** (`p, r, d`) : p = prime number, r = primitive root, d = random number

#### Examples

```
>>> from sympy.crypto.crypto import elgamal_private_key
>>> from sympy.ntheory import is_primitive_root, isprime
>>> a, b, _ = elgamal_private_key()
>>> isprime(a)
True
>>> is_primitive_root(b, a)
True
```

`sympy.crypto.crypto.encipher_elgamal(m, puk)`  
Encrypt message with public key

m is plain text message in int. puk is public key (p, r, e). In order to encrypt a message, a random number a between 2 and p, encrypted message is  $c_1$  and  $c_2$

$$c_1 \equiv r^a \pmod{p}$$

$$c_2 \equiv me^a \pmod{p}$$

**Parameters** `m` : int of encoded message

`puk` : public key

**Returns** (`c1, c2`) : Encipher into two number

#### Examples

```
>>> from sympy.crypto.crypto import encipher_elgamal
>>> encipher_elgamal(100, (1031, 14, 212))
(835, 271)
```

```
sympy.crypto.crypto.decrypt_elgamal(ct, prk)
Decrypt message with private key
ct = (c1, c2)
prk = (p, r, d)
According to extended Eucliden theorem, uc1^d + pn = 1
u ≡ 1/c1^d (mod p)
uc2 ≡ 1/c1^d c2 ≡ 1/r^ad c2 (mod p)
1/r^ad m e^a ≡ 1/r^ad m r^da ≡ m (mod p)
```

### Examples

```
>>> from sympy.crypto.crypto import decrypt_elgamal
>>> decrypt_elgamal((835, 271), (1031, 14, 636))
100
```

## 3.5 Concrete Mathematics

### 3.5.1 Hypergeometric terms

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term  $a(n)$  then it is hypergeometric if its consecutive term ratio is a rational function in  $n$ .

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```
>>> from sympy import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
True
>>> ff(n, k).is_hypergeometric(k)
True
```

To say more, all previously shown examples are valid for integer linear arguments:

```
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise it will return `None` to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

### 3.5.2 Concrete Class Reference

```
class sympy.concrete.summations.Sum
    Represents unevaluated summation.
```

`Sum` represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.

**See also:**

[sympy.concrete.summations.summation](#) (page 300), [sympy.concrete.products.Product](#) (page 292), [sympy.concrete.products.product](#) (page 301)

#### References

[R17] (page 1237), [R18] (page 1237), [R19] (page 1237)

## Examples

```
>>> from sympy.abc import i, k, m, n, x
>>> from sympy import Sum, factorial, oo, IndexedBase, Function
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()
Piecewise((1/(-x + 1), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), True))
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
exp(x)
```

Here are examples to do summation with symbolic indices. You can use either Function or IndexedBase classes:

```
>>> f = Function('f')
>>> Sum(f(n), (n, 0, 3)).doit()
f(0) + f(1) + f(2) + f(3)
>>> Sum(f(n), (n, 0, oo)).doit()
Sum(f(n), (n, 0, oo))
>>> f = IndexedBase('f')
>>> Sum(f[n]**2, (n, 0, 3)).doit()
f[0]**2 + f[1]**2 + f[2]**2 + f[3]**2
```

An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

```
>>> S = Sum(i, (i, 1, n)).doit()
>>> S
n**2/2 + n/2
>>> S.subs(n, -4)
6
>>> Sum(i, (i, 1, -4)).doit()
6
>>> Sum(-i, (i, -3, 0)).doit()
6
```

An explicit example of the Karr summation convention:

```
>>> S1 = Sum(i**2, (i, m, m+n-1)).doit()
>>> S1
m**2*n + m*n**2 - m*n + n**3/3 - n**2/2 + n/6
>>> S2 = Sum(i**2, (i, m+n, m-1)).doit()
>>> S2
-m**2*n - m*n**2 + m*n - n**3/3 + n**2/2 - n/6
>>> S1 + S2
0
>>> S3 = Sum(i, (i, m, m-1)).doit()
>>> S3
0
```

---

```
euler_maclaurin(m=0, n=0, eps=0, eval_integral=True)
```

Return an Euler-Maclaurin approximation of self, where m is the number of leading terms to sum directly and n is the number of terms in the tail.

With  $m = n = 0$ , this is simply the corresponding integral plus a first-order endpoint correction.

Returns (s, e) where s is the Euler-Maclaurin approximation and e is the estimated error (taken to be the magnitude of the first omitted term in the tail):

```
>>> from sympy.abc import k, a, b
>>> from sympy import Sum
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.28333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> from sympy import sstr
>>> print(sstr((s.evalf(), e.evalf()), full_prec=True))
(1.26629073187415, 0.0175000000000000)
```

The endpoints may be symbolic:

```
>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))
```

If the function is a polynomial of degree at most  $2n+1$ , the Euler-Maclaurin formula becomes exact (and  $e = 0$  is returned):

```
>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
>>> Sum(k, (k, 2, b)).doit()
b**2/2 + b/2 - 1
```

With a nonzero eps specified, the summation is ended as soon as the remainder term is less than the epsilon.

```
findrecur(F=-F, n=None)
```

Find a recurrence formula for the summand of the sum.

Given a sum  $f(n) = \sum_k F(n, k)$ , where  $F(n, k)$  is doubly hypergeometric (that's, both  $F(n+1, k)/F(n, k)$  and  $F(n, k+1)/F(n, k)$  are rational functions of  $n$  and  $k$ ), we find a recurrence for the summand  $F(n, k)$  of the form

$$\sum_{i=0}^I \sum_{j=0}^J a_{i,j} F(n-j, k-i) = 0$$

## Notes

We use Sister Celine's algorithm, see [R20] (page 1237).

## References

[R20] (page 1237)

## Examples

```
>>> from sympy import symbols, Function, factorial, oo
>>> from sympy.concrete import Sum
>>> n, k = symbols('n, k', integer=True)
>>> s = Sum(factorial(n)/(factorial(k)*factorial(n - k)), (k, 0, oo))
>>> s.findrecur()
-F(n, k) + F(n - 1, k) + F(n - 1, k - 1)
```

### reverse\_order(\*indices)

Reverse the order of a limit in a Sum.

See also:

`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index` (page 299),  
`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit` (page 300),  
`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder` (page 299)

## References

[R21] (page 1237)

## Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y, a, b, c, d

>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order( 0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order( 1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order( x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order( y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

```
class sympy.concrete.products.Product
    Represents unevaluated products.
```

`Product` represents a finite or infinite product, with the first argument being the general form of terms in the series, and the second argument being (`dummy_variable`, `start`, `end`), with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the product.

See also:

`sympy.concrete.summations.Sum` (page 289), `sympy.concrete.summations.summation` (page 300),  
`sympy.concrete.products.product` (page 301)

## References

[R22] (page 1237), [R23] (page 1237), [R24] (page 1237)

## Examples

```
>>> from sympy.abc import a, b, i, k, m, n, x
>>> from sympy import Product, factorial, oo
>>> Product(k,(k,1,m))
Product(k, (k, 1, m))
>>> Product(k,(k,1,m)).doit()
factorial(m)
>>> Product(k**2,(k,1,m))
Product(k**2, (k, 1, m))
>>> Product(k**2,(k,1,m)).doit()
(factorial(m))**2
```

Wallis' product for pi:

```
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
>>> W
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

Direct computation currently fails:

```
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```
>>> from sympy import limit
>>> W2 = Product(2*i/(2*i-1)*2*i/(2*i+1), (i, 1, n))
>>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
>>> W2e = W2.doit()
>>> W2e
2**(-2*n)*4**n*(factorial(n))**2/(RisingFactorial(1/2, n)*RisingFactorial(3/2, n))
>>> limit(W2e, n, oo)
pi/2
```

By the same formula we can compute  $\sin(\pi/2)$ :

```
>>> from sympy import pi, gamma, simplify
>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
>>> P = P.subs(x, pi/2)
```

```
>>> P
pi**2*Product(1 - pi**2/(4*k**2), (k, 1, n))/2
>>> Pe = P.doit()
>>> Pe
pi**2*RisingFactorial(1 + pi/2, n)*RisingFactorial(-pi/2 + 1, n)/(2*(factorial(n))**2)
>>> Pe = Pe.rewrite(gamma)
>>> Pe
pi**2*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/(2*gamma(1 + pi/2)*gamma(-pi/2 + 1)*gamma(n + 1)**2)
>>> Pe = simplify(Pe)
>>> Pe
sin(pi**2/2)*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/gamma(n + 1)**2
>>> limit(Pe, n, oo)
sin(pi**2/2)
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/i, (i, 6, 1)).doit()
120
>>> Product(i, (i, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those products by interchanging the limits according to the above rules:

```
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**n
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
```

```
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> simplify(P1 * P2)
1
```

```
reverse_order(expr, *indices)
Reverse the order of a limit in a Product.
```

See also:

`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index` (page 299),  
`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit` (page 300),  
`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder` (page 299)

## References

[R25] (page 1237)

## Examples

```
>>> from sympy import Product, simplify, RisingFactorial, gamma, Sum
>>> from sympy.abc import x, y, a, b, c, d
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr)
gamma(b + 1)/gamma(a)
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P)
gamma(b + 1)/gamma(a)
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order( 0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order( 1)
>>> S1
Sum(x*y, (x, b + 1, a - 1), (y, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order( x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order( y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

```
class sympy.concrete.expr_with_limits.ExprWithLimits
```

### as\_dummy()

Replace instances of the given dummy variables with explicit dummy counterparts to make clear what are dummy variables and what are real-world symbols in an object.

**See also:**

`sympy.concrete.expr_with_limits.ExprWithLimits.variables` (page 297) Lists the integration variables

### Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, x, y), (y, x, y)).as_dummy()
Integral(_x, (_x, x, _y), (_y, x, y))
```

If the object supports the “integral at” limit (`x,`) it is not treated as a dummy, but the explicit form, (`x, x`) of length 2 does treat the variable as a dummy.

```
>>> Integral(x, x).as_dummy()
Integral(x, x)
>>> Integral(x, (x, x)).as_dummy()
Integral(_x, (_x, x))
```

If there were no dummies in the original expression, then the the symbols which cannot be changed by `subs()` are clearly seen as those with an underscore prefix.

### free\_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

### Examples

```
>>> from sympy import Sum
>>> from sympy.abc import x, y
>>> Sum(x, (x, y, 1)).free_symbols
set([y])
```

### function

Return the function applied across limits.

**See also:**

`sympy.concrete.expr_with_limits.ExprWithLimits.limits` (page 297),  
`sympy.concrete.expr_with_limits.ExprWithLimits.variables` (page 297),  
`sympy.concrete.expr_with_limits.ExprWithLimits.free_symbols` (page 296)

### Examples

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

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

is_number
Return True if the Sum has no free symbols, else False.

limits
Return the limits of expression.

See also:
sympy.concrete.expr_with_limits.ExprWithLimits.function      (page 296),
sympy.concrete.expr_with_limits.ExprWithLimits.variables    (page 297),
sympy.concrete.expr_with_limits.ExprWithLimits.free_symbols (page 296)
```

### Examples

```
>>> from sympy import Integral
>>> from sympy.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)

variables
Return a list of the dummy variables

>>> from sympy import Sum
>>> from sympy.abc import x, i
>>> Sum(x**i, (i, 1, 3)).variables
[i]

See also:
sympy.concrete.expr_with_limits.ExprWithLimits.function      (page 296),
sympy.concrete.expr_with_limits.ExprWithLimits.limits        (page 297),
sympy.concrete.expr_with_limits.ExprWithLimits.free_symbols (page 296)

sympy.concrete.expr_with_limits.ExprWithLimits.as_dummy (page 296) Rename dummy
variables
```

```
class sympy.concrete.expr_with_intlimits.ExprWithIntLimits
```

```
change_index(var, trafo, newvar=None)
Change index of a Sum or Product.

Perform a linear transformation  $x \mapsto ax + b$  on the index variable  $x$ . For  $a$  the only values allowed are  $\pm 1$ . A new variable to be used after the change of index can also be specified.

See also:
sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index      (page 299),
sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit
(page 300),      sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder
(page 299),      sympy.concrete.summations.Sum.reverse_order     (page 292),
sympy.concrete.products.Product.reverse_order (page 295)
```

## Examples

```
>>> from sympy import Sum, Product, simplify
>>> from sympy.abc import x, y, a, b, c, d, u, v, i, j, k, l

>>> S = Sum(x, (x, a, b))
>>> S.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, x + 1, y)
>>> Sn
Sum(y - 1, (y, a + 1, b + 1))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, -x, y)
>>> Sn
Sum(-y, (y, -b, -a))
>>> Sn.doit()
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, x+u)
>>> Sn
Sum(-u + x, (x, a + u, b + u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2

>>> Sn = S.change_index(x, -x - u, y)
>>> Sn
Sum(-u - y, (y, -b - u, -a - u))
>>> Sn.doit()
-a**2/2 - a*u + a/2 + b**2/2 + b*u + b/2 - u*(-a + b + 1) + u
>>> simplify(Sn.doit())
-a**2/2 + a/2 + b**2/2 + b/2

>>> P = Product(i*j**2, (i, a, b), (j, c, d))
>>> P
Product(i*j**2, (i, a, b), (j, c, d))
>>> P2 = P.change_index(i, i+3, k)
>>> P2
Product(j**2*(k - 3), (k, a + 3, b + 3), (j, c, d))
>>> P3 = P2.change_index(j, -j, l)
>>> P3
Product(l**2*(k - 3), (k, a + 3, b + 3), (l, -d, -c))
```

When dealing with symbols only, we can make a general linear transformation:

```
>>> Sn = S.change_index(x, u*x+v, y)
>>> Sn
Sum((-v + y)/u, (y, b*u + v, a*u + v))
>>> Sn.doit()
-v*(a*u - b*u + 1)/u + (a**2*u**2/2 + a*u*v + a*u/2 - b**2*u**2/2 - b*u*v + b*u/2 + v)/u
>>> simplify(Sn.doit())
a**2*u/2 + a/2 - b**2*u/2 + b/2
```

However, the last result can be inconsistent with usual summation where the index increment is always 1. This is obvious as we get back the original value only for u equal +1 or -1.

**index(expr, x)**

Return the index of a dummy variable in the list of limits.

**See also:**

`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit`  
(page 300), `sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder`  
(page 299), `sympy.concrete.summations.Sum.reverse_order` (page 292),  
`sympy.concrete.products.Product.reverse_order` (page 295)

**Examples**

```
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy import Sum, Product
>>> Sum(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Sum(x*y, (x, a, b), (y, c, d)).index(y)
1
>>> Product(x*y, (x, a, b), (y, c, d)).index(x)
0
>>> Product(x*y, (x, a, b), (y, c, d)).index(y)
1
```

**reorder(expr, \*arg)**

Reorder limits in a expression containing a Sum or a Product.

**See also:**

`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index` (page 299),  
`sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit`  
(page 300), `sympy.concrete.summations.Sum.reverse_order` (page 292),  
`sympy.concrete.products.Product.reverse_order` (page 295)

**Examples**

```
>>> from sympy import Sum, Product
>>> from sympy.abc import x, y, z, a, b, c, d, e, f

>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((x, y))
Sum(x*y, (y, c, d), (x, a, b))

>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder((x, y), (x, z), (y, z))
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))

>>> P = Product(x*y*z, (x, a, b), (y, c, d), (z, e, f))
>>> P.reorder((x, y), (x, z), (y, z))
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

We can also select the index variables by counting them, starting with the inner-most one:

```
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder((0, 1))
Sum(x**2, (x, c, d), (x, a, b))
```

And of course we can mix both schemes:

```
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, x))
Sum(x*y, (y, c, d), (x, a, b))
>>> Sum(x*y, (x, a, b), (y, c, d)).reorder((y, 0))
Sum(x*y, (y, c, d), (x, a, b))
```

`reorder_limit(expr, x, y)`

Interchange two limit tuples of a Sum or Product expression.

See also:

<code>sympy.concrete.expr_with_intlimits.ExprWithIntLimits.index</code>	(page 299),
<code>sympy.concrete.expr_with_intlimits.ExprWithIntLimits.reorder</code>	(page 299),
<code>sympy.concrete.summations.Sum.reverse_order</code>	(page 292),
<code>sympy.concrete.products.Product.reverse_order</code>	(page 295)

### Examples

```
>>> from sympy.abc import x, y, z, a, b, c, d, e, f
>>> from sympy import Sum, Product

>>> Sum(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Sum(x*y*z, (z, e, f), (y, c, d), (x, a, b))
>>> Sum(x**2, (x, a, b), (x, c, d)).reorder_limit(1, 0)
Sum(x**2, (x, c, d), (x, a, b))

>>> Product(x*y*z, (x, a, b), (y, c, d), (z, e, f)).reorder_limit(0, 2)
Product(x*y*z, (z, e, f), (y, c, d), (x, a, b))
```

### 3.5.3 Concrete Functions Reference

`sympy.concrete.summations.summation(f, *symbols, **kwargs)`

Compute the summation of  $f$  with respect to symbols.

The notation for symbols is similar to the notation used in Integral. `summation(f, (i, a, b))` computes the sum of  $f$  with respect to  $i$  from  $a$  to  $b$ , i.e.,

$$\sum_{i=a}^b f$$

If it cannot compute the sum, it returns an unevaluated `Sum` object. Repeated sums can be computed by introducing additional symbols tuples:

```
>>> from sympy import summation, oo, symbols, log
>>> i, n, m = symbols('i n m', integer=True)

>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
```

```
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3

>>> from sympy.abc import x
>>> from sympy import factorial
>>> summation(x**n/factorial(n), (n, 0, oo))
exp(x)
```

**See also:**

`sympy.concrete.summations.Sum` (page 289), `sympy.concrete.products.Product` (page 292),  
`sympy.concrete.products.product` (page 301)

`sympy.concrete.products.product(*args, **kwargs)`

Compute the product.

The notation for symbols is similar to the notation used in Sum or Integral. `product(f, (i, a, b))` computes the product of f with respect to i from a to b, i.e.,

$$\text{product}(f(n), (i, a, b)) = \prod_{i=a}^b f(n)$$

If it cannot compute the product, it returns an unevaluated Product object. Repeated products can be computed by introducing additional symbols tuples:

```
>>> from sympy import product, symbols
>>> i, n, m, k = symbols('i n m k', integer=True)

>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

`sympy.concrete.gosper.gosper_normal(f, g, n, polys=True)`

Compute the Gosper's normal form of f and g.

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

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

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

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

$$2. \gcd(B(n), C(n+1)) = 1$$

$$3. \gcd(A(n), C(n)) = 1$$

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

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

## Examples

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

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

`sympy.concrete.gosper.gosper_term(f, n)`  
Compute Gosper's hypergeometric term for  $f$ .

Suppose  $f$  is a hypergeometric term such that:

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

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

## Examples

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

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

`sympy.concrete.gosper.gosper_sum(f, k)`  
Gosper's hypergeometric summation algorithm.

Given a hypergeometric term  $f$  such that:

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

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

## References

[R26] (page 1237)

## Examples

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

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

---

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

## 3.6 Numerical evaluation

### 3.6.1 Basics

Exact SymPy expressions can be converted to floating-point approximations (decimal numbers) using either the `.evalf()` method or the `N()` function. `N(expr, <args>)` is equivalent to `sympify(expr).evalf(<args>)`.

```
>>> from sympy import *
>>> N(sqrt(2)*pi)
4.44288293815837
>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to `evalf` or `N`:

```
>>> N(sqrt(2)*pi, 5)
4.4429
>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:

```
>>> N(1/(pi + I), 20)
0.2890254822223624241 - 0.091999668350375232456*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.33333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert SymPy expressions to regular Python numbers:

```
>>> float(pi)
3.1415926535...
>>> complex(pi+E*I)
(3.1415926535...+2.7182818284...j)
```

If these functions are used, failure to evaluate the expression to an explicit number (for example if the expression contains symbols) will raise an exception.

There is essentially no upper precision limit. The following command, for example, computes the first 100,000 digits of  $\pi/e$ :

```
>>> N(pi/E, 100000)
...

```

This shows digits 999,951 through 1,000,000 of pi:

```
>>> str(N(pi, 10**6))[-50:]
'95678796130331164628399634646042209010610577945815'
```

High-precision calculations can be slow. It is recommended (but entirely optional) to install gmpy (<http://code.google.com/p/gmpy/>), which will significantly speed up computations such as the one above.

### 3.6.2 Floating-point numbers

Floating-point numbers in SymPy are instances of the class `Float`. A `Float` can be created with a custom precision as second argument:

```
>>> Float(0.1)
0.100000000000000
>>> Float(0.1, 10)
0.1000000000
>>> Float(0.125, 30)
0.125000000000000000000000000000000000000000000000000000000000000
>>> Float(0.1, 30)
0.100000000000000005551115123126
```

As the last example shows, some Python floats are only accurate to about 15 digits as inputs, while others (those that have a denominator that is a power of 2, like  $.125 = 1/4$ ) are exact. To create a `Float` from a high-precision decimal number, it is better to pass a string, `Rational`, or `evalf` a `Rational`:

```
>>> Float('0.1', 30)
0.10000000000000000000000000000000000000000000000000000000000000
>>> Float(Rational(1, 10), 30)
0.10000000000000000000000000000000000000000000000000000000000000
>>> Rational(1, 10).evalf(30)
0.10000000000000000000000000000000000000000000000000000000000000
```

The precision of a number determines 1) the precision to use when performing arithmetic with the number, and 2) the number of digits to display when printing the number. When two numbers with different precision are used together in an arithmetic operation, the higher of the precisions is used for the result. The product of  $0.1 \pm 0.001$  and  $3.1415 \pm 0.0001$  has an uncertainty of about 0.003 and yet 5 digits of precision are shown.

```
>>> Float(0.1, 3)*Float(3.1415, 5)
0.31417
```

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

`N` and `evalf` can be used to change the precision of existing floating-point numbers:

```
>>> N(3.5)
3.500000000000000
>>> N(3.5, 5)
3.5000
>>> N(3.5, 30)
3.50000000000000000000000000000000000000000000000000000000000000
```

### 3.6.3 Accuracy and error handling

When the input to `N` or `evalf` is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation

$\varphi^{100}/\sqrt{5}$  where  $\varphi$  is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

```
>>> a, b = GoldenRatio**1000/sqrt(5), fibonacci(1000)
>>> float(a)
4.34665576869e+208
>>> float(b)
4.34665576869e+208
>>> float(a) - float(b)
0.0
```

`N` and `evalf` keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

```
>>> N(fibonacci(100) - GoldenRatio**100/sqrt(5))
-5.64613129282185e-22
```

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000'th Fibonacci number, the following happens:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85
```

The lack of digits in the returned number indicates that `N` failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than  $10^{84}$ , but that is not a particularly good answer. To force a higher working precision, the `maxn` keyword argument can be used:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, `maxn` can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the `strict=True` option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:

-sqrt(5)*GoldenRatio**1000/5 + 434665576869374564356885276750406258025646605173717804024817290895365554179490518904
```

from zero. Try simplifying the input, using `chop=True`, or providing a higher `maxn` for `evalf`

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet's formula), we get an expression that is exactly zero, but `N` does not know this:

```
>>> f = fibonacci(100) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the `chop` options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.000000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the `round` method are useful:

```
>>> Float('.1', '')*Float('.12345', '')
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the `round` method is useful:

```
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

### 3.6.4 Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf()
1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf()
1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval discontinuities. In many cases, `evalf/N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf()
2.346
```

It is better to split this integral into two pieces:

```
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf()
2.34635637913639
```

A similar example is the following oscillatory integral:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20)
0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor  $\cos(ax+b)$  or  $\sin(ax+b)$ . Note that many other oscillatory integrals can be transformed to this form with a change of variables:

```
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> intgrl = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgrl
oo
/
|
| sin(x)
| -----
| 2
| x
|
/
1
>>> N(intgrl, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.57721566490153286060651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.57721566490153286060651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```
>>> Sum(1/k, (k, 10000000, 20000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for pi can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> n = Symbol('n', integer=True)
>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**(4*n),
... (n, 0, oo))
```

```
>>> N(R, 10000)
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
99862803482534211706798214808651328230664709384460955058223172535940812848111745
02841027019385211055596446229489549303819644288109756659334461284756482337867831
...
```

### 3.6.5 Numerical simplification

The function `nsimplify` attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by `nsimplify` is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, `nsimplify` can be passed a list of constants to include (e.g. `pi`) and a minimum numerical tolerance. Here are some elementary examples:

```
>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
---
113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> nsimplify(2.0** (1/3.), tolerance=0.001)
635
---
504
>>> nsimplify(2.0** (1/3.), tolerance=0.001, full=True)
3 ---
\ / 2
```

Here are several more advanced examples:

```
>>> nsimplify(Float('0.130198866629986772369127970337', 30), [pi, E])
1
-----
5*pi
----- + 2*E
7
>>> nsimplify(cos(atan('1/3'))))
-----
3*\ / 10
-----
10
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49     8*I
-- + ---
17     17
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
```

```

1      /   ___
-- I* /   ----- + -
2      \V      10    4
>>> nsimplify(I**I, [pi])
-pi
-----
2
e
>>> n = Symbol('n')
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
2
pi
---
6
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])
---_
\V 2 *pi

```

## 3.7 Numeric Computation

Symbolic computer algebra systems like SymPy facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately SymPy offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in SymPy and then ship them off to the numeric system of your choice. This page documents many of the options available including the `math` library, the popular array computing package `numpy`, code generation in `Fortran` or `C`, and the use of the array compiler `Theano`.

### 3.7.1 Subs/evalf

`Subs` is the slowest but simplest option. It runs at SymPy speeds. The `.subs(...).evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within SymPy.

```

>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640

```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

### 3.7.2 Lambdify

The `lambdify` function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```

>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x

```

```
>>> f = lambdify(x, expr)
>>> f(3.14)
0.000507214304614
```

Here `lambdify` makes a function that computes  $f(x) = \sin(x)/x$ . By default `lambdify` relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between SymPy and raw Python.

`Lambdify` can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")

>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> f(data)
[ 0.84147098  0.84119981  0.84092844 ..., -0.05426074 -0.05433146
 -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nano-seconds per element. Unfortunately `numpy` incurs some start-up time and introduces an overhead of a few microseconds.

### 3.7.3 uFuncify

While NumPy operations are very efficient for vectorized data they sometimes incur unnecessary costs when chained together. Consider the following operation

```
>>> x = get_numpy_array(...)
>>> y = sin(x) / x
```

The operators `sin` and `/` call routines that execute tight for loops in C. The resulting computation looks something like this

```
for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = i; i < n; i++)
{
    y[i] = temp[i] / x[i];
}
```

This is slightly sub-optimal because

1. We allocate an extra `temp` array
2. We walk over `x` memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, SymPy is able to generate efficient low-level C or Fortran code. It can then depend on projects like Cython or f2py to compile and reconnect that code back up to Python. Fortunately this process is well automated and a SymPy user wishing to make use of this code generation should call the `ufuncify` function

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x

>>> from sympy.utilities.autowrap import ufuncify
>>> f = ufuncify([x], expr)
```

This function `f` consumes and returns a NumPy array. Generally `ufuncify` performs at least as well as `lambdify`. If the expression is complicated then `ufuncify` often significantly outperforms the NumPy backed solution. Jensen has a good [blog post](#) on this topic.

### 3.7.4 Theano

SymPy has a strong connection with [Theano](#), a mathematical array compiler. SymPy expressions can be easily translated to Theano graphs and then compiled using the Theano compiler chain.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x

>>> from sympy.printing.theanocode import theano_function
>>> f = theano_function([x], [expr])
```

If array broadcasting or types are desired then Theano requires this extra information

```
>>> f = theano_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Theano has a more sophisticated code generation system than SymPy's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Theano also supports SymPy Matrix and Matrix Expression objects.

### 3.7.5 So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Theano installed then that will often be the best choice. If you don't have Theano but do have f2py then you should use `ufuncify`.

Tool	Speed	Qualities	Dependencies
subs/evalf	50us	Simple	None
lambdify	1us	Scalar functions	math
lambdify-numpy	10ns	Vector functions	numpy
ufuncify	10ns	Complex vector expressions	f2py, Cython
Theano	10ns	Many outputs, CSE, GPUs	Theano

## 3.8 Functions Module

All functions support the methods documented below, inherited from `sympy.core.function.Function` (page 140).

```
class sympy.core.function.Function
    Base class for applied mathematical functions.

    It also serves as a constructor for undefined function classes.
```

### Examples

First example shows how to use Function as a constructor for undefined function classes:

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

In the following example Function is used as a base class for `my_func` that represents a mathematical function *my\_func*. Suppose that it is well known, that *my\_func(0)* is 1 and *my\_func* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *my\_func(x)* is real exactly when *x* is real. Here is an implementation that honours those requirements:

```
>>> from sympy import Function, S, oo, I, sin
>>> class my_func(Function):
...     ...
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...
...     def _eval_is_extended_real(self):
...         return self.args[0].is_extended_real
...
>>> x = S('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_extended_real
False
```

In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```
>>> class my_func(Function):
...     nargs = (1, 2)
...
>>>
as_base_exp()
    Returns the method as the 2-tuple (base, exponent).
fdiff(argindex=1)
    Returns the first derivative of the function.
is_commutative
    Returns whether the function is commutative.
```

### 3.8.1 Contents

#### Elementary

This module implements elementary functions, as well as functions like `Abs`, `Max`, etc.

##### `Abs`

Returns the absolute value of the argument.

Examples:

```
>>> from sympy.functions import Abs
>>> Abs(-1)
1
```

```
class sympy.functions.elementary.complexes.Abs
    Return the absolute value of the argument.
```

This is an extension of the built-in function `abs()` to accept symbolic values. If you pass a SymPy expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

See also:

`sympy.functions.elementary.complexes.sign` (page 339), `sympy.functions.elementary.complexes.conjugate` (page 322)

#### Examples

```
>>> from sympy import Abs, Symbol, S
>>> Abs(-1)
1
>>> x = Symbol('x', extended_real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
```

Note that the Python built-in will return either an `Expr` or `int` depending on the argument:

```
>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'sympy.core.numbers.One'>
```

Abs will always return a sympy object.

`fdiff(argindex=1)`  
Get the first derivative of the argument to Abs().

### Examples

```
>>> from sympy.abc import x
>>> from sympy.functions import Abs
>>> Abs(-x).fdiff()
sign(x)
```

## acos

`class sympy.functions.elementary.trigonometric.acos`  
The inverse cosine function.

Returns the arc cosine of x (measured in radians).

See also:

`sympy.functions.elementary.trigonometric.sin` (page 335), `sympy.functions.elementary.trigonometric.csc` (page 324), `sympy.functions.elementary.trigonometric.cos` (page 322),  
`sympy.functions.elementary.trigonometric.sec` (page 337), `sympy.functions.elementary.trigonometric.tan` (page 339), `sympy.functions.elementary.trigonometric.cot` (page 323),  
`sympy.functions.elementary.trigonometric.asin` (page 317), `sympy.functions.elementary.trigonometric.acs` (page 316), `sympy.functions.elementary.trigonometric.asec` (page 318),  
`sympy.functions.elementary.trigonometric.atan` (page 318), `sympy.functions.elementary.trigonometric.acc` (page 315), `sympy.functions.elementary.trigonometric.atan2` (page 319)

### Notes

`acos(x)` will evaluate automatically in the cases oo, -oo, 0, 1, -1.

### References

[R100] (page 1237), [R101] (page 1237), [R102] (page 1237)

### Examples

```
>>> from sympy import acos, oo, pi
>>> acos(1)
0
>>> acos(0)
pi/2
>>> acos(oo)
oo*I
```

**inverse(argindex=1)**

Returns the inverse of this function.

**acosh****class sympy.functions.elementary.hyperbolic.acosh**

The inverse hyperbolic cosine function.

- `acosh(x)` -> Returns the inverse hyperbolic cosine of  $x$

See also:

[sympy.functions.elementary.hyperbolic.asinh](#) (page 317), [sympy.functions.elementary.hyperbolic.atanh](#) (page 321), [sympy.functions.elementary.hyperbolic.cosh](#) (page 323)

**inverse(argindex=1)**

Returns the inverse of this function.

**acot****class sympy.functions.elementary.trigonometric.acot**

The inverse cotangent function.

Returns the arc cotangent of  $x$  (measured in radians).

See also:

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acs](#) (page 316), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318), [sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

**References**

[R103] (page 1237), [R104] (page 1237), [R105] (page 1237)

**inverse(argindex=1)**

Returns the inverse of this function.

**acoth****class sympy.functions.elementary.hyperbolic.acoth**

The inverse hyperbolic cotangent function.

- `acoth(x)` -> Returns the inverse hyperbolic cotangent of  $x$

**inverse(argindex=1)**

Returns the inverse of this function.

## acsc

```
class sympy.functions.elementary.trigonometric.acsc
```

The inverse cosecant function.

Returns the arc cosecant of x (measured in radians).

See also:

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318), [sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.acot](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

## Notes

`acsc(x)` will evaluate automatically in the cases  $\infty$ ,  $-\infty$ , 0, 1, -1.

## References

[R106] (page 1237), [R107] (page 1237), [R108] (page 1237)

## Examples

```
>>> from sympy import acsc, oo, pi
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
```

`inverse(argindex=1)`

Returns the inverse of this function.

## arg

Returns the argument (in radians) of a complex number. For a real number, the argument is always 0.

Examples:

```
>>> from sympy.functions import arg
>>> from sympy import I, sqrt
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
```

```
class sympy.functions.elementary.complexes.arg
```

Returns the argument (in radians) of a complex number

**asin**

```
class sympy.functions.elementary.trigonometric.asin
    The inverse sine function.
```

Returns the arcsine of x in radians.

**See also:**

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.acsc](#) (page 316), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318), [sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.acot](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

**Notes**

`asin(x)` will evaluate automatically in the cases  $\infty$ ,  $-\infty$ , 0, 1, -1 and for some instances when the result is a rational multiple of  $\pi$  (see the `eval` class method).

**References**

[R109] (page 1237), [R110] (page 1237), [R111] (page 1237)

**Examples**

```
>>> from sympy import asin, oo, pi
>>> asin(1)
pi/2
>>> asin(-1)
-pi/2

inverse(argindex=1)
    Returns the inverse of this function.
```

**asinh**

```
class sympy.functions.elementary.hyperbolic.asinh
    The inverse hyperbolic sine function.
```

•`asinh(x)` -> Returns the inverse hyperbolic sine of x

**See also:**

[sympy.functions.elementary.hyperbolic.cosh](#) (page 323), [sympy.functions.elementary.hyperbolic.tanh](#) (page 340), [sympy.functions.elementary.hyperbolic.sinh](#) (page 336)

```
inverse(argindex=1)
    Returns the inverse of this function.
```

**asec**

```
class sympy.functions.elementary.trigonometric.asec  
The inverse secant function.
```

Returns the arc secant of x (measured in radians).

**See also:**

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acs](#) (page 316), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.acc](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

**Notes**

`asec(x)` will evaluate automatically in the cases  $\infty$ ,  $-\infty$ , 0, 1, -1.

**References**

[R112] (page 1237), [R113] (page 1237), [R114] (page 1237)

**Examples**

```
>>> from sympy import asec, oo, pi  
>>> asec(1)  
0  
>>> asec(-1)  
pi  
  
inverse(argindex=1)
```

Returns the inverse of this function.

**atan**

```
class sympy.functions.elementary.trigonometric.atan  
The inverse tangent function.
```

Returns the arc tangent of x (measured in radians).

**See also:**

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acs](#) (page 316), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318), [sympy.functions.elementary.trigonometric.acc](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

**Notes**

`atan(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1`.

**References**

[R115] (page 1237), [R116] (page 1237), [R117] (page 1237)

**Examples**

```
>>> from sympy import atan, oo, pi
>>> atan(0)
0
>>> atan(1)
pi/4
>>> atan(oo)
pi/2

inverse(argindex=1)
```

Returns the inverse of this function.

**atan2**

This function is like `atan`, but considers the sign of both arguments in order to correctly determine the quadrant of its result.

`class sympy.functions.elementary.trigonometric.atan2`

The function `atan2(y, x)` computes  $\text{atan}(y/x)$  taking two arguments  $y$  and  $x$ . Signs of both  $y$  and  $x$  are considered to determine the appropriate quadrant of  $\text{atan}(y/x)$ . The range is  $(-\pi, \pi]$ . The complete definition reads as follows:

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The  $y$ -coordinate is the first argument and the  $x$ -coordinate the second.

**See also:**

`sympy.functions.elementary.trigonometric.sin` (page 335), `sympy.functions.elementary.trigonometric.csc` (page 324), `sympy.functions.elementary.trigonometric.cos` (page 322), `sympy.functions.elementary.trigonometric.sec` (page 337), `sympy.functions.elementary.trigonometric.tan` (page 339), `sympy.functions.elementary.trigonometric.cot` (page 323), `sympy.functions.elementary.trigonometric.asin` (page 317), `sympy.functions.elementary.trigonometric.acs` (page 316), `sympy.functions.elementary.trigonometric.acos` (page 314), `sympy.functions.elementary.trigonometric.asec` (page 318), `sympy.functions.elementary.trigonometric.atan` (page 318), `sympy.functions.elementary.trigonometric.acot` (page 315)

## References

[R118] (page 1237), [R119] (page 1237), [R120] (page 1237)

## Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> from sympy import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary atan function for the point  $(x, y) = (-1, 1)$

```
>>> from sympy import atan, S
>>> atan(S(1) / -1)
-pi/4
>>> atan2(1, -1)
3*pi/4
```

where only the atan2 function returns what we expect. We can differentiate the function with respect to both arguments:

```
>>> from sympy import diff
>>> from sympy.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)

>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the atan2 function in terms of complex logarithms:

```
>>> from sympy import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of (*atan*):

```
>>> from sympy import atan
>>> atan2(y, x).rewrite(atan)
2*atan(y/(x + sqrt(x**2 + y**2)))
```

but note that this form is undefined on the negative real axis.

### atanh

```
class sympy.functions.elementary.hyperbolic.atanh
```

The inverse hyperbolic tangent function.

- `atanh(x)` -> Returns the inverse hyperbolic tangent of  $x$

See also:

`sympy.functions.elementary.hyperbolic.asinh` (page 317), `sympy.functions.elementary.hyperbolic.acosh` (page 315), `sympy.functions.elementary.hyperbolic.tanh` (page 340)

`inverse(argindex=1)`

Returns the inverse of this function.

### ceiling

```
class sympy.functions.elementary.integers.ceiling
```

Ceiling is a univariate function which returns the smallest integer value not less than its argument.

Ceiling function is generalized in this implementation to complex numbers.

More information can be found in “Concrete mathematics” by Graham, pp. 87 or visit <http://mathworld.wolfram.com/CeilingFunction.html>.

```
>>> from sympy import ceiling, E, I, Float, Rational
>>> ceiling(17)
17
>>> ceiling(Rational(23, 10))
3
>>> ceiling(2*E)
6
>>> ceiling(-Float(0.567))
0
>>> ceiling(I/2)
I
```

See also:

`sympy.functions.elementary.integers.floor` (page 326)

### conjugate

Returns the complex conjugate of an argument. In mathematics, the complex conjugate of a complex number is given by changing the sign of the imaginary part. Thus, the conjugate of the complex number

$$a + ib$$

(where  $a$  and  $b$  are real numbers) is

$$a - ib$$

Examples:

```
>>> from sympy.functions import conjugate
>>> from sympy import I
>>> conjugate(2)
```

```
2
>>> conjugate(I)
-I
```

**class** `sympy.functions.elementary.complexes.conjugate`  
Changes the sign of the imaginary part of a complex number.

**See also:**

`sympy.functions.elementary.complexes.sign` (page 339), `sympy.functions.elementary.complexes.Abs` (page 313)

### Examples

```
>>> from sympy import conjugate, I

>>> conjugate(1 + I)
1 - I
```

**cos**

**class** `sympy.functions.elementary.trigonometric.cos`  
The cosine function.

Returns the cosine of x (measured in radians).

**See also:**

`sympy.functions.elementary.trigonometric.sin` (page 335), `sympy.functions.elementary.trigonometric.csc` (page 324), `sympy.functions.elementary.trigonometric.sec` (page 337),  
`sympy.functions.elementary.trigonometric.tan` (page 339), `sympy.functions.elementary.trigonometric.cot` (page 323), `sympy.functions.elementary.trigonometric.asin` (page 317),  
`sympy.functions.elementary.trigonometric.acsc` (page 316), `sympy.functions.elementary.trigonometric.acosec` (page 314),  
`sympy.functions.elementary.trigonometric.atan` (page 318), `sympy.functions.elementary.trigonometric.acot` (page 315), `sympy.functions.elementary.trigonometric.atan2` (page 319)

### Notes

See `sin()` (page 335) for notes about automatic evaluation.

### References

[R121] (page 1238), [R122] (page 1238), [R123] (page 1238)

### Examples

```
>>> from sympy import cos, pi
>>> from sympy.abc import x
>>> cos(x**2).diff(x)
-2*x*sin(x**2)
>>> cos(1).diff(x)
```

```
0
>>> cos(pi)
-1
>>> cos(pi/2)
0
>>> cos(2*pi/3)
-1/2
>>> cos(pi/12)
sqrt(2)/4 + sqrt(6)/4
```

### cosh

`class sympy.functions.elementary.hyperbolic.cosh`

The hyperbolic cosine function,  $\frac{e^x + e^{-x}}{2}$ .

•`cosh(x)` -> Returns the hyperbolic cosine of x

See also:

[sympy.functions.elementary.hyperbolic.sinh](#) (page 336), [sympy.functions.elementary.hyperbolic.tanh](#) (page 340), [sympy.functions.elementary.hyperbolic.acosh](#) (page 315)

### cot

`class sympy.functions.elementary.trigonometric.cot`

The cotangent function.

Returns the cotangent of x (measured in radians).

See also:

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acsc](#) (page 316), [sympy.functions.elementary.trigonometric.acosec](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318), [sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.acot](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

### Notes

See `sin()` (page 335) for notes about automatic evaluation.

### References

[R124] (page 1238), [R125] (page 1238), [R126] (page 1238)

### Examples

```
>>> from sympy import cot, pi
>>> from sympy.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
>>> cot(1).diff(x)
0
>>> cot(pi/12)
sqrt(3) + 2
```

**inverse(argindex=1)**

Returns the inverse of this function.

## coth

```
class sympy.functions.elementary.hyperbolic.coth
The hyperbolic cotangent function,  $\frac{\cosh(x)}{\sinh(x)}$ .
```

•`coth(x)` -> Returns the hyperbolic cotangent of x

**inverse(argindex=1)**

Returns the inverse of this function.

## csc

```
class sympy.functions.elementary.trigonometric.csc
The cosecant function.
```

Returns the cosecant of x (measured in radians).

**See also:**

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337),  
[sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317),  
[sympy.functions.elementary.trigonometric.acsc](#) (page 316), [sympy.functions.elementary.trigonometric.acot](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318),  
[sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

## Notes

See `sin()` (page 335) for notes about automatic evaluation.

## References

[R127] (page 1238), [R128] (page 1238), [R129] (page 1238)

## Examples

```
>>> from sympy import csc
>>> from sympy.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
>>> csc(1).diff(x)
0
```

### exp

class `sympy.functions.elementary.exponential.exp`

The exponential function,  $e^x$ .

See also:

`sympy.functions.elementary.exponential.log` (page 329)

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

Returns this function as a 2-tuple representing a complex number.

See also:

`sympy.functions.elementary.complexes.re` (page 333), `sympy.functions.elementary.complexes.im` (page 327)

### Examples

```
>>> from sympy import I
>>> from sympy.abc import x
>>> from sympy.functions import exp
>>> exp(x).as_real_imag()
(exp(re(x))*cos(im(x)), exp(re(x))*sin(im(x)))
>>> exp(1).as_real_imag()
(E, 0)
>>> exp(I).as_real_imag()
(cos(1), sin(1))
>>> exp(1+I).as_real_imag()
(E*cos(1), E*sin(1))
```

### base

Returns the base of the exponential function.

`fdiff(argindex=1)`

Returns the first derivative of this function.

`static taylor_term(*args, **kwargs)`

Calculates the next term in the Taylor series expansion.

See also:

classes `sympy.functions.elementary.exponential.log` (page 329)

### exp\_polar

class `sympy.functions.elementary.exponential.exp_polar`

Represent a ‘polar number’ (see g-function Sphinx documentation).

`exp_polar` represents the function  $Exp : \mathbb{C} \rightarrow \mathcal{S}$ , sending the complex number  $z = a + bi$  to the polar number  $r = \exp(a), \theta = b$ . It is one of the main functions to construct polar numbers.

```
>>> from sympy import exp_polar, pi, I, exp
```

The main difference is that polar numbers don't "wrap around" at  $2\pi$ :

```
>>> exp(2*pi*I)
1
>>> exp_polar(2*pi*I)
exp_polar(2*I*pi)
```

apart from that they behave mostly like classical complex numbers:

```
>>> exp_polar(2)*exp_polar(3)
exp_polar(5)
```

See also:

`sympy.simplify.simplify.powsimp` (page 925), `sympy.functions.elementary.complexes.polar_lift` (page 340), `sympy.functions.elementary.complexes.periodic_argument` (page 341), `sympy.functions.elementary.complexes.principal_branch` (page 341)

## ExprCondPair

```
class sympy.functions.elementary.piecewise.ExprCondPair
```

Represents an expression, condition pair.

`cond`

Returns the condition of this pair.

`expr`

Returns the expression of this pair.

`free_symbols`

Return the free symbols of this pair.

## floor

```
class sympy.functions.elementary.integers.floor
```

Floor is a univariate function which returns the largest integer value not greater than its argument. However this implementation generalizes floor to complex numbers.

More information can be found in "Concrete mathematics" by Graham, pp. 87 or visit <http://mathworld.wolfram.com/FloorFunction.html>.

```
>>> from sympy import floor, E, I, Float, Rational
>>> floor(17)
17
>>> floor(Rational(23, 10))
2
>>> floor(2*E)
5
>>> floor(-Float(0.567))
-1
>>> floor(-I/2)
-I
```

See also:

`sympy.functions.elementary.integers.ceiling` (page 321)

## HyperbolicFunction

`class sympy.functions.elementary.hyperbolic.HyperbolicFunction`  
Base class for hyperbolic functions.

See also:

`sympy.functions.elementary.hyperbolic.sinh` (page 336), `sympy.functions.elementary.hyperbolic.cosh` (page 323), `sympy.functions.elementary.hyperbolic.tanh` (page 340),  
`sympy.functions.elementary.hyperbolic.coth` (page 324)

## IdentityFunction

`class sympy.functions.elementary.miscellaneous.IdentityFunction`  
The identity function

### Examples

```
>>> from sympy import Id, Symbol  
>>> x = Symbol('x')  
>>> Id(x)  
x
```

## im

Returns the imaginary part of an expression.

Examples:

```
>>> from sympy.functions import im  
>>> from sympy import I  
>>> im(2+3*I)  
3
```

`class sympy.functions.elementary.complexes.im`

Returns imaginary part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

See also:

`sympy.functions.elementary.complexes.re` (page 333)

### Examples

```
>>> from sympy import re, im, E, I  
>>> from sympy.abc import x, y
```

```
>>> im(2*I)
0

>>> re(2*I + 17)
17

>>> im(x*I)
re(x)

>>> im(re(x) + y)
im(y)

as_real_imag(deep=True, **hints)
Return the imaginary part with a zero real part.
```

### Examples

```
>>> from sympy.functions import im
>>> from sympy import I
>>> im(2 + 3*I).as_real_imag()
(3, 0)
```

### See also:

[sympy.functions.elementary.complexes.re](#) (page 333)

## LambertW

`class sympy.functions.elementary.exponential.LambertW`

The Lambert W function  $W(z)$  is defined as the inverse function of  $w \exp(w)$  [R130] (page 1238).

In other words, the value of  $W(z)$  is such that  $z = W(z) \exp(W(z))$  for any complex number  $z$ . The Lambert W function is a multivalued function with infinitely many branches  $W_k(z)$ , indexed by  $k \in \mathbb{Z}$ . Each branch gives a different solution  $w$  of the equation  $z = w \exp(w)$ .

The Lambert W function has two partially real branches: the principal branch ( $k = 0$ ) is real for real  $z > -1/e$ , and the  $k = -1$  branch is real for  $-1/e < z < 0$ . All branches except  $k = 0$  have a logarithmic singularity at  $z = 0$ .

### References

[R130] (page 1238)

### Examples

```
>>> from sympy import LambertW
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_extended_real
False
```

**fdiff(argindex=1)**

Return the first derivative of this function.

**log****class sympy.functions.elementary.exponential.log**

The natural logarithm function  $\ln(x)$  or  $\log(x)$ . Logarithms are taken with the natural base,  $e$ . To get a logarithm of a different base  $b$ , use  $\log(x, b)$ , which is essentially short-hand for  $\log(x)/\log(b)$ .

**See also:**

[sympy.functions.elementary.exponential.exp](#) (page 325)

**as\_base\_exp()**

Returns this function in the form (base, exponent).

**as\_real\_imag(deep=True, \*\*hints)**

Returns this function as a complex coordinate.

**Examples**

```
>>> from sympy import I
>>> from sympy.abc import x
>>> from sympy.functions import log
>>> log(x).as_real_imag()
(log(Abs(x)), arg(x))
>>> log(I).as_real_imag()
(0, pi/2)
>>> log(1 + I).as_real_imag()
(log(sqrt(2)), pi/4)
>>> log(I*x).as_real_imag()
(log(Abs(x)), arg(I*x))
```

**fdiff(argindex=1)**

Returns the first derivative of the function.

**inverse(argindex=1)**

Returns  $e^x$ , the inverse function of  $\log(x)$ .

**static taylor\_term(\*args, \*\*kwargs)**

Returns the next term in the Taylor series expansion of  $\log(1 + x)$ .

**See also:**

classes [sympy.functions.elementary.exponential.exp](#) (page 325)

**Min**

Returns the minimum of two (comparable) expressions.

Examples:

```
>>> from sympy.functions import Min
>>> Min(1,2)
1
>>> from sympy.abc import x
```

```
>>> Min(1, x)
Min(1, x)
```

It is named `Min` and not `min` to avoid conflicts with the built-in function `min`.

```
class sympy.functions.elementary.miscellaneous.Min
    Return, if possible, the minimum value of the list.
```

See also:

`sympy.functions.elementary.miscellaneous.Max` ([page 330](#)) find maximum values

### Examples

```
>>> from sympy import Min, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)

>>> Min(x, -2)
Min(x, -2)

>>> Min(x, -2).subs(x, 3)
-2

>>> Min(p, -3)
-3

>>> Min(x, y)
Min(x, y)

>>> Min(n, 8, p, -7, p, oo)
Min(n, -7)
```

### Max

Returns the maximum of two (comparable) expressions

It is named `Max` and not `max` to avoid conflicts with the built-in function `max`.

```
class sympy.functions.elementary.miscellaneous.Max
    Return, if possible, the maximum value of the list.
```

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is  $\geq$  the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

See also:

`sympy.functions.elementary.miscellaneous.Min` ([page 330](#)) find minimum values

## References

[R131] ([page 1238](#)), [R132] ([page 1238](#))

## Examples

```
>>> from sympy import Max, Symbol, oo
>>> from sympy.abc import x, y
>>> p = Symbol('p', positive=True)
>>> n = Symbol('n', negative=True)

>>> Max(x, -2)
Max(x, -2)

>>> Max(x, -2).subs(x, 3)
3

>>> Max(p, -2)
p

>>> Max(x, y)
Max(x, y)

>>> Max(x, y) == Max(y, x)
True

>>> Max(x, Max(y, z))
Max(x, y, z)

>>> Max(n, 8, p, 7, -oo)
Max(8, p)

>>> Max(1, x, oo)
oo
```

### Algorithm

The task can be considered as searching of supremums in the directed complete partial orders [R131] ([page 1238](#)).

The source values are sequentially allocated by the isolated subsets in which supremums are searched and result as Max arguments.

If the resulted supremum is single, then it is returned.

The isolated subsets are the sets of values which are only the comparable with each other in the current set. E.g. natural numbers are comparable with each other, but not comparable with the  $x$  symbol. Another example: the symbol  $x$  with negative assumption is comparable with a natural number.

Also there are “least” elements, which are comparable with all others, and have a zero property (maximum or minimum for all elements). E.g.  $\infty$ . In case of it the allocation operation is terminated and only this value is returned.

**Assumption:**

- if  $A > B > C$  then  $A > C$
- if  $A == B$  then  $B$  can be removed

## Piecewise

```
class sympy.functions.elementary.piecewise.Piecewise
    Represents a piecewise function.
```

Usage:

```
Piecewise( (expr,cond), (expr,cond), ... )
```

- Each argument is a 2-tuple defining an expression and condition
- Theconds are evaluated in turn returning the first that is True. If any of the evaluatedconds are not determined explicitly False, e.g.  $x < 1$ , the function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, a ValueError exception will be raised.
- Pairs where the cond is explicitly False, will be removed.

See also:

[sympy.functions.elementary.piecewise.piecewise\\_fold](#) (page 332)

## Examples

```
>>> from sympy import Piecewise, log
>>> from sympy.abc import x
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise( (0, x<-1), (f, x<=1), (g, True))
>>> p.subs(x,1)
1
>>> p.subs(x,5)
log(5)
```

`doit(**hints)`

Evaluate this piecewise function.

[sympy.functions.elementary.piecewise.piecewise\\_fold\(expr\)](#)

Takes an expression containing a piecewise function and returns the expression in piecewise form.

See also:

[sympy.functions.elementary.piecewise.Piecewise](#) (page 332)

## Examples

```
>>> from sympy import Piecewise, piecewise_fold, sympify as S
>>> from sympy.abc import x
>>> p = Piecewise((x, x < 1), (1, S(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, 1 <= x))
```

**re**

Return the real part of an expression.

Examples:

```
>>> from sympy.functions import re
>>> from sympy import I
>>> re(2+3*I)
2
```

```
class sympy.functions.elementary.complexes.re
```

Returns real part of expression. This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use Basic.as\_real\_imag() or perform complex expansion on instance of this function.

```
>>> from sympy import re, im, I, E
>>> from sympy.abc import x, y

>>> re(2*E)
2*E

>>> re(2*I + 17)
17

>>> re(2*I)
0

>>> re(im(x) + x*I + 2)
2
```

See also:

[sympy.functions.elementary.complexes.im](#) (page 327)

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

Returns the real number with a zero complex part.

See also:

[sympy.functions.elementary.complexes.im](#) (page 327)

**real\_root**

```
sympy.functions.elementary.miscellaneous.real_root(arg, n=None)
```

Return the real nth-root of arg if possible. If n is omitted then all instances of  $(-n)^{**}(1/\text{odd})$  will be changed to  $-n^{**}(1/\text{odd})$ ; this will only create a real root of a principle root – the presence of other factors may cause the result to not be real.

See also:

[sympy.polys.rootof.RootOf](#) (page 722), [sympy.core.power.integer\\_nthroot](#) (page 112), [sympy.functions.elementary.miscellaneous.root](#) (page 334), [sympy.functions.elementary.miscellaneous.sqrt](#) (page 337)

## Examples

```
>>> from sympy import root, real_root, Rational
>>> from sympy.abc import x, n

>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*(-1)**(1/3)
>>> real_root(_)
-2
```

If one creates a non-principle root and applies `real_root`, the result will not be real (so use with caution):

```
>>> root(-8, 3, 2)
-2*(-1)**(2/3)
>>> real_root(_)
-2*(-1)**(2/3)
```

## root

`sympy.functions.elementary.miscellaneous.root(x, n, k)` → Returns the k-th n-th root of x, defaulting to the principle root (k=0).

See also:

`sympy.polys.rootoftools.RootOf` (page 722), `sympy.core.power.integer_nthroot` (page 112), `sympy.functions.elementary.miscellaneous.sqrt` (page 337), `sympy.functions.elementary.miscellaneous.real_root` (page 333)

## References

- [http://en.wikipedia.org/wiki/Square\\_root](http://en.wikipedia.org/wiki/Square_root)
- [http://en.wikipedia.org/wiki/Real\\_root](http://en.wikipedia.org/wiki/Real_root)
- [http://en.wikipedia.org/wiki/Root\\_of\\_unity](http://en.wikipedia.org/wiki/Root_of_unity)
- [http://en.wikipedia.org/wiki/Principal\\_value](http://en.wikipedia.org/wiki/Principal_value)
- <http://mathworld.wolfram.com/CubeRoot.html>

## Examples

```
>>> from sympy import root, Rational
>>> from sympy.abc import x, n

>>> root(x, 2)
sqrt(x)

>>> root(x, 3)
x**(1/3)
```

```
>>> root(x, n)
x**(1/n)

>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
-(-1)**(2/3)*2***(1/3)
```

To get all n n-th roots you can use the RootOf function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> from sympy import RootOf, I

>>> [ RootOf(x**2 - 1, i) for i in range(2) ]
[-1, 1]

>>> [ RootOf(x**3 - 1,i) for i in range(3) ]
[1, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]

>>> [ RootOf(x**4 - 1,i) for i in range(4) ]
[-1, 1, -I, I]
```

SymPy, like other symbolic algebra systems, returns the complex root of negative numbers. This is the principal root and differs from the text-book result that one might be expecting. For example, the cube root of -8 does not come back as -2:

```
>>> root(-8, 3)
2*(-1)**(1/3)
```

The real\_root function can be used to either make the principle result real (or simply to return the real root directly):

```
>>> from sympy import real_root
>>> real_root(_)
-2
>>> real_root(-32, 5)
-2
```

Alternatively, the n//2-th n-th root of a negative number can be computed with root:

```
>>> root(-32, 5, 5//2)
-2
```

## RoundFunction

```
class sympy.functions.elementary.integers.RoundFunction
    The base class for rounding functions.
```

**sin**

```
class sympy.functions.elementary.trigonometric.sin
    The sine function.
```

Returns the sine of x (measured in radians).

See also:

`sympy.functions.elementary.trigonometric.csc` (page 324), `sympy.functions.elementary.trigonometric.cos` (page 322), `sympy.functions.elementary.trigonometric.sec` (page 337),  
`sympy.functions.elementary.trigonometric.tan` (page 339), `sympy.functions.elementary.trigonometric.cot` (page 323), `sympy.functions.elementary.trigonometric.asin` (page 317),  
`sympy.functions.elementary.trigonometric.acsc` (page 316), `sympy.functions.elementary.trigonometric.acos` (page 314), `sympy.functions.elementary.trigonometric.asec` (page 318),  
`sympy.functions.elementary.trigonometric.atan` (page 318), `sympy.functions.elementary.trigonometric.atan2` (page 319),  
`sympy.functions.elementary.trigonometric.acot` (page 315), `sympy.functions.elementary.trigonometric.atan2` (page 319)

## Notes

This function will evaluate automatically in the case  $x/\pi$  is some rational number [R136] (page 1238). For example, if  $x$  is a multiple of  $\pi$ ,  $\pi/2$ ,  $\pi/3$ ,  $\pi/4$  and  $\pi/6$ .

## References

[R133] (page 1238), [R134] (page 1238), [R135] (page 1238), [R136] (page 1238)

## Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x
>>> sin(x**2).diff(x)
2*x*cos(x**2)
>>> sin(1).diff(x)
0
>>> sin(pi)
0
>>> sin(pi/2)
1
>>> sin(pi/6)
1/2
>>> sin(pi/12)
-sqrt(2)/4 + sqrt(6)/4
```

## sinh

`class sympy.functions.elementary.hyperbolic.sinh`

The hyperbolic sine function,  $\frac{e^x - e^{-x}}{2}$ .

•`sinh(x)` -> Returns the hyperbolic sine of  $x$

See also:

`sympy.functions.elementary.hyperbolic.cosh` (page 323), `sympy.functions.elementary.hyperbolic.tanh` (page 340), `sympy.functions.elementary.hyperbolic.asinh` (page 317)

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

Returns this function as a complex coordinate.

`fdiff(argindex=1)`

Returns the first derivative of this function.

```
inverse(argindex=1)
```

Returns the inverse of this function.

```
static taylor_term(*args, **kwargs)
```

Returns the next term in the Taylor series expansion.

## sec

```
class sympy.functions.elementary.trigonometric.sec
```

The secant function.

Returns the secant of  $x$  (measured in radians).

See also:

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322),  
[sympy.functions.elementary.trigonometric.tan](#) (page 339), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317),  
[sympy.functions.elementary.trigonometric.acsc](#) (page 316), [sympy.functions.elementary.trigonometric.ac](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318),  
[sympy.functions.elementary.trigonometric.atan](#) (page 318), [sympy.functions.elementary.trigonometric.ac](#) (page 315), [sympy.functions.elementary.trigonometric.atan2](#) (page 319)

## Notes

See [sin\(\)](#) (page 335) for notes about automatic evaluation.

## References

[R137] (page 1238), [R138] (page 1238), [R139] (page 1238)

## Examples

```
>>> from sympy import sec
>>> from sympy.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
>>> sec(1).diff(x)
0
```

## sqrt

Returns the square root of an expression. It is equivalent to raise to `Rational(1,2)`.

```
>>> from sympy.functions import sqrt
>>> from sympy import Rational
>>> sqrt(2) == 2**Rational(1,2)
True
```

```
sympy.functions.elementary.miscellaneous.sqrt(arg)
```

The square root function

`sqrt(x)` -> Returns the principal square root of  $x$ .

**See also:**

`sympy.polys.rootof.RootOf` (page 722), `sympy.functions.elementary.miscellaneous.root` (page 334), `sympy.functions.elementary.miscellaneous.real_root` (page 333)

## References

- [http://en.wikipedia.org/wiki/Square\\_root](http://en.wikipedia.org/wiki/Square_root)
- [http://en.wikipedia.org/wiki/Principal\\_value](http://en.wikipedia.org/wiki/Principal_value)

## Examples

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

>>> sqrt(x)
sqrt(x)

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

Note that  $\sqrt{x^2}$  does not simplify to  $x$ .

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

This is because the two are not equal to each other in general. For example, consider  $x == -1$ :

```
>>> from sympy import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
False
```

This is because `sqrt` computes the principal square root, so the square may put the argument in a different branch. This identity does hold if  $x$  is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the `force` option set to `True`:

```
>>> from sympy import powdenest
>>> sqrt(x**2)
sqrt(x**2)
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the `RootOf` function:

```
>>> from sympy import RootOf
>>> [ RootOf(x**2-3,i) for i in (0,1) ]
[-sqrt(3), sqrt(3)]
```

**sign**

```
class sympy.functions.elementary.complexes.sign
    Returns the complex sign of an expression:
```

If the expression is real the sign will be:

- 1 if expression is positive
- 0 if expression is equal to zero
- -1 if expression is negative

If the expression is imaginary the sign will be:

- I if  $\text{im}(\text{expression})$  is positive
- -I if  $\text{im}(\text{expression})$  is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be  $\cos(\arg(\text{expr})) + I * \sin(\arg(\text{expr}))$ .

**See also:**

[sympy.functions.elementary.complexes.Abs](#) (page 313), [sympy.functions.elementary.complexes.conjugate](#) (page 322)

**Examples**

```
>>> from sympy.functions import sign
>>> from sympy.core.numbers import I

>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> _.evalf()
0.707106781186548 + 0.707106781186548*I
```

**tan**

```
class sympy.functions.elementary.trigonometric.tan
    The tangent function.
```

Returns the tangent of x (measured in radians).

**See also:**

[sympy.functions.elementary.trigonometric.sin](#) (page 335), [sympy.functions.elementary.trigonometric.csc](#) (page 324), [sympy.functions.elementary.trigonometric.cos](#) (page 322), [sympy.functions.elementary.trigonometric.sec](#) (page 337), [sympy.functions.elementary.trigonometric.cot](#) (page 323), [sympy.functions.elementary.trigonometric.asin](#) (page 317), [sympy.functions.elementary.trigonometric.acsc](#) (page 316), [sympy.functions.elementary.trigonometric.acos](#) (page 314), [sympy.functions.elementary.trigonometric.asec](#) (page 318),

`sympy.functions.elementary.trigonometric.atan` (page 318), `sympy.functions.elementary.trigonometric.acot` (page 315), `sympy.functions.elementary.trigonometric.atan2` (page 319)

## Notes

See `sin()` (page 335) for notes about automatic evaluation.

## References

[R140] (page 1238), [R141] (page 1238), [R142] (page 1238)

## Examples

```
>>> from sympy import tan, pi
>>> from sympy.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(1).diff(x)
0
>>> tan(pi/8).expand()
-1 + sqrt(2)
```

`inverse(argindex=1)`

Returns the inverse of this function.

## tanh

`class sympy.functions.elementary.hyperbolic.tanh`

The hyperbolic tangent function,  $\frac{\sinh(x)}{\cosh(x)}$ .

•`tanh(x)` -> Returns the hyperbolic tangent of x

See also:

`sympy.functions.elementary.hyperbolic.sinh` (page 336), `sympy.functions.elementary.hyperbolic.cosh` (page 323), `sympy.functions.elementary.hyperbolic.atanh` (page 321)

`inverse(argindex=1)`

Returns the inverse of this function.

## polar\_lift

`class sympy.functions.elementary.complexes.polar_lift`

Lift argument to the Riemann surface of the logarithm, using the standard branch.

```
>>> from sympy import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
```

```
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)

>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

[sympy.functions.elementary.exponential.exp\\_polar](#) (page 325),  
[sympy.functions.elementary.complexes.periodic\\_argument](#) (page 341)

### **periodic\_argument**

**class sympy.functions.elementary.complexes.periodic\_argument**

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P, always return a value in  $(-P/2, P/2]$ , by using  $\exp(P*I) == 1$ .

```
>>> from sympy import exp, exp_polar, periodic_argument, unbranched_argument
>>> from sympy import I, pi
>>> unbranched_argument(exp(5*I*pi))
pi
>>> unbranched_argument(exp_polar(5*I*pi))
5*pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

[sympy.functions.elementary.exponential.exp\\_polar](#) (page 325)  
[sympy.functions.elementary.complexes.polar\\_lift](#) (page 340) Lift argument to the Riemann surface of the logarithm  
[sympy.functions.elementary.complexes.principal\\_branch](#) (page 341)

### **principal\_branch**

**class sympy.functions.elementary.complexes.principal\_branch**

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

This is a function of two arguments. The first argument is a polar number  $z$ , and the second one a positive real number of infinity,  $p$ . The result is “ $z \bmod \exp_polar(I*p)$ ”.

```
>>> from sympy import exp_polar, principal_branch, oo, I, pi
>>> from sympy.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
```

```
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)
```

See also:

[sympy.functions.elementary.exponential.exp\\_polar](#) (page 325)

[sympy.functions.elementary.complexes.polar\\_lift](#) ([page 340](#)) Lift argument to the Riemann surface of the logarithm

[sympy.functions.elementary.complexes.periodic\\_argument](#) (page 341)

## Combinatorial

This module implements various combinatorial functions.

### bell

```
class sympy.functions.combinatorial.numbers.bell
Bell numbers / Bell polynomials
```

The Bell numbers satisfy  $B_0 = 1$  and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by  $B_0(x) = 1$  and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, \dots, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots=k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!\dots j_{n-k+1}!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \dots \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- `bell(n)` gives the  $n^{th}$  Bell number,  $B_n$ .
- `bell(n, x)` gives the  $n^{th}$  Bell polynomial,  $B_n(x)$ .
- `bell(n, k, (x1, x2, ...))` gives Bell polynomials of the second kind,  $B_{n,k}(x_1, x_2, \dots, x_{n-k+1})$ .

See also:

[sympy.functions.combinatorial.numbers.bernoulli](#) (page 343), [sympy.functions.combinatorial.numbers.catalan](#) (page 345), [sympy.functions.combinatorial.numbers.euler](#) (page 347), [sympy.functions.combinatorial.numbers.fibonacci](#) (page 350), [sympy.functions.combinatorial.numbers.harmonic](#) (page 351), [sympy.functions.combinatorial.numbers.lucas](#) (page 353)

## Notes

Not to be confused with Bernoulli numbers and Bernoulli polynomials, which use the same notation.

## References

[R69] (page 1238), [R70] (page 1238), [R71] (page 1238)

## Examples

```
>>> from sympy import bell, Symbol, symbols

>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
>>> bell(30)
846749014511809332450147
>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t
>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

## bernieulli

```
class sympy.functions.combinatorial.numbers.bernieulli
Bernoulli numbers / Bernoulli polynomials
```

The Bernoulli numbers are a sequence of rational numbers defined by  $B_0 = 1$  and the recursive relation ( $n > 0$ ):

$$B_n = \frac{1}{n+1} \sum_{k=0}^n \frac{(n+1)_k}{k!} B_k$$

They are also commonly defined by their exponential generating function, which is  $x/(e^x - 1)$ . For odd indices  $> 1$ , the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \frac{1}{n!} \sum_{k=0}^n \frac{(n)_k}{k!} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as  $B_n(0) = B_n$ .

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = \frac{(A(n) - S(n))}{n!}$$

where  $A(n) = (n+3)/3$  when  $n = 0$  or  $2 \pmod{6}$ ,  $A(n) = -(n+3)/6$  when  $n = 4 \pmod{6}$ , and:

```
[n/6]
  ___      / n + 3 \
S(n) = )      | * B
  /___      \ n - 6*k /   n-6*k
    k = 1
```

This formula is similar to the sum given in the definition, but cuts  $2/3$  of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the nth Bernoulli number,  $B_n$
- `bernoulli(n, x)` gives the nth Bernoulli polynomial in  $x$ ,  $B_n(x)$

See also:

`sympy.functions.combinatorial.numbers.bell` (page 342), `sympy.functions.combinatorial.numbers.catalan` (page 345), `sympy.functions.combinatorial.numbers.euler` (page 347),  
`sympy.functions.combinatorial.numbers.fibonacci` (page 350), `sympy.functions.combinatorial.numbers.harmonic` (page 351), `sympy.functions.combinatorial.numbers.lucas` (page 353)

## References

[R72] (page 1238), [R73] (page 1238), [R74] (page 1238), [R75] (page 1238)

## Examples

```
>>> from sympy import bernoulli

>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]
>>> bernoulli(1000001)
0
```

## binomial

`class sympy.functions.combinatorial.factorials.binomial`

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$C(n,k) = n!/(k!(n-k)!) \text{ or } C(n, k) = f(n, k)/k!$$

First, in a strict combinatorial sense it defines the number of ways we can choose ‘ $k$ ’ elements from a set of ‘ $n$ ’ elements. In this case both arguments are nonnegative integers and `binomial` is computed using an efficient algorithm based on prime factorization.

The other definition is generalization for arbitrary ‘ $n$ ’, however ‘ $k$ ’ must also be nonnegative. This case is very useful when evaluating summations.

For the sake of convenience for negative ‘ $k$ ’ this function will return zero no matter what valued is the other argument.

To expand the binomial when  $n$  is a symbol, use either `expand_func()` or `expand(func=True)`. The former will keep the polynomial in factored form while the latter will expand the polynomial itself. See examples for details.

## Examples

```
>>> from sympy import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)

>>> binomial(15, 8)
6435

>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
...     print([binomial(N, i) for i in range(N + 1)])
...
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]

>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128

>>> binomial(n, 3)
binomial(n, 3)

>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3

>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```

## catalan

```
class sympy.functions.combinatorial.numbers.catalan
    Catalan numbers
```

The n-th catalan number is given by:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- catalan(n) gives the n-th Catalan number, C\_n

See also:

`sympy.functions.combinatorial.numbers.bell` (page 342), `sympy.functions.combinatorial.numbers.bernoulli` (page 343), `sympy.functions.combinatorial.numbers.euler` (page 347), `sympy.functions.combinatorial.numbers.fibonacci` (page 350), `sympy.functions.combinatorial.numbers.harmonic` (page 351), `sympy.functions.combinatorial.numbers.lucas` (page 353), `sympy.functions.combinatorial.factorials.binomial` (page 344)

## References

[R76] (page 1238), [R77] (page 1238), [R78] (page 1238), [R79] (page 1238)

## Examples

```
>>> from sympy import (Symbol, binomial, gamma, hyper, polygamma,
...                      catalan, diff, combsimp, Rational, I)

>>> [ catalan(i) for i in range(1,10) ]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]

>>> n = Symbol("n", integer=True)

>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)

>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))

>>> catalan(n).rewrite(hyper)
hyper((-n + 1, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1,2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers  $C(n)$  interpreted as a continuous real function in n:

```
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)
4*I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

## euler

```
class sympy.functions.combinatorial.numbers.euler
Euler numbers
```

The euler numbers are given by:

$$\frac{\sum_{k=0}^{2n+1} \sum_{j=0}^k (-1)^j (k-2j)!}{\sum_{k=0}^{2n+1} j! k!}$$

$$E_{2n+1} = 0$$

- euler(n) gives the n-th Euler number, E\_n

See also:

[sympy.functions.combinatorial.numbers.bell](#) (page 342), [sympy.functions.combinatorial.numbers.bernoulli](#) (page 343), [sympy.functions.combinatorial.numbers.fibonacci](#) (page 350), [sympy.functions.combinatorial.numbers.harmonic](#) (page 351), [sympy.functions.combinatorial.numbers.lucas](#) (page 353)

## References

[R80] (page 1238), [R81] (page 1238), [R82] (page 1238), [R83] (page 1239)

## Examples

```
>>> from sympy import Symbol
>>> from sympy.functions import euler
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n+2*n)
euler(3*n)
```

## factorial

```
class sympy.functions.combinatorial.factorials.factorial
```

Implementation of factorial function over nonnegative integers. By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which  $n$  objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact  $n! = \text{gamma}(n+1)$  for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments naive product is evaluated. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes  $n!$  via prime factorization of special class of numbers, called here the ‘Swing Numbers’.

See also:

`sympy.functions.combinatorial.factorials.factorial2` (page 349),  
`sympy.functions.combinatorial.factorials.RisingFactorial` (page 354),  
`sympy.functions.combinatorial.factorials.FallingFactorial` (page 350)

### Examples

```
>>> from sympy import Symbol, factorial, S
>>> n = Symbol('n', integer=True)

>>> factorial(0)
1

>>> factorial(7)
5040

>>> factorial(-2)
zoo

>>> factorial(n)
factorial(n)

>>> factorial(2*n)
factorial(2*n)

>>> factorial(S(1)/2)
factorial(1/2)
```

### subfactorial

```
class sympy.functions.combinatorial.factorials.subfactorial
```

The subfactorial counts the derangements of  $n$  items and is defined for non-negative integers as:

```
    ,
    |   1                               for n = 0
!n = {  0                               for n = 1
    |   (n - 1)*(!!(n - 1) + !(n - 2)) for n > 1
    ,
```

It can also be written as `int(round(n!/exp(1)))` but the recursive definition with caching is implemented for this function.

This function is generalized to noninteger arguments [R85] (page 1239) as

$$!x = \Gamma(x + 1, -1)/e$$

See also:

`sympy.functions.combinatorial.factorials.factorial` (page 347),  
`sympy.utilities.iterables.generate_derangements` (page 1134),  
`sympy.functions.special.gamma_functions.uppergamma` (page 366)

## References

[R84] (page 1239), [R85] (page 1239)

## Examples

```
>>> from sympy import subfactorial
>>> from sympy.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

### factorial2 / double factorial

```
class sympy.functions.combinatorial.factorials.factorial2
The double factorial n!!, not to be confused with (n)!
```

The double factorial is defined for nonnegative integers and for odd negative integers as:

```
'      n*(n - 2)*(n - 4)* ... * 1    for n positive odd
n!! = {  n*(n - 2)*(n - 4)* ... * 2    for n positive even
      | 1                           for n = 0
      | (n+2)!! / (n+2)           for n negative odd
      '
```

See also:

`sympy.functions.combinatorial.factorials.factorial` (page 347),  
`sympy.functions.combinatorial.factorials.RisingFactorial` (page 354),  
`sympy.functions.combinatorial.factorials.FallingFactorial` (page 350)

## References

[R86] (page 1239)

## Examples

```
>>> from sympy import factorial2, var
>>> var('n')
n
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
```

```
1
>>> factorial2(-5)
1/3
```

## FallingFactorial

`class sympy.functions.combinatorial.factorials.FallingFactorial`

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

$$\text{ff}(x, k) = x * (x-1) * \dots * (x - k+1)$$

where ‘x’ can be arbitrary expression and ‘k’ is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or visit <http://mathworld.wolfram.com/FallingFactorial.html> page.

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

>>> ff(x, 0)
1

>>> ff(5, 5)
120

>>> ff(x, 5) == x*(x-1)*(x-2)*(x-3)*(x-4)
True
```

See also:

`sympy.functions.combinatorial.factorials.factorial`

(page 347),

`sympy.functions.combinatorial.factorials.factorial2`

(page 349),

`sympy.functions.combinatorial.factorials.RisingFactorial` (page 354)

## fibonacci

`class sympy.functions.combinatorial.numbers.fibonacci`

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms  $F_0 = 0$ ,  $F_1 = 1$  and the two-term recurrence relation  $F_n = F_{n-1} + F_{n-2}$ . This definition extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by  $F_1(x) = 1$ ,  $F_2(x) = x$ , and  $F_n(x) = x*F_{n-1}(x) + F_{n-2}(x)$  for  $n > 2$ . For all positive integers  $n$ ,  $F_n(1) = F_n$ .

- `fibonacci(n)` gives the nth Fibonacci number,  $F_n$
- `fibonacci(n, x)` gives the nth Fibonacci polynomial in  $x$ ,  $F_n(x)$

See also:

`sympy.functions.combinatorial.numbers.bell` (page 342), `sympy.functions.combinatorial.numbers.bernoulli` (page 343), `sympy.functions.combinatorial.numbers.catalan` (page 345), `sympy.functions.combinatorial.numbers.euler` (page 347), `sympy.functions.combinatorial.numbers.harmonic` (page 351), `sympy.functions.combinatorial.numbers.lucas` (page 353)

## References

[R87] (page 1239), [R88] (page 1239)

## Examples

```
>>> from sympy import fibonacci, Symbol

>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

## harmonic

```
class sympy.functions.combinatorial.numbers.harmonic
    Harmonic numbers
```

The nth harmonic number is given by  $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$ .

More generally:

$$H_{n,m} = \sum_{k=1}^n \frac{1}{k^m}$$

As  $n \rightarrow \infty$ ,  $H_{n,m} \rightarrow \zeta(m)$ , the Riemann zeta function.

- `harmonic(n)` gives the nth harmonic number,  $H_n$
- `harmonic(n, m)` gives the nth generalized harmonic number of order  $m$ ,  $H_{n,m}$ , where `harmonic(n) == harmonic(n, 1)`

See also:

`sympy.functions.combinatorial.numbers.bell` (page 342), `sympy.functions.combinatorial.numbers.bernoulli` (page 343), `sympy.functions.combinatorial.numbers.catalan` (page 345),  
`sympy.functions.combinatorial.numbers.euler` (page 347), `sympy.functions.combinatorial.numbers.fibonacci` (page 350), `sympy.functions.combinatorial.numbers.lucas` (page 353)

## References

[R89] (page 1239), [R90] (page 1239), [R91] (page 1239)

## Examples

```
>>> from sympy import harmonic, oo

>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]
>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]
>>> harmonic(oo, 2)
pi**2/6
```

```
>>> from sympy import Symbol, Sum
>>> n = Symbol("n")

>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> from sympy import S, expand_func, simplify
>>> harmonic(8)
761/280
>>> harmonic(11)
83711/27720

>>> H = harmonic(1/S(3))
>>> H
harmonic(1/3)
>>> He = expand_func(H)
>>> He
-log(6) - sqrt(3)*pi/6 + 2*Sum(log(sin(_k*pi/3))*cos(2*_k*pi/3), (_k, 1, 1))
+ 3*Sum(1/(3*_k + 1), (_k, 0, 0))
>>> He.doit()
-log(6) - sqrt(3)*pi/6 - log(sqrt(3)/2) + 3
>>> H = harmonic(25/S(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(pi/7)**(-2*cos(pi/7))*sin(2*pi/7)**(2*cos(16*pi/7))*cos(pi/14)**(-2*sin(pi/14))/14)
+ pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/S(7)).n(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> from sympy import digamma, polygamma
>>> m = Symbol("m")

>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma

>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma

>>> harmonic(n,3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2

>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)
```

Integer offsets in the argument can be pulled out:

```
>>> from sympy import expand_func

>>> expand_func(harmonic(n+4))
harmonic(n) + 1/(n + 4) + 1/(n + 3) + 1/(n + 2) + 1/(n + 1)
```

```
>>> expand_func(harmonic(n-4))
harmonic(n) - 1/(n - 1) - 1/(n - 2) - 1/(n - 3) - 1/n
```

Some limits can be computed as well:

```
>>> from sympy import limit, oo

>>> limit(harmonic(n), n, oo)
oo

>>> limit(harmonic(n, 2), n, oo)
pi**2/6

>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we can not compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals `zeta(m)` for  $m > 1$ .

## lucas

```
class sympy.functions.combinatorial.numbers.lucas
    Lucas numbers
```

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values  $L_0 = 2$  and  $L_1 = 1$ .

• `lucas(n)` gives the nth Lucas number

See also:

`sympy.functions.combinatorial.numbers.bell` (page 342), `sympy.functions.combinatorial.numbers.bernoulli` (page 343), `sympy.functions.combinatorial.numbers.catalan` (page 345),  
`sympy.functions.combinatorial.numbers.euler` (page 347), `sympy.functions.combinatorial.numbers.fibonacci` (page 350), `sympy.functions.combinatorial.numbers.harmonic` (page 351)

## References

[R92] (page 1239), [R93] (page 1239)

## Examples

```
>>> from sympy import lucas

>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

**MultiFactorial**

```
class sympy.functions.combinatorial.factorials.MultiFactorial
```

**RisingFactorial**

```
class sympy.functions.combinatorial.factorials.RisingFactorial
```

Rising factorial (also called Pochhammer symbol) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by:

$$rf(x, k) = x * (x+1) * \dots * (x + k-1)$$

where ‘x’ can be arbitrary expression and ‘k’ is an integer. For more information check “Concrete mathematics” by Graham, pp. 66 or visit <http://mathworld.wolfram.com/RisingFactorial.html> page.

**See also:**

<code>sympy.functions.combinatorial.factorials.factorial</code>	(page 347),
<code>sympy.functions.combinatorial.factorials.factorial2</code>	(page 349),
<code>sympy.functions.combinatorial.factorials.FallingFactorial</code>	(page 350)

**Examples**

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

>>> rf(x, 0)
1

>>> rf(1, 5)
120

>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
```

**stirling**

```
sympy.functions.combinatorial.numbers.stirling(n, k, d=None, kind=2, signed=False)
```

Return Stirling number  $S(n, k)$  of the first or second (default) kind.

The sum of all Stirling numbers of the second kind for  $k = 1$  through  $n$  is  $\text{bell}(n)$ . The recurrence relationship for these numbers is:

$$\begin{aligned} \{0\} &= \{n\} & \{0\} &= \{n+1\} & \{n\} &= \{n\} \\ \{ \} &= 1; & \{ \} &= \{ \} = 0; & \{ \} &= j * \{ \} + \{ \} \\ \{0\} &= \{0\} & \{k\} &= \{k\} & \{k\} &= \{k-1\} \end{aligned}$$

**where  $j$  is:**  $n$  for Stirling numbers of the first kind  $-n$  for signed Stirling numbers of the first kind  $k$  for Stirling numbers of the second kind

The first kind of Stirling number counts the number of permutations of  $n$  distinct items that have  $k$  cycles; the second kind counts the ways in which  $n$  distinct items can be partitioned into  $k$  parts. If  $d$  is given, the “reduced Stirling number of the second kind” is returned:  $S^{\{d\}}(n, k) = S(n - d +$

$1, k - d + 1)$  with  $n \geq k \geq d$ . (This counts the ways to partition  $n$  consecutive integers into  $k$  groups with no pairwise difference less than  $d$ . See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

See also:

`sympy.utilities.iterables.multiset_partitions` (page 1139)

## References

[R94] (page 1239), [R95] (page 1239)

## Examples

```
>>> from sympy.functions.combinatorial.numbers import stirling, bell
>>> from sympy.combinatorics import Permutation
>>> from sympy.utilities.iterables import multiset_partitions, permutations
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
>>> [stirling(4, i, kind=1) for i in range(5)]
[0, 6, 11, 6, 1]
```

First kind (signed):

```
>>> [stirling(4, i, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```

Second kind:

```
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```

Reduced second kind:

```
>>> from sympy import subsets, oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

## Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The `k` parameter indicates the number of elements to pick (or the number of partitions to make). When `k` is `None`, the sum of the enumeration for all `k` (from 0 through the number of items represented by `n`) is returned. A `replacement` parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.

```
>>> from sympy.functions.combinatorial.numbers import nC, nP, nT  
>>> items = 'baby'  
  
sympy.functions.combinatorial.numbers.nC(n, k=None, replacement=False)
```

Return the number of combinations of `n` items taken `k` at a time.

**Possible values for `n`:** integer - set of length `n` sequence - converted to a multiset internally multiset - {element: multiplicity}

If `k` is `None` then the total of all combinations of length 0 through the number of items represented in `n` will be returned.

If `replacement` is `True` then a given item can appear more than once in the `k` items. (For example, for ‘ab’ sets of 2 would include ‘aa’, ‘ab’, and ‘bb’.) The multiplicity of elements in `n` is ignored when `replacement` is `True` but the total number of elements is considered since no element can appear more times than the number of elements in `n`.

See also:

[sympy.utilities.iterables.multiset\\_combinations](#) (page 1139)

## References

[R96] (page 1239), [R97] (page 1239)

## Examples

```
>>> from sympy.functions.combinatorial.numbers import nC  
>>> from sympy.utilities.iterables import multiset_combinations  
>>> nC(3, 2)  
3  
>>> nC('abc', 2)  
3  
>>> nC('aab', 2)  
2
```

When `replacement` is `True`, each item can have multiplicity equal to the length represented by `n`:

```
>>> nC('aabc', replacement=True)  
35  
>>> [len(list(multiset_combinations('aaaabbbbcccc', i))) for i in range(5)]  
[1, 3, 6, 10, 15]  
>>> sum(_)  
35
```

If there are `k` items with multiplicities `m_1, m_2, ..., m_k` then the total of all combinations of length 0 through `k` is the product,  $(m_1 + 1)*(m_2 + 1)*\dots*(m_k + 1)$ . When the multiplicity of each item

is 1 (i.e., k unique items) then there are  $2^{**k}$  combinations. For example, if there are 4 unique items, the total number of combinations is 16:

```
>>> sum(nC(4, i) for i in range(5))
16
```

`sympy.functions.combinatorial.numbers.nP(n, k=None, replacement=False)`

Return the number of permutations of n items taken k at a time.

**Possible values for n::** integer - set of length n sequence - converted to a multiset internally multiset  
- {element: multiplicity}

If k is None then the total of all permutations of length 0 through the number of items represented by n will be returned.

If replacement is True then a given item can appear more than once in the k items. (For example, for ‘ab’ permutations of 2 would include ‘aa’, ‘ab’, ‘ba’ and ‘bb’.) The multiplicity of elements in n is ignored when replacement is True but the total number of elements is considered since no element can appear more times than the number of elements in n.

See also:

`sympy.utilities.iterables.multiset_permutations` (page 1140)

## References

[R98] (page 1239)

## Examples

```
>>> from sympy.functions.combinatorial.numbers import nP
>>> from sympy.utilities.iterables import multiset_permutations, multiset
>>> nP(3, 2)
6
>>> nP('abc', 2) == nP(multiset('abc'), 2) == 6
True
>>> nP('aab', 2)
3
>>> nP([1, 2, 2], 2)
3
>>> [nP(3, i) for i in range(4)]
[1, 3, 6, 6]
>>> nP(3) == sum(_)
True
```

When replacement is True, each item can have multiplicity equal to the length represented by n:

```
>>> nP('aabc', replacement=True)
121
>>> [len(list(multiset_permutations('aaaabbbbcccc', i))) for i in range(5)]
[1, 3, 9, 27, 81]
>>> sum(_)
121
```

`sympy.functions.combinatorial.numbers.nT(n, k=None)`

Return the number of k-sized partitions of n items.

**Possible values for n::** integer -  $n$  identical items sequence - converted to a multiset internally multisets - {element: multiplicity}

Note: the convention for `nT` is different than that of `nC` and `nP` in that here an integer indicates  $n$  *identical* items instead of a set of length  $n$ ; this is in keeping with the `partitions` function which treats its integer- $n$  input like a list of  $n$  1s. One can use `range(n)` for  $n$  to indicate  $n$  distinct items.

If `k` is `None` then the total number of ways to partition the elements represented in `n` will be returned.

**See also:**

`sympy.utilities.iterables.partitions` (page 1141), `sympy.utilities.iterables.multiset_partitions` (page 1139)

## References

[R99] (page 1239)

## Examples

```
>>> from sympy.functions.combinatorial.numbers import nT
```

Partitions of the given multiset:

```
>>> [nT('aabbc', i) for i in range(1, 7)]
[1, 8, 11, 5, 1, 0]
>>> nT('aabbc') == sum(_)
True
```

```
>>> [nT("mississippi", i) for i in range(1, 12)]
[1, 74, 609, 1521, 1768, 1224, 579, 197, 50, 9, 1]
```

Partitions when all items are identical:

```
>>> [nT(5, i) for i in range(1, 6)]
[1, 2, 2, 1, 1]
>>> nT('1'*5) == sum(_)
True
```

When all items are different:

```
>>> [nT(range(5), i) for i in range(1, 6)]
[1, 15, 25, 10, 1]
>>> nT(range(5)) == sum(_)
True
```

Note that the integer for `n` indicates *identical* items for `nT` but indicates  $n$  *different* items for `nC` and `nP`.

## Special

### DiracDelta

```
class sympy.functions.special.delta_functions.DiracDelta
The DiracDelta function and its derivatives.
```

DiracDelta function has the following properties:

```
1.diff(Heaviside(x),x) = DiracDelta(x)
2.integrate(DiracDelta(x-a)*f(x),(x,-oo,oo)) = f(a) and integrate(DiracDelta(x-a)*f(x),(x,a-e,a+e))
   = f(a)
3.DiracDelta(x) = 0 for all x != 0
4.DiracDelta(g(x)) = Sum_i(DiracDelta(x-x_i)/abs(g'(x_i))) Where x_i-s are the roots of g
Derivatives of k-th order of DiracDelta have the following property:
5.DiracDelta(x,k) = 0, for all x != 0
```

See also:

[sympy.functions.special.delta\\_functions.Heaviside](#) (page 360),  
[sympy.simplify.simplify](#) (page 916), [sympy.functions.special.delta\\_functions.DiracDelta.is\\_simple](#) (page 359), [sympy.functions.special.tensor\\_functions.KroneckerDelta](#) (page 425)

## References

[R143] (page 1239)

`is_simple(self, x)`

Tells whether the argument(args[0]) of DiracDelta is a linear expression in x.

x can be:

- a symbol

See also:

[sympy.simplify.simplify.simplify](#) (page 916), [sympy.functions.special.delta\\_functions.DiracDelta](#) (page 358)

## Examples

```
>>> from sympy import DiracDelta, cos
>>> from sympy.abc import x, y

>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True

>>> DiracDelta(x**2+x-2).is_simple(x)
False

>>> DiracDelta(cos(x)).is_simple(x)
False
```

`simplify(self, x)`

Compute a simplified representation of the function using property number 4.

x can be:

- a symbol

See also:

[sympy.functions.special.delta\\_functions.DiracDelta.is\\_simple](#) (page 359),  
[sympy.functions.special.delta\\_functions.DiracDelta](#) (page 358)

## Examples

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

>>> DiracDelta(x*y).simplify(x)
DiracDelta(x)/Abs(y)
>>> DiracDelta(x*y).simplify(y)
DiracDelta(y)/Abs(x)

>>> DiracDelta(x**2 + x - 2).simplify(x)
DiracDelta(x - 1)/3 + DiracDelta(x + 2)/3
```

## Heaviside

```
class sympy.functions.special.delta_functions.Heaviside
    Heaviside Piecewise function
```

Heaviside function has the following properties<sup>5</sup>:

```
1.diff(Heaviside(x),x) = DiracDelta(x) ( 0, if x < 0
2.Heaviside(x) = < ( 1/2 if x==0 [*] ( 1, if x > 0
```

I think is better to have  $H(0) = 1/2$ , due to the following:

```
integrate(DiracDelta(x), x) = Heaviside(x)
integrate(DiracDelta(x), (x, -oo, oo)) = 1
```

and since DiracDelta is a symmetric function, `integrate(DiracDelta(x), (x, 0, oo))` should be 1/2 (which is what Maple returns).

If we take  $\text{Heaviside}(0) = 1/2$ , we would have `integrate(DiracDelta(x), (x, 0, oo)) = ``Heaviside(oo) - Heaviside(0) = 1 - 1/2 = 1/2` and `integrate(DiracDelta(x), (x, -oo, 0)) = ``Heaviside(0) - Heaviside(-oo) = 1/2 - 0 = 1/2`

If we consider, instead  $\text{Heaviside}(0) = 1$ , we would have `integrate(DiracDelta(x), (x, 0, oo)) = Heaviside(oo) - Heaviside(0) = 0` and `integrate(DiracDelta(x), (x, -oo, 0)) = Heaviside(0) - Heaviside(-oo) = 1`

See also:

`sympy.functions.special.delta_functions.DiracDelta` (page 358)

## References

[R144] (page 1239)

## Gamma, Beta and related Functions

```
class sympy.functions.special.gamma_functions.gamma
    The gamma function
```

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^t dt.$$

<sup>5</sup> Regarding to the value at 0, Mathematica defines  $H(0) = 1$ , but Maple uses  $H(0) = \text{undefined}$

The `gamma` function implements the function which passes through the values of the factorial function, i.e.  $\Gamma(n) = (n - 1)!$  when  $n$  is an integer. More general,  $\Gamma(z)$  is defined in the whole complex plane except at the negative integers where there are simple poles.

**See also:**

[lowergamma](#) ([page 367](#)) Lower incomplete gamma function.  
[uppergamma](#) ([page 366](#)) Upper incomplete gamma function.  
[polygamma](#) ([page 363](#)) Polygamma function.  
[loggamma](#) ([page 362](#)) Log Gamma function.  
[digamma](#) ([page 365](#)) Digamma function.  
[trigamma](#) ([page 365](#)) Trigamma function.  
[sympy.functions.special.beta\\_functions.betac](#) ([page 367](#)) Euler Beta function.

## References

[R145] ([page 1239](#)), [R146] ([page 1239](#)), [R147] ([page 1239](#)), [R148] ([page 1239](#))

## Examples

```
>>> from sympy import S, I, pi, oo, gamma
>>> from sympy.abc import x
```

Several special values are known:

```
>>> gamma(1)
1
>>> gamma(4)
6
>>> gamma(S(3)/2)
sqrt(pi)/2
```

The Gamma function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to  $x$  is supported:

```
>>> from sympy import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 + polygamma(2, 1)/6 - EulerGamma*pi**3/3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

class `sympy.functions.special.gamma_functions.loggamma`

The `loggamma` function implements the logarithm of the gamma function i.e,  $\log \Gamma(x)$ .

See also:

`gamma` ([page 360](#)) Gamma function.

`lowergamma` ([page 367](#)) Lower incomplete gamma function.

`uppergamma` ([page 366](#)) Upper incomplete gamma function.

`polygamma` ([page 363](#)) Polygamma function.

`digamma` ([page 365](#)) Digamma function.

`trigamma` ([page 365](#)) Trigamma function.

`sympy.functions.special.beta_functions.betac` ([page 367](#)) Euler Beta function.

## References

[R149] ([page 1239](#)), [R150] ([page 1239](#)), [R151] ([page 1239](#)), [R152] ([page 1239](#))

## Examples

Several special values are known. For numerical integral arguments we have:

```
>>> from sympy import loggamma
>>> loggamma(-2)
oo
>>> loggamma(0)
oo
>>> loggamma(1)
0
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

and for symbolic values:

```
>>> from sympy import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

for half-integral values:

```
>>> from sympy import S, pi
>>> loggamma(S(5)/2)
log(3*sqrt(pi)/4)
```

---

```
>>> loggamma(n/2)
log(2**(-n + 1)*sqrt(pi)*gamma(n)/gamma(n/2 + 1/2))
```

and general rational arguments:

```
>>> from sympy import expand_func
>>> L = loggamma(S(16)/3)
>>> expand_func(L).doit()
-5*log(3) + loggamma(1/3) + log(4) + log(7) + log(10) + log(13)
>>> L = loggamma(S(19)/4)
>>> expand_func(L).doit()
-4*log(4) + loggamma(3/4) + log(3) + log(7) + log(11) + log(15)
>>> L = loggamma(S(23)/7)
>>> expand_func(L).doit()
-3*log(7) + log(2) + loggamma(2/7) + log(9) + log(16)
```

The loggamma function has the following limits towards infinity:

```
>>> from sympy import oo
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if  $x \in \mathbb{C} \setminus \{-\infty, 0\}$ :

```
>>> from sympy.abc import x
>>> from sympy import conjugate
>>> conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(loggamma(x), x, 0, 4)
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + O(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import I
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.872436647264298171*I
```

`class sympy.functions.special.gamma_functions.polygamma`

The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`.

It is a meromorphic function on  $\mathbb{C}$  and defined as the  $(n+1)$ -th derivative of the logarithm of the gamma function:

$$\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).$$

See also:

[gamma](#) (page 360) Gamma function.  
[lowergamma](#) (page 367) Lower incomplete gamma function.  
[uppergamma](#) (page 366) Upper incomplete gamma function.  
[loggamma](#) (page 362) Log Gamma function.  
[digamma](#) (page 365) Digamma function.  
[trigamma](#) (page 365) Trigamma function.  
[sympy.functions.special.betafuctions.bet](#) (page 367) Euler Beta function.

## References

[R153] (page 1239), [R154] (page 1239), [R155] (page 1239), [R156] (page 1239)

## Examples

Several special values are known:

```
>>> from sympy import S, polygamma
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, 1/S(2))
-2*log(2) - EulerGamma
>>> polygamma(0, 1/S(3))
-3*log(3)/2 - sqrt(3)*pi/6 - EulerGamma
>>> polygamma(0, 1/S(4))
-3*log(2) - pi/2 - EulerGamma
>>> polygamma(0, 2)
-EulerGamma + 1
>>> polygamma(0, 23)
-EulerGamma + 19093197/5173168

>>> from sympy import oo, I
>>> polygamma(0, oo)
oo
>>> polygamma(0, -oo)
oo
>>> polygamma(0, I*oo)
oo
>>> polygamma(0, -I*oo)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import Symbol, diff
>>> x = Symbol("x")
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
```

---

```

polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)

>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)

```

We can rewrite polygamma functions in terms of harmonic numbers:

```

>>> from sympy import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)

sympy.functions.special.gamma_functions.digamma(x)

```

The digamma function is the first derivative of the loggamma function i.e,

$$\psi(x) := \frac{d}{dz} \log \Gamma(z) = \frac{\Gamma'(z)}{\Gamma(z)}$$

In this case, `digamma(z)` = `polygamma(0, z)`.

**See also:**

[gamma \(page 360\)](#) Gamma function.

[lowergamma \(page 367\)](#) Lower incomplete gamma function.

[uppergamma \(page 366\)](#) Upper incomplete gamma function.

[polygamma \(page 363\)](#) Polygamma function.

[loggamma \(page 362\)](#) Log Gamma function.

[trigamma \(page 365\)](#) Trigamma function.

[sympy.functions.special.beta\\_functions.betac \(page 367\)](#) Euler Beta function.

## References

[R157] (page 1239), [R158] (page 1239), [R159] (page 1239)

`sympy.functions.special.gamma_functions.trigamma(x)`

The trigamma function is the second derivative of the loggamma function i.e,

$$\psi^{(1)}(z) := \frac{d^2}{dz^2} \log \Gamma(z).$$

In this case, `trigamma(z)` = `polygamma(1, z)`.

**See also:**

[gamma](#) (page 360) Gamma function.  
[lowergamma](#) (page 367) Lower incomplete gamma function.  
[uppergamma](#) (page 366) Upper incomplete gamma function.  
[polygamma](#) (page 363) Polygamma function.  
[loggamma](#) (page 362) Log Gamma function.  
[digamma](#) (page 365) Digamma function.  
[sympy.functions.special.beta\\_functions.betac](#) (page 367) Euler Beta function.

## References

[R160] (page 1239), [R161] (page 1239), [R162] (page 1240)

**class** `sympy.functions.special.gamma_functions.uppergamma`  
The upper incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).$$

where  $\gamma(s, x)$  is the lower incomplete gamma function, [lowergamma](#) (page 367). This can be shown to be the same as

$$\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} {}_1F_1\left(\begin{array}{c} s \\ s+1 \end{array} \middle| -x\right),$$

where  ${}_1F_1$  is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt = x^{n-1} \Gamma(1-n, x).$$

## See also:

[gamma](#) (page 360) Gamma function.  
[lowergamma](#) (page 367) Lower incomplete gamma function.  
[polygamma](#) (page 363) Polygamma function.  
[loggamma](#) (page 362) Log Gamma function.  
[digamma](#) (page 365) Digamma function.  
[trigamma](#) (page 365) Trigamma function.  
[sympy.functions.special.beta\\_functions.betac](#) (page 367) Euler Beta function.

## References

[R163] (page 1240), [R164] (page 1240), [R165] (page 1240), [R166] (page 1240), [R167] (page 1240),  
[R168] (page 1240)

## Examples

```
>>> from sympy import uppergamma, S
>>> from sympy.abc import s, x
>>> uppergamma(s, x)
uppergamma(s, x)
>>> uppergamma(3, x)
x**2*exp(-x) + 2*x*exp(-x) + 2*exp(-x)
>>> uppergamma(-S(1)/2, x)
-2*sqrt(pi)*(-erf(sqrt(x)) + 1) + 2*exp(-x)/sqrt(x)
>>> uppergamma(-2, x)
expint(3, x)/x**2
```

`class sympy.functions.special.gamma_functions.lowergamma`  
The lower incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1\left(\begin{array}{c} s \\ s+1 \end{array} \middle| -x\right),$$

where  ${}_1F_1$  is the (confluent) hypergeometric function.

See also:

[gamma \(page 360\)](#) Gamma function.

[uppergamma \(page 366\)](#) Upper incomplete gamma function.

[polygamma \(page 363\)](#) Polygamma function.

[loggamma \(page 362\)](#) Log Gamma function.

[digamma \(page 365\)](#) Digamma function.

[trigamma \(page 365\)](#) Trigamma function.

[sympy.functions.special.beta\\_functions.betac](#) (page 367) Euler Beta function.

## References

[R169] (page 1240), [R170] (page 1240), [R171] (page 1240), [R172] (page 1240), [R173] (page 1240)

## Examples

```
>>> from sympy import lowergamma, S
>>> from sympy.abc import s, x
>>> lowergamma(s, x)
lowergamma(s, x)
>>> lowergamma(3, x)
-x**2*exp(-x) - 2*x*exp(-x) + 2 - 2*exp(-x)
>>> lowergamma(-S(1)/2, x)
-2*sqrt(pi)*erf(sqrt(x)) - 2*exp(-x)/sqrt(x)
```

```
class sympy.functions.special.beta_functions.beta
```

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$B(x, y) := \int_0^1 t^{x-1} (1-t)^{y-1} dt.$$

Beta function or Euler's first integral is closely associated with gamma function. The Beta function often used in probability theory and mathematical statistics. It satisfies properties like:

$$\begin{aligned} B(a, 1) &= \frac{1}{a} \\ B(a, b) &= B(b, a) \\ B(a, b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \end{aligned}$$

Therefore for integral values of a and b:

$$B = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

#### See also:

[sympy.functions.special.gamma\\_functions.gamma](#) (page 360) Gamma function.

[sympy.functions.special.gamma\\_functions.uppergamma](#) (page 366) Upper incomplete gamma function.

[sympy.functions.special.gamma\\_functions.lowergamma](#) (page 367) Lower incomplete gamma function.

[sympy.functions.special.gamma\\_functions.polygamma](#) (page 363) Polygamma function.

[sympy.functions.special.gamma\\_functions.loggamma](#) (page 362) Log Gamma function.

[sympy.functions.special.gamma\\_functions.digamma](#) (page 365) Digamma function.

[sympy.functions.special.gamma\\_functions.trigamma](#) (page 365) Trigamma function.

#### References

[R174] (page 1240), [R175] (page 1240), [R176] (page 1240)

#### Examples

```
>>> from sympy import I, pi
>>> from sympy.abc import x,y
```

The Beta function obeys the mirror symmetry:

```
>>> from sympy import beta
>>> from sympy import conjugate
>>> conjugate(beta(x,y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> from sympy import beta
>>> from sympy import diff
>>> diff(beta(x,y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)

>>> from sympy import beta
>>> from sympy import diff
>>> diff(beta(x,y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from sympy import beta
>>> beta(pi,pi).evalf(40)
0.02671848900111377452242355235388489324562

>>> beta(1+I,1+I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

## Error Functions and Fresnel Integrals

```
class sympy.functions.special.error_functions.erf
```

The Gauss error function. This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

See also:

[erfc](#) (page 370) Complementary error function.  
[erfi](#) (page 371) Imaginary error function.  
[erf2](#) (page 372) Two-argument error function.  
[erfinv](#) (page 373) Inverse error function.  
[erfcinv](#) (page 374) Inverse Complementary error function.  
[erf2inv](#) (page 375) Inverse two-argument error function.

## References

[R177] (page 1240), [R178] (page 1240), [R179] (page 1240), [R180] (page 1240)

## Examples

```
>>> from sympy import I, oo, erf
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erf(0)
0
>>> erf(oo)
1
```

```
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erf(z), z)
2*exp(-z**2)/sqrt(pi)
```

We can numerically evaluate the error function to arbitrary precision on the whole complex plane:

```
>>> erf(4).evalf(30)
0.99999984582742099719981147840

>>> erf(-4*I).evalf(30)
-1296959.73071763923152794095062*I
```

`class sympy.functions.special.error_functions.erfc`  
Complementary Error Function. The function is defined as:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

See also:

[erf \(page 369\)](#) Gaussian error function.  
[erfi \(page 371\)](#) Imaginary error function.  
[erf2 \(page 372\)](#) Two-argument error function.  
[erfinv \(page 373\)](#) Inverse error function.  
[erfcinv \(page 374\)](#) Inverse Complementary error function.  
[erf2inv \(page 375\)](#) Inverse two-argument error function.

## References

[R181] (page 1240), [R182] (page 1240), [R183] (page 1240), [R184] (page 1240)

## Examples

```
>>> from sympy import I, oo, erfc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfc(z), z)
-2*exp(-z**2)/sqrt(pi)
```

It also follows

```
>>> erfc(-z)
-erfc(z) + 2
```

We can numerically evaluate the complementary error function to arbitrary precision on the whole complex plane:

```
>>> erfc(4).evalf(30)
0.000000154172579002800188521596734869

>>> erfc(4*I).evalf(30)
1.0 - 1296959.73071763923152794095062*I
```

```
class sympy.functions.special.error_functions.erfi
```

Imaginary error function. The function erfi is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$$

See also:

[erf \(page 369\)](#) Gaussian error function.

[erfc \(page 370\)](#) Complementary error function.

[erf2 \(page 372\)](#) Two-argument error function.

[erfinv \(page 373\)](#) Inverse error function.

[erfcinv \(page 374\)](#) Inverse Complementary error function.

`erf2inv` (page 375) Inverse two-argument error function.

## References

[R185] (page 1240), [R186] (page 1240), [R187] (page 1240)

## Examples

```
>>> from sympy import I, oo, erfi
>>> from sympy.abc import z
```

Several special values are known:

```
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erfi(-z)
-erfi(z)

>>> from sympy import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(erfi(z), z)
2*exp(z**2)/sqrt(pi)
```

We can numerically evaluate the imaginary error function to arbitrary precision on the whole complex plane:

```
>>> erfi(2).evalf(30)
18.5648024145755525987042919132

>>> erfi(-2*I).evalf(30)
-0.995322265018952734162069256367*I
```

class `sympy.functions.special.error_functions.erf2`

Two-argument error function. This function is defined as:

$$\text{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt$$

See also:

`erf` (page 369) Gaussian error function.

[erfc](#) (page 370) Complementary error function.  
[erfi](#) (page 371) Imaginary error function.  
[erfinv](#) (page 373) Inverse error function.  
[erfcinv](#) (page 374) Inverse Complementary error function.  
[erf2inv](#) (page 375) Inverse two-argument error function.

## References

[R188] (page 1240)

## Examples

```
>>> from sympy import I, oo, erf2
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
-erf(x) + 1
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
>>> erf2(-oo, y)
erf(y) + 1
```

In general one can pull out factors of -1:

```
>>> erf2(-x, -y)
-erf2(x, y)
```

The error function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> from sympy import diff
>>> diff(erf2(x, y), x)
-2*exp(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*exp(-y**2)/sqrt(pi)
```

**class** `sympy.functions.special.error_functions.erfinv`  
Inverse Error Function. The erfinv function is defined as:

$$\operatorname{erf}(x) = y \Rightarrow \operatorname{erfinv}(y) = x$$

See also:

[erf](#) (page 369) Gaussian error function.  
[erfc](#) (page 370) Complementary error function.  
[erfi](#) (page 371) Imaginary error function.  
[erf2](#) (page 372) Two-argument error function.  
[erfcinv](#) (page 374) Inverse Complementary error function.  
[erf2inv](#) (page 375) Inverse two-argument error function.

## References

[R189] (page 1240), [R190] (page 1240)

## Examples

```
>>> from sympy import I, oo, erfinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to x is supported:

```
>>> from sympy import diff
>>> diff(erfinv(x), x)
sqrt(pi)*exp(erfinv(x)**2)/2
```

We can numerically evaluate the inverse error function to arbitrary precision on [-1, 1]:

```
>>> erfinv(0.2).evalf(30)
0.179143454621291692285822705344
```

`class sympy.functions.special.error_functions.erfcinv`  
Inverse Complementary Error Function. The erfcinv function is defined as:

$$\text{erfc}(x) = y \Rightarrow \text{erfcinv}(y) = x$$

See also:

[erf](#) (page 369) Gaussian error function.  
[erfc](#) (page 370) Complementary error function.  
[erfi](#) (page 371) Imaginary error function.  
[erf2](#) (page 372) Two-argument error function.  
[erfinv](#) (page 373) Inverse error function.  
[erf2inv](#) (page 375) Inverse two-argument error function.

## References

[R191] (page 1240), [R192] (page 1240)

## Examples

```
>>> from sympy import I, oo, erfcinv
>>> from sympy.abc import x
```

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to  $x$  is supported:

```
>>> from sympy import diff
>>> diff(erfcinv(x), x)
-sqrt(pi)*exp(erfcinv(x)**2)/2
```

class sympy.functions.special.error\_functions.erf2inv

Two-argument Inverse error function. The erf2inv function is defined as:

$$\text{erf}2(x, w) = y \Rightarrow \text{erf}2\text{inv}(x, y) = w$$

See also:

[erf](#) (page 369) Gaussian error function.

[erfc](#) (page 370) Complementary error function.

[erfi](#) (page 371) Imaginary error function.

[erf2](#) (page 372) Two-argument error function.

[erfinv](#) (page 373) Inverse error function.

[erfcinv](#) (page 374) Inverse complementary error function.

## References

[R193] (page 1240)

## Examples

```
>>> from sympy import I, oo, erf2inv, erfinv, erfcinv
>>> from sympy.abc import x, y
```

Several special values are known:

```
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
```

```
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
erfinv(y)
>>> erf2inv(oo, y)
erfcinv(-y)
```

Differentiation with respect to x and y is supported:

```
>>> from sympy import diff
>>> diff(erf2inv(x, y), x)
exp(-x**2 + erf2inv(x, y)**2)
>>> diff(erf2inv(x, y), y)
sqrt(pi)*exp(erf2inv(x, y)**2)/2
```

```
class sympy.functions.special.error_functions.FresnelIntegral
Base class for the Fresnel integrals.
```

```
class sympy.functions.special.error_functions.fresnels
Fresnel integral S.
```

This function is defined by

$$S(z) = \int_0^z \sin \frac{\pi}{2} t^2 dt.$$

It is an entire function.

See also:

[fresnelc](#) (page 377) Fresnel cosine integral.

## References

[R194] (page 1240), [R195] (page 1240), [R196] (page 1240), [R197] (page 1241), [R198] (page 1241)

## Examples

```
>>> from sympy import I, oo, fresnels
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and  $i$  from the argument:

---

```
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry  $\overline{S(z)} = S(\bar{z})$ :

```
>>> from sympy import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from sympy import integrate, pi, sin, gamma, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
>>> expand_func(integrate(sin(pi*z**2/2), z))
fresnels(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958

>>> fresnels(-2*I).evalf(30)
0.343415678363698242195300815958*I
```

**class** `sympy.functions.special.error_functions.fresnelc`  
Fresnel integral C.

This function is defined by

$$C(z) = \int_0^z \cos \frac{\pi}{2} t^2 dt.$$

It is an entire function.

**See also:**

`fresnels` (page 376) Fresnel sine integral.

## References

[R199] (page 1241), [R200] (page 1241), [R201] (page 1241), [R202] (page 1241), [R203] (page 1241)

## Examples

```
>>> from sympy import I, oo, fresnelc
>>> from sympy.abc import z
```

Several special values are known:

```
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
I/2
>>> fresnelc(-I*oo)
-I/2
```

In general one can pull out factors of  $-1$  and  $i$  from the argument:

```
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry  $\overline{C(z)} = C(\bar{z})$ :

```
>>> from sympy import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from sympy import integrate, pi, cos, gamma, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357

>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

## Exponential, Logarithmic and Trigonometric Integrals

```
class sympy.functions.special.error_functions.Ei
    The classical exponential integral.
```

For use in SymPy, this function is defined as

$$\text{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,$$

where  $\gamma$  is the Euler-Mascheroni constant.

If  $x$  is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane  $\mathbb{C} \setminus (-\infty, 0]$ .

## Background

The name *exponential integral* comes from the following statement:

$$\text{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

If the integral is interpreted as a Cauchy principal value, this statement holds for  $x > 0$  and  $\text{Ei}(x)$  as defined above.

Note that we carefully avoided defining  $\text{Ei}(x)$  for negative real  $x$ . This is because above integral formula does not hold for any polar lift of such  $x$ , indeed all branches of  $\text{Ei}(x)$  above the negative reals are imaginary.

However, the following statement holds for all  $x \in \mathbb{R}^*$ :

$$\int_{-\infty}^x \frac{e^t}{t} dt = \frac{\text{Ei}(|x|e^{i\arg(x)}) + \text{Ei}(|x|e^{-i\arg(x)})}{2},$$

where the integral is again understood to be a principal value if  $x > 0$ , and  $|x|e^{i\arg(x)}$ ,  $|x|e^{-i\arg(x)}$  denote two conjugate polar lifts of  $x$ .

**See also:**

[expint \(page 380\)](#) Generalised exponential integral.

[E1 \(page 382\)](#) Special case of the generalised exponential integral.

[li \(page 382\)](#) Logarithmic integral.

[Li \(page 383\)](#) Offset logarithmic integral.

[Si \(page 384\)](#) Sine integral.

[Ci \(page 385\)](#) Cosine integral.

[Shi \(page 387\)](#) Hyperbolic sine integral.

[Chi \(page 388\)](#) Hyperbolic cosine integral.

[sympy.functions.special.gamma\\_functions.uppergamma \(page 366\)](#) Upper incomplete gamma function.

## References

[R204] (page 1241), [R205] (page 1241), [R206] (page 1241)

## Examples

```
>>> from sympy import Ei, polar_lift, exp_polar, I, pi
>>> from sympy.abc import x
```

The exponential integral in SymPy is strictly undefined for negative values of the argument. For convenience, exponential integrals with negative arguments are immediately converted into an expression that agrees with the classical integral definition:

```
>>> Ei(-1)
-I*pi + Ei(exp_polar(I*pi))
```

This yields a real value:

```
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```
>>> Ei(x).diff(x)
exp(x)/x
```

The exponential integral is related to many other special functions. For example:

```
>>> from sympy import uppergamma, expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)

class sympy.functions.special.error_functions.expint
Generalized exponential integral.
```

This function is defined as

$$E_\nu(z) = z^{\nu-1} \Gamma(1-\nu, z),$$

where  $\Gamma(1-\nu, z)$  is the upper incomplete gamma function (`uppergamma`).

Hence for  $z$  with positive real part we have

$$E_\nu(z) = \int_1^\infty \frac{e^{-zt}}{z^\nu} dt,$$

which explains the name.

The representation as an incomplete gamma function provides an analytic continuation for  $E_\nu(z)$ . If  $\nu$  is a non-positive integer the exponential integral is thus an unbranched function of  $z$ , otherwise there is a branch point at the origin. Refer to the incomplete gamma function documentation for details of the branching behavior.

See also:

[Ei \(page 378\)](#) Another related function called exponential integral.

[E1 \(page 382\)](#) The classical case, returns  $\text{expint}(1, z)$ .

[li \(page 382\)](#) Logarithmic integral.

[Li \(page 383\)](#) Offset logarithmic integral.

[Si \(page 384\)](#) Sine integral.

[Ci](#) (page 385) Cosine integral.

[Shi](#) (page 387) Hyperbolic sine integral.

[Chi](#) (page 388) Hyperbolic cosine integral.

`sympy.functions.special.gamma_functions.uppergamma` (page 366)

## References

[R207] (page 1241), [R208] (page 1241), [R209] (page 1241)

## Examples

```
>>> from sympy import expint, S
>>> from sympy.abc import nu, z
```

Differentiation is supported. Differentiation with respect to  $z$  explains further the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to  $\nu$  has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**{nu - 1}*meijerg(((), (1, 1)), ((0, 0, -nu + 1), ()), z)
```

At non-positive integer orders, the exponential integral reduces to the exponential function:

```
>>> expint(0, z)
exp(-z)/z
>>> expint(-1, z)
exp(-z)/z + exp(-z)/z**2
```

At half-integers it reduces to error functions:

```
>>> expint(S(1)/2, z)
-sqrt(pi)*erf(sqrt(z))/sqrt(z) + sqrt(pi)/sqrt(z)
```

At positive integer orders it can be rewritten in terms of exponentials and  $\text{expint}(1, z)$ . Use `expand_func()` to do this:

```
>>> from sympy import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + (-z**3 + z**2 - 2*z + 6)*exp(-z)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> from sympy import uppergamma
>>> expint(nu, z).rewrite(uppergamma)
z**{nu - 1}*uppergamma(-nu + 1, z)
```

As such it is branched at the origin:

```
>>> from sympy import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
>>> expint(nu, z*exp_polar(2*pi*I))
z**2*(nu - 1)*(exp(2*I*pi*nu) - 1)*gamma(-nu + 1) + expint(nu, z)
```

`sympy.functions.special.error_functions.E1(z)`  
Classical case of the generalized exponential integral.

This is equivalent to `expint(1, z)`.

See also:

[Ei \(page 378\)](#) Exponential integral.  
[expint \(page 380\)](#) Generalised exponential integral.  
[li \(page 382\)](#) Logarithmic integral.  
[Li \(page 383\)](#) Offset logarithmic integral.  
[Si \(page 384\)](#) Sine integral.  
[Ci \(page 385\)](#) Cosine integral.  
[Shi \(page 387\)](#) Hyperbolic sine integral.  
[Chi \(page 388\)](#) Hyperbolic cosine integral.

```
class sympy.functions.special.error_functions.li
```

The classical logarithmic integral.

For the use in SymPy, this function is defined as

$$\text{li}(x) = \int_0^x \frac{1}{\log(t)} dt.$$

See also:

[Li \(page 383\)](#) Offset logarithmic integral.  
[Ei \(page 378\)](#) Exponential integral.  
[expint \(page 380\)](#) Generalised exponential integral.  
[E1 \(page 382\)](#) Special case of the generalised exponential integral.  
[Si \(page 384\)](#) Sine integral.  
[Ci \(page 385\)](#) Cosine integral.  
[Shi \(page 387\)](#) Hyperbolic sine integral.  
[Chi \(page 388\)](#) Hyperbolic cosine integral.

## References

[R210] (page 1241), [R211] (page 1241), [R212] (page 1241), [R213] (page 1241)

## Examples

```
>>> from sympy import I, oo, li
>>> from sympy.abc import z
```

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the  $li$  function via an integral:

The logarithmic integral can also be defined in terms of  $Ei$ :

```
>>> from sympy import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919

>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> findroot(li, 2)
1.45136923488338
```

Further transformations include rewriting  $li$  in terms of the trigonometric integrals  $Si$ ,  $Ci$ ,  $Shi$  and  $Chi$ :

```
>>> from sympy import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Ci)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Shi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
>>> li(z).rewrite(Chi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
```

```
class sympy.functions.special.error_functions.Li
```

The offset logarithmic integral.

For the use in SymPy, this function is defined as

$$\text{Li}(x) = \text{li}(x) - \text{li}(2)$$

**See also:**

[li \(page 382\)](#) Logarithmic integral.

[Ei \(page 378\)](#) Exponential integral.

[expint \(page 380\)](#) Generalised exponential integral.

[E1 \(page 382\)](#) Special case of the generalised exponential integral.

[Si \(page 384\)](#) Sine integral.

[Ci \(page 385\)](#) Cosine integral.

[Shi \(page 387\)](#) Hyperbolic sine integral.

[Chi \(page 388\)](#) Hyperbolic cosine integral.

## References

[R214] (page 1241), [R215] (page 1241), [R216] (page 1241)

## Examples

```
>>> from sympy import I, oo, Li  
>>> from sympy.abc import z
```

The following special value is known:

```
>>> Li(2)  
0
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff  
>>> diff(Li(z), z)  
1/log(z)
```

The shifted logarithmic integral can be written in terms of  $li(z)$ :

```
>>> from sympy import li  
>>> Li(z).rewrite(li)  
li(z) - li(2)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> Li(2).evalf(30)  
0  
  
>>> Li(4).evalf(30)  
1.92242131492155809316615998938
```

```
class sympy.functions.special.error_functions.Si
Sine integral.
```

This function is defined by

$$\text{Si}(z) = \int_0^z \frac{\sin t}{t} dt.$$

It is an entire function.

See also:

[Ci](#) (page 385) Cosine integral.

[Shi](#) (page 387) Hyperbolic sine integral.

[Chi](#) (page 388) Hyperbolic cosine integral.

[Ei](#) (page 378) Exponential integral.

[expint](#) (page 380) Generalised exponential integral.

[E1](#) (page 382) Special case of the generalised exponential integral.

[li](#) (page 382) Logarithmic integral.

[Li](#) (page 383) Offset logarithmic integral.

## References

[R217] (page 1241)

## Examples

```
>>> from sympy import Si
>>> from sympy.abc import z
```

The sine integral is an antiderivative of  $\sin(z)/z$ :

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by  $I$ :

```
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
    expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

```
class sympy.functions.special.error_functions.Ci
    Cosine integral.
```

This function is defined for positive  $x$  by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt = - \int_x^\infty \frac{\cos t}{t} dt,$$

where  $\gamma$  is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = -\frac{\text{E}_1(e^{i\pi/2}z) + \text{E}_1(e^{-i\pi/2}z)}{2}$$

which holds for all polar  $z$  and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for  $z \in \mathbb{C}$  with  $\Re(z) > 0$ . By lifting to the principal branch we obtain an analytic function on the cut complex plane.

#### See also:

[Si \(page 384\)](#) Sine integral.

[Shi \(page 387\)](#) Hyperbolic sine integral.

[Chi \(page 388\)](#) Hyperbolic cosine integral.

[Ei \(page 378\)](#) Exponential integral.

[expint \(page 380\)](#) Generalised exponential integral.

[E1 \(page 382\)](#) Special case of the generalised exponential integral.

[li \(page 382\)](#) Logarithmic integral.

[Li \(page 383\)](#) Offset logarithmic integral.

#### References

[R218] (page 1241)

#### Examples

```
>>> from sympy import Ci
>>> from sympy.abc import z
```

The cosine integral is a primitive of  $\cos(z)/z$ :

```
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary cos under multiplication by  $i$ :

```
>>> from sympy import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

class `sympy.functions.special.error_functions.Shi`  
Sinh integral.

This function is defined by

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt.$$

It is an entire function.

See also:

[Si \(page 384\)](#) Sine integral.

[Ci \(page 385\)](#) Cosine integral.

[Chi \(page 388\)](#) Hyperbolic cosine integral.

[Ei \(page 378\)](#) Exponential integral.

[expint \(page 380\)](#) Generalised exponential integral.

[E1 \(page 382\)](#) Special case of the generalised exponential integral.

[li \(page 382\)](#) Logarithmic integral.

[Li \(page 383\)](#) Offset logarithmic integral.

## References

[R219] (page 1241)

## Examples

```
>>> from sympy import Shi
>>> from sympy.abc import z
```

The Sinh integral is a primitive of  $\sinh(z)/z$ :

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> from sympy import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary sinh under multiplication by  $i$ :

```
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from sympy import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

class `sympy.functions.special.error_functions.Chi`  
Cosh integral.

This function is defined for positive  $x$  by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt,$$

where  $\gamma$  is the Euler-Mascheroni constant.

We have

$$\text{Chi}(z) = \text{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

which holds for all polar  $z$  and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

See also:

[Si \(page 384\)](#) Sine integral.

[Ci \(page 385\)](#) Cosine integral.

[Shi \(page 387\)](#) Hyperbolic sine integral.

[Ei \(page 378\)](#) Exponential integral.

[expint \(page 380\)](#) Generalised exponential integral.

[E1 \(page 382\)](#) Special case of the generalised exponential integral.

[li \(page 382\)](#) Logarithmic integral.

[Li \(page 383\)](#) Offset logarithmic integral.

## References

[R220] (page 1241)

## Examples

```
>>> from sympy import Chi
>>> from sympy.abc import z
```

The cosh integral is a primitive of  $\cosh(z)/z$ :

---

```
>>> Chi(z).diff(z)
cosh(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from sympy import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by  $i$ :

```
>>> from sympy import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from sympy import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

## Bessel Type Functions

```
class sympy.functions.special.bessel.BesselBase
Abstract base class for bessel-type functions.
```

This class is meant to reduce code duplication. All Bessel type functions can 1) be differentiated, and the derivatives expressed in terms of similar functions and 2) be rewritten in terms of other bessel-type functions.

Here “bessel-type functions” are assumed to have one complex parameter.

To use this base class, define class attributes `_a` and `_b` such that  $2*F_n' = -_a*F_{n+1} + b*F_{n-1}$ .

### `argument`

The argument of the bessel-type function.

### `order`

The order of the bessel-type function.

```
class sympy.functions.special.bessel.besselj
Bessel function of the first kind.
```

The Bessel  $J$  function of order  $\nu$  is defined to be the function satisfying Bessel’s differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left( \frac{1}{\Gamma(\nu + 1)2^\nu} + O(z^2) \right),$$

if  $\nu$  is not a negative integer. If  $\nu = -n \in \mathbb{Z}_{<0}$  is a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

See also:

[bessely](#) (page 390), [besseli](#) (page 391), [besselk](#) (page 391)

## References

[R221] (page 1241), [R222] (page 1241), [R223] (page 1241), [R224] (page 1241)

## Examples

Create a Bessel function object:

```
>>> from sympy import besselj, jn
>>> from sympy.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```

Rewrite in terms of spherical Bessel functions:

```
>>> b.rewrite(jn)
sqrt(2)*sqrt(z)*jn(n - 1/2, z)/sqrt(pi)
```

Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

```
class sympy.functions.special.bessel.bessely
Bessel function of the second kind.
```

The Bessel  $Y$  function of order  $\nu$  is defined as

$$Y_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{J_\mu(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where  $J_\mu(z)$  is the Bessel function of the first kind.

It is a solution to Bessel's equation, and linearly independent from  $J_\nu$ .

See also:

[besselj](#) (page 389), [besseli](#) (page 391), [besselk](#) (page 391)

## References

[R225] (page 1241)

## Examples

```
>>> from sympy import bessely, yn
>>> from sympy.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
bessely(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

**class sympy.functions.special.bessel.besseli**  
Modified Bessel function of the first kind.

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2)^2 w = 0.$$

It can be defined as

$$I_\nu(z) = i^{-\nu} J_\nu(iz),$$

where  $J_\nu(z)$  is the Bessel function of the first kind.

**See also:**

[besselj](#) (page 389), [bessely](#) (page 390), [besselk](#) (page 391)

## References

[R226] (page 1241)

## Examples

```
>>> from sympy import besseli
>>> from sympy.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```

**class sympy.functions.special.bessel.besselk**  
Modified Bessel function of the second kind.

The Bessel K function of order  $\nu$  is defined as

$$K_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

where  $I_\mu(z)$  is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from  $Y_\nu$ .

**See also:**

[besselj](#) (page 389), [besseli](#) (page 391), [bessely](#) (page 390)

## References

[R227] (page 1241)

## Examples

```
>>> from sympy import besselk
>>> from sympy.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

```
class sympy.functions.special.bessel.hankel1
Hankel function of the first kind.
```

This function is defined as

$$H_{\nu}^{(1)} = J_{\nu}(z) + iY_{\nu}(z),$$

where  $J_{\nu}(z)$  is the Bessel function of the first kind, and  $Y_{\nu}(z)$  is the Bessel function of the second kind.

It is a solution to Bessel's equation.

### See also:

[hankel2](#) (page 392), [besselj](#) (page 389), [bessely](#) (page 390)

## References

[R228] (page 1241)

## Examples

```
>>> from sympy import hankel1
>>> from sympy.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

```
class sympy.functions.special.bessel.hankel2
Hankel function of the second kind.
```

This function is defined as

$$H_{\nu}^{(2)} = J_{\nu}(z) - iY_{\nu}(z),$$

where  $J_{\nu}(z)$  is the Bessel function of the first kind, and  $Y_{\nu}(z)$  is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from  $H_{\nu}^{(1)}$ .

### See also:

[hankel1](#) (page 392), [besselj](#) (page 389), [bessely](#) (page 390)

## References

[R229] (page 1241)

## Examples

```
>>> from sympy import hankel2
>>> from sympy.abc import z, n
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

**class sympy.functions.special.bessel.jn**  
Spherical Bessel function of the first kind.

This function is a solution to the spherical Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where  $J_\nu(z)$  is the Bessel function of the first kind.

**See also:**

[besselj](#) (page 389), [bessely](#) (page 390), [besselk](#) (page 391), [yn](#) (page 393)

### Examples

```
>>> from sympy import Symbol, jn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print(jn(0, z).expand(func=True))
sin(z)/z
>>> jn(1, z).expand(func=True) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
```

The spherical Bessel functions of integral order are calculated using the formula:

$$j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z,$$

where the coefficients  $f_n(z)$  are available as `sympy.polys.orthopolys.spherical_bessel_fn()` (page 723).

**class sympy.functions.special.bessel.yn**  
Spherical Bessel function of the second kind.

This function is another solution to the spherical Bessel equation, and linearly independent from  $j_n$ . It can be defined as

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where  $Y_\nu(z)$  is the Bessel function of the second kind.

**See also:**

[besselj](#) (page 389), [bessely](#) (page 390), [besselk](#) (page 391), [jn](#) (page 393)

### Examples

```
>>> from sympy import Symbol, yn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2-sin(z)/z
True
```

For integral orders  $n$ ,  $y_n$  is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

```
sympy.functions.special.bessel.jn_zeros(n, k, method='sympy', dps=15)
Zeros of the spherical Bessel function of the first kind.
```

This returns an array of zeros of  $j_n$  up to the  $k$ -th zero.

- `method = "sympy"`: uses mpmath's function `besseljzero`
- `method = "scipy"`: uses the SciPy's `sph_jn` and `newton` to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. [The function used with `method="sympy"` is a recent addition to mpmath, before that a general solver was used.]

See also:

[jn](#) (page 393), [yn](#) (page 393), [besselj](#) (page 389), [besselk](#) (page 391), [bessely](#) (page 390)

## Examples

```
>>> from sympy import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

## Airy Functions

```
class sympy.functions.special.bessel.AiryBase
Abstract base class for Airy functions.
```

This class is meant to reduce code duplication.

```
class sympy.functions.special.bessel.airyai
The Airy function  $A_i$  of the first kind.
```

The Airy function  $A_i(z)$  is defined to be the function satisfying Airy's differential equation

$$\frac{d^2w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real  $z$

$$Ai(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

[airybi](#) (page 395) Airy function of the second kind.

[airyaiprime](#) (page 397) Derivative of the Airy function of the first kind.

[airybiprime](#) (page 398) Derivative of the Airy function of the second kind.

## References

[R230] (page 1241), [R231] (page 1241), [R232] (page 1242), [R233] (page 1242)

## Examples

Create an Airy function object:

```
>>> from sympy import airyai
>>> from sympy.abc import z

>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
3**((1/3)/(3*gamma(2/3)))
>>> from sympy import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyai(z), z, 0, 3)
3**((5/6)*gamma(1/3)/(6*pi) - 3**((1/6)*z*gamma(2/3)/(2*pi) + O(z**3))
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite  $\text{Ai}(z)$  in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyai(z).rewrite(hyper)
-3**((2/3)*z*hyper((), (4/3,), z**3/9)/(3*gamma(1/3)) + 3**((1/3)*hyper((), (2/3,), z**3/9)/(3*gamma(2/3)))
```

```
class sympy.functions.special.bessel.airybi
The Airy function Bi of the second kind.
```

The Airy function  $\text{Bi}(z)$  is defined to be the function satisfying Airy's differential equation

$$\frac{d^2w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real  $z$

$$\text{Bi}(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

[airyai](#) (page 394) Airy function of the first kind.

[airyaprime](#) (page 397) Derivative of the Airy function of the first kind.

[airybiprime](#) (page 398) Derivative of the Airy function of the second kind.

## References

[R234] (page 1242), [R235] (page 1242), [R236] (page 1242), [R237] (page 1242)

## Examples

Create an Airy function object:

```
>>> from sympy import airybi
>>> from sympy.abc import z

>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
3**((5/6)/(3*gamma(2/3)))
>>> from sympy import oo
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

---

```
>>> from sympy import series
>>> series(airybi(z), z, 0, 3)
3** (1/3)*gamma(1/3)/(2*pi) + 3** (2/3)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite  $\text{Bi}(z)$  in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybi(z).rewrite(hyper)
3** (1/6)*z*hyper((), (4/3,), z**3/9)/gamma(1/3) + 3** (5/6)*hyper((), (2/3,), z**3/9)/(3*gamma(2/3))
```

`class sympy.functions.special.bessel.airyaiprime`

The derivative  $\text{Ai}'$  of the Airy function of the first kind.

The Airy function  $\text{Ai}'(z)$  is defined to be the function

$$\text{Ai}'(z) := \frac{d \text{Ai}(z)}{dz}.$$

See also:

`airyai` (page 394) Airy function of the first kind.

`airybi` (page 395) Airy function of the second kind.

`airybiprime` (page 398) Derivative of the Airy function of the second kind.

## References

[R238] (page 1242), [R239] (page 1242), [R240] (page 1242), [R241] (page 1242)

## Examples

Create an Airy function object:

```
>>> from sympy import airyaiprime
>>> from sympy.abc import z

>>> airyaiprime(z)
airyaiprime(z)
```

Several special values are known:

```
>>> airyaiprime(0)
-3** (2/3)/(3*gamma(1/3))
>>> from sympy import oo
>>> airyaiprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airyaiprime(z))
airyaiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from sympy import diff
>>> diff(airyaiprime(z), z)
z*airyai(z)
>>> diff(airyaiprime(z), z, 2)
z*airyaiprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airyaiprime(z), z, 0, 3)
-3**{2/3}/(3*gamma(1/3)) + 3**{1/3}*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyaiprime(-2).evalf(50)
0.61825902074169104140626429133247528291577794512415
```

Rewrite  $\text{Ai}'(z)$  in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airyaiprime(z).rewrite(hyper)
3**{1/3}*z**2*hyper(((), (5/3, ), z**3/9)/(6*gamma(2/3)) - 3**{2/3}*hyper(((), (1/3, ), z**3/9)/(3*gamma(1/3)))
```

```
class sympy.functions.special.bessel.airybiprime
The derivative Bi' of the Airy function of the first kind.
```

The Airy function  $\text{Bi}'(z)$  is defined to be the function

$$\text{Bi}'(z) := \frac{d\text{Bi}(z)}{dz}.$$

See also:

[airyai](#) (page 394) Airy function of the first kind.

[airybi](#) (page 395) Airy function of the second kind.

[airyaiprime](#) (page 397) Derivative of the Airy function of the first kind.

## References

[R242] (page 1242), [R243] (page 1242), [R244] (page 1242), [R245] (page 1242)

## Examples

Create an Airy function object:

```
>>> from sympy import airybiprime
>>> from sympy.abc import z

>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
3**(1/6)/gamma(1/3)
>>> from sympy import oo
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from sympy import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to  $z$  is supported:

```
>>> from sympy import diff
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> from sympy import series
>>> series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**5/6*z**2/(6*gamma(2/3)) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybiprime(-2).evalf(50)
0.27879516692116952268509756941098324140300059345163
```

Rewrite  $\text{Bi}'(z)$  in terms of hypergeometric functions:

```
>>> from sympy import hyper
>>> airybiprime(z).rewrite(hyper)
3**5/6*z**2*hyper(((), (5/3,), z**3/9)/(6*gamma(2/3)) + 3**1/6*hyper(((), (1/3,), z**3/9)/gamma(1/3))
```

## B-Splines

```
sympy.functions.special.bsplines.bspline_basis(d, knots, n, x, close=True)
```

The  $n$ -th B-spline at  $x$  of degree  $d$  with knots.

B-Splines are piecewise polynomials of degree  $d$  [R246] (page 1242). They are defined on a set of knots, which is a sequence of integers or floats.

The 0th degree splines have a value of one on a single interval:

```
>>> from sympy import bspline_basis
>>> from sympy.abc import x
>>> d = 0
>>> knots = range(5)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, And(x <= 1, x >= 0)), (0, True))
```

For a given  $(d, \text{knots})$  there are  $\text{len}(\text{knots}) - d - 1$  B-splines defined, that are indexed by  $n$  (starting at 0).

Here is an example of a cubic B-spline:

```
>>> bspline_basis(3, range(5), 0, x)
Piecewise((x**3/6, And(x < 1, x >= 0)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3, And(x < 2, x >= 1)),
          (x**3/2 - 4*x**2 + 10*x - 22/3, And(x < 3, x >= 2)),
          (-x**3/6 + 2*x**2 - 8*x + 32/3, And(x <= 4, x >= 3)),
          (0, True))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = [0,0,2,3,4]
>>> bspline_basis(d, knots, 0, x)
Piecewise((-x/2 + 1, And(x <= 2, x >= 0)), (0, True))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-splines many times, it is best to lambdify them first:

```
>>> from sympy import lambdify
>>> d = 3
>>> knots = range(10)
>>> b0 = bspline_basis(d, knots, 0, x)
>>> f = lambdify(x, b0)
>>> y = f(0.5)
```

See also:

[sympy.functions.special.bsplines.bspline\\_basis\\_set](#) (page 400)

## References

[R246] (page 1242)

[sympy.functions.special.bsplines.bspline\\_basis\\_set\(d, knots, x\)](#)  
Return the len(knots)-d-1 B-splines at x of degree d with knots.

This function returns a list of Piecewise polynomials that are the len(knots)-d-1 B-splines of degree d for the given knots. This function calls `bspline_basis(d, knots, n, x)` for different values of n.

See also:

[sympy.functions.special.bsplines.bspline\\_basis](#) (page 399)

## Examples

```
>>> from sympy import bspline_basis_set
>>> from sympy.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, And(x < 1, x >= 0)),
           (-x**2 + 3*x - 3/2, And(x < 2, x >= 1)),
           (x**2/2 - 3*x + 9/2, And(x <= 3, x >= 2)),
           (0, True)),
 Piecewise((x**2/2 - x + 1/2, And(x < 2, x >= 1)),
           (-x**2 + 5*x - 11/2, And(x < 3, x >= 2))),
```

---

```
(x**2/2 - 4*x + 8, And(x <= 4, x >= 3)),
(0, True))]
```

## Riemann Zeta and Related Functions

`class sympy.functions.special.zeta_functions.zeta`  
Hurwitz zeta function (or Riemann zeta function).

For  $\operatorname{Re}(a) > 0$  and  $\operatorname{Re}(s) > 1$ , this function is defined as

$$\zeta(s, a) = \sum_{n=0}^{\infty} \frac{1}{(n+a)^s},$$

where the standard choice of argument for  $n+a$  is used. For fixed  $a$  with  $\operatorname{Re}(a) > 0$  the Hurwitz zeta function admits a meromorphic continuation to all of  $\mathbb{C}$ , it is an unbranched function with a simple pole at  $s = 1$ .

Analytic continuation to other  $a$  is possible under some circumstances, but this is not typically done.

The Hurwitz zeta function is a special case of the Lerch transcendent:

$$\zeta(s, a) = \Phi(1, s, a).$$

This formula defines an analytic continuation for all possible values of  $s$  and  $a$  (also  $\operatorname{Re}(a) < 0$ ), see the documentation of `lerchphi` (page 404) for a description of the branching behavior.

If no value is passed for  $a$ , by this function assumes a default value of  $a = 1$ , yielding the Riemann zeta function.

**See also:**

`dirichlet_eta` (page 402), `lerchphi` (page 404), `polylog` (page 403)

## References

[R247] (page 1242), [R248] (page 1242)

## Examples

For  $a = 1$  the Hurwitz zeta function reduces to the famous Riemann zeta function:

$$\zeta(s, 1) = \zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

```
>>> from sympy import zeta
>>> from sympy.abc import s
>>> zeta(s, 1)
zeta(s)
>>> zeta(s)
zeta(s)
```

The Riemann zeta function can also be expressed using the Dirichlet eta function:

```
>>> from sympy import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(-2**(-s + 1) + 1)
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n}(2\pi)^{2n}}{2(2n)!}$$

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}$$

At negative even integers the Riemann zeta function is zero:

```
>>> zeta(-4)
0
```

No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```
>>> zeta(3).n()
1.20205690315959
```

The derivative of  $\zeta(s, a)$  with respect to  $a$  is easily computed:

```
>>> from sympy.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to  $s$  has no useful closed form expression:

```
>>> zeta(s, a).diff(s)
Derivative(zeta(s, a), s)
```

The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, `sympy.functions.special.zeta_functions.lerchphi` (page 404):

```
>>> from sympy import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

`class sympy.functions.special.zeta_functions.dirichlet_eta`  
Dirichlet eta function.

For  $\text{Re}(s) > 0$ , this function is defined as

$$\eta(s) = \sum_{n=1}^{\infty} \frac{(-1)^n}{n^s}.$$

It admits a unique analytic continuation to all of  $\mathbb{C}$ . It is an entire, unbranched function.

**See also:**

`zeta` (page 401)

## References

[R249] (page 1242)

## Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from sympy import dirichlet_eta, zeta
>>> from sympy.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(-2**(-s + 1) + 1)*zeta(s)

class sympy.functions.special.zeta_functions.polylog
Polylogarithm function.
```

For  $|z| < 1$  and  $s \in \mathbb{C}$ , the polylogarithm is defined by

$$\text{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for  $n$ . It admits an analytic continuation which is branched at  $z = 1$  (notably not on the sheet of initial definition),  $z = 0$  and  $z = \infty$ .

The name polylogarithm comes from the fact that for  $s = 1$ , the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\text{Li}_{s+1}(z) = \int_0^z \frac{\text{Li}_s(t)}{t} dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\text{Li}_s(z) = z\Phi(z, s, 1)$$

## See also:

[zeta](#) (page 401), [lerchphi](#) (page 404)

## Examples

For  $z \in \{0, 1, -1\}$ , the polylogarithm is automatically expressed using other functions:

```
>>> from sympy import polylog
>>> from sympy.abc import s
>>> polylog(s, 0)
0
>>> polylog(s, 1)
zeta(s)
>>> polylog(s, -1)
dirichlet_eta(s)
```

If  $s$  is a negative integer, 0 or 1, the polylogarithm can be expressed using elementary functions. This can be done using `expand_func()`:

```
>>> from sympy import expand_func
>>> from sympy.abc import z
>>> expand_func(polylog(1, z))
-log(z*exp_polar(-I*pi) + 1)
>>> expand_func(polylog(0, z))
z/(-z + 1)
```

The derivative with respect to  $z$  can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> from sympy import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

```
class sympy.functions.special.zeta_functions.lerchphi
Lerch transcendent (Lerch phi function).
```

For  $\operatorname{Re}(a) > 0$ ,  $|z| < 1$  and  $s \in \mathbb{C}$ , the Lerch transcendent is defined as

$$\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s},$$

where the standard branch of the argument is used for  $n+a$ , and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

$$L(q, s, a) = \Phi(e^{2\pi i q}, s, a).$$

### Analytic Continuation and Branching Behavior

It can be shown that

$$\Phi(z, s, a) = z\Phi(z, s, a+1) + a^{-s}.$$

This provides the analytic continuation to  $\operatorname{Re}(a) \leq 0$ .

Assume now  $\operatorname{Re}(a) > 0$ . The integral representation

$$\Phi_0(z, s, a) = \int_0^\infty \frac{t^{s-1} e^{-at}}{1 - ze^{-t}} \frac{dt}{\Gamma(s)}$$

provides an analytic continuation to  $\mathbb{C} - [1, \infty)$ . Finally, for  $x \in (1, \infty)$  we find

$$\lim_{\epsilon \rightarrow 0^+} \Phi_0(x + i\epsilon, s, a) - \lim_{\epsilon \rightarrow 0^+} \Phi_0(x - i\epsilon, s, a) = \frac{2\pi i \log^{s-1} x}{x^a \Gamma(s)},$$

using the standard branch for both  $\log x$  and  $\log \log x$  (a branch of  $\log \log x$  is needed to evaluate  $\log x^{s-1}$ ). This concludes the analytic continuation. The Lerch transcendent is thus branched at  $z \in \{0, 1, \infty\}$  and  $a \in \mathbb{Z}_{\leq 0}$ . For fixed  $z, a$  outside these branch points, it is an entire function of  $s$ .

**See also:**

[polylog](#) (page 403), [zeta](#) (page 401)

### References

[R250] (page 1242), [R251] (page 1242), [R252] (page 1242)

## Examples

The Lerch transcendent is a fairly general function, for this reason it does not automatically evaluate to simpler functions. Use `expand_func()` to achieve this.

If  $z = 1$ , the Lerch transcendent reduces to the Hurwitz zeta function:

```
>>> from sympy import lerchphi, expand_func
>>> from sympy.abc import z, s, a
>>> expand_func(lerchphi(1, s, a))
zeta(s, a)
```

More generally, if  $z$  is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:

```
>>> expand_func(lerchphi(-1, s, a))
2**(-s)*zeta(s, a/2) - 2**(-s)*zeta(s, a/2 + 1/2)
```

If  $a = 1$ , the Lerch transcendent reduces to the polylogarithm:

```
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if  $a$  is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```
>>> from sympy import S
>>> expand_func(lerchphi(z, s, S(1)/2))
2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
            polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))
>>> expand_func(lerchphi(z, s, S(3)/2))
-2**s/z + 2**(s - 1)*(polylog(s, sqrt(z))/sqrt(z) -
                        polylog(s, sqrt(z)*exp_polar(I*pi))/sqrt(z))/z
```

The derivatives with respect to  $z$  and  $a$  can be computed in closed form:

```
>>> lerchphi(z, s, a).diff(z)
(-a*lerchphi(z, s, a) + lerchphi(z, s - 1, a))/z
>>> lerchphi(z, s, a).diff(a)
-s*lerchphi(z, s + 1, a)
```

## Hypergeometric Functions

```
class sympy.functions.special.hyper.hyper
```

The (generalized) hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index. When convergent, it is continued analytically to the largest possible domain.

The hypergeometric function depends on two vectors of parameters, called the numerator parameters  $a_p$ , and the denominator parameters  $b_q$ . It also has an argument  $z$ . The series definition is

$${}_pF_q \left( \begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!},$$

where  $(a)_n = (a)(a+1)\dots(a+n-1)$  denotes the rising factorial.

If one of the  $b_q$  is a non-positive integer then the series is undefined unless one of the  $a_p$  is a larger (i.e. smaller in magnitude) non-positive integer. If none of the  $b_q$  is a non-positive integer and one of the  $a_p$  is a non-positive integer, then the series reduces to a polynomial. To simplify the following discussion, we assume that none of the  $a_p$  or  $b_q$  is a non-positive integer. For more details, see the references.

The series converges for all  $z$  if  $p \leq q$ , and thus defines an entire single-valued function in this case. If  $p = q + 1$  the series converges for  $|z| < 1$ , and can be continued analytically into a half-plane. If  $p > q + 1$  the series is divergent for all  $z$ .

Note: The hypergeometric function constructor currently does *not* check if the parameters actually yield a well-defined function.

#### See also:

`sympy.simplify.hyperexpand` (page 935), `sympy.functions.special.gamma_functions.gamma` (page 360), `sympy.functions.special.hyper.meijerg` (page 407)

#### References

[R253] (page 1242), [R254] (page 1242)

#### Examples

The parameters  $a_p$  and  $b_q$  can be passed as arbitrary iterables, for example:

```
>>> from sympy.functions import hyper
>>> from sympy.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using unicode):

```
>>> from sympy import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)
      - /1, 2, 3 | \
      | |           | x|
3 2 \ 3, 4 | /
```

The parameters must always be iterables, even if they are vectors of length one or zero:

```
>>> hyper((1, ), [], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on  $x$  then you should not expect much implemented functionality):

```
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```
>>> from sympy import hyperexpand
>>> hyperexpand(hyper([], [], x))
exp(x)
```

You can also use `expand_func`:

```
>>> from sympy import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> from sympy import S
>>> hyperexpand(hyper([], [S(1)/2], -x**2/4))
cos(x)
>>> hyperexpand(x*hyper([S(1)/2, S(1)/2], [S(3)/2], x**2))
asin(x)
```

We can also sometimes hyperexpand parametric functions:

```
>>> from sympy.abc import a
>>> hyperexpand(hyper([-a, [], x)))
(-x + 1)**a
```

**ap**

Numerator parameters of the hypergeometric function.

**argument**

Argument of the hypergeometric function.

**bq**

Denominator parameters of the hypergeometric function.

**convergence\_statement**

Return a condition on z under which the series converges.

**eta**

A quantity related to the convergence of the series.

**radius\_of\_convergence**

Compute the radius of convergence of the defining series.

Note that even if this is not oo, the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

```
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
oo
```

**class sympy.functions.special.hyper.meijerg**

The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

The Meijer G-function depends on four sets of parameters. There are “*numerator parameters*”  $a_1, \dots, a_n$  and  $a_{n+1}, \dots, a_p$ , and there are “*denominator parameters*”  $b_1, \dots, b_m$  and  $b_{m+1}, \dots, b_q$ . Confusingly, it is traditionally denoted as follows (note the position of  $m, n, p, q$ , and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left( \begin{matrix} a_1, \dots, a_n & a_{n+1}, \dots, a_p \\ b_1, \dots, b_m & b_{m+1}, \dots, b_q \end{matrix} \middle| z \right).$$

However, in sympy the four parameter vectors are always available separately (see examples), so that there is no need to keep track of the decorating sub- and super-scripts on the G symbol.

The G function is defined as the following integral:

$$\frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where  $\Gamma(z)$  is the gamma function. There are three possible contours which we will not describe in detail here (see the references). If the integral converges along more than one of them the definitions agree. The contours all separate the poles of  $\Gamma(1 - a_j + s)$  from the poles of  $\Gamma(b_k - s)$ , so in particular the G function is undefined if  $a_j - b_k \in \mathbb{Z}_{>0}$  for some  $j \leq n$  and  $k \leq m$ .

The conditions under which one of the contours yields a convergent integral are complicated and we do not state them here, see the references.

Note: Currently the Meijer G-function constructor does *not* check any convergence conditions.

#### See also:

[sympy.functions.special.hyper.hyper](#) (page 405), [sympy.simplify.hyperexpand](#) (page 935)

#### References

[R255] (page 1242), [R256] (page 1242)

#### Examples

You can pass the parameters either as four separate vectors:

```
>>> from sympy.functions import meijerg
>>> from sympy.abc import x, a
>>> from sympy.core.containers import Tuple
>>> from sympy import pprint
>>> pprint(meijerg((1, 2), (a, 4), (5,), [], x), use_unicode=False)
  _1, 2 /1, 2 a, 4 | \
/_ |           | x|
\_|4, 1 \ 5       | /
```

or as two nested vectors:

```
>>> pprint(meijerg([(1, 2), (3, 4)], ([5], Tuple()), x), use_unicode=False)
  _1, 2 /1, 2 3, 4 | \
/_ |           | x|
\_|4, 1 \ 5       | /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:

```
>>> from sympy import pprint
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g, use_unicode=False)
  _1, 1 /1 2 | \
/_ |           | x|
\_|2, 2 \3 4 | /
>>> g.an
(1,)
>>> g.ap
```

```
(1, 2)
>>> g.aother
(2,)
>>> g.bm
(3,)
>>> g.bq
(3, 4)
>>> g.bother
(4,)
```

The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater's theorem. For example:

```
>>> from sympy import hyperexpand
>>> from sympy.abc import a, b, c
>>> hyperexpand(meijerg([a], [], [c], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,), (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use expand\_func or hyperexpand to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```
>>> from sympy import expand_func, S
>>> expand_func(meijerg([],[], [[0],[]], -x))
exp(x)
>>> hyperexpand(meijerg([],[], [[S(1)/2],[0]], (x/2)**2))
sin(x)/sqrt(pi)
```

**an**

First set of numerator parameters.

**aother**

Second set of numerator parameters.

**ap**

Combined numerator parameters.

**argument**

Argument of the Meijer G-function.

**bm**

First set of denominator parameters.

**bother**

Second set of denominator parameters.

**bq**

Combined denominator parameters.

**delta**

A quantity related to the convergence region of the integral, c.f. references.

**get\_period()**

Return a number P such that  $G(x \cdot \exp(IP)) == G(x)$ .

```
>>> from sympy.functions.special.hyper import meijerg
>>> from sympy.abc import z
>>> from sympy import pi, S
```

```
>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1,1], [2], [1, S(1)/2, S(1)/3], [1], z).get_period()
12*pi
```

### integrand(*s*)

Get the defining integrand D(s).

### nu

A quantity related to the convergence region of the integral, c.f. references.

## Elliptic integrals

**class** `sympy.functions.special.elliptic_integrals.elliptic_k`

The complete elliptic integral of the first kind, defined by

$$K(z) = F\left(\frac{\pi}{2} \middle| z\right)$$

where  $F(z|m)$  is the Legendre incomplete elliptic integral of the first kind.

The function  $K(z)$  is a single-valued function on the complex plane with branch cut along the interval  $(1, \infty)$ .

**See also:**

`elliptic_f` (page 410)

## References

[R257] (page 1242), [R258] (page 1242)

## Examples

```
>>> from sympy import elliptic_k, I, pi
>>> from sympy.abc import z
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(z).series(z, n=3)
pi/2 + pi*z/8 + 9*pi*z**2/128 + O(z**3)
```

**class** `sympy.functions.special.elliptic_integrals.elliptic_f`

The Legendre incomplete elliptic integral of the first kind, defined by

$$F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

This function reduces to a complete elliptic integral of the first kind,  $K(m)$ , when  $z = \pi/2$ .

**See also:**

`elliptic_k` (page 410)

## References

[R259] (page 1242), [R260] (page 1242)

## Examples

```
>>> from sympy import elliptic_f, I, 0
>>> from sympy.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I
```

**class** `sympy.functions.special.elliptic_integrals.elliptic_e`

Called with two arguments  $z$  and  $m$ , evaluates the incomplete elliptic integral of the second kind, defined by

$$E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} dt$$

Called with a single argument  $z$ , evaluates the Legendre complete elliptic integral of the second kind

$$E(z) = E\left(\frac{\pi}{2}|z\right)$$

The function  $E(z)$  is a single-valued function on the complex plane with branch cut along the interval  $(1, \infty)$ .

## References

[R261] (page 1242), [R262] (page 1242), [R263] (page 1242)

## Examples

```
>>> from sympy import elliptic_e, I, pi, 0
>>> from sympy.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + O(z**6)
>>> elliptic_e(z).series(z, n=4)
pi/2 - pi*z/8 - 3*pi*z**2/128 - 5*pi*z**3/512 + O(z**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

**class** `sympy.functions.special.elliptic_integrals.elliptic_pi`

Called with three arguments  $n$ ,  $z$  and  $m$ , evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi(n; z|m) = \int_0^z \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}}$$

Called with two arguments  $n$  and  $m$ , evaluates the complete elliptic integral of the third kind:

$$\Pi(n|m) = \Pi\left(n; \frac{\pi}{2}|m\right)$$

## References

[R264] (page 1242), [R265] (page 1242), [R266] (page 1242)

## Examples

```
>>> from sympy import elliptic_pi, I, pi, 0, S
>>> from sympy.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + O(z**4)
>>> elliptic_pi(0.5 + I, 1.0 - I, 1.2)
2.50232379629182 - 0.760939574180767*I
>>> elliptic_pi(0, 0)
pi/2
>>> elliptic_pi(1.0 - I/3, 2.0 + I)
3.29136443417283 + 0.32555634906645*I
```

## Orthogonal Polynomials

This module mainly implements special orthogonal polynomials.

See also `functions.combinatorial.numbers` which contains some combinatorial polynomials.

### Jacobi Polynomials

```
class sympy.functions.special.polynomials.jacobi
Jacobi polynomial  $P_n^{(\alpha, \beta)}(x)$ 
```

`jacobi(n, alpha, beta, x)` gives the nth Jacobi polynomial in  $x$ ,  $P_n^{(\alpha, \beta)}(x)$ .

The Jacobi polynomials are orthogonal on  $[-1, 1]$  with respect to the weight  $(1 - x)^\alpha (1 + x)^\beta$ .

See also:

`gegenbauer` (page 414), `chebyshev_root` (page 416), `chebyshev` (page 416),  
`chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418),  
`hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420),  
`sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly`  
(page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723),  
`sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly`  
(page 723), `sympy.polys.orthopolys.legendre_poly` (page 723),  
`sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R267] (page 1243), [R268] (page 1243), [R269] (page 1243)

## Examples

```
>>> from sympy import jacobi, S, conjugate, diff
>>> from sympy.abc import n,a,b,x
```

```

>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
>>> jacobi(2, a, b, x)
(a**2/8 - a*b/4 - a/8 + b**2/8 - b/8 + x**2*(a**2/8 + a*b/4 + 7*a/8 +
b**2/8 + 7*b/8 + 3/2) + x*(a**2/4 + 3*a/4 - b**2/4 - 3*b/4) - 1/2)

>>> jacobi(n, a, b, x)
jacobi(n, a, b, x)

>>> jacobi(n, a, a, x)
RisingFactorial(a + 1, n)*gegenbauer(n,
a + 1/2, x)/RisingFactorial(2*a + 1, n)

>>> jacobi(n, 0, 0, x)
legendre(n, x)

>>> jacobi(n, S(1)/2, S(1)/2, x)
RisingFactorial(3/2, n)*chebyshev(n, x)/factorial(n + 1)

>>> jacobi(n, -S(1)/2, -S(1)/2, x)
RisingFactorial(1/2, n)*chebyshevt(n, x)/factorial(n)

>>> jacobi(n, a, b, -x)
(-1)**n*jacobi(n, b, a, x)

>>> jacobi(n, a, b, 0)
2*(-n)*gamma(a + n + 1)*hyper((-b - n, -n), (a + 1,), -1)/(factorial(n)*gamma(a + 1))
>>> jacobi(n, a, b, 1)
RisingFactorial(a + 1, n)/factorial(n)

>>> conjugate(jacobi(n, a, b, x))
jacobi(n, conjugate(a), conjugate(b), conjugate(x))

>>> diff(jacobi(n,a,b,x), x)
(a/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)

```

`sympy.functions.special.polynomials.jacobi_normalized(n, a, b, x)`  
Jacobi polynomial  $P_n^{(\alpha, \beta)}(x)$

`jacobi_normalized(n, alpha, beta, x)` gives the nth Jacobi polynomial in x,  $P_n^{(\alpha, \beta)}(x)$ .

The Jacobi polynomials are orthogonal on  $[-1, 1]$  with respect to the weight  $(1 - x)^\alpha (1 + x)^\beta$ .

This functions returns the polynomials normilzed:

$$\int_{-1}^1 P_m^{(\alpha, \beta)}(x) P_n^{(\alpha, \beta)}(x) (1 - x)^\alpha (1 + x)^\beta dx = \delta_{m,n}$$

See also:

`gegenbauer` (page 414), `chebyshevt_root` (page 416), `chebyshev` (page 416),  
`chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418),  
`hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420),  
`sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly`  
(page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723),  
`sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly`

(page 723), [sympy.polys.orthopolys.legendre\\_poly](#) (page 723), [sympy.polys.orthopolys.laguerre\\_poly](#) (page 723)

## References

[R270] (page 1243), [R271] (page 1243), [R272] (page 1243)

## Examples

```
>>> from sympy import jacobi_normalized
>>> from sympy.abc import n,a,b,x

>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**((a + b + 1))*gamma(a + n + 1)*gamma(b + n + 1)/((a + b + 2*n + 1)*factorial(n)*gamma(n + 1)))
```

## Gegenbauer Polynomials

`class sympy.functions.special.polynomials.gegenbauer`

Gegenbauer polynomial  $C_n^{(\alpha)}(x)$

`gegenbauer(n, alpha, x)` gives the nth Gegenbauer polynomial in  $x$ ,  $C_n^{(\alpha)}(x)$ .

The Gegenbauer polynomials are orthogonal on  $[-1, 1]$  with respect to the weight  $(1 - x^2)^{\alpha - \frac{1}{2}}$ .

### See also:

`jacobi` (page 412), `chebyshevt_root` (page 416), `chebyshev` (page 416),  
`chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418),  
`hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420),  
`sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723),  
`sympy.polys.orthopolys.chebyshevt_poly` (page 723),  
`sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723),  
`sympy.polys.orthopolys.legendre_poly` (page 723),  
`sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R273] (page 1243), [R274] (page 1243), [R275] (page 1243)

## Examples

```
>>> from sympy import gegenbauer, conjugate, diff
>>> from sympy.abc import n,a,x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)
```

```
>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)

>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(-n/2 + 1/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))

>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))

>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)
```

## Chebyshev Polynomials

class `sympy.functions.special.polynomials.chebyshevt`  
Chebyshev polynomial of the first kind,  $T_n(x)$

`chebyshevt(n, x)` gives the nth Chebyshev polynomial (of the first kind) in  $x$ ,  $T_n(x)$ .

The Chebyshev polynomials of the first kind are orthogonal on  $[-1, 1]$  with respect to the weight  $\frac{1}{\sqrt{1-x^2}}$ .

See also:

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418), `hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420), `sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723), `sympy.polys.orthopolys.legendre_poly` (page 723), `sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R276] (page 1243), [R277] (page 1243), [R278] (page 1243), [R279] (page 1243), [R280] (page 1243)

## Examples

```
>>> from sympy import chebyshevt, chebyshev, diff
>>> from sympy.abc import n,x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1

>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
```

```
>>> chebyshevt(-n, x)
chebyshevt(n, x)

>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n

>>> diff(chebyshevt(n, x), x)
n*chebyshev(n - 1, x)
```

class sympy.functions.special.polynomials.chebyshev  
Chebyshev polynomial of the second kind,  $U_n(x)$

chebyshev(n, x) gives the nth Chebyshev polynomial of the second kind in x,  $U_n(x)$ .

The Chebyshev polynomials of the second kind are orthogonal on  $[-1, 1]$  with respect to the weight  $\sqrt{1 - x^2}$ .

#### See also:

jacobi (page 412), gegenbauer (page 414), chebyshevt (page 415), chebyshevt\_root (page 416), chebyshev\_root (page 417), legendre (page 417), assoc\_legendre (page 418), hermite (page 419), laguerre (page 419), assoc\_laguerre (page 420), sympy.polys.orthopolys.jacobi\_poly (page 723), sympy.polys.orthopolys.gegenbauer\_poly (page 723), sympy.polys.orthopolys.chebyshevt\_poly (page 723), sympy.polys.orthopolys.chebyshev\_poly (page 723), sympy.polys.orthopolys.hermite\_poly (page 723), sympy.polys.orthopolys.legendre\_poly (page 723), sympy.polys.orthopolys.laguerre\_poly (page 723)

#### References

[R281] (page 1243), [R282] (page 1243), [R283] (page 1243), [R284] (page 1243), [R285] (page 1243)

#### Examples

```
>>> from sympy import chebyshevt, chebyshev, diff
>>> from sympy.abc import n,x
>>> chebyshev(0, x)
1
>>> chebyshev(1, x)
2*x
>>> chebyshev(2, x)
4*x**2 - 1

>>> chebyshev(n, x)
chebyshev(n, x)
>>> chebyshev(n, -x)
(-1)**n*chebyshev(n, x)
>>> chebyshev(-n, x)
-chebyshev(n - 2, x)

>>> chebyshev(n, 0)
cos(pi*n/2)
>>> chebyshev(n, 1)
n + 1
```

---

```
>>> diff(chebyshev(n, x), x)
(-x*chebyshev(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

**class sympy.functions.special.polynomials.chebyshevt\_root**

chebyshev\_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if  $0 \leq k < n$ ,  $\text{chebyshevt}(n, \text{chebyshevt\_root}(n, k)) == 0$ .

**See also:**

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshev](#) (page 416), [chebyshev\\_root](#) (page 417), [legendre](#) (page 417), [assoc\\_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 419), [assoc\\_laguerre](#) (page 420), [sympy.polys.orthopolys.jacobi\\_poly](#) (page 723), [sympy.polys.orthopolys.gegenbauer\\_poly](#) (page 723), [sympy.polys.orthopolys.chebyshevt\\_poly](#) (page 723), [sympy.polys.orthopolys.hermite\\_poly](#) (page 723), [sympy.polys.orthopolys.legendre\\_poly](#) (page 723), [sympy.polys.orthopolys.laguerre\\_poly](#) (page 723)

### Examples

```
>>> from sympy import chebyshevt, chebyshevt_root
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

**class sympy.functions.special.polynomials.chebyshev\_root**

chebyshev\_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if  $0 \leq k < n$ ,  $\text{chebyshev}(n, \text{chebyshev_root}(n, k)) == 0$ .

**See also:**

[chebyshevt](#) (page 415), [chebyshevt\\_root](#) (page 416), [chebyshev](#) (page 416), [legendre](#) (page 417), [assoc\\_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 419), [assoc\\_laguerre](#) (page 420), [sympy.polys.orthopolys.jacobi\\_poly](#) (page 723), [sympy.polys.orthopolys.gegenbauer\\_poly](#) (page 723), [sympy.polys.orthopolys.chebyshevt\\_poly](#) (page 723), [sympy.polys.orthopolys.hermite\\_poly](#) (page 723), [sympy.polys.orthopolys.legendre\\_poly](#) (page 723), [sympy.polys.orthopolys.laguerre\\_poly](#) (page 723)

### Examples

```
>>> from sympy import chebyshev, chebyshev_root
>>> chebyshev_root(3, 2)
-sqrt(2)/2
>>> chebyshev(3, chebyshev_root(3, 2))
0
```

## Legendre Polynomials

**class sympy.functions.special.polynomials.legendre**

legendre(n, x) gives the nth Legendre polynomial of x,  $P_n(x)$

The Legendre polynomials are orthogonal on  $[-1, 1]$  with respect to the constant weight 1. They satisfy  $P_n(1) = 1$  for all n; further,  $P_n$  is odd for odd n and even for even n.

See also:

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt` (page 415), `chebyshevt_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `assoc_legendre` (page 418), `hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420), `sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723), `sympy.polys.orthopolys.legendre_poly` (page 723), `sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R286] (page 1243), [R287] (page 1243), [R288] (page 1243), [R289] (page 1243)

## Examples

```
>>> from sympy import legendre, diff
>>> from sympy.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n,x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)

class sympy.functions.special.polynomials.assoc_legendre
assoc.legendre(n,m, x) gives  $P_n^m(x)$ , where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial,  $P_n(x)$  in the following manner:
```

$$P_n^m(x) = (-1)^m (1-x^2)^{\frac{m}{2}} \frac{d^m P_n(x)}{dx^m}$$

Associated Legendre polynomial are orthogonal on  $[-1, 1]$  with:

- weight = 1 for the same m, and different n.
- weight =  $1/(1-x^2)$  for the same n, and different m.

See also:

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt` (page 415), `chebyshevt_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `legendre` (page 417), `hermite` (page 419), `laguerre` (page 419), `assoc_laguerre` (page 420), `sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723), `sympy.polys.orthopolys.legendre_poly` (page 723), `sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R290] (page 1243), [R291] (page 1243), [R292] (page 1243), [R293] (page 1243)

## Examples

```
>>> from sympy import assoc_legendre
>>> from sympy.abc import x, m, n
>>> assoc_legendre(0,0, x)
1
>>> assoc_legendre(1,0, x)
x
>>> assoc_legendre(1,1, x)
-sqrt(-x**2 + 1)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

## Hermite Polynomials

`class sympy.functions.special.polynomials.hermite`  
`hermite(n, x)` gives the nth Hermite polynomial in x,  $H_n(x)$

The Hermite polynomials are orthogonal on  $(-\infty, \infty)$  with respect to the weight  $\exp\left(-\frac{x^2}{2}\right)$ .

### See also:

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt` (page 415), `chebyshevt_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418), `laguerre` (page 419), `assoc_laguerre` (page 420), `sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723), `sympy.polys.orthopolys.legendre_poly` (page 723), `sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R294] (page 1243), [R295] (page 1243), [R296] (page 1243)

## Examples

```
>>> from sympy import hermite, diff
>>> from sympy.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

## Laguerre Polynomials

class `sympy.functions.special.polynomials.laguerre`  
Returns the nth Laguerre polynomial in x,  $L_n(x)$ .

**Parameters** `n` : int

Degree of Laguerre polynomial. Must be  $n \geq 0$ .

**See also:**

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt` (page 415), `chebyshev_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418), `hermite` (page 419), `assoc_laguerre` (page 420), `sympy.polys.orthopolys.jacobi_poly` (page 723), `sympy.polys.orthopolys.gegenbauer_poly` (page 723), `sympy.polys.orthopolys.chebyshevt_poly` (page 723), `sympy.polys.orthopolys.chebyshev_poly` (page 723), `sympy.polys.orthopolys.hermite_poly` (page 723), `sympy.polys.orthopolys.legendre_poly` (page 723), `sympy.polys.orthopolys.laguerre_poly` (page 723)

## References

[R297] (page 1243), [R298] (page 1243), [R299] (page 1243), [R300] (page 1243)

## Examples

```
>>> from sympy import laguerre, diff
>>> from sympy.abc import x, n
>>> laguerre(0, x)
1
>>> laguerre(1, x)
-x + 1
>>> laguerre(2, x)
x**2/2 - 2*x + 1
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1
```

```
>>> laguerre(n, x)
laguerre(n, x)
```

```
>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

class `sympy.functions.special.polynomials.assoc_laguerre`  
Returns the nth generalized Laguerre polynomial in x,  $L_n(x)$ .

**Parameters** `n` : int

Degree of Laguerre polynomial. Must be  $n \geq 0$ .

`alpha` : Expr

Arbitrary expression. For `alpha=0` regular Laguerre polynomials will be generated.

**See also:**

`jacobi` (page 412), `gegenbauer` (page 414), `chebyshevt` (page 415), `chebyshev_root` (page 416), `chebyshev` (page 416), `chebyshev_root` (page 417), `legendre` (page 417), `assoc_legendre` (page 418), `hermite` (page 419), `laguerre` (page 419),

[sympy.polys.orthopolys.jacobi\\_poly](#) (page 723), [sympy.polys.orthopolys.gegenbauer\\_poly](#) (page 723), [sympy.polys.orthopolys.chebyshev\\_t\\_poly](#) (page 723), [sympy.polys.orthopolys.chebyshev\\_u\\_poly](#) (page 723), [sympy.polys.orthopolys.hermite\\_poly](#) (page 723), [sympy.polys.orthopolys.legendre\\_poly](#) (page 723), [sympy.polys.orthopolys.laguerre\\_poly](#) (page 723)

## References

[R301] (page 1243), [R302] (page 1243), [R303] (page 1244), [R304] (page 1244)

## Examples

```
>>> from sympy import laguerre, assoc_laguerre, diff
>>> from sympy.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) +
x*(-a**2/2 - 5*a/2 - 3) + 1

>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)

>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)

>>> assoc_laguerre(n, 0, x)
laguerre(n, x)

>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)

>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))
```

## Spherical Harmonics

```
class sympy.functions.special.spherical_harmonics.Ynm
Spherical harmonics defined as
```

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) P_n^m(\cos(\theta))$$

`Ynm()` gives the spherical harmonic function of order  $n$  and  $m$  in  $\theta$  and  $\varphi$ ,  $Y_n^m(\theta, \varphi)$ . The four parameters are as follows:  $n \geq 0$  an integer and  $m$  an integer such that  $-n \leq m \leq n$  holds. The two angles are real-valued with  $\theta \in [0, \pi]$  and  $\varphi \in [0, 2\pi]$ .

See also:

`sympy.functions.special.spherical_harmonics.Ynm_c` (page 423),  
`sympy.functions.special.spherical_harmonics.Znm` (page 423)

## References

[R305] (page 1244), [R306] (page 1244), [R307] (page 1244), [R308] (page 1244)

## Examples

```
>>> from sympy import Ynm, Symbol
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order

```
>>> from sympy import Ynm, Symbol
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> Ynm(n, -m, theta, phi)
(-1)**m*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

as well as for the angles

```
>>> from sympy import Ynm, Symbol, simplify
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)

>>> Ynm(n, m, theta, -phi)
exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

For specific integers n and m we can evaluate the harmonics to more useful expressions

```
>>> simplify(Ynm(0, 0, theta, phi).expand(func=True))
1/(2*sqrt(pi))

>>> simplify(Ynm(1, -1, theta, phi).expand(func=True))
sqrt(6)*exp(-I*phi)*sin(theta)/(4*sqrt(pi))

>>> simplify(Ynm(1, 0, theta, phi).expand(func=True))
sqrt(3)*cos(theta)/(2*sqrt(pi))

>>> simplify(Ynm(1, 1, theta, phi).expand(func=True))
-sqrt(6)*exp(I*phi)*sin(theta)/(4*sqrt(pi))
```

```
>>> simplify(Ynm(2, -2, theta, phi).expand(func=True))
sqrt(30)*exp(-2*I*phi)*sin(theta)**2/(8*sqrt(pi))

>>> simplify(Ynm(2, -1, theta, phi).expand(func=True))
sqrt(30)*exp(-I*phi)*sin(2*theta)/(8*sqrt(pi))

>>> simplify(Ynm(2, 0, theta, phi).expand(func=True))
sqrt(5)*(3*cos(theta)**2 - 1)/(4*sqrt(pi))

>>> simplify(Ynm(2, 1, theta, phi).expand(func=True))
-sqrt(30)*exp(I*phi)*sin(2*theta)/(8*sqrt(pi))

>>> simplify(Ynm(2, 2, theta, phi).expand(func=True))
sqrt(30)*exp(2*I*phi)*sin(theta)**2/(8*sqrt(pi))
```

We can differentiate the functions with respect to both angles

```
>>> from sympy import Ynm, Symbol, diff
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + sqrt((-m + n)*(m + n + 1))*exp(-I*phi)*Ynm(n, m + 1, theta, phi)

>>> diff(Ynm(n, m, theta, phi), phi)
I*m*Ynm(n, m, theta, phi)
```

Further we can compute the complex conjugation

```
>>> from sympy import Ynm, Symbol, conjugate
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> conjugate(Ynm(n, m, theta, phi))
(-1)**(2*m)*exp(-2*I*m*phi)*Ynm(n, m, theta, phi)
```

To get back the well known expressions in spherical coordinates we use full expansion

```
>>> from sympy import Ynm, Symbol, expand_func
>>> from sympy.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")

>>> expand_func(Ynm(n, m, theta, phi))
sqrt((2*n + 1)*factorial(-m + n)/factorial(m + n))*exp(I*m*phi)*assoc_legendre(n, m, cos(theta))/(2*sqrt(pi))
```

`sympy.functions.special.spherical_harmonics.Ynm_c(n, m, theta, phi)`

Conjugate spherical harmonics defined as

$$\overline{Y_n^m(\theta, \varphi)} := (-1)^m Y_n^{-m}(\theta, \varphi)$$

See also:

`sympy.functions.special.spherical_harmonics.Ynm` (page 421), `sympy.functions.special.spherical_harmonic` (page 423)

## References

[R309] (page 1244), [R310] (page 1244), [R311] (page 1244)

`class sympy.functions.special.spherical_harmonics.Znm`

Real spherical harmonics defined as

$$Z_n^m(\theta, \varphi) := \begin{cases} \frac{Y_n^m(\theta, \varphi) + \overline{Y_n^m(\theta, \varphi)}}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - \overline{Y_n^m(\theta, \varphi)}}{i\sqrt{2}} & m < 0 \end{cases}$$

which gives in simplified form

$$Z_n^m(\theta, \varphi) = \begin{cases} \frac{Y_n^m(\theta, \varphi) + (-1)^m Y_n^{-m}(\theta, \varphi)}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - (-1)^m Y_n^{-m}(\theta, \varphi)}{i\sqrt{2}} & m < 0 \end{cases}$$

See also:

`sympy.functions.special.spherical_harmonics.Ynm` (page 421), `sympy.functions.special.spherical_harmonic` (page 423)

## References

[R312] (page 1244), [R313] (page 1244), [R314] (page 1244)

## Tensor Functions

`sympy.functions.special.tensor_functions.Eijk(*args, **kwargs)`

Represent the Levi-Civita symbol.

This is just compatibility wrapper to `LeviCivita()`.

See also:

`sympy.functions.special.tensor_functions.LeviCivita` (page 424)

`sympy.functions.special.tensor_functions.eval_levicivita(*args)`

Evaluate Levi-Civita symbol.

`class sympy.functions.special.tensor_functions.LeviCivita`

Represent the Levi-Civita symbol.

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

See also:

`sympy.functions.special.tensor_functions.Eijk` (page 424)

## Examples

```
>>> from sympy import LeviCivita
>>> from sympy.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

class `sympy.functions.special.tensor_functions.KroneckerDelta`  
The discrete, or Kronecker, delta function.

A function that takes in two integers  $i$  and  $j$ . It returns 0 if  $i$  and  $j$  are not equal or it returns 1 if  $i$  and  $j$  are equal.

**Parameters** `i` : Number, Symbol

The first index of the delta function.

`j` : Number, Symbol

The second index of the delta function.

**See also:**

`sympy.functions.special.tensor_functions.KroneckerDelta.eval` (page 425),  
`sympy.functions.special.delta_functions.DiracDelta` (page 358)

## References

[R315] (page 1244)

## Examples

A simple example with integer indices:

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

**classmethod eval(i, j)**  
Evaluates the discrete delta function.

#### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k

>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)

# indirect doctest
```

**indices\_contain\_equal\_information**  
Returns True if indices are either both above or below fermi.

#### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

**is\_above\_fermi**  
True if Delta can be non-zero above fermi

#### See also:

`sympy.functions.special.tensor_functions.KroneckerDelta.is_below_fermi` (page 427),  
`sympy.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi` (page 427), `sympy.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi` (page 427)

#### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
```

```
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

`is_below_fermi`  
True if Delta can be non-zero below fermi

See also:

[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_above\\_fermi](#) (page 426),  
[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_only\\_above\\_fermi](#)  
(page 427), [sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_only\\_below\\_fermi](#)  
(page 427)

## Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

`is_only_above_fermi`  
True if Delta is restricted to above fermi

See also:

[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_above\\_fermi](#) (page 426),  
[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_below\\_fermi](#) (page 427),  
[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_only\\_below\\_fermi](#)  
(page 427)

## Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

`is_only_below_fermi`  
True if Delta is restricted to below fermi

See also:

[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_above\\_fermi](#) (page 426),  
[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_below\\_fermi](#) (page 427),  
[sympy.functions.special.tensor\\_functions.KroneckerDelta.is\\_only\\_above\\_fermi](#)  
(page 427)

### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

### killable\_index

Returns the index which is preferred to substitute in the final expression.

The index to substitute is the index with less information regarding fermi level. If indices contain same information, ‘a’ is preferred before ‘b’.

See also:

[sympy.functions.special.tensor\\_functions.KroneckerDelta.preferred\\_index](#) (page 428)

### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

### preferred\_index

Returns the index which is preferred to keep in the final expression.

The preferred index is the index with more information regarding fermi level. If indices contain same information, ‘a’ is preferred before ‘b’.

See also:

`sympy.functions.special.tensor_functions.KroneckerDelta.killable_index` (page 428)

### Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

## 3.9 Geometry Module

### 3.9.1 Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

### 3.9.2 Available Entities

The following entities are currently available in the geometry module:

- Point
- Line, Ray, Segment
- Ellipse, Circle
- Polygon, RegularPolygon, Triangle

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

### 3.9.3 Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment(Point(0, 0), Point(1, 1/2))
>>> Segment(Point(1, S(1)/2), Point(0, 0))
Segment(Point(0, 0), Point(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
>>> intersection(c, l)
[Point(-5*sqrt(2)/2, -5*sqrt(2)/2), Point(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

### 3.9.4 Intersection of medians

```
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment(Point(0, 0), Point(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point(a, b/3)]
```

### 3.9.5 An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([\[WikiPappus\]](#) (page 1244)):

Given one set of collinear points  $A, B, C$ , and another set of collinear points  $a, b, c$ , then the intersection points  $X, Y, Z$  of line pairs  $Ab$  and  $aB$ ,  $Ac$  and  $aC$ ,  $Bc$  and  $bC$  are collinear.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>>
>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))
>>>
>>> def subs_point(l, val):
...     """Take an arbitrary point and make it a fixed point."""
...     t = Symbol('t', extended_real=True)
...     ap = l.arbitrary_point()
...     return Point(ap.x.subs(t, val), ap.y.subs(t, val))
...
>>> p11 = subs_point(l1, 5)
>>> p12 = subs_point(l1, 6)
>>> p13 = subs_point(l1, 11)
>>>
>>> p21 = subs_point(l2, -1)
>>> p22 = subs_point(l2, 2)
>>> p23 = subs_point(l2, 13)
>>>
>>> l11 = Line(p11, p22)
>>> l12 = Line(p11, p23)
>>> l13 = Line(p12, p21)
>>> l14 = Line(p12, p23)
>>> l15 = Line(p13, p21)
>>> l16 = Line(p13, p22)
>>>
>>> pp1 = intersection(l11, l13)[0]
>>> pp2 = intersection(l12, l15)[0]
>>> pp3 = intersection(l14, l16)[0]
>>>
>>> Point.is_collinear(pp1, pp2, pp3)
True
```

## References

### 3.9.6 Miscellaneous Notes

- The area property of `Polygon` and `Triangle` may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the `abs` function.
- Although `Polygon` can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return `True` (e.g., `Point.is_collinear`) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

### 3.9.7 Future Work

#### Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x,y,z = map(Symbol, 'xyz')
>>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
>>> Point.is_collinear(p1, p2, p3)
False
```

Even though the result is currently `False`, this is not *always* true. If the quantity  $z - y - 2 * y * z + 2 * y^{**} 2 == 0$  then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at the very least, being nice to know. This could be potentially done by returning an object (e.g., `GeometryResult`) that the user could use. This actually would not involve an extensive amount of work.

#### Three Dimensions and Beyond

Currently there are no plans for extending the module to three dimensions, but it certainly would be a good addition. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

#### Submodules

##### Entities

```
class sympy.geometry.entity.GeometryEntity
    The base class for all geometrical entities.
```

This class doesn't represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

`encloses(o)`

Return True if o is inside (not on or outside) the boundaries of self.

The object will be decomposed into Points and individual Entities need only define an `encloses_point` method for their class.

**See also:**

`sympy.geometry.ellipse.Ellipse.encloses_point` (page 485),  
`sympy.geometry.polygon.Polygon.encloses_point` (page 499)

#### Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(*RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t2.encloses(t)
True
```

```
>>> t.encloses(t2)
False

intersection(o)
    Returns a list of all of the intersections of self with o.

See also:
    sympy.geometry.util.intersection (page 435)
```

#### Notes

An entity is not required to implement this method.

If two different types of entities can intersect, the item with higher index in ordering\_of\_classes should implement intersections with anything having a lower index.

```
is_similar(other)
    Is this geometrical entity similar to another geometrical entity?

    Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will
    allow one to obtain the other.

See also:
    scale (page 433)
```

#### Notes

This method is not intended to be used directly but rather through the *are\_similar* function found in util.py. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

```
rotate(angle, pt=None)
    Rotate angle radians counterclockwise about Point pt.

    The default pt is the origin, Point(0, 0)

See also:
    scale (page 433), translate (page 434)
```

#### Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point(0, 1), Point(-sqrt(3)/2, -1/2), Point(sqrt(3)/2, -1/2))

scale(x=1, y=1, pt=None)
    Scale the object by multiplying the x,y-coordinates by x and y.

    If pt is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and
    shifted by pt.

See also:
```

[rotate](#) (page 433), [translate](#) (page 434)

### Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point(2, 0), Point(-1, sqrt(3)/2), Point(-1, -sqrt(3)/2))
>>> t.scale(2,2)
Triangle(Point(2, 0), Point(-1, sqrt(3)), Point(-1, -sqrt(3)))
```

[translate](#)( $x=0, y=0$ )

Shift the object by adding to the x,y-coordinates the values x and y.

See also:

[rotate](#) (page 433), [scale](#) (page 433)

### Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point(1, 0), Point(-1/2, sqrt(3)/2), Point(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point(3, 0), Point(3/2, sqrt(3)/2), Point(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point(3, 2), Point(3/2, sqrt(3)/2 + 2),
         Point(3/2, -sqrt(3)/2 + 2))
```

## Utils

[sympy.geometry.util.idiff](#)( $eq, y, x, n=1$ )

Return  $dy/dx$  assuming that  $eq == 0$ .

**Parameters** **y** : the dependent variable or a list of dependent variables (with y first)

**x** : the variable that the derivative is being taken with respect to

**n** : the order of the derivative (default is 1)

See also:

[sympy.core.function.Derivative](#) (page 135) represents unevaluated derivatives

[sympy.core.function.diff](#) (page 138) explicitly differentiates wrt symbols

### Examples

```
>>> from sympy.abc import x, y, a
>>> from sympy.geometry.util import idiff
```

```
>>> circ = x**2 + y**2 - 4
>>> idiff(circ, y, x)
-x/y
>>> idiff(circ, y, x, 2).simplify()
-(x**2 + y**2)/y**3
```

Here, `a` is assumed to be independent of `x`:

```
>>> idiff(x + a + y, y, x)
-1
```

Now the `x`-dependence of `a` is made explicit by listing `a` after `y` in a list.

```
>>> idiff(x + a + y, [y, a], x)
-Derivative(a, x) - 1
```

`sympy.geometry.util.intersection(*entities)`

The intersection of a collection of `GeometryEntity` instances.

**Parameters** `entities` : sequence of `GeometryEntity`

**Returns** `intersection` : list of `GeometryEntity`

**Raises** `NotImplementedError`

When unable to calculate intersection.

**See also:**

[sympy.geometry.entity.GeometryEntity.intersection](#) (page 432)

## Notes

The intersection of any geometrical entity with itself should return a list with one item: the entity in question. An intersection requires two or more entities. If only a single entity is given then the function will return an empty list. It is possible for `intersection` to miss intersections that one knows exists because the required quantities were not fully simplified internally. Reals should be converted to `Rationals`, e.g. `Rational(str(real_num))` or else failures due to floating point issues may result.

## Examples

```
>>> from sympy.geometry import Point, Line, Circle, intersection
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 5)
>>> l1, l2 = Line(p1, p2), Line(p3, p2)
>>> c = Circle(p2, 1)
>>> intersection(l1, p2)
[Point(1, 1)]
>>> intersection(l1, l2)
[Point(1, 1)]
>>> intersection(c, p2)
[]
>>> intersection(c, Point(1, 0))
[Point(1, 0)]
>>> intersection(c, l2)
[Point(-sqrt(5)/5 + 1, 2*sqrt(5)/5 + 1),
 Point(sqrt(5)/5 + 1, -2*sqrt(5)/5 + 1)]
```

`sympy.geometry.util.convex_hull(*args)`

The convex hull surrounding the Points contained in the list of entities.

**Parameters** `args` : a collection of Points, Segments and/or Polygons

**Returns** `convex_hull` : Polygon

**See also:**

`sympy.geometry.point.Point` (page 437), `sympy.geometry.polygon.Polygon` (page 495)

## Notes

This can only be performed on a set of non-symbolic points.

## References

[1] [http://en.wikipedia.org/wiki/Graham\\_scan](http://en.wikipedia.org/wiki/Graham_scan)

[2] Andrew's Monotone Chain Algorithm (A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", 1979) [http://geomalgorithms.com/a10\\_hull-1.html](http://geomalgorithms.com/a10_hull-1.html)

## Examples

```
>>> from sympy.geometry import Point, convex_hull
>>> points = [(1,1), (1,2), (3,1), (-5,2), (15,4)]
>>> convex_hull(*points)
Polygon(Point(-5, 2), Point(1, 1), Point(3, 1), Point(15, 4))
```

`sympy.geometry.util.are_similar(e1, e2)`

Are two geometrical entities similar.

Can one geometrical entity be uniformly scaled to the other?

**Parameters** `e1` : GeometryEntity

`e2` : GeometryEntity

**Returns** `are_similar` : boolean

**Raises** `GeometryError`

    When `e1` and `e2` cannot be compared.

**See also:**

`sympy.geometry.entity.GeometryEntity.is_similar` (page 433)

## Notes

If the two objects are equal then they are similar.

## Examples

```
>>> from sympy import Point, Circle, Triangle, are_similar
>>> c1, c2 = Circle(Point(0, 0), 4), Circle(Point(1, 4), 3)
>>> t1 = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
>>> t2 = Triangle(Point(0, 0), Point(2, 0), Point(0, 2))
>>> t3 = Triangle(Point(0, 0), Point(3, 0), Point(0, 1))
>>> are_similar(t1, t2)
True
>>> are_similar(t1, t3)
False
```

### `sympy.geometry.util.centroid(*args)`

Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons. The centroid is the weighted average of the individual centroid where the weights are the lengths (of segments) or areas (of polygons). Overlapping regions will add to the weight of that region.

If there are no objects (or a mixture of objects) then None is returned.

See also:

`sympy.geometry.point.Point` (page 437), `sympy.geometry.line.Segment` (page 461),  
`sympy.geometry.polygon.Polygon` (page 495)

### Examples

```
>>> from sympy import Point, Segment, Polygon
>>> from sympy.geometry.util import centroid
>>> p = Polygon((0, 0), (10, 0), (10, 10))
>>> q = p.translate(0, 20)
>>> p.centroid, q.centroid
(Point(20/3, 10/3), Point(20/3, 70/3))
>>> centroid(p, q)
Point(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point(1, -sqrt(2) + 2)
>>> centroid(Point(0, 0), Point(2, 0))
Point(1, 0)
```

Stacking 3 polygons on top of each other effectively triples the weight of that polygon:

```
>>> p = Polygon((0, 0), (1, 0), (1, 1), (0, 1))
>>> q = Polygon((1, 0), (3, 0), (3, 1), (1, 1))
>>> centroid(p, q)
Point(3/2, 1/2)
>>> centroid(p, p, p, q) # centroid x-coord shifts left
Point(11/10, 1/2)
```

Stacking the squares vertically above and below p has the same effect:

```
>>> centroid(p, p.translate(0, 1), p.translate(0, -1), q)
Point(11/10, 1/2)
```

### Points

```
class sympy.geometry.point.Point
A point in a 2-dimensional Euclidean space.
```

**Parameters** `coords` : sequence of 2 coordinate values.

**Raises** `TypeError`

When trying to add or subtract points with different dimensions. When trying to create a point with more than two dimensions. When `intersection` is called with object other than a Point.

See also:

`sympy.geometry.line.Segment` (page 461) Connects two Points

### Examples

```
>>> from sympy.geometry import Point
>>> from sympy.abc import x
>>> Point(1, 2)
Point(1, 2)
>>> Point([1, 2])
Point(1, 2)
>>> Point(0, x)
Point(0, x)
```

FLOATS are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point(0.5, 0.25)
Point(1/2, 1/4)
>>> Point(0.5, 0.25, evaluate=False)
Point(0.5, 0.25)
```

### Attributes

---

<code>x</code> (page 442)	Returns the X coordinate of the Point.
<code>y</code> (page 442)	Returns the Y coordinate of the Point.
<code>length</code> (page 440)	Treating a Point as a Line, this returns 0 for the length of a Point.

---

### `distance(p)`

The Euclidean distance from self to point p.

**Parameters** `p` : Point

**Returns** `distance` : number or symbolic expression.

See also:

`sympy.geometry.line.Segment.length` (page 463)

### Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.distance(p2)
5
```

```
>>> from sympy.abc import x, y
>>> p3 = Point(x, y)
>>> p3.distance(Point(0, 0))
sqrt(x**2 + y**2)
```

**dot(*p2*)**  
Return dot product of self with another Point.

**evalf(*prec=None*, \*\**options*)**  
Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

**Returns point : Point**

### Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point(1/2, 3/2)
>>> p1.evalf()
Point(0.5, 1.5)
```

**intersection(*o*)**  
The intersection between this point and another point.

**Parameters other : Point**

**Returns intersection : list of Points**

### Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

### Examples

```
>>> from sympy import Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point(0, 0)]
```

**is\_collinear(\**points*)**  
Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

**Parameters points : sequence of Point**

**Returns is\_collinear : boolean**

See also:

[sympy.geometry.line.Line](#) (page 456)

#### Notes

Slope is preserved everywhere on a line, so the slope between any two points on the line should be the same. Take the first two points,  $p_1$  and  $p_2$ , and create a translated point  $v_1$  with  $p_1$  as the origin. Now for every other point we create a translated point,  $v_i$  with  $p_1$  also as the origin. Note that these translations preserve slope since everything is consistently translated to a new origin of  $p_1$ . Since slope is preserved then we have the following equality:

- $v_1.\text{slope} = v_i.\text{slope}$
- $v_1.y/v_1.x = v_i.y/v_i.x$  (due to translation)
- $v_1.y*v_i.x = v_i.y*v_1.x$
- $v_1.y*v_i.x - v_i.y*v_1.x = 0$  (\*)

Hence, if we have a  $v_i$  such that the equality in (\*) is False then the points are not collinear. We do this test for every point in the list, and if all pass then they are collinear.

#### Examples

```
>>> from sympy import Point
>>> from sympy.abc import x
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4, p5 = Point(2, 2), Point(x, x), Point(1, 2)
>>> Point.is_collinear(p1, p2, p3, p4)
True
>>> Point.is_collinear(p1, p2, p3, p5)
False
```

**is\_concyclic(\*points)**  
Is a sequence of points concyclic?

Test whether or not a sequence of points are concyclic (i.e., they lie on a circle).

**Parameters** `points` : sequence of Points

**Returns** `is_concyclic` : boolean

True if points are concyclic, False otherwise.

See also:

[sympy.geometry.ellipse.Circle](#) (page 493)

#### Notes

No points are not considered to be concyclic. One or two points are definitely concyclic and three points are concyclic iff they are not collinear.

For more than three points, create a circle from the first three points. If the circle cannot be created (i.e., they are collinear) then all of the points cannot be concyclic. If the circle is created successfully then simply check the remaining points for containment in the circle.

## Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(-1, 0), Point(1, 0)
>>> p3, p4 = Point(0, 1), Point(-1, 2)
>>> Point.is_concyclic(p1, p2, p3)
True
>>> Point.is_concyclic(p1, p2, p3, p4)
False
```

### length

Treating a Point as a Line, this returns 0 for the length of a Point.

## Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.length
0
```

### midpoint(*p*)

The midpoint between self and point *p*.

**Parameters** *p* : Point

**Returns** *midpoint* : Point

**See also:**

[sympy.geometry.line.Segment.midpoint](#) (page 463)

## Examples

```
>>> from sympy.geometry import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point(7, 3)
```

### n(*prec=None, \*\*options*)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

**Returns** *point* : Point

## Examples

```
>>> from sympy import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point(1/2, 3/2)
>>> p1.evalf()
Point(0.5, 1.5)
```

`rotate(angle, pt=None)`  
Rotate `angle` radians counterclockwise about Point `pt`.

See also:

`rotate` (page 441), `scale` (page 441)

### Examples

```
>>> from sympy import Point, pi
>>> t = Point(1, 0)
>>> t.rotate(pi/2)
Point(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point(2, -1)
```

`scale(x=1, y=1, pt=None)`

Scale the coordinates of the Point by multiplying by `x` and `y` after subtracting `pt` – default is `(0, 0)` – and then adding `pt` back again (i.e. `pt` is the point of reference for the scaling).

See also:

`rotate` (page 441), `translate` (page 442)

### Examples

```
>>> from sympy import Point
>>> t = Point(1, 1)
>>> t.scale(2)
Point(2, 1)
>>> t.scale(2, 2)
Point(2, 2)
```

`transform(matrix)`

Return the point after applying the transformation described by the 3x3 Matrix, `matrix`.

See also:

`sympy.geometry.entity.GeometryEntity.rotate` (page 433),  
`sympy.geometry.entity.GeometryEntity.scale` (page 433), `sympy.geometry.entity.GeometryEntity.translate` (page 434)

`translate(x=0, y=0)`

Shift the Point by adding `x` and `y` to the coordinates of the Point.

See also:

`rotate` (page 441), `scale` (page 441)

### Examples

```
>>> from sympy import Point
>>> t = Point(0, 1)
>>> t.translate(2)
Point(2, 1)
>>> t.translate(2, 2)
Point(2, 3)
```

```
>>> t + Point(2, 2)
Point(2, 3)
```

x

Returns the X coordinate of the Point.

### Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.x
0
```

y

Returns the Y coordinate of the Point.

### Examples

```
>>> from sympy import Point
>>> p = Point(0, 1)
>>> p.y
1
```

## 3D Point

Geometrical Points.

**Contains** Point3D

```
class sympy.geometry.point3d.Point3D
A point in a 3-dimensional Euclidean space.
```

**Parameters** coords : sequence of 3 coordinate values.

**Raises** **NotImplementedError**

When trying to create a point other than 2 or 3 dimensions. When *intersection* is called with object other than a Point.

**TypeError**

When trying to add or subtract points with different dimensions.

### Notes

Currently only 2-dimensional and 3-dimensional points are supported.

### Examples

```
>>> from sympy import Point3D
>>> from sympy.abc import x
>>> Point3D(1, 2, 3)
Point3D(1, 2, 3)
>>> Point3D([1, 2, 3])
Point3D(1, 2, 3)
>>> Point3D(0, x, 3)
Point3D(0, x, 3)
```

FLOATS are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point3D(0.5, 0.25, 2)
Point3D(1/2, 1/4, 2)
>>> Point3D(0.5, 0.25, 3, evaluate=False)
Point3D(0.5, 0.25, 3)
```

## Attributes

---

x (page 448)	Returns the X coordinate of the Point.
y (page 448)	Returns the Y coordinate of the Point.
z (page 448)	Returns the Z coordinate of the Point.
length (page 446)	Treating a Point as a Line, this returns 0 for the length of a Point.

---

**static are\_collinear(\*points)**

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

**Parameters** points : sequence of Point

**Returns** are\_collinear : boolean

**See also:**

[sympy.geometry.line3d.Line3D](#) (page 464)

## Examples

```
>>> from sympy import Point3D, Matrix
>>> from sympy.abc import x
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> p3, p4, p5 = Point3D(2, 2, 2), Point3D(x, x, x), Point3D(1, 2, 6)
>>> Point3D.are_collinear(p1, p2, p3, p4)
True
>>> Point3D.are_collinear(p1, p2, p3, p5)
False
```

**static are\_coplanar(\*points)**

This function tests whether passed points are coplanar or not. It uses the fact that the triple scalar product of three vectors vanishes if the vectors are coplanar. Which means that the volume of the solid described by them will have to be zero for coplanarity.

**Parameters** A set of points 3D points

**Returns** boolean

## Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 2)
>>> p2 = Point3D(2, 7, 2)
>>> p3 = Point3D(0, 0, 2)
>>> p4 = Point3D(1, 1, 2)
>>> Point3D.are_coplanar(p1, p2, p3, p4)
True
>>> p5 = Point3D(0, 1, 3)
>>> Point3D.are_coplanar(p1, p2, p3, p5)
False
```

`direction_cosine(point)`

Gives the direction cosine between 2 points

**Parameters** `p` : Point3D

**Returns** list

## Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_cosine(Point3D(2, 3, 5))
[sqrt(6)/6, sqrt(6)/6, sqrt(6)/3]
```

`direction_ratio(point)`

Gives the direction ratio between 2 points

**Parameters** `p` : Point3D

**Returns** list

## Examples

```
>>> from sympy import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_ratio(Point3D(2, 3, 5))
[1, 1, 2]
```

`distance(p)`

The Euclidean distance from self to point p.

**Parameters** `p` : Point

**Returns** `distance` : number or symbolic expression.

**See also:**

`sympy.geometry.line.Segment.length` (page 463)

## Examples

```
>>> from sympy import Point3D
>>> p1, p2 = Point3D(1, 1, 1), Point3D(4, 5, 0)
>>> p1.distance(p2)
sqrt(26)

>>> from sympy.abc import x, y, z
>>> p3 = Point3D(x, y, z)
>>> p3.distance(Point3D(0, 0, 0))
sqrt(x**2 + y**2 + z**2)
```

### dot(*p2*)

Return dot product of self with another Point.

### evalf(*prec=None, \*\*options*)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

**Returns** `point` : Point

### Examples

```
>>> from sympy import Point3D, Rational
>>> p1 = Point3D(Rational(1, 2), Rational(3, 2), Rational(5, 2))
>>> p1
Point3D(1/2, 3/2, 5/2)
>>> p1.evalf()
Point3D(0.5, 1.5, 2.5)
```

### intersection(*o*)

The intersection between this point and another point.

**Parameters** `other` : Point

**Returns** `intersection` : list of Points

### Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

### Examples

```
>>> from sympy import Point3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point3D(0, 0, 0)]
```

### length

Treating a Point as a Line, this returns 0 for the length of a Point.

## Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.length
0
```

```
midpoint(p)
```

The midpoint between self and point p.

**Parameters** `p` : Point

**Returns** `midpoint` : Point

**See also:**

[sympy.geometry.line.Segment.midpoint](#) (page 463)

## Examples

```
>>> from sympy import Point3D
>>> p1, p2 = Point3D(1, 1, 1), Point3D(13, 5, 1)
>>> p1.midpoint(p2)
Point3D(7, 3, 1)
```

```
n(prec=None, **options)
```

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

**Returns** `point` : Point

## Examples

```
>>> from sympy import Point3D, Rational
>>> p1 = Point3D(Rational(1, 2), Rational(3, 2), Rational(5, 2))
>>> p1
Point3D(1/2, 3/2, 5/2)
>>> p1.evalf()
Point3D(0.5, 1.5, 2.5)
```

```
scale(x=1, y=1, z=1, pt=None)
```

Scale the coordinates of the Point by multiplying by `x` and `y` after subtracting `pt` – default is (0, 0) – and then adding `pt` back again (i.e. `pt` is the point of reference for the scaling).

**See also:**

[translate](#) (page 447)

## Examples

```
>>> from sympy import Point3D
>>> t = Point3D(1, 1, 1)
>>> t.scale(2)
Point3D(2, 1, 1)
```

```
>>> t.scale(2, 2)
Point3D(2, 2, 1)
```

#### transform(*matrix*)

Return the point after applying the transformation described by the 3x3 Matrix, *matrix*.

See also:

[sympy.geometry.entity.GeometryEntity.rotate](#) (page 433),  
[sympy.geometry.entity.GeometryEntity.scale](#) (page 433), [sympy.geometry.entity.GeometryEntity.translate](#) (page 434)

#### translate(*x=0, y=0, z=0*)

Shift the Point by adding x and y to the coordinates of the Point.

See also:

[sympy.geometry.entity.GeometryEntity.rotate](#) (page 433), [scale](#) (page 447)

### Examples

```
>>> from sympy import Point3D
>>> t = Point3D(0, 1, 1)
>>> t.translate(2)
Point3D(2, 1, 1)
>>> t.translate(2, 2)
Point3D(2, 3, 1)
>>> t + Point3D(2, 2, 2)
Point3D(2, 3, 3)
```

#### x

Returns the X coordinate of the Point.

### Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 3)
>>> p.x
0
```

#### y

Returns the Y coordinate of the Point.

### Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.y
1
```

#### z

Returns the Z coordinate of the Point.

## Examples

```
>>> from sympy import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.z
1
```

## Lines

`class sympy.geometry.line.LinearEntity`

A base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.

See also:

[sympy.geometry.entity.GeometryEntity](#) (page 432)

## Notes

This is an abstract class and is not meant to be instantiated.

## Attributes

---

<a href="#">p1</a> (page 453)	The first defining point of a linear entity.
<a href="#">p2</a> (page 453)	The second defining point of a linear entity.
<a href="#">coefficients</a> (page 450)	The coefficients ( $a, b, c$ ) for $ax + by + c = 0$ .
<a href="#">slope</a> (page 456)	The slope of this linear entity, or infinity if vertical.
<a href="#">points</a> (page 454)	The two points used to define this linear entity.

---

`angle_between(l1, l2)`

The angle formed between the two linear entities.

Parameters `l1` : `LinearEntity`

`l2` : `LinearEntity`

Returns `angle` : angle in radians

See also:

[is\\_perpendicular](#) (page 452)

## Notes

From the dot product of vectors  $v_1$  and  $v_2$  it is known that:

$$\text{dot}(v_1, v_2) = |v_1| * |v_2| * \cos(A)$$

where  $A$  is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, 0)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.angle_between(l2)
pi/2
```

`arbitrary_point(parameter='t')`

A parameterized point on the Line.

**Parameters** `parameter` : str, optional

The name of the parameter which will be used for the parametric point. The default value is ‘t’. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

**Returns** `point` : Point

**Raises** ValueError

When `parameter` already appears in the Line’s definition.

**See also:**

[sympy.geometry.point.Point](#) (page 437)

**Examples**

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point(4*t + 1, 3*t)
```

`static are_concurrent(*lines)`

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

**Parameters** `lines` : a sequence of linear entities.

**Returns** True : if the set of linear entities are concurrent,

False : otherwise.

**See also:**

[sympy.geometry.util.intersection](#) (page 435)

**Notes**

Simply take the first two lines and find their intersection. If there is no intersection, then the first two lines were parallel and had no intersection so concurrency is impossible amongst the whole set. Otherwise, check to see if the intersection point of the first two lines is a member on the rest of the lines. If so, the lines are concurrent.

## Examples

```
>>> from sympy import Point, Line, Line3D
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> p3, p4 = Point(-2, -2), Point(0, 2)
>>> l1, l2, l3 = Line(p1, p2), Line(p1, p3), Line(p1, p4)
>>> Line.are_concurrent(l1, l2, l3)
True

>>> l4 = Line(p2, p3)
>>> Line.are_concurrent(l2, l3, l4)
False
```

### coefficients

The coefficients  $(a, b, c)$  for  $ax + by + c = 0$ .

See also:

[sympy.geometry.line.Line.equation](#) (page 458)

## Examples

```
>>> from sympy import Point, Line
>>> from sympy.abc import x, y
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.coefficients
(-3, 5, 0)

>>> p3 = Point(x, y)
>>> l2 = Line(p1, p3)
>>> l2.coefficients
(-y, x, 0)
```

### contains(*other*)

Subclasses should implement this method and should return True if other is on the boundaries of self; False if not on the boundaries of self; None if a determination cannot be made.

### intersection(*o*)

The intersection with another geometrical entity.

**Parameters** *o* : Point or LinearEntity

**Returns** intersection : list of geometrical entities

See also:

[sympy.geometry.point.Point](#) (page 437)

## Examples

```
>>> from sympy import Point, Line, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(7, 7)
>>> l1 = Line(p1, p2)
>>> l1.intersection(p3)
[Point(7, 7)]
```

```
>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point(15/8, 15/8)]
```

```
>>> p6, p7 = Point(0, 5), Point(2, 6)
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
```

**is\_parallel(*l1*, *l2*)**  
Are two linear entities parallel?

**Parameters** *l1* : LinearEntity

*l2* : LinearEntity

**Returns** **True** : if *l1* and *l2* are parallel,  
**False** : otherwise.

**See also:**

[coefficients](#) (page 450)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4 = Point(3, 4), Point(6, 7)
>>> l1, l2 = Line(p1, p2), Line(p3, p4)
>>> Line.is_parallel(l1, l2)
True
```

```
>>> p5 = Point(6, 6)
>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
```

**is\_perpendicular(*l1*, *l2*)**  
Are two linear entities perpendicular?

**Parameters** *l1* : LinearEntity

*l2* : LinearEntity

**Returns** **True** : if *l1* and *l2* are perpendicular,  
**False** : otherwise.

**See also:**

[coefficients](#) (page 450)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 1)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
```

```
>>> l1.is_perpendicular(l2)
True

>>> p4 = Point(5, 3)
>>> l3 = Line(p1, p4)
>>> l1.is_perpendicular(l3)
False

is_similar(other)
Return True if self and other are contained in the same line.
```

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 1), Point(3, 4), Point(2, 3)
>>> l1 = Line(p1, p2)
>>> l2 = Line(p1, p3)
>>> l1.is_similar(l2)
True
```

### length

The length of the line.

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.length
oo
```

### p1

The first defining point of a linear entity.

#### See also:

`sympy.geometry.point.Point` (page 437)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point(0, 0)
```

### p2

The second defining point of a linear entity.

#### See also:

`sympy.geometry.point.Point` (page 437)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point(5, 3)
```

`parallel_line(p)`

Create a new Line parallel to this linear entity which passes through the point *p*.

**Parameters** *p* : Point

**Returns** line : Line

**See also:**

[is\\_parallel](#) (page 451)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

`perpendicular_line(p)`

Create a new Line perpendicular to this linear entity which passes through the point *p*.

**Parameters** *p* : Point

**Returns** line : Line

**See also:**

[is\\_perpendicular](#) (page 452), [perpendicular\\_segment](#) (page 454)

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

`perpendicular_segment(p)`

Create a perpendicular line segment from *p* to this line.

The endpoints of the segment are *p* and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

**Parameters** *p* : Point

**Returns segment** : Segment

**See also:**

[perpendicular\\_line](#) (page 454)

#### Notes

Returns  $p$  itself if  $p$  is on this linear entity.

#### Examples

```
>>> from sympy import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 2)
>>> l1 = Line(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point(4, 0))
Segment(Point(2, 2), Point(4, 0))
```

#### points

The two points used to define this linear entity.

**Returns points** : tuple of Points

**See also:**

[sympy.geometry.point.Point](#) (page 437)

#### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point(0, 0), Point(5, 11))
```

#### projection( $o$ )

Project a point, line, ray, or segment onto this linear entity.

**Parameters other** : Point or LinearEntity (Line, Ray, Segment)

**Returns projection** : Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter **other**.

**Raises GeometryError**

When method is unable to perform projection.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [perpendicular\\_line](#) (page 454)

## Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point P is projected onto a line L by finding the point on L that is closest to P. This point is the intersection of L and the line perpendicular to L that passes through P.

## Examples

```
>>> from sympy import Point, Line, Segment, Rational
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(Rational(1, 2), 0)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point(1/4, 1/4)

>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
>>> l1.projection(s1)
Segment(Point(5, 5), Point(13/2, 13/2))
```

### random\_point()

A random point on a LinearEntity.

**Returns** point : Point

**See also:**

[sympy.geometry.point.Point](#) (page 437)

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> p3 = l1.random_point()
>>> # random point - don't know its coords in advance
>>> p3
Point(...)

>>> # point should belong to the line
>>> p3 in l1
True
```

### slope

The slope of this linear entity, or infinity if vertical.

**Returns** slope : number or sympy expression

**See also:**

[coefficients](#) (page 450)

## Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.slope
5/3

>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

**class sympy.geometry.line.Line**  
An infinite line in space.

A line is declared with two distinct points or a point and slope as defined using keyword *slope*.

**Parameters** **p1** : Point

**pt** : Point

**slope** : sympy expression

**See also:**

[sympy.geometry.point.Point](#) (page 437)

## Notes

At the moment only lines in a 2D space can be declared, because Points can be defined only for 2D spaces.

## Examples

```
>>> import sympy
>>> from sympy import Point
>>> from sympy.abc import L
>>> from sympy.geometry import Line, Segment
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line(Point(2, 3), Point(3, 5))
>>> L.points
(Point(2, 3), Point(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword *slope*:

```
>>> Line(Point(0, 0), slope=0)
Line(Point(0, 0), Point(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

`contains(o)`  
Return True if o is on this Line, or False otherwise.

#### Examples

```
>>> from sympy import Line, Point  
>>> p1, p2 = Point(0, 1), Point(3, 4)  
>>> l = Line(p1, p2)  
>>> l.contains(p1)  
True  
>>> l.contains((0, 1))  
True  
>>> l.contains((0, 0))  
False
```

`distance(o)`  
Finds the shortest distance between a line and a point.

**Raises** `NotImplementedError` is raised if o is not a `Point`

#### Examples

```
>>> from sympy import Point, Line  
>>> p1, p2 = Point(0, 0), Point(1, 1)  
>>> s = Line(p1, p2)  
>>> s.distance(Point(-1, 1))  
sqrt(2)  
>>> s.distance((-1, 2))  
3*sqrt(2)/2
```

`equal(other)`  
Returns True if self and other are the same mathematical entities

`equation(x='x', y='y')`  
The equation of the line: ax + by + c.

**Parameters** `x` : str, optional

The name to use for the x-axis, default value is ‘x’.

`y` : str, optional

The name to use for the y-axis, default value is ‘y’.

**Returns** `equation` : sympy expression

**See also:**

`LinearEntity.coefficients` (page 450)

#### Examples

```
>>> from sympy import Point, Line  
>>> p1, p2 = Point(1, 0), Point(5, 3)  
>>> l1 = Line(p1, p2)  
>>> l1.equation()  
-3*x + 4*y + 3
```

`plot_interval(parameter='t')`

The plot interval for the default geometric plot of line. Gives values that will produce a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

**Parameters** `parameter` : str, optional

Default value is ‘t’.

**Returns** `plot_interval` : list (plot interval)

[parameter, lower\_bound, upper\_bound]

### Examples

```
>>> from sympy import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

`class sympy.geometry.line.Ray`

A Ray is a semi-line in the space with a source point and a direction.

**Parameters** `p1` : Point

The source of the Ray

`p2` : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [Line](#) (page 456)

### Notes

At the moment only rays in a 2D space can be declared, because Points can be defined only for 2D spaces.

### Examples

```
>>> import sympy
>>> from sympy import Point, pi
>>> from sympy.abc import r
>>> from sympy.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray(Point(2, 3), Point(3, 5))
>>> r.points
(Point(2, 3), Point(3, 5))
>>> r.source
Point(2, 3)
>>> r.xdirection
oo
>>> r.ydirection
```

```
oo
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

## Attributes

<a href="#">source</a> (page 460)	The point from which the ray emanates.
<a href="#">xdirection</a> (page 461)	The x direction of the ray.
<a href="#">ydirection</a> (page 461)	The y direction of the ray.

`contains(o)`  
Is other GeometryEntity contained in this Ray?

## Examples

```
>>> from sympy import Ray, Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 4)
>>> r = Ray(p1, p2)
>>> r.contains(p1)
True
>>> r.contains((1, 1))
True
>>> r.contains((1, 3))
False
>>> s = Segment((1, 1), (2, 2))
>>> r.contains(s)
True
>>> s = Segment((1, 2), (2, 5))
>>> r.contains(s)
False
>>> r1 = Ray((2, 2), (3, 3))
>>> r.contains(r1)
True
>>> r1 = Ray((2, 2), (3, 5))
>>> r.contains(r1)
False
```

`distance(o)`  
Finds the shortest distance between the ray and a point.

**Raises** `NotImplementedError` is raised if `o` is not a `Point`

## Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Ray(p1, p2)
>>> s.distance(Point(-1, -1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
```

`equals(other)`

Returns True if self and other are the same mathematical entities

`plot_interval(parameter='t')`

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

**Parameters** `parameter` : str, optional

Default value is ‘t’.

**Returns** `plot_interval` : list

[parameter, lower\_bound, upper\_bound]

## Examples

```
>>> from sympy import Point, Ray, pi
>>> r = Ray((0, 0), angle=pi/4)
>>> r.plot_interval()
[t, 0, 10]
```

`source`

The point from which the ray emanates.

**See also:**

[sympy.geometry.point.Point](#) (page 437)

## Examples

```
>>> from sympy import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point(0, 0)
```

`xdirection`

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

**See also:**

[ydirection](#) (page 461)

## Examples

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, -1)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

**ydirection**

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 461)

**Examples**

```
>>> from sympy import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(-1, -1), Point(-1, 0)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

**class sympy.geometry.line.Segment**

A undirected line segment in space.

**Parameters** **p1** : Point

**p2** : Point

See also:

[sympy.geometry.point.Point](#) (page 437), [Line](#) (page 456)

**Notes**

At the moment only segments in a 2D space can be declared, because Points can be defined only for 2D spaces.

**Examples**

```
>>> import sympy
>>> from sympy import Point
>>> from sympy.abc import s
>>> from sympy.geometry import Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment(Point(1, 0), Point(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1))
>>> s
Segment(Point(1, 1), Point(4, 3))
>>> s.points
(Point(1, 1), Point(4, 3))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point(5/2, 2)
```

## Attributes

---

<a href="#">length</a> (page 463)	The length of the line segment.
<a href="#">midpoint</a> (page 463)	The midpoint of the line segment.

---

### `contains(other)`

Is the other GeometryEntity contained within this Segment?

## Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s2 = Segment(p2, p1)
>>> s.contains(s2)
True
```

### `distance(o)`

Finds the shortest distance between a line segment and a point.

**Raises** `NotImplementedError` is raised if *o* is not a `Point`

## Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s.distance(Point(10, 15))
sqrt(170)
>>> s.distance((0, 12))
sqrt(73)
```

### `length`

The length of the line segment.

**See also:**

[sympy.geometry.point.Point.distance](#) (page 438)

## Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
```

### `midpoint`

The midpoint of the line segment.

**See also:**

[sympy.geometry.point.Point.midpoint](#) (page 441)

### Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point(2, 3/2)
```

`perpendicular_bisector(p=None)`

The perpendicular bisector of this segment.

If no point is specified or the point specified is not on the bisector then the bisector is returned as a Line. Otherwise a Segment is returned that joins the point specified and the intersection of the bisector and the segment.

**Parameters** `p` : Point

**Returns** `bisector` : Line or Segment

See also:

`LinearEntity.perpendicular_segment` (page 454)

### Examples

```
>>> from sympy import Point, Segment
>>> p1, p2, p3 = Point(0, 0), Point(6, 6), Point(5, 1)
>>> s1 = Segment(p1, p2)
>>> s1.perpendicular_bisector()
Line(Point(3, 3), Point(9, -3))

>>> s1.perpendicular_bisector(p3)
Segment(Point(3, 3), Point(5, 1))
```

`plot_interval(parameter='t')`

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

**Parameters** `parameter` : str, optional

Default value is ‘t’.

**Returns** `plot_interval` : list

[`parameter`, `lower_bound`, `upper_bound`]

### Examples

```
>>> from sympy import Point, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> s1 = Segment(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

## 3D Line

Line-like geometrical entities.

**Contains** LinearEntity3D Line3D Ray3D Segment3D

```
class sympy.geometry.line3d.Line3D
    An infinite 3D line in space.
```

A line is declared with two distinct points or a point and direction\_ratio as defined using keyword *direction\_ratio*.

**Parameters** **p1** : Point3D  
    **pt** : Point3D

**direction\_ratio** : list

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443)

### Examples

```
>>> import sympy
>>> from sympy import Point3D
>>> from sympy.abc import L
>>> from sympy.geometry import Line3D, Segment3D
>>> L = Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L
Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L.points
(Point3D(2, 3, 4), Point3D(3, 5, 1))
```

**contains(o)**

Return True if o is on this Line, or False otherwise.

### Examples

```
>>> from sympy import Line3D
>>> a = (0, 0, 0)
>>> b = (1, 1, 1)
>>> c = (2, 2, 2)
>>> l1 = Line3D(a, b)
>>> l2 = Line3D(b, a)
>>> l1 == l2
False
>>> l1 in l2
True
```

**distance(o)**

Finds the shortest distance between a line and a point.

**Raises** `NotImplementedError` **is raised if o is not an instance of Point3D**

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> s = Line3D(p1, p2)
>>> s.distance(Point3D(-1, 1, 1))
```

```
2*sqrt(6)/3
>>> s.distance((-1, 1, 1))
2*sqrt(6)/3
```

### equals(*other*)

Returns True if self and other are the same mathematical entities

### equation(*x*=*x'*, *y*=*y'*, *z*=*z'*, *k*=*k*)

The equation of the line in 3D

#### Parameters *x* : str, optional

The name to use for the x-axis, default value is ‘x’.

#### *y* : str, optional

The name to use for the y-axis, default value is ‘y’.

#### *z* : str, optional

The name to use for the z-axis, default value is ‘z’.

#### Returns **equation** : tuple

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 0)
>>> l1 = Line3D(p1, p2)
>>> l1.equation()
(x/4 - 1/4, y/3, z0*z, k)
```

### plot\_interval(*parameter*=‘t’)

The plot interval for the default geometric plot of line. Gives values that will produce a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

#### Parameters *parameter* : str, optional

Default value is ‘t’.

#### Returns **plot\_interval** : list (plot interval)

[*parameter*, lower\_bound, upper\_bound]

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

### class `sympy.geometry.line3d.LinearEntity3D`

An base class for all linear entities (line, ray and segment) in a 3-dimensional Euclidean space.

### Notes

This is a base class and is not meant to be instantiated.

**Attributes**


---

<code>p1</code> (page 470)	The first defining point of a linear entity.
<code>p2</code> (page 471)	The second defining point of a linear entity.
<code>direction_ratio</code> (page 468)	The direction ratio of a given line in 3D.
<code>direction_cosine</code> (page 468)	The normalized direction ratio of a given line in 3D.
<code>points</code> (page 472)	The two points used to define this linear entity.

---

`angle_between(l1, l2)`

The angle formed between the two linear entities.

**Parameters** `l1` : LinearEntity`l2` : LinearEntity**Returns** `angle` : angle in radians**See also:**`is_perpendicular` (page 470)**Notes**

From the dot product of vectors v1 and v2 it is known that:

$$\text{dot}(v1, v2) = |v1| * |v2| * \cos(A)$$

where A is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

**Examples**

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.angle_between(l2)
acos(-sqrt(2)/3)
```

`arbitrary_point(parameter='t')`

A parameterized point on the Line.

**Parameters** `parameter` : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

**Returns** `point` : Point3D**Raises** ValueError

When `parameter` already appears in the Line's definition.

**See also:**`sympy.geometry.point3d.Point3D` (page 443)

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.arbitrary_point()
Point3D(4*t + 1, 3*t, t)
```

**static are\_concurrent(\*lines)**

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

**Parameters** `lines` : a sequence of linear entities.

**Returns** `True` : if the set of linear entities are concurrent,

`False` : otherwise.

**See also:**

`sympy.geometry.util.intersection` (page 435)

## Notes

Simply take the first two lines and find their intersection. If there is no intersection, then the first two lines were parallel and had no intersection so concurrency is impossible amongst the whole set. Otherwise, check to see if the intersection point of the first two lines is a member on the rest of the lines. If so, the lines are concurrent.

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 2)
>>> p3, p4 = Point3D(-2, -2, -2), Point3D(0, 2, 1)
>>> l1, l2, l3 = Line3D(p1, p2), Line3D(p1, p3), Line3D(p1, p4)
>>> Line3D.are_concurrent(l1, l2, l3)
True

>>> l4 = Line3D(p2, p3)
>>> Line3D.are_concurrent(l2, l3, l4)
False
```

**contains(other)**

Subclasses should implement this method and should return `True` if other is on the boundaries of self; `False` if not on the boundaries of self; `None` if a determination cannot be made.

**direction cosine**

The normalized direction ratio of a given line in 3D.

**See also:**

`sympy.geometry.line.Line.equation` (page 458)

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_cosine
[sqrt(35)/7, 3*sqrt(35)/35, sqrt(35)/35]
>>> sum(i**2 for i in _)
1
```

### direction\_ratio

The direction ratio of a given line in 3D.

See also:

[sympy.geometry.line.Line.equation](#) (page 458)

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_ratio
[5, 3, 1]
```

### intersection(o)

The intersection with another geometrical entity.

Parameters **o** : Point or LinearEntity3D

Returns **intersection** : list of geometrical entities

See also:

[sympy.geometry.point3d.Point3D](#) (page 443)

## Examples

```
>>> from sympy import Point3D, Line3D, Segment3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(7, 7, 7)
>>> l1 = Line3D(p1, p2)
>>> l1.intersection(p3)
[Point3D(7, 7, 7)]

>>> l1 = Line3D(Point3D(4,19,12), Point3D(5,25,17))
>>> l2 = Line3D(Point3D(-3, -15, -19), direction_ratio=[2,8,8])
>>> l1.intersection(l2)
[Point3D(1, 1, -3)]

>>> p6, p7 = Point3D(0, 5, 2), Point3D(2, 6, 3)
>>> s1 = Segment3D(p6, p7)
>>> l1.intersection(s1)
[]
```

### is\_parallel(l1, l2)

Are two linear entities parallel?

**Parameters** `l1` : LinearEntity

`l2` : LinearEntity

**Returns** `True` : if `l1` and `l2` are parallel,

`False` : otherwise.

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 4, 5)
>>> p3, p4 = Point3D(2, 1, 1), Point3D(8, 9, 11)
>>> l1, l2 = Line3D(p1, p2), Line3D(p3, p4)
>>> Line3D.is_parallel(l1, l2)
True

>>> p5 = Point3D(6, 6, 6)
>>> l3 = Line3D(p3, p5)
>>> Line3D.is_parallel(l1, l3)
False
```

`is_perpendicular(l1, l2)`

Are two linear entities perpendicular?

**Parameters** `l1` : LinearEntity

`l2` : LinearEntity

**Returns** `True` : if `l1` and `l2` are perpendicular,

`False` : otherwise.

### See also:

[direction\\_ratio](#) (page 468)

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.is_perpendicular(l2)
False

>>> p4 = Point3D(5, 3, 7)
>>> l3 = Line3D(p1, p4)
>>> l1.is_perpendicular(l3)
False
```

`is_similar(other)`

Return `True` if self and other are contained in the same line.

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(2, 2, 2)
>>> l1 = Line3D(p1, p2)
>>> l2 = Line3D(p1, p3)
>>> l1.is_similar(l2)
True
```

**length**  
The length of the line.

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.length
oo
```

**p1**  
The first defining point of a linear entity.

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443)

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.p1
Point3D(0, 0, 0)
```

**p2**  
The second defining point of a linear entity.

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443)

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.p2
Point3D(5, 3, 1)
```

**parallel\_line(*p*)**  
Create a new Line parallel to this linear entity which passes through the point *p*.

**Parameters** **p** : Point3D

**Returns** **line** : Line3D

See also:

[is\\_parallel](#) (page 469)

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

`perpendicular_line(p)`

Create a new Line perpendicular to this linear entity which passes through the point  $p$ .

**Parameters** `p` : Point3D

**Returns** `line` : Line3D

See also:

[is\\_perpendicular](#) (page 470), [perpendicular\\_segment](#) (page 472)

### Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

`perpendicular_segment(p)`

Create a perpendicular line segment from  $p$  to this line.

The endpoints of the segment are  $p$  and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

**Parameters** `p` : Point3D

**Returns** `segment` : Segment3D

See also:

[perpendicular\\_line](#) (page 471)

### Notes

Returns  $p$  itself if  $p$  is on this linear entity.

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point3D(4, 0, 0))
Segment3D(Point3D(4/3, 4/3, 4/3), Point3D(4, 0, 0))
```

### points

The two points used to define this linear entity.

**Returns** `points` : tuple of Points

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443)

## Examples

```
>>> from sympy import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 11, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.points
(Point3D(0, 0, 0), Point3D(5, 11, 1))
```

### projection(*o*)

Project a point, line, ray, or segment onto this linear entity.

**Parameters** `other` : Point or LinearEntity (Line, Ray, Segment)

**Returns** `projection` : Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter `other`.

**Raises** `GeometryError`

When method is unable to perform projection.

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443), [perpendicular\\_line](#) (page 471)

## Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point P is projected onto a line L by finding the point on L that is closest to P. This point is the intersection of L and the line perpendicular to L that passes through P.

## Examples

```
>>> from sympy import Point3D, Line3D, Segment3D, Rational
>>> p1, p2, p3 = Point3D(0, 0, 1), Point3D(1, 1, 2), Point3D(2, 0, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.projection(p3)
Point3D(2/3, 2/3, 5/3)

>>> p4, p5 = Point3D(10, 0, 1), Point3D(12, 1, 3)
>>> s1 = Segment3D(p4, p5)
>>> l1.projection(s1)
[Segment3D(Point3D(10/3, 10/3, 13/3), Point3D(5, 5, 6))]
```

**class** `sympy.geometry.line3d.Ray3D`

A Ray is a semi-line in the space with a source point and a direction.

**Parameters** `p1` : `Point3D`

The source of the Ray

`p2` : Point or a direction vector

`direction_ratio`: Determines the direction in which the Ray propagates.

See also:

`sympy.geometry.point3d.Point3D` (page 443), `Line3D` (page 464)

## Examples

```
>>> import sympy
>>> from sympy import Point3D, pi
>>> from sympy.abc import r
>>> from sympy.geometry import Ray3D
>>> r = Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r
Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.points
(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.source
Point3D(2, 3, 4)
>>> r.xdirection
oo
>>> r.ydirection
oo
>>> r.direction_ratio
[1, 2, -4]
```

## Attributes

---

<code>source</code> (page 475)	The point from which the ray emanates.
<code>xdirection</code> (page 475)	The x direction of the ray.
<code>ydirection</code> (page 475)	The y direction of the ray.
<code>zdirection</code> (page 476)	The z direction of the ray.

---

`contains(o)`  
Is other GeometryEntity contained in this Ray?

`distance(o)`  
Finds the shortest distance between the ray and a point.

**Raises** `NotImplementedError` **is raised if o is not a Point**

### Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 2)
>>> s = Ray3D(p1, p2)
>>> s.distance(Point3D(-1, -1, 2))
sqrt(6)
>>> s.distance((-1, -1, 2))
sqrt(6)
```

`equals(other)`  
Returns True if self and other are the same mathematical entities

`plot_interval(parameter='t')`  
The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

**Parameters** `parameter` : str, optional

Default value is 't'.

**Returns** `plot_interval` : list

[parameter, lower\_bound, upper\_bound]

### Examples

```
>>> from sympy import Point3D, Ray3D, pi
>>> r = Ray3D(Point3D(0, 0, 0), Point3D(1, 1, 1))
>>> r.plot_interval()
[t, 0, 10]
```

`source`  
The point from which the ray emanates.

**See also:**

[sympy.geometry.point3d.Point3D](#) (page 443)

### Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 1, 5)
>>> r1 = Ray3D(p1, p2)
>>> r1.source
Point3D(0, 0, 0)
```

`xdirection`  
The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

See also:

[ydirection](#) (page 475)

### Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, -1, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

### ydirection

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 475)

### Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

### zdirection

The z direction of the ray.

Positive infinity if the ray points in the positive z direction, negative infinity if the ray points in the negative z direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 475)

### Examples

```
>>> from sympy import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.zdirection
-oo
>>> r2.zdirection
0
```

```
>>> r2.zdirection
0
```

class sympy.geometry.line3d.Segment3D  
A undirected line segment in a 3D space.

**Parameters** **p1** : Point3D

**p2** : Point3D

See also:

[sympy.geometry.point3d.Point3D](#) (page 443), [Line3D](#) (page 464)

### Examples

```
>>> import sympy
>>> from sympy import Point3D
>>> from sympy.abc import s
>>> from sympy.geometry import Segment3D
>>> Segment3D((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s
Segment3D(Point3D(1, 1, 7), Point3D(4, 3, 9))
>>> s.points
(Point3D(1, 1, 7), Point3D(4, 3, 9))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)
```

### Attributes

---

<a href="#">length</a> (page 477)	The length of the line segment.
<a href="#">midpoint</a> (page 477)	The midpoint of the line segment.

---

**contains(*other*)**

Is the other GeometryEntity contained within this Segment?

### Examples

```
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 1, 1), Point3D(3, 4, 5)
>>> s = Segment3D(p1, p2)
>>> s2 = Segment3D(p2, p1)
>>> s.contains(s2)
True
```

**distance(*o*)**

Finds the shortest distance between a line segment and a point.

**Raises** `NotImplementedError` **is raised if *o* is not a Point3D**

### Examples

```
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 3), Point3D(1, 1, 4)
>>> s = Segment3D(p1, p2)
>>> s.distance(Point3D(10, 15, 12))
sqrt(341)
>>> s.distance((10, 15, 12))
sqrt(341)
```

#### length

The length of the line segment.

#### See also:

[sympy.geometry.point3d.Point3D.distance](#) (page 445)

### Examples

```
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.length
sqrt(34)
```

#### midpoint

The midpoint of the line segment.

#### See also:

[sympy.geometry.point3d.Point3D.midpoint](#) (page 446)

### Examples

```
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.midpoint
Point3D(2, 3/2, 3/2)
```

#### plot\_interval(parameter='t')

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

**Parameters** `parameter` : str, optional

Default value is 't'.

**Returns** `plot_interval` : list

[parameter, lower\_bound, upper\_bound]

### Examples

---

```
>>> from sympy import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 0)
>>> s1 = Segment3D(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

## Curves

`class sympy.geometry.curve.Curve`  
A curve in space.

A curve is defined by parametric functions for the coordinates, a parameter and the lower and upper bounds for the parameter value.

**Parameters** `function` : list of functions

`limits` : 3-tuple

Function parameter and lower and upper bounds.

**Raises** `ValueError`

When `functions` are specified incorrectly. When `limits` are specified incorrectly.

**See also:**

[sympy.core.function.Function](#) (page 140), [sympy.polys.polyfuncs.interpolate](#) (page 718)

## Examples

```
>>> from sympy import sin, cos, Symbol, interpolate
>>> from sympy.abc import t, a
>>> from sympy.geometry import Curve
>>> C = Curve((sin(t), cos(t)), (t, 0, 2))
>>> C.functions
(sin(t), cos(t))
>>> C.limits
(t, 0, 2)
>>> C.parameter
t
>>> C = Curve((t, interpolate([1, 4, 9, 16], t)), (t, 0, 1)); C
Curve((t, t**2), (t, 0, 1))
>>> C.subs(t, 4)
Point(4, 16)
>>> C.arbitrary_point(a)
Point(a, a**2)
```

## Attributes

---

<code>functions</code> (page 480)	The functions specifying the curve.
<code>parameter</code> (page 480)	The curve function variable.
<code>limits</code> (page 480)	The limits for the curve.

---

`arbitrary_point(parameter='t')`  
A parameterized point on the curve.

**Parameters** `parameter` : str or Symbol, optional

Default value is ‘t’; the Curve’s parameter is selected with `None` or `self.parameter` otherwise the provided symbol is used.

**Returns** `arbitrary_point` : Point

**Raises** ValueError

When `parameter` already appears in the functions.

**See also:**

[sympy.geometry.point.Point](#) (page 437)

### Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import s
>>> from sympy.geometry import Curve
>>> C = Curve([2*s, s**2], (s, 0, 2))
>>> C.arbitrary_point()
Point(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point(2*s, s**2)
>>> C.arbitrary_point(None)
Point(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point(2*a, a**2)
```

### free\_symbols

Return a set of symbols other than the bound symbols used to parametrically define the Curve.

### Examples

```
>>> from sympy.abc import t, a
>>> from sympy.geometry import Curve
>>> Curve((t, t**2), (t, 0, 2)).free_symbols
set()
>>> Curve((t, t**2), (t, a, 2)).free_symbols
set([a])
```

### functions

The functions specifying the curve.

**Returns** `functions` : list of parameterized coordinate functions.

**See also:**

[parameter](#) (page 480)

### Examples

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

**limits**

The limits for the curve.

**Returns** `limits` : tuple

Contains parameter and lower and upper limits.

**See also:**

[plot\\_interval](#) (page 481)

**Examples**

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**3], (t, -2, 2))
>>> C.limits
(t, -2, 2)
```

**parameter**

The curve function variable.

**Returns** `parameter` : SymPy symbol

**See also:**

[functions](#) (page 480)

**Examples**

```
>>> from sympy.abc import t
>>> from sympy.geometry import Curve
>>> C = Curve([t, t**2], (t, 0, 2))
>>> C.parameter
t
```

**plot\_interval(*parameter*=*t*)**

The plot interval for the default geometric plot of the curve.

**Parameters** `parameter` : str or Symbol, optional

Default value is ‘t’; otherwise the provided symbol is used.

**Returns** `plot_interval` : list (plot interval)

[parameter, lower\_bound, upper\_bound]

**See also:**

[limits](#) (page 480) Returns limits of the parameter interval

**Examples**

```
>>> from sympy import Curve, sin
>>> from sympy.abc import x, t, s
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval()
[t, 1, 2]
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval(s)
[s, 1, 2]
```

```
rotate(angle=0, pt=None)
    Rotate angle radians counterclockwise about Point pt.

    The default pt is the origin, Point(0, 0).
```

### Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy.abc import x
>>> from sympy import pi
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))

scale(x=1, y=1, pt=None)
    Override GeometryEntity.scale since Curve is not made up of Points.
```

### Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy import pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))

translate(x=0, y=0)
    Translate the Curve by (x, y).
```

### Examples

```
>>> from sympy.geometry.curve import Curve
>>> from sympy import pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).translate(1, 2)
Curve((x + 1, x + 2), (x, 0, 1))
```

## Ellipses

```
class sympy.geometry.ellipse.Ellipse
    An elliptical GeometryEntity.
```

**Parameters** `center` : Point, optional

Default value is Point(0, 0)

`hradius` : number or SymPy expression, optional

`vradius` : number or SymPy expression, optional

`eccentricity` : number or SymPy expression, optional

Two of `hradius`, `vradius` and `eccentricity` must be supplied to create an Ellipse.  
The third is derived from the two supplied.

**Raises** `GeometryError`

When `hradius`, `vradius` and `eccentricity` are incorrectly supplied as parameters.

## TypeError

When *center* is not a Point.

See also:

[Circle](#) (page 493)

## Notes

Constructed from a center and two radii, the first being the horizontal radius (along the x-axis) and the second being the vertical radius (along the y-axis).

When symbolic value for *hradius* and *vradius* are used, any calculation that refers to the foci or the major or minor axis will assume that the ellipse has its major radius on the x-axis. If this is not true then a manual rotation is necessary.

## Examples

```
>>> from sympy import Ellipse, Point, Rational
>>> e1 = Ellipse(Point(0, 0), 5, 1)
>>> e1.hradius, e1.vradius
(5, 1)
>>> e2 = Ellipse(Point(3, 1), hradius=3, eccentricity=Rational(4, 5))
>>> e2
Ellipse(Point(3, 1), 3, 9/5)
```

## Attributes

---

<a href="#">center</a> (page 484)	The center of the ellipse.
<a href="#">hradius</a> (page 487)	The horizontal radius of the ellipse.
<a href="#">vradius</a> (page 492)	The vertical radius of the ellipse.
<a href="#">area</a> (page 484)	The area of the ellipse.
<a href="#">circumference</a> (page 484)	The circumference of the ellipse.
<a href="#">eccentricity</a> (page 484)	The eccentricity of the ellipse.
<a href="#">periapsis</a> (page 490)	The periapsis of the ellipse.
<a href="#">apoapsis</a> (page 483)	The apoapsis of the ellipse.
<a href="#">focus_distance</a> (page 486)	The focale distance of the ellipse.
<a href="#">foci</a> (page 486)	The foci of the ellipse.

---

## apoapsis

The apoapsis of the ellipse.

The greatest distance between the focus and the contour.

**Returns** `apoapsis` : number

See also:

[periapsis](#) (page 490) Returns shortest distance between foci and contour

**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.apoapsis
2*sqrt(2) + 3

arbitrary_point(parameter='t')
```

A parameterized point on the ellipse.

**Parameters** `parameter` : str, optional

Default value is ‘t’.

**Returns** `arbitrary_point` : Point

**Raises** ValueError

When `parameter` already appears in the functions.

**See also:**

[sympy.geometry.point.Point](#) (page 437)

**Examples**

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point(3*cos(t), 2*sin(t))
```

**area**

The area of the ellipse.

**Returns** `area` : number

**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.area
3*pi
```

**center**

The center of the ellipse.

**Returns** `center` : number

**See also:**

[sympy.geometry.point.Point](#) (page 437)

### Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point(0, 0)
```

#### circumference

The circumference of the ellipse.

### Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.circumference
12*Integral(sqrt((-8*_x**2/9 + 1)/(-_x**2 + 1)), (_x, 0, 1))
```

#### eccentricity

The eccentricity of the ellipse.

**Returns** eccentricity : number

### Examples

```
>>> from sympy import Point, Ellipse, sqrt
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, sqrt(2))
>>> e1.eccentricity
sqrt(7)/3
```

#### encloses\_point(p)

Return True if p is enclosed by (is inside of) self.

**Parameters** p : Point

**Returns** encloses\_point : True, False or None

**See also:**

[sympy.geometry.point.Point](#) (page 437)

### Notes

Being on the border of self is considered False.

### Examples

```
>>> from sympy import Ellipse, S
>>> from sympy.abc import t
>>> e = Ellipse((0, 0), 3, 2)
>>> e.encloses_point((0, 0))
True
```

```
>>> e.encloses_point(e.arbitrary_point(t).subs(t, S.Half))
False
>>> e.encloses_point((4, 0))
False

equation(x='x', y='y')
The equation of the ellipse.
```

**Parameters** `x` : str, optional

Label for the x-axis. Default value is ‘x’.

`y` : str, optional

Label for the y-axis. Default value is ‘y’.

**Returns** `equation` : sympy expression

**See also:**

[arbitrary\\_point \(page 483\)](#) Returns parameterized point on ellipse

### Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.equation()
y**2/4 + (x/3 - 1/3)**2 - 1

evolute(x='x', y='y')
The equation of evolute of the ellipse.
```

**Parameters** `x` : str, optional

Label for the x-axis. Default value is ‘x’.

`y` : str, optional

Label for the y-axis. Default value is ‘y’.

**Returns** `equation` : sympy expression

### Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.evolute()
2**((2/3)*y**((2/3)) + (3*x - 3)**((2/3)) - 5**((2/3))

foci
The foci of the ellipse.
```

**Raises** `ValueError`

When the major and minor axis cannot be determined.

**See also:**

[sympy.geometry.point.Point \(page 437\)](#)

[focus\\_distance \(page 486\)](#) Returns the distance between focus and center

## Notes

The foci can only be calculated if the major/minor axes are known.

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.foci
(Point(-2*sqrt(2), 0), Point(2*sqrt(2), 0))
```

### focus\_distance

The focale distance of the ellipse.

The distance between the center and one focus.

**Returns** `focus_distance` : number

**See also:**

[foci](#) (page 486)

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.focus_distance
2*sqrt(2)
```

### hradius

The horizontal radius of the ellipse.

**Returns** `hradius` : number

**See also:**

[vradius](#) (page 492), [major](#) (page 488), [minor](#) (page 489)

## Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.hradius
3
```

### intersection(*o*)

The intersection of this ellipse and another geometrical entity *o*.

**Parameters** `o` : GeometryEntity

**Returns** `intersection` : list of GeometryEntity objects

**See also:**

[sympy.geometry.entity.GeometryEntity](#) (page 432)

## Notes

Currently supports intersections with Point, Line, Segment, Ray, Circle and Ellipse types.

## Examples

```
>>> from sympy import Ellipse, Point, Line, sqrt
>>> e = Ellipse(Point(0, 0), 5, 7)
>>> e.intersection(Point(0, 0))
[]
>>> e.intersection(Point(5, 0))
[Point(5, 0)]
>>> e.intersection(Line(Point(0,0), Point(0, 1)))
[Point(0, -7), Point(0, 7)]
>>> e.intersection(Line(Point(5,0), Point(5, 1)))
[Point(5, 0)]
>>> e.intersection(Line(Point(6,0), Point(6, 1)))
[]
>>> e = Ellipse(Point(-1, 0), 4, 3)
>>> e.intersection(Ellipse(Point(1, 0), 4, 3))
[Point(0, -3*sqrt(15)/4), Point(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point(2, -3*sqrt(7)/4), Point(2, 3*sqrt(7)/4)]
>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point(-363/175, -48*sqrt(111)/175), Point(-363/175, 48*sqrt(111)/175), Point(3, 0)]

>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point(-17/5, -12/5), Point(-17/5, 12/5), Point(7/5, -12/5), Point(7/5, 12/5)]
```

### is\_tangent(o)

Is  $o$  tangent to the ellipse?

**Parameters**  $o$  : GeometryEntity

An Ellipse, LinearEntity or Polygon

**Returns** `is_tangent`: boolean

True if  $o$  is tangent to the ellipse, False otherwise.

**Raises** `NotImplementedError`

When the wrong type of argument is supplied.

**See also:**

[tangent\\_lines](#) (page 492)

## Examples

```
>>> from sympy import Point, Ellipse, Line
>>> p0, p1, p2 = Point(0, 0), Point(3, 0), Point(3, 3)
>>> e1 = Ellipse(p0, 3, 2)
>>> l1 = Line(p1, p2)
>>> e1.is_tangent(l1)
True
```

**major**

Longer axis of the ellipse (if it can be determined) else hradius.

**Returns** `major` : number or expression

**See also:**

`hradius` (page 487), `vradius` (page 492), `minor` (page 489)

**Examples**

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.major
3

>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).major
a
>>> Ellipse(p1, b, a).major
b

>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).major
m + 1
```

**minor**

Shorter axis of the ellipse (if it can be determined) else vradius.

**Returns** `minor` : number or expression

**See also:**

`hradius` (page 487), `vradius` (page 492), `major` (page 488)

**Examples**

```
>>> from sympy import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.minor
1

>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).minor
b
>>> Ellipse(p1, b, a).minor
a

>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).minor
m
```

`normal_lines(p, prec=None)`  
Normal lines between  $p$  and the ellipse.

**Parameters** `p` : Point

**Returns** `normal_lines` : list with 1, 2 or 4 Lines

### Examples

```
>>> from sympy import Line, Point, Ellipse
>>> e = Ellipse((0, 0), 2, 3)
>>> c = e.center
>>> e.normal_lines(c + Point(1, 0))
[Line(Point(0, 0), Point(1, 0))]
>>> e.normal_lines(c)
[Line(Point(0, 0), Point(0, 1)), Line(Point(0, 0), Point(1, 0))]
```

Off-axis points require the solution of a quartic equation. This often leads to very large expressions that may be of little practical use. An approximate solution of `prec` digits can be obtained by passing in the desired value:

```
>>> e.normal_lines((3, 3), prec=2)
[Line(Point(-38/47, -85/31), Point(9/47, -21/17)),
 Line(Point(19/13, -43/21), Point(32/13, -8/3))]
```

Whereas the above solution has an operation count of 12, the exact solution has an operation count of 2020.

### periapsis

The periapsis of the ellipse.

The shortest distance between the focus and the contour.

**Returns** `periapsis` : number

**See also:**

[apoapsis \(page 483\)](#) Returns greatest distance between focus and contour

### Examples

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.periapsis
-2*sqrt(2) + 3
```

### plot\_interval(parameter='t')

The plot interval for the default geometric plot of the Ellipse.

**Parameters** `parameter` : str, optional

Default value is ‘t’.

**Returns** `plot_interval` : list

[parameter, lower\_bound, upper\_bound]

## Examples

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.plot_interval()
[t, -pi, pi]
```

```
random_point(seed=None)
A random point on the ellipse.
```

**Returns** point : Point

### See also:

[sympy.geometry.point.Point](#) (page 437)

[arbitrary\\_point](#) (page 483) Returns parameterized point on ellipse

## Notes

An arbitrary\_point with a random value of t substituted into it may not test as being on the ellipse because the expression tested that a point is on the ellipse doesn't simplify to zero and doesn't evaluate exactly to zero:

```
>>> from sympy.abc import t
>>> e1.arbitrary_point(t)
Point(3*cos(t), 2*sin(t))
>>> p2 = _.subs(t, 0.1)
>>> p2 in e1
False
```

Note that arbitrary\_point routine does not take this approach. A value for cos(t) and sin(t) (not t) is substituted into the arbitrary point. There is a small chance that this will give a point that will not test as being in the ellipse, so the process is repeated (up to 10 times) until a valid point is obtained.

## Examples

```
>>> from sympy import Point, Ellipse, Segment
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point(...)
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point(2.1, 1.4)
```

The random\_point method assures that the point will test as being in the ellipse:

```
>>> p1 in e1
True
```

### reflect(*line*)

Override GeometryEntity.reflect since the radius is not a GeometryEntity.

## Notes

Until the general ellipse (with no axis parallel to the x-axis) is supported a `NotImplemented` error is raised and the equation whose zeros define the rotated ellipse is given.

## Examples

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point(1, 0), -1)
>>> from sympy import Ellipse, Line, Point
>>> Ellipse(Point(3, 4), 1, 3).reflect(Line(Point(0, -4), Point(5, 0)))
Traceback (most recent call last):
...
NotImplementedError:
General Ellipse is not supported but the equation of the reflected
Ellipse is given by the zeros of: f(x, y) = (9*x/41 + 40*y/41 +
37/41)**2 + (40*x/123 - 3*y/41 - 364/123)**2 - 1
```

`rotate(angle=0, pt=None)`

Rotate `angle` radians counterclockwise about Point `pt`.

Note: since the general ellipse is not supported, only rotations that are integer multiples of  $\pi/2$  are allowed.

## Examples

```
>>> from sympy import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point(0, 1), 1, 2)
>>> Ellipse((1, 0), 2, 1).rotate(pi)
Ellipse(Point(-1, 0), 2, 1)
```

`scale(x=1, y=1, pt=None)`

Override `GeometryEntity.scale` since it is the major and minor axes which must be scaled and they are not `GeometryEntities`.

## Examples

```
>>> from sympy import Ellipse
>>> Ellipse((0, 0), 2, 1).scale(2, 4)
Circle(Point(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point(0, 0), 4, 1)
```

`tangent_lines(p)`

Tangent lines between `p` and the ellipse.

If `p` is on the ellipse, returns the tangent line through point `p`. Otherwise, returns the tangent line(s) from `p` to the ellipse, or `None` if no tangent line is possible (e.g., `p` inside ellipse).

**Parameters** `p` : Point

**Returns** `tangent_lines` : list with 1 or 2 Lines

**Raises** `NotImplementedError`

Can only find tangent lines for a point,  $p$ , on the ellipse.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.Line](#) (page 456)

**Examples**

```
>>> from sympy import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line(Point(3, 0), Point(3, -12))]
```

**vradius**

The vertical radius of the ellipse.

**Returns** `vradius` : number

**See also:**

[hradius](#) (page 487), [major](#) (page 488), [minor](#) (page 489)

**Examples**

```
>>> from sympy import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.vradius
1
```

**class** `sympy.geometry.ellipse.Circle`

A circle in space.

Constructed simply from a center and a radius, or from three non-collinear points.

**Parameters** `center` : Point

`radius` : number or sympy expression

`points` : sequence of three Points

**Raises** `GeometryError`

When trying to construct circle from three collinear points. When trying to construct circle from incorrect parameters.

**See also:**

[Ellipse](#) (page 482), [sympy.geometry.point.Point](#) (page 437)

**Examples**

```
>>> from sympy.geometry import Point, Circle
>>> # a circle constructed from a center and radius
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.hradius, c1.vradius, c1.radius
(5, 5, 5)
```

```
>>> # a circle costructed from three points
>>> c2 = Circle(Point(0, 0), Point(1, 1), Point(1, 0))
>>> c2.hradius, c2.vradius, c2.radius, c2.center
(sqrt(2)/2, sqrt(2)/2, sqrt(2)/2, Point(1/2, 1/2))
```

## Attributes

---

<code>circumference</code> (page 493)	The circumference of the circle.
<code>equation</code> (page 494)([x, y])	The equation of the circle.

---

`radius` (synonymous with `hradius`, `vradius`, `major` and `minor`)

### `circumference`

The circumference of the circle.

**Returns** `circumference` : number or SymPy expression

## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.circumference
12*pi

equation(x='x', y='y')
The equation of the circle.
```

**Parameters** `x` : str or Symbol, optional

Default value is ‘x’.

`y` : str or Symbol, optional

Default value is ‘y’.

**Returns** `equation` : SymPy expression

## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.equation()
x**2 + y**2 - 25

intersection(o)
The intersection of this circle with another geometrical entity.
```

**Parameters** `o` : GeometryEntity

**Returns** `intersection` : list of GeometryEntities

## Examples

```
>>> from sympy import Point, Circle, Line, Ray
>>> p1, p2, p3 = Point(0, 0), Point(5, 5), Point(6, 0)
>>> p4 = Point(5, 0)
>>> c1 = Circle(p1, 5)
>>> c1.intersection(p2)
[]
>>> c1.intersection(p4)
[Point(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point(5*sqrt(2)/2, 5*sqrt(2)/2)]
>>> c1.intersection(Line(p2, p3))
[]
[]
```

### radius

The radius of the circle.

**Returns** `radius` : number or sympy expression

**See also:**

`Ellipse.major` (page 488), `Ellipse.minor` (page 489), `Ellipse.hradius` (page 487),  
`Ellipse.vradius` (page 492)

## Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.radius
6
```

### reflect(*line*)

Override `GeometryEntity.reflect` since the radius is not a `GeometryEntity`.

## Examples

```
>>> from sympy import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point(1, 0), -1)
```

### scale(*x=1*, *y=1*, *pt=None*)

Override `GeometryEntity.scale` since the radius is not a `GeometryEntity`.

## Examples

```
>>> from sympy import Circle
>>> Circle((0, 0), 1).scale(2, 2)
Circle(Point(0, 0), 2)
>>> Circle((0, 0), 1).scale(2, 4)
Ellipse(Point(0, 0), 2, 4)
```

### vradius

This `Ellipse` property is an alias for the `Circle`'s radius.

Whereas `hradius`, `major` and `minor` can use Ellipse's conventions, the `vradius` does not exist for a circle. It is always a positive value in order that the Circle, like Polygons, will have an area that can be positive or negative as determined by the sign of the `hradius`.

### Examples

```
>>> from sympy import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.vradius
6
```

## Polygons

```
class sympy.geometry.polygon.Polygon
A two-dimensional polygon.
```

A simple polygon in space. Can be constructed from a sequence of points or from a center, radius, number of sides and rotation angle.

**Parameters** `vertices` : sequence of Points

**Raises** `GeometryError`

If all parameters are not Points.

If the Polygon has intersecting sides.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.Segment](#) (page 461), [Triangle](#) (page 509)

### Notes

Polygons are treated as closed paths rather than 2D areas so some calculations can be be negative or positive (e.g., area) based on the orientation of the points.

Any consecutive identical points are reduced to a single point and any points collinear and between two points will be removed unless they are needed to define an explicit intersection (see examples).

A Triangle, Segment or Point will be returned when there are 3 or fewer points provided.

### Examples

```
>>> from sympy import Point, Polygon, pi
>>> p1, p2, p3, p4, p5 = [(0, 0), (1, 0), (5, 1), (0, 1), (3, 0)]
>>> Polygon(p1, p2, p3, p4)
Polygon(Point(0, 0), Point(1, 0), Point(5, 1), Point(0, 1))
>>> Polygon(p1, p2)
Segment(Point(0, 0), Point(1, 0))
>>> Polygon(p1, p2, p5)
Segment(Point(0, 0), Point(3, 0))
```

While the sides of a polygon are not allowed to cross implicitly, they can do so explicitly. For example, a polygon shaped like a Z with the top left connecting to the bottom right of the Z must have the point in the middle of the Z explicitly given:

```
>>> mid = Point(1, 1)
>>> Polygon((0, 2), (2, 2), mid, (0, 0), (2, 0), mid).area
0
>>> Polygon((0, 2), (2, 2), mid, (2, 0), (0, 0), mid).area
-2
```

When the keyword *n* is used to define the number of sides of the Polygon then a RegularPolygon is created and the other arguments are interpreted as center, radius and rotation. The unrotated RegularPolygon will always have a vertex at Point(*r*, 0) where *r* is the radius of the circle that circumscribes the RegularPolygon. Its method *spin* can be used to increment that angle.

```
>>> p = Polygon((0,0), 1, n=3)
>>> p
RegularPolygon(Point(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point(1, 0)
>>> p.args[0]
Point(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point(0, 1)
```

## Attributes

---

<a href="#">area</a> (page 498)	The area of the polygon.
<a href="#">angles</a> (page 497)	The internal angle at each vertex.
<a href="#">perimeter</a> (page 500)	The perimeter of the polygon.
<a href="#">vertices</a> (page 501)	The vertices of the polygon.
<a href="#">centroid</a> (page 498)	The centroid of the polygon.
<a href="#">sides</a> (page 501)	The line segments that form the sides of the polygon.

---

### angles

The internal angle at each vertex.

**Returns angles : dict**

A dictionary where each key is a vertex and each value is the internal angle at that vertex. The vertices are represented as Points.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.LinearEntity.angle\\_between](#) (page 449)

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.angles[p1]
pi/2
```

```
>>> poly.angles[p2]
acos(-4*sqrt(17)/17)
```

`arbitrary_point(parameter='t')`

A parameterized point on the polygon.

The parameter, varying from 0 to 1, assigns points to the position on the perimeter that is that fraction of the total perimeter. So the point evaluated at  $t=1/2$  would return the point from the first vertex that is  $1/2$  way around the polygon.

**Parameters** `parameter` : str, optional

Default value is 't'.

**Returns** `arbitrary_point` : Point

**Raises** ValueError

When `parameter` already appears in the Polygon's definition.

**See also:**

`sympy.geometry.point.Point` (page 437)

## Examples

```
>>> from sympy import Polygon, S, Symbol
>>> t = Symbol('t', extended_real=True)
>>> tri = Polygon((0, 0), (1, 0), (1, 1))
>>> p = tri.arbitrary_point('t')
>>> perimeter = tri.perimeter
>>> s1, s2 = [s.length for s in tri.sides[:2]]
>>> p.subs(t, (s1 + s2/2)/perimeter)
Point(1, 1/2)
```

`area`

The area of the polygon.

**See also:**

`sympy.geometry.ellipse.Ellipse.area` (page 484)

## Notes

The area calculation can be positive or negative based on the orientation of the points.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.area
3
```

`centroid`

The centroid of the polygon.

**Returns** `centroid` : Point

See also:

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.util.centroid](#) (page 436)

### Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.centroid
Point(31/18, 11/18)
```

`distance(o)`

Returns the shortest distance between self and o.

If o is a point, then self does not need to be convex. If o is another polygon self and o must be complex.

### Examples

```
>>> from sympy import Point, Polygon, RegularPolygon
>>> p1, p2 = map(Point, [(0, 0), (7, 5)])
>>> poly = Polygon(*RegularPolygon(p1, 1, 3).vertices)
>>> poly.distance(p2)
sqrt(61)
```

`encloses_point(p)`

Return True if p is enclosed by (is inside of) self.

**Parameters** `p` : Point

**Returns** `encloses_point` : True, False or None

See also:

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.ellipse.Ellipse.encloses\\_point](#) (page 485)

### Notes

Being on the border of self is considered False.

### References

[1] <http://www.ariel.com.au/a/python-point-int-poly.html>

### Examples

```
>>> from sympy import Polygon, Point
>>> from sympy.abc import t
>>> p = Polygon((0, 0), (4, 0), (4, 4))
>>> p.encloses_point(Point(2, 1))
True
```

```
>>> p.encloses_point(Point(2, 2))
False
>>> p.encloses_point(Point(5, 5))
False
```

### intersection(*o*)

The intersection of two polygons.

The intersection may be empty and can contain individual Points and complete Line Segments.

**Parameters other:** Polygon

**Returns intersection :** list

The list of Segments and Points

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.Segment](#) (page 461)

### Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly1 = Polygon(p1, p2, p3, p4)
>>> p5, p6, p7 = map(Point, [(3, 2), (1, -1), (0, 2)])
>>> poly2 = Polygon(p5, p6, p7)
>>> poly1.intersection(poly2)
[Point(2/3, 0), Point(9/5, 1/5), Point(7/3, 1), Point(1/3, 1)]
```

### is\_convex()

Is the polygon convex?

A polygon is convex if all its interior angles are less than 180 degrees.

**Returns is\_convex :** boolean

True if this polygon is convex, False otherwise.

**See also:**

[sympy.geometry.util.convex\\_hull](#) (page 435)

### Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.is_convex()
True
```

### perimeter

The perimeter of the polygon.

**Returns perimeter :** number or Basic instance

**See also:**

[sympy.geometry.line.Segment.length](#) (page 463)

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.perimeter
sqrt(17) + 7
```

plot\_interval(parameter='t')

The plot interval for the default geometric plot of the polygon.

**Parameters** parameter : str, optional

Default value is ‘t’.

**Returns** plot\_interval : list (plot interval)

[parameter, lower\_bound, upper\_bound]

## Examples

```
>>> from sympy import Polygon
>>> p = Polygon((0, 0), (1, 0), (1, 1))
>>> p.plot_interval()
[t, 0, 1]
```

### sides

The line segments that form the sides of the polygon.

**Returns** sides : list of sides

Each side is a Segment.

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.Segment](#) (page 461)

## Notes

The Segments that represent the sides are an undirected line segment so cannot be used to tell the orientation of the polygon.

## Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.sides
[Segment(Point(0, 0), Point(1, 0)),
 Segment(Point(1, 0), Point(5, 1)),
 Segment(Point(0, 1), Point(5, 1)), Segment(Point(0, 0), Point(0, 1))]
```

### vertices

The vertices of the polygon.

**Returns** vertices : tuple of Points

See also:

[sympy.geometry.point.Point](#) (page 437)

#### Notes

When iterating over the vertices, it is more efficient to index self rather than to request the vertices and index them. Only use the vertices when you want to process all of them at once. This is even more important with RegularPolygons that calculate each vertex.

#### Examples

```
>>> from sympy import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.vertices
(Point(0, 0), Point(1, 0), Point(5, 1), Point(0, 1))
>>> poly.args[0]
Point(0, 0)
```

class `sympy.geometry.polygon.RegularPolygon`  
A regular polygon.

Such a polygon has all internal angles equal and all sides the same length.

**Parameters** `center` : `Point`

`radius` : number or `Basic` instance

The distance from the center to a vertex

`n` : int

The number of sides

**Raises** `GeometryError`

If the `center` is not a `Point`, or the `radius` is not a number or `Basic` instance, or the number of sides, `n`, is less than three.

See also:

[sympy.geometry.point.Point](#) (page 437), [Polygon](#) (page 495)

#### Notes

A `RegularPolygon` can be instantiated with `Polygon` with the kwarg `n`.

Regular polygons are instantiated with a `center`, `radius`, `number of sides` and a `rotation angle`. Whereas the arguments of a `Polygon` are vertices, the vertices of the `RegularPolygon` must be obtained with the `vertices` method.

#### Examples

---

```
>>> from sympy.geometry import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r
RegularPolygon(Point(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point(5, 0)
```

## Attributes

---

<code>vertices</code> (page 508)	The vertices of the RegularPolygon.
<code>center</code> (page 504)	The center of the RegularPolygon
<code>radius</code> (page 507)	Radius of the RegularPolygon
<code>rotation</code> (page 508)	CCW angle by which the RegularPolygon is rotated
<code>apothem</code> (page 503)	The inradius of the RegularPolygon.
<code>interior_angle</code> (page 506)	Measure of the interior angles.
<code>exterior_angle</code> (page 506)	Measure of the exterior angles.
<code>circumcircle</code> (page 504)	The circumcircle of the RegularPolygon.
<code>incircle</code> (page 506)	The incircle of the RegularPolygon.
<code>angles</code> (page 503)	Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

### angles

Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

## Examples

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.angles
{Point(-5/2, -5*sqrt(3)/2): pi/3,
 Point(-5/2, 5*sqrt(3)/2): pi/3,
 Point(5, 0): pi/3}
```

### apothem

The inradius of the RegularPolygon.

The apothem/inradius is the radius of the inscribed circle.

**Returns** `apothem` : number or instance of Basic

## See also:

`sympy.geometry.line.Segment.length` (page 463), `sympy.geometry.ellipse.Circle.radius` (page 494)

## Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.apothem
sqrt(2)*r/2
```

**area**

Returns the area.

**Examples**

```
>>> from sympy.geometry import RegularPolygon
>>> square = RegularPolygon((0, 0), 1, 4)
>>> square.area
2
>>> _ == square.length**2
True
```

**args**

Returns the center point, the radius, the number of sides, and the orientation angle.

**Examples**

```
>>> from sympy import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.args
(Point(0, 0), 5, 3, 0)
```

**center**

The center of the RegularPolygon

This is also the center of the circumscribing circle.

**Returns** center : Point

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.ellipse.Ellipse.center](#) (page 484)

**Examples**

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point(0, 0)
```

**centroid**

The center of the RegularPolygon

This is also the center of the circumscribing circle.

**Returns** center : Point

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.ellipse.Ellipse.center](#) (page 484)

### Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point(0, 0)
```

#### circumcenter

Alias for center.

### Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point(0, 0)
```

#### circumcircle

The circumcircle of the RegularPolygon.

**Returns** `circumcircle` : Circle

**See also:**

`circumcenter` (page 504), `sympy.geometry.ellipse.Circle` (page 493)

### Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point(0, 0), 4)
```

#### circumradius

Alias for radius.

### Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.circumradius
r
```

#### encloses\_point(p)

Return True if p is enclosed by (is inside of) self.

**Parameters** `p` : Point

**Returns** `encloses_point` : True, False or None

**See also:**

`sympy.geometry.ellipse.Ellipse.encloses_point` (page 485)

## Notes

Being on the border of self is considered False.

The general Polygon.encloses\_point method is called only if a point is not within or beyond the incircle or circumcircle, respectively.

## Examples

```
>>> from sympy import RegularPolygon, S, Point, Symbol
>>> p = RegularPolygon((0, 0), 3, 4)
>>> p.encloses_point(Point(0, 0))
True
>>> r, R = p.inradius, p.circumradius
>>> p.encloses_point(Point((r + R)/2, 0))
True
>>> p.encloses_point(Point(R/2, R/2 + (R - r)/10))
False
>>> t = Symbol('t', extended_real=True)
>>> p.encloses_point(p.arbitrary_point().subs(t, S.Half))
False
>>> p.encloses_point(Point(5, 5))
False
```

### exterior\_angle

Measure of the exterior angles.

**Returns** `exterior_angle` : number

**See also:**

[sympy.geometry.line.LinearEntity.angle\\_between](#) (page 449)

## Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.exterior_angle
pi/4
```

### incircle

The incircle of the RegularPolygon.

**Returns** `incircle` : Circle

**See also:**

[inradius](#) (page 506), [sympy.geometry.ellipse.Circle](#) (page 493)

## Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incircle
Circle(Point(0, 0), 4*cos(pi/7))
```

**inradius**  
Alias for apothem.

#### Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.inradius
sqrt(2)*r/2
```

**interior\_angle**  
Measure of the interior angles.

**Returns** `interior_angle` : number

**See also:**

[sympy.geometry.line.LinearEntity.angle\\_between](#) (page 449)

#### Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.interior_angle
3*pi/4
```

**length**  
Returns the length of the sides.

The half-length of the side and the apothem form two legs of a right triangle whose hypotenuse is the radius of the regular polygon.

#### Examples

```
>>> from sympy.geometry import RegularPolygon
>>> from sympy import sqrt
>>> s = square_in_unit_circle = RegularPolygon((0, 0), 1, 4)
>>> s.length
sqrt(2)
>>> sqrt(_/2)**2 + s.apothem**2) == s.radius
True
```

**radius**  
Radius of the RegularPolygon

This is also the radius of the circumscribing circle.

**Returns** `radius` : number or instance of Basic

**See also:**

[sympy.geometry.line.Segment.length](#) (page 463), [sympy.geometry.ellipse.Circle.radius](#) (page 494)

## Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.radius
r

reflect(line)
Override GeometryEntity.reflect since this is not made of only points.
```

```
>>> from sympy import RegularPolygon, Line

>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point(4/5, 2/5), -1, 4, acos(3/5))
```

**rotate(*angle*, *pt=None*)**

Override GeometryEntity.rotate to first rotate the RegularPolygon about its center.

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = RegularPolygon(Point(1, 0), 1, 3)
>>> t.vertices[0] # vertex on x-axis
Point(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point(0, 2)
```

**See also:**

[rotation](#) (page 508)

[spin](#) (**page 508**) Rotates a RegularPolygon in place

**rotation**

CCW angle by which the RegularPolygon is rotated

**Returns** **rotation** : number or instance of Basic

## Examples

```
>>> from sympy import pi
>>> from sympy.geometry import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
pi
```

**scale(*x=1*, *y=1*, *pt=None*)**

Override GeometryEntity.scale since it is the radius that must be scaled (if *x == y*) or else a new Polygon must be returned.

```
>>> from sympy import RegularPolygon
```

Symmetric scaling returns a RegularPolygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 2)
RegularPolygon(Point(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point(2, 0), Point(0, 1), Point(-2, 0), Point(0, -1))
```

### spin(*angle*)

Increment *in place* the virtual Polygon's rotation by ccw angle.

See also: `rotate` method which moves the center.

```
>>> from sympy import Polygon, Point, pi
>>> r = Polygon(Point(0,0), 1, n=3)
>>> r.vertices[0]
Point(1, 0)
>>> r.spin(pi/6)
>>> r.vertices[0]
Point(sqrt(3)/2, 1/2)
```

See also:

[rotation](#) (page 508)

[rotate](#) (page 507) Creates a copy of the `RegularPolygon` rotated about a `Point`

### vertices

The vertices of the `RegularPolygon`.

Returns `vertices` : list

Each vertex is a `Point`.

See also:

[sympy.geometry.point.Point](#) (page 437)

## Examples

```
>>> from sympy.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point(5, 0), Point(0, 5), Point(-5, 0), Point(0, -5)]
```

## class `sympy.geometry.polygon.Triangle`

A polygon with three vertices and three sides.

Parameters `points` : sequence of `Points`

keyword: `asa`, `sas`, or `sss` to specify sides/angles of the triangle

Raises `GeometryError`

If the number of vertices is not equal to three, or one of the vertices is not a `Point`, or a valid keyword is not given.

See also:

[sympy.geometry.point.Point](#) (page 437), [Polygon](#) (page 495)

## Examples

```
>>> from sympy.geometry import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
```

Keywords sss, sas, or asa can be used to give the desired side lengths (in order) and interior angles (in degrees) that define the triangle:

```
>>> Triangle(sss=(3, 4, 5))
Triangle(Point(0, 0), Point(3, 0), Point(3, 4))
>>> Triangle(asa=(30, 1, 30))
Triangle(Point(0, 0), Point(1, 0), Point(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point(0, 0), Point(2, 0), Point(sqrt(2)/2, sqrt(2)/2))
```

## Attributes

---

<code>vertices</code> (page 515)	The triangle's vertices
<code>altitudes</code> (page 510)	The altitudes of the triangle.
<code>orthocenter</code> (page 515)	The orthocenter of the triangle.
<code>circumcenter</code> (page 510)	The circumcenter of the triangle
<code>circumradius</code> (page 511)	The radius of the circumcircle of the triangle.
<code>circumcircle</code> (page 511)	The circle which passes through the three vertices of the triangle.
<code>inradius</code> (page 512)	The radius of the incircle.
<code>incircle</code> (page 512)	The incircle of the triangle.
<code>medians</code> (page 514)	The medians of the triangle.
<code>medial</code> (page 514)	The medial triangle of the triangle.

---

### altitudes

The altitudes of the triangle.

An altitude of a triangle is a segment through a vertex, perpendicular to the opposite side, with length being the height of the vertex measured from the line containing the side.

**Returns altitudes : dict**

The dictionary consists of keys which are vertices and values which are Segments.

**See also:**

`sympy.geometry.point.Point` (page 437), `sympy.geometry.line.Segment.length` (page 463)

## Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.altitudes[p1]
Segment(Point(0, 0), Point(1/2, 1/2))
```

### bisectors()

The angle bisectors of the triangle.

An angle bisector of a triangle is a straight line through a vertex which cuts the corresponding angle in half.

**Returns bisectors** : dict

Each key is a vertex (Point) and each value is the corresponding bisector (Segment).

**See also:**

[sympy.geometry.point.Point](#) (page 437), [sympy.geometry.line.Segment](#) (page 461)

**Examples**

```
>>> from sympy.geometry import Point, Triangle, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> from sympy import sqrt
>>> t.bisectors()[p2] == Segment(Point(0, sqrt(2) - 1), Point(1, 0))
True
```

**circumcenter**

The circumcenter of the triangle

The circumcenter is the center of the circumcircle.

**Returns circumcenter** : Point

**See also:**

[sympy.geometry.point.Point](#) (page 437)

**Examples**

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcenter
Point(1/2, 1/2)
```

**circumcircle**

The circle which passes through the three vertices of the triangle.

**Returns circumcircle** : Circle

**See also:**

[sympy.geometry.ellipse.Circle](#) (page 493)

**Examples**

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcircle
Circle(Point(1/2, 1/2), sqrt(2)/2)
```

**circumradius**

The radius of the circumcircle of the triangle.

**Returns circumradius** : number of Basic instance

See also:

`sympy.geometry.ellipse.Circle.radius` (page 494)

### Examples

```
>>> from sympy import Symbol
>>> from sympy.geometry import Point, Triangle
>>> a = Symbol('a')
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, a)
>>> t = Triangle(p1, p2, p3)
>>> t.circumradius
sqrt(a**2/4 + 1/4)
```

#### incenter

The center of the incircle.

The incircle is the circle which lies inside the triangle and touches all three sides.

**Returns** `incenter` : `Point`

See also:

`incircle` (page 512), `sympy.geometry.point.Point` (page 437)

### Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.incenter
Point(-sqrt(2)/2 + 1, -sqrt(2)/2 + 1)
```

#### incircle

The incircle of the triangle.

The incircle is the circle which lies inside the triangle and touches all three sides.

**Returns** `incircle` : `Circle`

See also:

`sympy.geometry.ellipse.Circle` (page 493)

### Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(2, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.incircle
Circle(Point(-sqrt(2) + 2, -sqrt(2) + 2), -sqrt(2) + 2)
```

#### inradius

The radius of the incircle.

**Returns** `inradius` : number of `Basic` instance

See also:

[incircle](#) (page 512), [sympy.geometry.ellipse.Circle.radius](#) (page 494)

Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(4, 0), Point(0, 3)
>>> t = Triangle(p1, p2, p3)
>>> t.inradius
1
```

`is_equilateral()`

Are all the sides the same length?

Returns `is_equilateral` : boolean

See also:

[sympy.geometry.entity.GeometryEntity.is\\_similar](#) (page 433), [RegularPolygon](#) (page 502),  
[is\\_isosceles](#) (page 513), [is\\_right](#) (page 513), [is\\_scalene](#) (page 513)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_equilateral()
False

>>> from sympy import sqrt
>>> t2 = Triangle(Point(0, 0), Point(10, 0), Point(5, 5*sqrt(3)))
>>> t2.is_equilateral()
True
```

`is_isosceles()`

Are two or more of the sides the same length?

Returns `is_isosceles` : boolean

See also:

[is\\_equilateral](#) (page 512), [is\\_right](#) (page 513), [is\\_scalene](#) (page 513)

Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(2, 4))
>>> t1.is_isosceles()
True
```

`is_right()`

Is the triangle right-angled.

Returns `is_right` : boolean

See also:

[sympy.geometry.line.LinearEntity.is\\_perpendicular](#) (page 452), [is\\_equilateral](#) (page 512), [is\\_isosceles](#) (page 513), [is\\_scalene](#) (page 513)

## Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_right()
True

is_scalene()
```

Are all the sides of the triangle of different lengths?

**Returns** `is_scalene` : boolean

**See also:**

`is_equilateral` (page 512), `is_isosceles` (page 513), `is_right` (page 513)

## Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

`is_similar(t1, t2)`

Is another triangle similar to this one.

Two triangles are similar if one can be uniformly scaled to the other.

**Parameters** `other: Triangle`

**Returns** `is_similar` : boolean

**See also:**

`sympy.geometry.entity.GeometryEntity.is_similar` (page 433)

## Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -3))
>>> t1.is_similar(t2)
True

>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -4))
>>> t1.is_similar(t2)
False
```

`medial`

The medial triangle of the triangle.

The triangle which is formed from the midpoints of the three sides.

**Returns** `medial` : Triangle

**See also:**

`sympy.geometry.line.Segment.midpoint` (page 463)

## Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medial
Triangle(Point(1/2, 0), Point(1/2, 1/2), Point(0, 1/2))
```

### medians

The medians of the triangle.

A median of a triangle is a straight line through a vertex and the midpoint of the opposite side, and divides the triangle into two equal areas.

**Returns** `medians` : dict

Each key is a vertex (Point) and each value is the median (Segment) at that point.

**See also:**

[sympy.geometry.point.Point.midpoint](#) (page 441), [sympy.geometry.line.Segment.midpoint](#) (page 463)

## Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medians[p1]
Segment(Point(0, 0), Point(1/2, 1/2))
```

### orthocenter

The orthocenter of the triangle.

The orthocenter is the intersection of the altitudes of a triangle. It may lie inside, outside or on the triangle.

**Returns** `orthocenter` : Point

**See also:**

[sympy.geometry.point.Point](#) (page 437)

## Examples

```
>>> from sympy.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.orthocenter
Point(0, 0)
```

### vertices

The triangle's vertices

**Returns** `vertices` : tuple

Each element in the tuple is a Point

See also:

`sympy.geometry.point.Point` (page 437)

### Examples

```
>>> from sympy.geometry import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point(0, 0), Point(4, 0), Point(4, 3))
```

## Plane

Geometrical Planes.

**Contains** Plane

**class** `sympy.geometry.plane.Plane`

A plane is a flat, two-dimensional surface. A plane is the two-dimensional analogue of a point (zero-dimensions), a line (one-dimension) and a solid (three-dimensions). A plane can generally be constructed by two types of inputs. They are three non-collinear points and a point and the plane's normal vector.

### Examples

```
>>> from sympy import Plane, Point3D
>>> from sympy.abc import x
>>> Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane((1, 1, 1), (2, 3, 4), (2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane(Point3D(1, 1, 1), normal_vector=(1,4,7))
Plane(Point3D(1, 1, 1), (1, 4, 7))
```

### Attributes

---

<code>p1</code> (page 519)	The only defining point of the plane.
<code>normal_vector</code> (page 519)	Normal vector of the given plane.

---

`angle_between(o)`

Angle between the plane and other geometric entity.

**Parameters** `LinearEntity3D, Plane.`

**Returns** `angle` : angle in radians

### Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the angle between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a

desired plane and then proceed to calculate the angle.

### Examples

```
>>> from sympy import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 2), normal_vector=(1, 2, 3))
>>> b = Line3D(Point3D(1, 3, 4), Point3D(2, 2, 2))
>>> a.angle_between(b)
-asin(sqrt(21)/6)
```

#### arbitrary\_point(*t=None*)

Returns an arbitrary point on the Plane; varying *t* from 0 to  $2\pi$  will move the point in a circle of radius 1 about p1 of the Plane.

**Returns** Point3D

### Examples

```
>>> from sympy.geometry.plane import Plane
>>> from sympy.abc import t
>>> p = Plane((0, 0, 0), (0, 0, 1), (0, 1, 0))
>>> p.arbitrary_point(t)
Point3D(0, cos(t), sin(t))
>>> _.distance(p.p1).simplify()
1
```

#### static are\_concurrent(\**planes*)

Is a sequence of Planes concurrent?

Two or more Planes are concurrent if their intersections are a common line.

**Parameters** planes: list

**Returns** Boolean

### Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(5, 0, 0), normal_vector=(1, -1, 1))
>>> b = Plane(Point3D(0, -2, 0), normal_vector=(3, 1, 1))
>>> c = Plane(Point3D(0, -1, 0), normal_vector=(5, -1, 9))
>>> Plane.are_concurrent(a, b)
True
>>> Plane.are_concurrent(a, b, c)
False
```

#### distance(*o*)

Distance between the plane and another geometric entity.

**Parameters** Point3D, LinearEntity3D, Plane.

**Returns** distance

## Notes

This method accepts only 3D entities as its parameter, but if you want to calculate the distance between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the distance.

## Examples

```
>>> from sympy import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.distance(b)
sqrt(3)
>>> c = Line3D(Point3D(2, 3, 1), Point3D(1, 2, 2))
>>> a.distance(c)
0
```

`equation(x=None, y=None, z=None)`

The equation of the Plane.

## Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 2), Point3D(2, 4, 7), Point3D(3, 5, 1))
>>> a.equation()
-23*x + 11*y - 2*z + 16
>>> a = Plane(Point3D(1, 4, 2), normal_vector=(6, 6, 6))
>>> a.equation()
6*x + 6*y + 6*z - 42
```

`intersection(o)`

The intersection with other geometrical entity.

**Parameters** `Point, Point3D, LinearEntity, LinearEntity3D, Plane`

**Returns** List

## Examples

```
>>> from sympy import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.intersection(b)
[Point3D(1, 2, 3)]
>>> c = Line3D(Point3D(1, 4, 7), Point3D(2, 2, 2))
>>> a.intersection(c)
[Point3D(2, 2, 2)]
>>> d = Plane(Point3D(6, 0, 0), normal_vector=(2, -5, 3))
>>> e = Plane(Point3D(2, 0, 0), normal_vector=(3, 4, -3))
>>> d.intersection(e)
[Line3D(Point3D(78/23, -24/23, 0), Point3D(147/23, 321/23, 23))]
```

`is_coplanar(o)`

Returns True if `o` is coplanar with self, else False.

## Examples

```
>>> from sympy import Plane, Point3D
>>> o = (0, 0, 0)
>>> p = Plane(o, (1, 1, 1))
>>> p2 = Plane(o, (2, 2, 2))
>>> p == p2
False
>>> p.is_coplanar(p2)
True
```

`is_parallel(l)`

Is the given geometric entity parallel to the plane?

**Parameters** LinearEntity3D or Plane

**Returns** Boolean

## Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(3,1,3), normal_vector=(4, 8, 12))
>>> a.is_parallel(b)
True
```

`is_perpendicular(l)`

is the given geometric entity perpendicular to the given plane?

**Parameters** LinearEntity3D or Plane

**Returns** Boolean

## Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(2, 2, 2), normal_vector=(-1, 2, -1))
>>> a.is_perpendicular(b)
True
```

`normal_vector`

Normal vector of the given plane.

## Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.normal_vector
(-1, 2, -1)
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 4, 7))
>>> a.normal_vector
(1, 4, 7)
```

p1

The only defining point of the plane. Others can be obtained from the arbitrary\_point method.

**See also:**

`sympy.geometry.point3d.Point3D` (page 443)

### Examples

```
>>> from sympy import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.p1
Point3D(1, 1, 1)
```

`parallel_plane(pt)`

Plane parallel to the given plane and passing through the point pt.

**Parameters** `pt: Point3D`

**Returns** Plane

### Examples

```
>>> from sympy import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.parallel_plane(Point3D(2, 3, 5))
Plane(Point3D(2, 3, 5), (2, 4, 6))
```

`perpendicular_line(pt)`

A line perpendicular to the given plane.

**Parameters** `pt: Point3D`

**Returns** Line3D

### Examples

```
>>> from sympy import Plane, Point3D, Line3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> a.perpendicular_line(Point3D(9, 8, 7))
Line3D(Point3D(9, 8, 7), Point3D(11, 12, 13))
```

`perpendicular_plane(*pts)`

Return a perpendicular passing through the given points. If the direction ratio between the points is the same as the Plane's normal vector then, to select from the infinite number of possible planes, a third point will be chosen on the z-axis (or the y-axis if the normal vector is already parallel to the z-axis). If less than two points are given they will be supplied as follows: if no point is given then pt1 will be self.p1; if a second point is not given it will be a point through pt1 on a line parallel to the z-axis (if the normal is not already the z-axis, otherwise on the line parallel to the y-axis).

**Parameters** `pts: 0, 1 or 2 Point3D`

**Returns** Plane

### Examples

```
>>> from sympy import Plane, Point3D, Line3D
>>> a, b = Point3D(0, 0, 0), Point3D(0, 1, 0)
>>> Z = (0, 0, 1)
>>> p = Plane(a, normal_vector=Z)
>>> p.perpendicular_plane(a, b)
Plane(Point3D(0, 0, 0), (1, 0, 0))

projection(pt)
Project the given point onto the plane along the plane normal.
```

**Parameters** Point or Point3D

**Returns** Point3D

### Examples

```
>>> from sympy import Plane, Point, Point3D
>>> A = Plane(Point3D(1, 1, 2), normal_vector=(1, 1, 1))
```

The projection is along the normal vector direction, not the z axis, so (1, 1) does not project to (1, 1, 2) on the plane A:

```
>>> b = Point(1, 1)
>>> A.projection(b)
Point3D(5/3, 5/3, 2/3)
>>> _ in A
True
```

But the point (1, 1, 2) projects to (1, 1) on the XY-plane:

```
>>> XY = Plane((0, 0, 0), (0, 0, 1))
>>> XY.projection((1, 1, 2))
Point3D(1, 1, 0)
```

### projection\_line(line)

Project the given line onto the plane through the normal plane containing the line.

**Parameters** LinearEntity or LinearEntity3D

**Returns** Point3D, Line3D, Ray3D or Segment3D

### Notes

For the interaction between 2D and 3D lines(segments, rays), you should convert the line to 3D by using this method. For example for finding the intersection between a 2D and a 3D line, convert the 2D line to a 3D line by projecting it on a required plane and then proceed to find the intersection between those lines.

### Examples

```
>>> from sympy import Plane, Line, Line3D, Point, Point3D
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Line(Point(1, 1), Point(2, 2))
```

```
>>> a.projection_line(b)
Line3D(Point3D(4/3, 4/3, 1/3), Point3D(5/3, 5/3, -1/3))
>>> c = Line3D(Point3D(1, 1, 1), Point3D(2, 2, 2))
>>> a.projection_line(c)
Point3D(1, 1, 1)

random_point(seed=None)
    Returns a random point on the Plane.

    Returns Point3D
```

## 3.10 Symbolic Integrals

The `integrals` module in SymPy implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is `integrate()` (page 543)

- `integrate(f, x)` returns the indefinite integral  $\int f dx$
- `integrate(f, (x, a, b))` returns the definite integral  $\int_a^b f dx$

### 3.10.1 Examples

SymPy can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> from sympy import *
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
 3      2
x      x
--- + --- + x
 3      2
```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)
      1
log(x + 1) + -----
      x + 1
```

Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions `exp`, `cos` and `sin` can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, SymPy will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)
 2      2      x      x
x *e *sin(x)  x *e *cos(x)  x      e *sin(x)  e *cos(x)
----- + ----- - x*e *sin(x) + ----- - -----
 2          2          2          2
```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

---

```
>>> integrate(exp(-x**2)*erf(x), x)
      2
    --- 
  \ / pi *erf (x)
  -----
        4
```

### 3.10.2 Integral Transforms

Sympy has special support for definite integrals, and integral transforms.

`sympy.integrals.transforms.mellin_transform(f, x, s, **hints)`

Compute the Mellin transform  $F(s)$  of  $f(x)$ ,

$$F(s) = \int_0^\infty x^{s-1} f(x) dx.$$

For all “sensible” functions, this converges absolutely in a strip  $a < \operatorname{Re}(s) < b$ .

The Mellin transform is related via change of variables to the Fourier transform, and also to the (bilateral) Laplace transform.

This function returns `(F, (a, b), cond)` where  $F$  is the Mellin transform of  $f$ , `(a, b)` is the fundamental strip (as above), and `cond` are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated `MellinTransform` (page 555) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 555). If `noconds=False`, then only  $F$  will be returned (i.e. not `cond`, and also not the strip `(a, b)`).

```
>>> from sympy.integrals.transforms import mellin_transform
>>> from sympy import exp
>>> from sympy.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)
```

See also:

`inverse_mellin_transform` (page 523), `laplace_transform` (page 524), `fourier_transform` (page 525), `hankel_transform` (page 527), `inverse_hankel_transform` (page 528)

`sympy.integrals.transforms.inverse_mellin_transform(F, s, x, strip, **hints)`

Compute the inverse Mellin transform of  $F(s)$  over the fundamental strip given by `strip=(a, b)`.

This can be defined as

$$f(x) = \int_{c-i\infty}^{c+i\infty} x^{-s} F(s) ds,$$

for any  $c$  in the fundamental strip. Under certain regularity conditions on  $F$  and/or  $f$ , this recovers  $f$  from its Mellin transform  $F$  (and vice versa), for positive real  $x$ .

One of  $a$  or  $b$  may be passed as `None`; a suitable  $c$  will be inferred.

If the integral cannot be computed in closed form, this function returns an unevaluated `InverseMellinTransform` (page 555) object.

Note that this function will assume  $x$  to be positive and real, regardless of the `sympy` assumptions!

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 555).

```
>>> from sympy.integrals.transforms import inverse_mellin_transform
>>> from sympy import oo, gamma
>>> from sympy.abc import x, s
>>> inverse_mellin_transform(gamma(s), s, x, (0, oo))
exp(-x)
```

The fundamental strip matters:

```
>>> f = 1/(s**2 - 1)
>>> inverse_mellin_transform(f, s, x, (-oo, -1))
(x/2 - 1/(2*x))*Heaviside(x - 1)
>>> inverse_mellin_transform(f, s, x, (-1, 1))
-x*Heaviside(-x + 1)/2 - Heaviside(x - 1)/(2*x)
>>> inverse_mellin_transform(f, s, x, (1, oo))
(-x/2 + 1/(2*x))*Heaviside(-x + 1)
```

See also:

[mellin\\_transform](#) (page 522), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528)

`sympy.integrals.transforms.laplace_transform(f, t, s, **hints)`

Compute the Laplace Transform  $F(s)$  of  $f(t)$ ,

$$F(s) = \int_0^\infty e^{-st} f(t) dt.$$

For all “sensible” functions, this converges absolutely in a half plane  $a < \operatorname{Re}(s)$ .

This function returns  $(F, a, \text{cond})$  where  $F$  is the Laplace transform of  $f$ ,  $\operatorname{Re}(s) > a$  is the half-plane of convergence, and  $\text{cond}$  are auxiliary convergence conditions.

If the integral cannot be computed in closed form, this function returns an unevaluated [LaplaceTransform](#) (page 555) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). If `noconds=True`, only  $F$  will be returned (i.e. not  $\text{cond}$ , and also not the plane  $a$ ).

```
>>> from sympy.integrals import laplace_transform
>>> from sympy.abc import t, s, a
>>> laplace_transform(t**a, t, s)
(s**(-a)*gamma(a + 1)/s, 0, -re(a) < 1)
```

See also:

[inverse\\_laplace\\_transform](#) (page 524), [mellin\\_transform](#) (page 522), [fourier\\_transform](#) (page 525), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528)

`sympy.integrals.transforms.inverse_laplace_transform(F, s, t, plane=None, **hints)`

Compute the inverse Laplace transform of  $F(s)$ , defined as

$$f(t) = \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,$$

for  $c$  so large that  $F(s)$  has no singularities in the half-plane  $\operatorname{Re}(s) > c - \epsilon$ .

The plane can be specified by argument `plane`, but will be inferred if passed as `None`.

Under certain regularity conditions, this recovers  $f(t)$  from its Laplace Transform  $F(s)$ , for non-negative  $t$ , and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated [InverseLaplaceTransform](#) (page 555) object.

Note that this function will always assume  $t$  to be real, regardless of the sympy assumption on  $t$ .

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555).

```
>>> from sympy.integrals.transforms import inverse_laplace_transform
>>> from sympy import exp, Symbol
>>> from sympy.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

**See also:**

[laplace\\_transform](#) (page 524), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528)

[sympy.integrals.transforms.fourier\\_transform\(f, x, k, \\*\\*hints\)](#)

Compute the unitary, ordinary-frequency Fourier transform of  $f$ , defined as

$$F(k) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i xk} dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [FourierTransform](#) (page 555) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import fourier_transform, exp
>>> from sympy.abc import x, k
>>> fourier_transform(exp(-x**2), x, k)
sqrt(pi)*exp(-pi**2*k**2)
>>> fourier_transform(exp(-x**2), x, k, noconds=False)
(sqrt(pi)*exp(-pi**2*k**2), True)
```

**See also:**

[inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

[sympy.integrals.transforms.inverse\\_fourier\\_transform\(F, k, x, \\*\\*hints\)](#)

Compute the unitary, ordinary-frequency inverse Fourier transform of  $F$ , defined as

$$f(x) = \int_{-\infty}^{\infty} F(k)e^{2\pi i xk} dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseFourierTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_fourier_transform, exp, sqrt, pi
>>> from sympy.abc import x, k
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x)
exp(-x**2)
>>> inverse_fourier_transform(sqrt(pi)*exp(-(pi*k)**2), k, x, noconds=False)
(exp(-x**2), True)
```

See also:

[fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

`sympy.integrals.transforms.sine_transform(f, x, k, **hints)`

Compute the unitary, ordinary-frequency sine transform of  $f$ , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \sin(2\pi xk) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [SineTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import sine_transform, exp
>>> from sympy.abc import x, k, a
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*k*exp(-k**2/(4*a))/(4*a**(3/2))
>>> sine_transform(x**(-a), x, k)
2**(-a + 1/2)*k**(-a - 1)*gamma(-a/2 + 1)/gamma(a/2 + 1/2)
```

See also:

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

`sympy.integrals.transforms.inverse_sine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse sine transform of  $F$ , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \sin(2\pi xk) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseSineTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of `sympy.integrals.transforms.IntegralTransform.doit()` (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_sine_transform, exp, sqrt, gamma, pi
>>> from sympy.abc import x, k, a
>>> inverse_sine_transform(2**((1-2*a)/2)*k**(a - 1)*
...     gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)
>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k, x)
x*exp(-a*x**2)
```

See also:

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

```
sympy.integrals.transforms.cosine_transform(f, x, k, **hints)
```

Compute the unitary, ordinary-frequency cosine transform of  $f$ , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} f(x) \cos(2\pi xk) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [CosineTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import cosine_transform, exp, sqrt, cos
>>> from sympy.abc import x, k, a
>>> cosine_transform(exp(-a*x), x, k)
sqrt(2)*a/(sqrt(pi)*(a**2 + k**2))
>>> cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
a*exp(-a**2/(2*k))/(2*k**(3/2))
```

See also:

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

```
sympy.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)
```

Compute the unitary, ordinary-frequency inverse cosine transform of  $F$ , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \cos(2\pi xk) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseCosineTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import inverse_cosine_transform, exp, sqrt, pi
>>> from sympy.abc import x, k, a
>>> inverse_cosine_transform(sqrt(2)*a/(sqrt(pi)*(a**2 + k**2)), k, x)
exp(-a*x)
>>> inverse_cosine_transform(1/sqrt(k), k, x)
1/sqrt(x)
```

See also:

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [hankel\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

```
sympy.integrals.transforms.hankel_transform(f, r, k, nu, **hints)
```

Compute the Hankel transform of  $f$ , defined as

$$F_\nu(k) = \int_0^\infty f(r) J_\nu(kr) r dr.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [HankelTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import hankel_transform, inverse_hankel_transform
>>> from sympy import gamma, exp, sinh, cosh
>>> from sympy.abc import r, k, m, nu, a

>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)

>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)

>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))

>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

See also:

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [inverse\\_hankel\\_transform](#) (page 528), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

```
sympy.integrals.transforms.inverse_hankel_transform(F, k, r, nu, **hints)
```

Compute the inverse Hankel transform of  $F$  defined as

$$f(r) = \int_0^\infty F_\nu(k) J_\nu(kr) k dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated [InverseHankelTransform](#) (page 556) object.

For a description of possible hints, refer to the docstring of [sympy.integrals.transforms.IntegralTransform.doit\(\)](#) (page 555). Note that for this transform, by default `noconds=True`.

```
>>> from sympy import hankel_transform, inverse_hankel_transform, gamma
>>> from sympy import gamma, exp, sinh, cosh
>>> from sympy.abc import r, k, m, nu, a

>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)
```

---

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)

>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))

>>> inverse_hankel_transform(ht, k, r, 0)
exp(-a*r)
```

**See also:**

[fourier\\_transform](#) (page 525), [inverse\\_fourier\\_transform](#) (page 525), [sine\\_transform](#) (page 526), [inverse\\_sine\\_transform](#) (page 526), [cosine\\_transform](#) (page 526), [inverse\\_cosine\\_transform](#) (page 527), [hankel\\_transform](#) (page 527), [mellin\\_transform](#) (page 522), [laplace\\_transform](#) (page 524)

### 3.10.3 Internals

There is a general method for calculating antiderivatives of elementary functions, called the *Risch algorithm*. The Risch algorithm is a decision procedure that can determine whether an elementary solution exists, and in that case calculate it. It can be extended to handle many nonelementary functions in addition to the elementary ones.

SymPy currently uses a simplified version of the Risch algorithm, called the *Risch-Norman algorithm*. This algorithm is much faster, but may fail to find an antiderivative, although it is still very powerful. SymPy also uses pattern matching and heuristics to speed up evaluation of some types of integrals, e.g. polynomials.

For non-elementary definite integrals, SymPy uses so-called Meijer G-functions. Details are described here:

#### Computing Integrals using Meijer G-Functions

This text aims do describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

##### Overview

The algorithm to compute  $\int f(x)dx$  or  $\int_0^\infty f(x)dx$  generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function `hyperexpand` (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g.  $\int_{-\infty}^{\infty}$ ) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

##### Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being 0,  $\pm 1$ ,  $\infty$ ). This is not very important when e.g. expanding a single hypergeometric function into named

special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set  $\mathcal{S} = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$ . We have a map  $p : \mathcal{S} \rightarrow \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$ . Decreeing this to be a local biholomorphism gives  $\mathcal{S}$  both a topology and a complex structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps  $\text{Exp} : \mathbb{C} \rightarrow \mathcal{S}, (x + iy) \mapsto (\exp(x), y)$  and  $\text{Log} : \mathcal{S} \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$ . These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to  $\mathcal{S}$  and denote the resulting object  $\mathcal{S}_0$ . Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by  $\{\text{Exp}(z) : \Re(z) < k\}$ , which at least defines a topology. Elements of  $\mathcal{S}_0$  shall be called polar numbers. We further define functions  $\text{Arg} : \mathcal{S} \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$  and  $|.| : \mathcal{S}_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$ . These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from  $\mathbb{C}$  to  $\mathcal{S}$ . We define  $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a+b)$  for  $a, b \in \mathbb{C}$ , also for  $a \in \mathcal{S}$  and  $b \in \mathbb{C}$  we define  $a^b = \text{Exp}(b\text{Log}(a))$ . It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g.  $(ab)^c = a^c b^c$ ) which hold in  $\mathbb{C}$  only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions  $F(z) = \sqrt{1+z}$  and  $G(a, b) = \sqrt{a+b}$ , where  $a, b, z$  are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if  $S(z)$  denotes the standard branch of the square root function on  $\mathbb{C}$ , we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting  $|z| = 1$  here, this does not matter for integration.) Finally we define  $G(a, b) = \sqrt{a}F(b/a)$ .

### Representing Branched Functions on the Argand Plane

Suppose  $f : \mathcal{S} \rightarrow \mathbb{C}$  is a holomorphic function. We wish to define a function  $F$  on (part of) the complex numbers  $\mathbb{C}$  that represents  $f$  as closely as possible. This process is known as “introducing branch cuts”. In our situation, there is actually a canonical way of doing this (which is adhered to in all of SymPy), as follows: Introduce the “cut complex plane”  $C = \mathbb{C} \setminus \mathbb{R}_{\leq 0}$ . Define a function  $l : C \rightarrow \mathcal{S}$  via  $re^{i\theta} \mapsto r \text{Exp}(i\theta)$ . Here  $r > 0$  and  $-\pi < \theta \leq \pi$ . Then  $l$  is holomorphic, and we define  $G = f \circ l$ . This called “lifting to the principal branch” throughout the SymPy documentation.

### Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand  $f(x)$  and are trying to rewrite it as a single G-function. To do this, we first split  $f(x)$  into the form  $x^s g(x)$  (where  $g(x)$  is supposed to be simpler than  $f(x)$ ). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by `_split_mul(f, x)`. Then we assemble a tuple of functions that occur in  $f$  (e.g. if  $f(x) = e^x \cos x$ , we would assemble the tuple `(cos, exp)`). This is done by the function `_mytype(f, x)`. Next we index a lookup table (created using `_create_lookup_table()`) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting  $f(x)$  into a product  $f_1(x)f_2(x)$ . We could try these splittings in any order, but it is often a good idea to minimise (a) the number of powers occuring in  $f_i(x)$  and (b) the number of different functions occuring in  $f_i(x)$ . Thus given e.g.  $f(x) = \sin x e^x \sin 2x$  we should try  $f_1(x) = \sin x \sin 2x$ ,  $f_2(x) = e^x$  first. All of this is done by the function `_mul_as_two_parts(f)`.

Finally, we can try a recursive Mellin transform technique. Since the Meijer G-function is defined essentially as a certain inverse mellin transform, if we want to write a function  $f(x)$  as a G-function, we can compute its mellin transform  $F(s)$ . If  $F(s)$  is in the right form, the G-function expression can be read off. This technique generalises many standard rewritings, e.g.  $e^{ax}e^{bx} = e^{(a+b)x}$ .

One twist is that some functions don't have mellin transforms, even though they can be written as G-functions. This is true for example for  $f(x) = e^x \sin x$  (the function grows too rapidly to have a mellin transform). However if the function is recognised to be analytic, then we can try to compute the mellin-transform of  $f(ax)$  for a parameter  $a$ , and deduce the G-function expression by analytic continuation. (Checking for analyticity is easy. Since we can only deal with a certain subset of functions anyway, we only have to filter out those which are not analytic.)

The function `_rewrite_single` does the table lookup and recursive mellin transform. The functions `_rewrite1` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

### Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see `_rewrite_saxena_1` and `_rewrite_saxena`, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [BlogPost] (page 1244)).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing  $e^z$  is satisfies  $G(\text{Exp}(2\pi i)z) = G(z)$  for all  $z \in \mathcal{S}$ . The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z, period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

### The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually stated in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.

$$\int_0^\infty G_{p,q}^{m,n} \left( \begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \eta x \right) dx = \frac{\prod_{j=1}^m \Gamma(b_j + 1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j + 1)}$$

2.

$$\int_0^\infty G_{u,v}^{s,t} \left( \begin{matrix} c_1, \dots, c_u \\ d_1, \dots, d_v \end{matrix} \middle| \sigma x \right) G_{p,q}^{m,n} \left( \begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \omega x \right) dx = G_{v+p, u+q}^{m+t, n+s} \left( \begin{matrix} a_1, \dots, a_n, -d_1, \dots, -d_v, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, -c_1, \dots, -c_u, b_{m+1}, \dots, b_q \end{matrix} \middle| \frac{\omega}{\sigma} \right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in SymPy. They are an amalgamation of conditions found in [Prudnikov1990] (page 1247) and [Luke1969] (page 1247); please let us know if you find any errors.

### Conditions of Convergence for Integral (1)

We can without loss of generality assume  $p \leq q$ , since the G-functions of indices  $m, n, p, q$  and of indices  $n, m, q, p$  can be related easily (see e.g. [Luke1969] (page 1247), section 5.3). We introduce the following notation:

$$\begin{aligned}\xi &= m + n - p \\ \delta &= m + n - \frac{p + q}{2}\end{aligned}$$

$$\begin{aligned}C_3 : -\Re(b_j) &< 1 \text{ for } j = 1, \dots, m \\ 0 &< -\Re(a_j) \text{ for } j = 1, \dots, n\end{aligned}$$

$$\begin{aligned}C_3^* : -\Re(b_j) &< 1 \text{ for } j = 1, \dots, q \\ 0 &< -\Re(a_j) \text{ for } j = 1, \dots, p\end{aligned}$$

$$C_4 : -\Re(\delta) + \frac{q + 1 - p}{2} > q - p$$

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of  $x$ , whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on  $(0, \epsilon)$  for any  $\epsilon > 0$ . Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions  $n$  “at zero” and “at infinity” both have to be fulfilled, for some  $n$ .

These are the conditions “at infinity”:

1.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$

$$B = 1 \leq p \wedge 1 \leq m \wedge q = p + 1 \wedge \neg(n = 0 \wedge m = p + 1)$$

$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \dots, \left\lceil \frac{\delta}{2} \right\rceil.$$

2.

$$n = 0 \wedge p + 1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.

$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q - 2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.

$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re \left( \sum_{j=1}^p b_j - a_j \right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions “at zero”:

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

### Conditions of Convergence for Integral (2)

We introduce the following notation:

$$b^* = s + t - \frac{u + v}{2}$$

$$c^* = m + n - \frac{p + q}{2}$$

$$\rho = \sum_{j=1}^v d_j - \sum_{j=1}^u c_j + \frac{u - v}{2} + 1$$

$$\mu = \sum_{j=1}^q b_j - \sum_{j=1}^p a_j + \frac{p - q}{2} + 1$$

$$\phi = q - p - \frac{u - v}{2} + 1$$

$$\eta = 1 - (v - u) - \mu - \rho$$

$$\psi = \frac{\pi(q-m-n) + |\arg(\omega)|}{q-p}$$

$$\theta = \frac{\pi(v-s-t) + |\arg(\sigma)|}{v-u}$$

$$\lambda_c = (q-p)|\omega|^{1/(q-p)} \cos \psi + (v-u)|\sigma|^{1/(v-u)} \cos \theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q-p)|\omega|^{1/(q-p)} \sin \psi + c_2(v-u)|\sigma|^{1/(v-u)} \sin \theta$$

$$\lambda_s = \begin{cases} \lambda_{s0}(-1, -1) \lambda_{s0}(1, 1) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(\operatorname{sign}(\arg(\omega)), -1) \lambda_{s0}(\operatorname{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(-1, \operatorname{sign}(\arg(\sigma))) \lambda_{s0}(1, \operatorname{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0 \\ \lambda_{s0}(\operatorname{sign}(\arg(\omega)), \operatorname{sign}(\arg(\sigma))) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma} e^{-i\pi(b^*+c^*)}$$

$$z_1 = \frac{\sigma}{\omega} e^{-i\pi(b^*+c^*)}$$

The following conditions will be helpful:

$$\begin{aligned} C_1 : (a_i - b_j) &\notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, n, j = 1, \dots, m \\ &\wedge (c_i - d_j) \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, t, j = 1, \dots, s \end{aligned}$$

$$C_2 : \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \dots, m, j = 1, \dots, s$$

$$C_3 : \Re(a_i + c_j) < 1 \text{ for } i = 1, \dots, n, j = 1, \dots, t$$

$$C_4 : (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, t$$

$$C_5 : (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, s$$

$$C_6 : (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, n$$

$$C_7 : (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, m$$

$$C_8 : 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_9 : 0 < |\phi| - 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_{10} : |\arg(\sigma)| < \pi b^*$$

$$C_{11} : |\arg(\sigma)| = \pi b^*$$

$$C_{12} : |\arg(\omega)| < c^* \pi$$

$$C_{13} : |\arg(\omega)| = c^* \pi$$

$$C_{14}^1 : (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1)$$

$$C_{14}^2 : (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1)$$

$$C_{14} : \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2)$$

$$C_{15} : \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0)$$

$$C_{16} : \int_0^\infty G_{u,v}^{s,t}(\sigma x) dx \text{ converges at infinity}$$

$$C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x) dx \text{ converges at infinity}$$

Note that  $C_{16}$  and  $C_{17}$  are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.

$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re\mu < 1 \wedge \Re\rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.

$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.

$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.

$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.

$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.

$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.

$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re\rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.

$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re\mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.

$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re\mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.

$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{16}$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{16}$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{16}$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

## The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [Luke1969] (page 1247), section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument  $bz$ , and convergence conditions are also given. However, we need a formula for argument  $bz^a$  for rational  $a$ .

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) dz,$$

for positive real  $t$ . Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

### How to compute the integral

We shall work formally for now. Denote by  $\Delta(s)$  the product of gamma functions appearing in the definition of  $G$ , so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s ds.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds dz.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds.$$

The inner integral is easily seen to be  $\frac{1}{\Gamma(-as)} \frac{1}{t^{1+as}}$ . (Using Cauchy's theorem and Jordan's lemma deform the contour to run from  $-\infty$  to  $-\infty$ , encircling 0 once in the negative sense. For  $as$  real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t} \frac{1}{2\pi i} \int_L \Delta(s) \frac{1}{\Gamma(-as)} \left(\frac{b}{t^a}\right)^s ds,$$

which is a so-called Fox H function (of argument  $\frac{b}{t^a}$ ). For rational  $a$ , this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

### When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for  $\Re(as) < -1$ . Thus the contour  $L$  has to run left of the line  $\Re(as) = -1$ . Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient  $b$ . Hence so is  $f(t)$ . Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that  $a > 0$ , then the latter condition is fulfilled if  $G$  converges along contours (2) or (3) of [Luke1969] (page 1247), section 5.2, i.e. either  $\delta \geq \frac{a}{2}$  or  $p \geq 1, p \geq q$ .

### When the integral exists

Using [Luke1969] (page 1247), section 5.10, for any given meijer G-function we can find a dominant term of the form  $z^a e^{bz^c}$  (although this expression might not be the best possible, because of cancellation).

We must thus investigate

$$\lim_{T \rightarrow \infty} \int_{c-iT}^{c+iT} e^{zt} z^a e^{bz^c} dz.$$

(This principal value integral is the exact statement used in the Laplace inversion theorem.) We write  $z = c + i\tau$ . Then  $\arg(z) \rightarrow \pm\frac{\pi}{2}$ , and so  $e^{zt} \sim e^{it\tau}$  (where  $\sim$  shall always mean “asymptotically equivalent up to a positive real multiplicative constant”). Also  $z^{x+iy} \sim |\tau|^x e^{iy \log|\tau|} e^{\pm xi\frac{\pi}{2}}$ .

Set  $\omega_{\pm} = be^{\pm i\Re(c)\frac{\pi}{2}}$ . We have three cases:

1.  $b = 0$  or  $\Re(c) \leq 0$ . In this case the integral converges if  $\Re(a) \leq -1$ .
2.  $b \neq 0$ ,  $\Im(c) = 0$ ,  $\Re(c) > 0$ . In this case the integral converges if  $\Re(\omega_{\pm}) < 0$ .
3.  $b \neq 0$ ,  $\Im(c) = 0$ ,  $\Re(c) > 0$ ,  $\Re(\omega_{\pm}) \leq 0$ , and at least one of  $\Re(\omega_{\pm}) = 0$ . Here the same condition as in (1) applies.

## Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in SymPy. An entry “generated” means that the corresponding G-function has a variable number of parameters. This table is intended to shrink in future, when the algorithm’s capabilities of deriving new formulae improve. Of course it has to grow whenever a new class of special functions is to be dealt with. Elementary functions:

$$a = aG_{1,1}^{1,0} \left( \begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) + aG_{1,1}^{0,1} \left( \begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right)$$

$$(z^q p + b)^{-a} = \frac{b^{-a}}{\Gamma(a)} G_{1,1}^{1,1} \left( \begin{matrix} -a + 1 \\ 0 \end{matrix} \middle| \frac{z^q p}{b} \right)$$

$$\frac{-b^a + (z^q p)^a}{z^q p - b} = \frac{1}{\pi} b^{a-1} G_{2,2}^{2,2} \left( \begin{matrix} 0, a \\ 0, a \end{matrix} \middle| \frac{z^q p}{b} \right) \sin(\pi a)$$

$$\left( a + \sqrt{z^q p + a^2} \right)^b = -\frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2} \left( \begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)$$

$$\left( -a + \sqrt{z^q p + a^2} \right)^b = \frac{a^b b}{2\sqrt{\pi}} G_{2,2}^{1,2} \left( \begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} + 1 \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)$$

$$\frac{\left( a + \sqrt{z^q p + a^2} \right)^b}{\sqrt{z^q p + a^2}} = \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2} \left( \begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ 0 \end{matrix} \middle| \frac{z^q p}{a^2} \right)$$

$$\frac{\left( -a + \sqrt{z^q p + a^2} \right)^b}{\sqrt{z^q p + a^2}} = \frac{1}{\sqrt{\pi}} a^{b-1} G_{2,2}^{1,2} \left( \begin{matrix} \frac{b}{2} + \frac{1}{2}, \frac{b}{2} \\ b \end{matrix} \middle| \frac{z^q p}{a^2} \right)$$

$$\left( z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b = -\frac{a^{\frac{b}{2}} b}{2\sqrt{\pi}} G_{2,2}^{2,1} \left( \begin{matrix} \frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix} \middle| -\frac{b}{2} + 1 \middle| \frac{z^q p}{a} \right)$$

$$\left( -z^{\frac{q}{2}} \sqrt{p} + \sqrt{z^q p + a} \right)^b = \frac{a^{\frac{b}{2}} b}{2\sqrt{\pi}} G_{2,2}^{2,1} \left( \begin{matrix} -\frac{b}{2} + 1 \\ 0, \frac{1}{2} \end{matrix} \middle| \frac{b}{2} + 1 \middle| \frac{z^q p}{a} \right)$$

$$\frac{\left(z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^qp+a}\right)^b}{\sqrt{z^qp+a}} = \frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix} \frac{b}{2}+\frac{1}{2} & -\frac{b}{2}+\frac{1}{2} \\ 0, \frac{1}{2} & \end{matrix} \middle| \frac{z^qp}{a}\right)$$

$$\frac{\left(-z^{\frac{q}{2}}\sqrt{p} + \sqrt{z^qp+a}\right)^b}{\sqrt{z^qp+a}} = \frac{1}{\sqrt{\pi}}a^{\frac{b}{2}-\frac{1}{2}}G_{2,2}^{2,1}\left(\begin{matrix} -\frac{b}{2}+\frac{1}{2} & \frac{b}{2}+\frac{1}{2} \\ 0, \frac{1}{2} & \end{matrix} \middle| \frac{z^qp}{a}\right)$$

Functions involving  $|z^qp - b|$ :

$$|z^qp - b|^{-a} = \frac{\pi |b|^{-a}}{\cos\left(\frac{\pi a}{2}\right)\Gamma(a)}G_{2,2}^{1,1}\left(\begin{matrix} -a+1 & -\frac{a}{2}+\frac{1}{2} \\ 0 & -\frac{a}{2}+\frac{1}{2} \end{matrix} \middle| \frac{z^qp}{b}\right), \text{ if } \Re a < 1$$

Functions involving  $\text{Chi}(z^qp)$ :

$$\text{Chi}(z^qp) = -\frac{\pi^{\frac{3}{2}}}{2}G_{2,4}^{2,0}\left(\begin{matrix} \frac{1}{2}, 1 \\ 0, 0 \end{matrix} \middle| \frac{p^2}{4}z^{2q}\right)$$

Functions involving  $\text{Ci}(z^qp)$ :

$$\text{Ci}(z^qp) = -\frac{\sqrt{\pi}}{2}G_{1,3}^{2,0}\left(\begin{matrix} 1 \\ 0, 0 \end{matrix} \middle| \frac{p^2}{4}z^{2q}\right)$$

Functions involving  $\text{Ei}(z^qp)$ :

$$\text{Ei}(z^qp) = -i\pi G_{1,1}^{1,0}\left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z\right) - G_{1,2}^{2,0}\left(\begin{matrix} 1 \\ 0, 0 \end{matrix} \middle| z^q p e^{i\pi}\right) - i\pi G_{1,1}^{0,1}\left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z\right)$$

Functions involving  $\theta(z^qp - b)$ :

$$(z^qp - b)^{a-1}\theta(z^qp - b) = b^{a-1}G_{1,1}^{0,1}\left(\begin{matrix} a \\ 0 \end{matrix} \middle| \frac{z^qp}{b}\right)\Gamma(a), \text{ if } b > 0$$

$$(-z^qp + b)^{a-1}\theta(-z^qp + b) = b^{a-1}G_{1,1}^{1,0}\left(\begin{matrix} a \\ 0 \end{matrix} \middle| \frac{z^qp}{b}\right)\Gamma(a), \text{ if } b > 0$$

$$(z^qp - b)^{a-1}\theta\left(z - \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) = b^{a-1}G_{1,1}^{0,1}\left(\begin{matrix} a \\ 0 \end{matrix} \middle| \frac{z^qp}{b}\right)\Gamma(a), \text{ if } b > 0$$

$$(-z^qp + b)^{a-1}\theta\left(-z + \left(\frac{b}{p}\right)^{\frac{1}{q}}\right) = b^{a-1}G_{1,1}^{1,0}\left(\begin{matrix} a \\ 0 \end{matrix} \middle| \frac{z^qp}{b}\right)\Gamma(a), \text{ if } b > 0$$

Functions involving  $\theta(-z^qp + 1)$ ,  $\log(z^qp)$ :

$$\log^n(z^qp)\theta(-z^qp + 1) = \text{generated}$$

$$\log^n(z^qp)\theta(z^qp - 1) = \text{generated}$$

Functions involving  $\text{Shi}(z^qp)$ :

$$\text{Shi}(z^qp) = \frac{\sqrt{\pi}p}{4}z^qG_{1,3}^{1,1}\left(\begin{matrix} \frac{1}{2} & \\ 0 & -\frac{1}{2}, -\frac{1}{2} \end{matrix} \middle| \frac{p^2}{4}z^{2q}e^{i\pi}\right)$$

Functions involving  $\text{Si}(z^q p)$ :

$$\text{Si}(z^q p) = \frac{\sqrt{\pi}}{2} G_{1,3}^{1,1} \left( \begin{matrix} 1 \\ \frac{1}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $I_a(z^q p)$ :

$$I_a(z^q p) = \pi G_{1,3}^{1,0} \left( \begin{matrix} \frac{a}{2} & -\frac{a}{2} + \frac{1}{2} \\ -\frac{a}{2}, \frac{a}{2} + \frac{1}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $J_a(z^q p)$ :

$$J_a(z^q p) = G_{0,2}^{1,0} \left( \begin{matrix} \frac{a}{2} \\ -\frac{a}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $K_a(z^q p)$ :

$$K_a(z^q p) = \frac{1}{2} G_{0,2}^{2,0} \left( \begin{matrix} \frac{a}{2}, -\frac{a}{2} \\ \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $Y_a(z^q p)$ :

$$Y_a(z^q p) = G_{1,3}^{2,0} \left( \begin{matrix} \frac{a}{2}, -\frac{a}{2} \\ -\frac{a}{2} - \frac{1}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $\cos(z^q p)$ :

$$\cos(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left( \begin{matrix} 0 \\ \frac{1}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $\cosh(z^q p)$ :

$$\cosh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left( \begin{matrix} 0 & \frac{1}{2} \\ \frac{1}{2}, \frac{1}{2} \end{matrix} \mid \begin{matrix} p^2 \\ 4 z^{2q} \end{matrix} \right)$$

Functions involving  $E(z^q p)$ :

$$E(z^q p) = -\frac{1}{4} G_{2,2}^{1,2} \left( \begin{matrix} \frac{1}{2}, \frac{3}{2} \\ 0 \end{matrix} \mid \begin{matrix} -z^q p \\ 0 \end{matrix} \right)$$

Functions involving  $K(z^q p)$ :

$$K(z^q p) = \frac{1}{2} G_{2,2}^{1,2} \left( \begin{matrix} \frac{1}{2}, \frac{1}{2} \\ 0 \end{matrix} \mid \begin{matrix} -z^q p \\ 0 \end{matrix} \right)$$

Functions involving  $\text{erf}(z^q p)$ :

$$\text{erf}(z^q p) = \frac{1}{\sqrt{\pi}} G_{1,2}^{1,1} \left( \begin{matrix} 1 \\ \frac{1}{2} \end{matrix} \mid \begin{matrix} z^{2q} p^2 \\ 0 \end{matrix} \right)$$

Functions involving  $\text{erfc}(z^q p)$ :

$$\text{erfc}(z^q p) = \frac{1}{\sqrt{\pi}} G_{1,2}^{2,0} \left( \begin{matrix} 1 \\ 0, \frac{1}{2} \end{matrix} \mid \begin{matrix} z^{2q} p^2 \\ -z^q p^2 \end{matrix} \right)$$

Functions involving  $\text{erfi}(z^q p)$ :

$$\text{erfi}(z^q p) = \frac{z^q p}{\sqrt{\pi}} G_{1,2}^{1,1} \left( \begin{matrix} \frac{1}{2} \\ 0 \end{matrix} \mid \begin{matrix} -z^{2q} p^2 \\ -\frac{1}{2} \end{matrix} \right)$$

Functions involving  $e^{z^q p e^{i\pi}}$ :

$$e^{z^q p e^{i\pi}} = G_{0,1}^{1,0} \left( 0 \middle| z^q p \right)$$

Functions involving  $E_a(z^q p)$ :

$$E_a(z^q p) = G_{1,2}^{2,0} \left( a - 1, 0 \middle| z^q p \right)$$

Functions involving  $C(z^q p)$ :

$$C(z^q p) = \frac{1}{2} G_{1,3}^{1,1} \left( \frac{1}{4}, 0, \frac{3}{4} \middle| \frac{\pi^2 p^4}{16} z^{4q} \right)$$

Functions involving  $S(z^q p)$ :

$$S(z^q p) = \frac{1}{2} G_{1,3}^{1,1} \left( \frac{1}{4}, 0, \frac{1}{4} \middle| \frac{\pi^2 p^4}{16} z^{4q} \right)$$

Functions involving  $\log(z^q p)$ :

$$\log^n(z^q p) = \text{generated}$$

$$\log(z^q p + a) = G_{1,1}^{1,0} \left( 0, 1 \middle| z \right) \log(a) + G_{1,1}^{0,1} \left( 1, 0 \middle| z \right) \log(a) + G_{2,2}^{1,2} \left( 1, 1, 0 \middle| \frac{z^q p}{a} \right)$$

$$\log(|z^q p - a|) = G_{1,1}^{1,0} \left( 0, 1 \middle| z \right) \log(|a|) + G_{1,1}^{0,1} \left( 1, 0 \middle| z \right) \log(|a|) + \pi G_{3,3}^{1,2} \left( 1, 1, 0, \frac{1}{2} \middle| \frac{z^q p}{a} \right)$$

Functions involving  $\sin(z^q p)$ :

$$\sin(z^q p) = \sqrt{\pi} G_{0,2}^{1,0} \left( \frac{1}{2}, 0 \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving  $\sinh(z^q p)$ :

$$\sinh(z^q p) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left( \frac{1}{2}, 1, 0 \middle| \frac{p^2}{4} z^{2q} \right)$$

### 3.10.4 API reference

`sympy.integrals.integrals.integrate(f, var, ...)`

Compute definite or indefinite integral of one or more variables using Risch-Norman algorithm and table lookup. This procedure is able to handle elementary algebraic and transcendental functions and also a huge class of special functions, including Airy, Bessel, Whittaker and Lambert.

var can be:

- a symbol – indefinite integration
- a tuple (symbol, a) – **indefinite integration with result** given with a replacing symbol
- a tuple (symbol, a, b) – definite integration

Several variables can be specified, in which case the result is multiple integration. (If var is omitted and the integrand is univariate, the indefinite integral in that variable will be performed.)

Indefinite integrals are returned without terms that are independent of the integration variables. (see examples)

Definite improper integrals often entail delicate convergence conditions. Pass `conds='piecewise'`, 'separate' or 'none' to have these returned, respectively, as a Piecewise function, as a separate result (i.e. result will be a tuple), or not at all (default is 'piecewise').

### Strategy

Sympy uses various approaches to definite integration. One method is to find an antiderivative for the integrand, and then use the fundamental theorem of calculus. Various functions are implemented to integrate polynomial, rational and trigonometric functions, and integrands containing DiracDelta terms.

Sympy also implements the part of the Risch algorithm, which is a decision procedure for integrating elementary functions, i.e., the algorithm can either find an elementary antiderivative, or prove that one does not exist. There is also a (very successful, albeit somewhat slow) general implementation of the heuristic Risch algorithm. This algorithm will eventually be phased out as more of the full Risch algorithm is implemented. See the docstring of `Integral._eval_integral()` for more details on computing the antiderivative using algebraic methods.

The option `risch=True` can be used to use only the (full) Risch algorithm. This is useful if you want to know if an elementary function has an elementary antiderivative. If the indefinite Integral returned by this function is an instance of `NonElementaryIntegral`, that means that the Risch algorithm has proven that integral to be non-elementary. Note that by default, additional methods (such as the Meijer G method outlined below) are tried on these integrals, as they may be expressible in terms of special functions, so if you only care about elementary answers, use `risch=True`. Also note that an unevaluated Integral returned by this function is not necessarily a `NonElementaryIntegral`, even with `risch=True`, as it may just be an indication that the particular part of the Risch algorithm needed to integrate that function is not yet implemented.

Another family of strategies comes from re-writing the integrand in terms of so-called Meijer G-functions. Indefinite integrals of a single G-function can always be computed, and the definite integral of a product of two G-functions can be computed from zero to infinity. Various strategies are implemented to rewrite integrands as G-functions, and use this information to compute integrals (see the `meijerint` module).

The option `manual=True` can be used to use only an algorithm that tries to mimic integration by hand. This algorithm does not handle as many integrands as the other algorithms implemented but may return results in a more familiar form. The `manualintegrate` module has functions that return the steps used (see the module docstring for more information).

In general, the algebraic methods work best for computing antiderivatives of (possibly complicated) combinations of elementary functions. The G-function methods work best for computing definite integrals from zero to infinity of moderately complicated combinations of special functions, or indefinite integrals of very simple combinations of special functions.

The strategy employed by the integration code is as follows:

- If computing a definite integral, and both limits are real, and at least one limit is  $+\infty$ , try the G-function method of definite integration first.
- Try to find an antiderivative, using all available methods, ordered by performance (that is try fastest method first, slowest last; in particular polynomial integration is tried first, Meijer G-functions second to last, and heuristic Risch last).
- If still not successful, try G-functions irrespective of the limits.

The option `meijerg=True`, `False`, `None` can be used to, respectively: always use G-function methods and no others, never use G-function methods, or use all available methods (in order as described above). It defaults to `None`.

See also:

`sympy.integrals.integrals.Integral` (page 551), `sympy.integrals.integrals.Integral.doit` (page 552)

### Examples

```
>>> from sympy import integrate, log, exp, oo
>>> from sympy.abc import a, x, y

>>> integrate(x*y, x)
x**2*y/2

>>> integrate(log(x), x)
x*log(x) - x

>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1

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

Terms that are independent of `x` are dropped by indefinite integration:

```
>>> from sympy import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3

>>> integrate(x*y)
Traceback (most recent call last):
...
ValueError: specify integration variables to integrate x*y
```

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

```
>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), -re(a) < 1),
          (Integral(x**a*exp(-x), (x, 0, oo)), True))

>>> integrate(x**a*exp(-x), (x, 0, oo), conds='none')
gamma(a + 1)

>>> integrate(x**a*exp(-x), (x, 0, oo), conds='separate')
(gamma(a + 1), -re(a) < 1)

sympy.integrals.integrals.line_integrate(field, Curve, variables)
Compute the line integral.
```

See also:

`sympy.integrals.integrals.integrate` (page 543), `sympy.integrals.integrals.Integral` (page 551)

### Examples

```
>>> from sympy import Curve, line_integrate, E, ln
>>> from sympy.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)
```

`sympy.integrals.deltafunctions.deltaintegrate(f, x)`

The idea for integration is the following:

- If we are dealing with a DiracDelta expression, i.e. `DiracDelta(g(x))`, we try to simplify it.

If we could simplify it, then we integrate the resulting expression. We already know we can integrate a simplified expression, because only simple DiracDelta expressions are involved.

If we couldn't simplify it, there are two cases:

1. The expression is a simple expression: we return the integral, taking care if we are dealing with a Derivative or with a proper DiracDelta.
2. The expression is not simple (i.e. `DiracDelta(cos(x))`): we can do nothing at all.

- If the node is a multiplication node having a DiracDelta term:

First we expand it.

If the expansion did work, then we try to integrate the expansion.

If not, we try to extract a simple DiracDelta term, then we have two cases:

1. We have a simple DiracDelta term, so we return the integral.
2. We didn't have a simple term, but we do have an expression with simplified DiracDelta terms, so we integrate this expression.

### See also:

`sympy.functions.special.delta_functions.DiracDelta` (page 358),  
`sympy.integrals.integrals.Integral` (page 551)

### Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.integrals.deltafunctions import deltaintegrate
>>> from sympy import sin, cos, DiracDelta, Heaviside
>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)
>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

`sympy.integrals.rationaltools.ratint(f, x, **flags)`

Performs indefinite integration of rational functions.

Given a field  $K$  and a rational function  $f = p/q$ , where  $p$  and  $q$  are polynomials in  $K[x]$ , returns a function  $g$  such that  $f = g'$ .

---

```
>>> from sympy.integrals.rationaltools import ratint
>>> from sympy.abc import x

>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See also:

[sympy.integrals.integrals.Integral.doit](#) (page 552), [sympy.integrals.rationaltools.ratint\\_logpart](#) (page 547), [sympy.integrals.rationaltools.ratint\\_ratpart](#) (page 547)

## References

[Bro05] (page 1244)

`sympy.integrals.rationaltools.ratint_logpart(f, g, x, t=None)`  
Lazard-Rioboo-Trager algorithm.

Given a field K and polynomials f and g in K[x], such that f and g are coprime,  $\deg(f) < \deg(g)$  and g is square-free, returns a list of tuples (s\_i, q\_i) of polynomials, for i = 1..n, such that s\_i in K[t, x] and q\_i in K[t], and:

$$\frac{d}{dx} \frac{f}{g} = \frac{d}{dx} \left( \sum_{i=1}^n \frac{s_i(a)}{q_i(a)} \right) = \sum_{i=1}^n \frac{q_i'(a) \log(s_i(a), x)}{q_i(a)^2}$$

See also:

[sympy.integrals.rationaltools.ratint](#) (page 546), [sympy.integrals.rationaltools.ratint\\_ratpart](#) (page 547)

## Examples

```
>>> from sympy.integrals.rationaltools import ratint_logpart
>>> from sympy.abc import x
>>> from sympy import Poly
>>> ratint_logpart(Poly(1, x, domain='ZZ'), x)
[(Poly(x + 3*_t/2 + 1/2, x, domain='QQ[_t]'), Poly(3*_t**2 + 1, _t, domain='ZZ'))]
>>> ratint_logpart(Poly(12, x, domain='ZZ'), x)
[(Poly(x - 3*_t/8 - 1/2, x, domain='QQ[_t]'), Poly(-_t**2 + 16, _t, domain='ZZ'))]
```

`sympy.integrals.rationaltools.ratint_ratpart(f, g, x)`  
Horowitz-Ostrogradsky algorithm.

Given a field K and polynomials f and g in K[x], such that f and g are coprime and  $\deg(f) < \deg(g)$ , returns fractions A and B in K(x), such that  $f/g = A' + B$  and B has square-free denominator.

See also:

[sympy.integrals.rationaltools.ratint](#) (page 546), [sympy.integrals.rationaltools.ratint\\_logpart](#) (page 547)

## Examples

```
>>> from sympy.integrals.rationaltools import ratint_ratpart
>>> from sympy.abc import x, y
>>> from sympy import Poly
>>> ratint_ratpart(Poly(1, x, domain='ZZ'),
... Poly(x + 1, x, domain='ZZ'), x)
(0, 1/(x + 1))
>>> ratint_ratpart(Poly(1, x, domain='EX'),
... Poly(x**2 + y**2, x, domain='EX'), x)
(0, 1/(x**2 + y**2))
>>> ratint_ratpart(Poly(36, x, domain='ZZ'),
... Poly(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x, domain='ZZ'), x)
((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
```

`sympy.integrals.heurisch.components(f, x)`

Returns a set of all functional components of the given expression which includes symbols, function applications and compositions and non-integer powers. Fractional powers are collected with minimal, positive exponents.

```
>>> from sympy import cos, sin
>>> from sympy.abc import x, y
>>> from sympy.integrals.heurisch import components

>>> components(sin(x)*cos(x)**2, x)
set([x, sin(x), cos(x)])
```

### See also:

[sympy.integrals.heurisch.heurisch](#) (page 548)

`sympy.integrals.heurisch.heurisch(f, x, rewrite=False, hints=None, mappings=None, retries=3, degree_offset=0, unnecessary_permutations=None)`

Compute indefinite integral using heuristic Risch algorithm.

This is a heuristic approach to indefinite integration in finite terms using the extended heuristic (parallel) Risch algorithm, based on Manuel Bronstein’s “Poor Man’s Integrator”.

The algorithm supports various classes of functions including transcendental elementary or special functions like Airy, Bessel, Whittaker and Lambert.

Note that this algorithm is not a decision procedure. If it isn’t able to compute the antiderivative for a given function, then this is not a proof that such a function does not exist. One should use recursive Risch algorithm in such case. It’s an open question if this algorithm can be made a full decision procedure.

This is an internal integrator procedure. You should use toplevel ‘integrate’ function in most cases, as this procedure needs some preprocessing steps and otherwise may fail.

### See also:

[sympy.integrals.integrals.Integral.doit](#) (page 552), [sympy.integrals.integrals.Integral](#) (page 551), [sympy.integrals.heurisch.components](#) (page 548)

## Examples

```
>>> from sympy import tan
>>> from sympy.integrals.heurisch import heurisch
>>> from sympy.abc import x, y
```

```
>>> heurisch(y*tan(x), x)
y*log(tan(x)**2 + 1)/2
```

See Manuel Bronstein’s “Poor Man’s Integrator”:

[1] <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint/index.html>

For more information on the implemented algorithm refer to:

- [2] **K. Geddes, L. Stefanus, On the Risch-Norman Integration** Method and its Implementation in Maple, Proceedings of ISSAC‘89, ACM Press, 212-217.
- [3] **J. H. Davenport, On the Parallel Risch Algorithm (I)**, Proceedings of EUROCAM‘82, LNCS 144, Springer, 144-157.
- [4] **J. H. Davenport, On the Parallel Risch Algorithm (III): Use of Tangents**, SIGSAM Bulletin 16 (1982), 3-6.
- [5] **J. H. Davenport, B. M. Trager, On the Parallel Risch Algorithm (II)**, ACM Transactions on Mathematical Software 11 (1985), 356-362.

```
sympy.integrals.heurisch.heurisch_wrapper(f, x, rewrite=False, hints=None, map-
                                             pings=None, retries=3, degree_offset=0, unnec-
                                             essary_permutations=None)
```

A wrapper around the heurisch integration algorithm.

This method takes the result from heurisch and checks for poles in the denominator. For each of these poles, the integral is reevaluated, and the final integration result is given in terms of a Piecewise.

See also:

[sympy.integrals.heurisch.heurisch](#) (page 548)

## Examples

```
>>> from sympy.core import symbols
>>> from sympy.functions import cos
>>> from sympy.integrals.heurisch import heurisch, heurisch_wrapper
>>> n, x = symbols('n x')
>>> heurisch(cos(n*x), x)
sin(n*x)/n
>>> heurisch_wrapper(cos(n*x), x)
Piecewise((x, Eq(n, 0)), (sin(n*x)/n, True))
```

```
sympy.integrals.trigonometry.trigintegrate(f, x,conds='piecewise')
Integrate f = Mul(trig) over x
```

```
>>> from sympy import Symbol, sin, cos, tan, sec, csc, cot
>>> from sympy.integrals.trigonometry import trigintegrate
>>> from sympy.abc import x

>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2

>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)*cos(x)/2

>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)
```

```
>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)
```

[http://en.wikibooks.org/wiki/Calculus/Integration\\_techniques](http://en.wikibooks.org/wiki/Calculus/Integration_techniques)

See also:

`sympy.integrals.integrals.Integral.doit` (page 552), `sympy.integrals.integrals.Integral` (page 551)

`sympy.integrals.manualintegrate.manualintegrate(f, var)`

Compute indefinite integral of a single variable using an algorithm that resembles what a student would do by hand.

Unlike `integrate`, `var` can only be a single symbol.

See also:

`sympy.integrals.integrals.integrate` (page 543), `sympy.integrals.integrals.Integral.doit` (page 552), `sympy.integrals.integrals.Integral` (page 551)

## Examples

```
>>> from sympy import sin, cos, tan, exp, log, integrate
>>> from sympy.integrals.manualintegrate import manualintegrate
>>> from sympy.abc import x
>>> manualintegrate(1 / x, x)
log(x)
>>> integrate(1/x)
log(x)
>>> manualintegrate(log(x), x)
x*log(x) - x
>>> integrate(log(x))
x*log(x) - x
>>> manualintegrate(exp(x) / (1 + exp(2 * x)), x)
atan(exp(x))
>>> integrate(exp(x) / (1 + exp(2 * x)))
RootSum(4*_z**2 + 1, Lambda(_i, _i*log(2*_i + exp(x))))
>>> manualintegrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x), x)
-cos(x)**5/5
>>> manualintegrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> integrate(cos(x)**4 * sin(x)**3, x)
cos(x)**7/7 - cos(x)**5/5
>>> manualintegrate(tan(x), x)
-log(cos(x))
>>> integrate(tan(x), x)
-log(sin(x)**2 - 1)/2
```

`sympy.integrals.manualintegrate.integral_steps(integrand, symbol, **options)`

Returns the steps needed to compute an integral.

This function attempts to mirror what a student would do by hand as closely as possible.

SymPy Gamma uses this to provide a step-by-step explanation of an integral. The code it uses to format the results of this function can be found at [https://github.com/sympy/sympy\\_gamma/blob/master/app/logic/intsteps.py](https://github.com/sympy/sympy_gamma/blob/master/app/logic/intsteps.py).

**Returns rule** : namedtuple

The first step; most rules have substeps that must also be considered. These substeps can be evaluated using `manualintegrate` to obtain a result.

### Examples

```
>>> from sympy import exp, sin, cos
>>> from sympy.integrals.manualintegrate import integral_steps
>>> from sympy.abc import x
>>> print(repr(integral_steps(exp(x) / (1 + exp(2 * x)), x)))
URule(u_var=_u, u_func=exp(x), constant=1,
      substep=ArctanRule(context=1/(_u**2 + 1), symbol=_u),
      context=exp(x)/(exp(2*x) + 1), symbol=x)
>>> print(repr(integral_steps(sin(x), x)))
TrigRule(func='sin', arg=x, context=sin(x), symbol=x)
>>> print(repr(integral_steps((x**2 + 3)**2 , x)))
RewriteRule(rewritten=x**4 + 6*x**2 + 9,
            substep=AddRule(substeps=[PowerRule(base=x, exp=4, context=x**4, symbol=x),
                                       ConstantTimesRule(constant=6, other=x**2,
                                             substep=PowerRule(base=x, exp=2, context=x**2, symbol=x),
                                             context=6*x**2, symbol=x),
                                       ConstantRule(constant=9, context=9, symbol=x)],
                           context=x**4 + 6*x**2 + 9, symbol=x), context=(x**2 + 3)**2, symbol=x)
```

The class `Integral` represents an unevaluated integral and has some methods that help in the integration of an expression.

`class sympy.integrals.integrals.Integral`

Represents unevaluated integral.

`is_commutative`

Returns whether all the free symbols in the integral are commutative.

`as_sum(n, method='midpoint')`

Approximates the definite integral by a sum.

method ... one of: left, right, midpoint, trapezoid

These are all basically the rectangle method [1], the only difference is where the function value is taken in each interval to define the rectangle.

[1] [http://en.wikipedia.org/wiki/Rectangle\\_method](http://en.wikipedia.org/wiki/Rectangle_method)

**See also:**

`sympy.integrals.integrals.Integral.doit` ([page 552](#)) Perform the integration using any hints

### Examples

```
>>> from sympy import sin, sqrt
>>> from sympy.abc import x
>>> from sympy.integrals import Integral
>>> e = Integral(sin(x), (x, 3, 7))
>>> e
Integral(sin(x), (x, 3, 7))
```

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].

The left-hand rule uses function evaluations at the left of each interval:

```
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```

The midpoint rule uses evaluations at the center of each interval:

```
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```

The right-hand rule uses function evaluations at the right of each interval:

```
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```

The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```

All but the trapexoid method may be used when dealing with a function with a discontinuity. Here, the discontinuity at  $x = 0$  can be avoided by using the midpoint or right-hand method:

```
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4)  # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return oo:

```
>>> e.as_sum(5, 'left')
oo
>>> e.as_sum(5, 'trapezoid')
oo

doit(**hints)
```

Perform the integration using any hints given.

**See also:**

[sympy.integrals.trigonometry.trigintegrate](#) (page 549), [sympy.integrals.heurisch.heurisch](#) (page 548), [sympy.integrals.rationaltools.ratint](#) (page 546)

[sympy.integrals.integrals.Integral.as\\_sum](#) (page 551) Approximate the integral using a sum

## Examples

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

---

```
>>> Integral(x**i, (i, 1, 3)).doit()
Piecewise((2, Eq(log(x), 0)), (x**3/log(x) - x/log(x), True))
```

**free\_symbols**

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

**See also:**

<a href="#">sympy.concrete.expr_with_limits.ExprWithLimits.function</a> <a href="#">sympy.concrete.expr_with_limits.ExprWithLimits.limits</a> <a href="#">sympy.concrete.expr_with_limits.ExprWithLimits.variables</a>	<span style="font-size: small;">(page 296),</span> <span style="font-size: small;">(page 297),</span> <span style="font-size: small;">(page 297)</span>
--	---

**Examples**

```
>>> from sympy import Integral
>>> from sympy.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols
set([y])
```

**transform(*x, u*)**

Performs a change of variables from *x* to *u* using the relationship given by *x* and *u* which will define the transformations *f* and *F* (which are inverses of each other) as follows:

1. If *x* is a Symbol (which is a variable of integration) then *u* will be interpreted as some function, *f(u)*, with inverse *F(u)*. This, in effect, just makes the substitution of *x* with *f(x)*.
2. If *u* is a Symbol then *x* will be interpreted as some function, *F(x)*, with inverse *f(u)*. This is commonly referred to as u-substitution.

Once *f* and *F* have been identified, the transformation is made as follows:

$$\int_a^b x dx \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{dx}{du}$$

where *F(x)* is the inverse of *f(x)* and the limits and integrand have been corrected so as to retain the same value after integration.

**See also:**

[sympy.concrete.expr\\_with\\_limits.ExprWithLimits.variables](#) (page 297) Lists the integration variables

[sympy.concrete.expr\\_with\\_limits.ExprWithLimits.as\\_dummy](#) (page 296) Replace integration variables with dummy ones

**Notes**

The mappings, *F(x)* or *f(u)*, must lead to a unique integral. Linear or rational linear expression,  $2 * x$ ,  $1/x$  and *sqrt(x)*, will always work; quadratic expressions like  $x * * 2 - 1$  are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if *x* is not a variable of integration.

*x* must be (or contain) only one of the integration variables. If *u* has more than one free symbol then it should be sent as a tuple (*u, uvar*) where *uvar* identifies which variable is replacing the integration variable. XXX can it contain another integration variable?

## Examples

```
>>> from sympy.abc import a, b, c, d, x, u, y
>>> from sympy import Integral, S, cos, sqrt
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u-substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because  $x = \pm\sqrt{u+1}$  and the sign does not cancel out of the integrand:

```
>>> Integral(cos(x**2 - 1), (x, 0, 1)).transform(x**2 - 1, u)
Traceback (most recent call last):
...
ValueError:
The mapping between F(x) and f(u) did not give a unique integrand.
```

transform can do a substitution. Here, the previous result is transformed back into the original expression using “u-substitution”:

```
>>> ui = _
>>> _.transform(sqrt(u + 1), x) == i
True
```

We can accomplish the same with a regular substitution:

```
>>> ui.transform(u, x**2 - 1) == i
True
```

If the  $x$  does not contain a symbol of integration then the integral will be returned unchanged. Integral  $i$  does not have an integration variable  $a$  so no change is made:

```
>>> i.transform(a, x) == i
True
```

When  $u$  has more than one free symbol the symbol that is replacing  $x$  must be identified by passing  $u$  as a tuple:

```
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, u))
Integral(a + u, (u, -a, -a + 1))
>>> Integral(x, (x, 0, 1)).transform(x, (u + a, a))
Integral(a + u, (a, -u, -u + 1))
```

```
class sympy.integrals.transforms.IntegralTransform
Base class for integral transforms.
```

This class represents unevaluated transforms.

To implement a concrete transform, derive from this class and implement the `_compute_transform(f, x, s, **hints)` and `_as_integral(f, x, s)` functions. If the transform cannot be computed, raise `IntegralTransformError`.

Also set `cls._name`.

Implement self.\_collapse\_extra if your function returns more than just a number and possibly a convergence condition.

**doit(\*\*hints)**

Try to evaluate the transform in closed form.

This general function handles linearity, but apart from that leaves pretty much everything to \_compute\_transform.

Standard hints are the following:

- simplify**: whether or not to simplify the result
- noconds**: if True, don't return convergence conditions
- needeval**: if True, raise **IntegralTransformError** instead of returning IntegralTransform objects

The default values of these hints depend on the concrete transform, usually the default is (**simplify**, **noconds**, **needeval**) = (True, False, False).

**free\_symbols**

This method returns the symbols that will exist when the transform is evaluated.

**function**

The function to be transformed.

**function\_variable**

The dependent variable of the function to be transformed.

**transform\_variable**

The independent transform variable.

**class sympy.integrals.transforms.MellinTransform**

Class representing unevaluated Mellin transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute Mellin transforms, see the [mellin\\_transform\(\)](#) (page 522) docstring.

**class sympy.integrals.transforms.InverseMellinTransform**

Class representing unevaluated inverse Mellin transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse Mellin transforms, see the [inverse\\_mellin\\_transform\(\)](#) (page 523) docstring.

**class sympy.integrals.transforms.LaplaceTransform**

Class representing unevaluated Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute Laplace transforms, see the [laplace\\_transform\(\)](#) (page 524) docstring.

**class sympy.integrals.transforms.InverseLaplaceTransform**

Class representing unevaluated inverse Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse Laplace transforms, see the [inverse\\_laplace\\_transform\(\)](#) (page 524) docstring.

**class sympy.integrals.transforms.FourierTransform**

Class representing unevaluated Fourier transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute Fourier transforms, see the [fourier\\_transform\(\)](#) (page 525) docstring.

**class sympy.integrals.transforms.InverseFourierTransform**

Class representing unevaluated inverse Fourier transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse Fourier transforms, see the [inverse\\_fourier\\_transform\(\)](#) (page 525) docstring.

**class sympy.integrals.transforms.SineTransform**

Class representing unevaluated sine transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute sine transforms, see the [sine\\_transform\(\)](#) (page 526) docstring.

**class sympy.integrals.transforms.InverseSineTransform**

Class representing unevaluated inverse sine transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse sine transforms, see the [inverse\\_sine\\_transform\(\)](#) (page 526) docstring.

**class sympy.integrals.transforms.CosineTransform**

Class representing unevaluated cosine transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute cosine transforms, see the [cosine\\_transform\(\)](#) (page 526) docstring.

**class sympy.integrals.transforms.InverseCosineTransform**

Class representing unevaluated inverse cosine transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse cosine transforms, see the [inverse\\_cosine\\_transform\(\)](#) (page 527) docstring.

**class sympy.integrals.transforms.HankelTransform**

Class representing unevaluated Hankel transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute Hankel transforms, see the [hankel\\_transform\(\)](#) (page 527) docstring.

**class sympy.integrals.transforms.InverseHankelTransform**

Class representing unevaluated inverse Hankel transforms.

For usage of this class, see the [IntegralTransform](#) (page 554) docstring.

For how to compute inverse Hankel transforms, see the [inverse\\_hankel\\_transform\(\)](#) (page 528) docstring.

### 3.10.5 TODO and Bugs

There are still lots of functions that SymPy does not know how to integrate. For bugs related to this module, see <https://github.com/sympy/sympy/issues?labels=Integration>

## 3.11 Numeric Integrals

SymPy has functions to calculate points and weights for Gaussian quadrature of any order and any precision:

`sympy.integrals.quadrature.gauss_legendre(n, n_digits)`

Computes the Gauss-Legendre quadrature [R316] (page 1244) points and weights.

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $P_n$  and the weights  $w_i$  are given by:

$$w_i = \frac{2}{(1 - x_i^2) (P'_n(x_i))^2}$$

**Parameters** `n` : the order of quadrature

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

**See also:**

`sympy.integrals.quadrature.gauss_laguerre` (page 557), `sympy.integrals.quadrature.gauss_gen_laguerre` (page 559), `sympy.integrals.quadrature.gauss_hermite` (page 558), `sympy.integrals.quadrature.gauss_chebyt` (page 560), `sympy.integrals.quadrature.gauss_chebyu` (page 561), `sympy.integrals.quadrature.gauss_jacobi` (page 562)

### References

[R316] (page 1244), [R317] (page 1244)

### Examples

```
>>> from sympy.integrals.quadrature import gauss_legendre
>>> x, w = gauss_legendre(3, 5)
>>> x
[-0.7746, 0, 0.7746]
>>> w
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> x
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> w
[0.34786, 0.65215, 0.65215, 0.34786]
```

`sympy.integrals.quadrature.gauss_laguerre(n, n_digits)`

Computes the Gauss-Laguerre quadrature [R318] (page 1244) points and weights.

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^\infty e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $L_n$  and the weights  $w_i$  are given by:

$$w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}$$

**Parameters** `n` : the order of quadrature

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

**See also:**

`sympy.integrals.quadrature.gauss_legendre` (page 557), `sympy.integrals.quadrature.gauss_gen_laguerre` (page 559), `sympy.integrals.quadrature.gauss_hermite` (page 558),  
`sympy.integrals.quadrature.gauss_chebyshev_t` (page 560), `sympy.integrals.quadrature.gauss_chebyshev_u` (page 561), `sympy.integrals.quadrature.gauss_jacobi` (page 562)

## References

[R318] (page 1244), [R319] (page 1244)

## Examples

```
>>> from sympy.integrals.quadrature import gauss_laguerre
>>> x, w = gauss_laguerre(3, 5)
>>> x
[0.41577, 2.2943, 6.2899]
>>> w
[0.71109, 0.27852, 0.010389]
>>> x, w = gauss_laguerre(6, 5)
>>> x
[0.22285, 1.1889, 2.9927, 5.7751, 9.8375, 15.983]
>>> w
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

`sympy.integrals.quadrature.gauss_hermite(n, n_digits)`

Computes the Gauss-Hermite quadrature [R320] (page 1244) points and weights.

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $H_n$  and the weights  $w_i$  are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}$$

**Parameters** `n` : the order of quadrature

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

See also:

[sympy.integrals.quadrature.gauss\\_legendre](#) (page 557), [sympy.integrals.quadrature.gauss\\_laguerre](#) (page 557), [sympy.integrals.quadrature.gauss\\_gen\\_laguerre](#) (page 559), [sympy.integrals.quadrature.gauss\\_chebyshev\\_t](#) (page 560), [sympy.integrals.quadrature.gauss\\_chebyshev\\_u](#) (page 561), [sympy.integrals.quadrature.gauss\\_jacobi](#) (page 562)

## References

[R320] (page 1244), [R321] (page 1244), [R322] (page 1244)

## Examples

```
>>> from sympy.integrals.quadrature import gauss_hermite
>>> x, w = gauss_hermite(3, 5)
>>> x
[-1.2247, 0, 1.2247]
>>> w
[0.29541, 1.1816, 0.29541]

>>> x, w = gauss_hermite(6, 5)
>>> x
[-2.3506, -1.3358, -0.43608, 0.43608, 1.3358, 2.3506]
>>> w
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

`sympy.integrals.quadrature.gauss_gen_laguerre(n, alpha, n_digits)`

Computes the generalized Gauss-Laguerre quadrature [R323] (page 1244) points and weights.

The generalized Gauss-Laguerre quadrature approximates the integral:

$$\int_0^\infty x^\alpha e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $L_n^\alpha$  and the weights  $w_i$  are given by:

$$w_i = \frac{\Gamma(\alpha + n)}{n\Gamma(n)L_{n-1}^\alpha(x_i)L_{n-1}^{\alpha+1}(x_i)}$$

**Parameters** `n` : the order of quadrature

`alpha` : the exponent of the singularity,  $\alpha > -1$

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

See also:

[sympy.integrals.quadrature.gauss\\_legendre](#) (page 557), [sympy.integrals.quadrature.gauss\\_laguerre](#) (page 557), [sympy.integrals.quadrature.gauss\\_hermite](#) (page 558), [sympy.integrals.quadrature.gauss\\_chebyshev\\_t](#) (page 560), [sympy.integrals.quadrature.gauss\\_chebyshev\\_u](#) (page 561), [sympy.integrals.quadrature.gauss\\_jacobi](#) (page 562)

## References

[R323] (page 1244), [R324] (page 1244)

## Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, -S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> w
[1.4493, 0.31413, 0.00906]

>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

`sympy.integrals.quadrature.gauss_chebyshev_t(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R325] (page 1244) points and weights of the first kind.

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $T_n$  and the weights  $w_i$  are given by:

$$w_i = \frac{\pi}{n}$$

**Parameters** `n` : the order of quadrature

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

**See also:**

`sympy.integrals.quadrature.gauss_legendre` (page 557), `sympy.integrals.quadrature.gauss_laguerre` (page 557), `sympy.integrals.quadrature.gauss_hermite` (page 558), `sympy.integrals.quadrature.gauss_gen_laguerre` (page 559), `sympy.integrals.quadrature.gauss_chebyshev_u` (page 561), `sympy.integrals.quadrature.gauss_jacobi` (page 562)

## References

[R325] (page 1244), [R326] (page 1244)

## Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]

>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> w
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

`sympy.integrals.quadrature.gauss_chebyshev_u(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R327] (page 1244) points and weights of the second kind.

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

$$\int_{-1}^1 \sqrt{1-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $U_n$  and the weights  $w_i$  are given by:

$$w_i = \frac{\pi}{n+1} \sin^2\left(\frac{i}{n+1}\pi\right)$$

**Parameters** `n` : the order of quadrature

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

**See also:**

`sympy.integrals.quadrature.gauss_legendre` (page 557), `sympy.integrals.quadrature.gauss_laguerre` (page 557), `sympy.integrals.quadrature.gauss_hermite` (page 558), `sympy.integrals.quadrature.gauss_gen_laguerre` (page 559), `sympy.integrals.quadrature.gauss_chebyshev_t` (page 560), `sympy.integrals.quadrature.gauss_jacobi` (page 562)

## References

[R327] (page 1244), [R328] (page 1244)

## Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> w
[0.3927, 0.7854, 0.3927]
```

```
>>> x, w = gauss_chebyshev_u(6, 5)
>>> x
[0.90097, 0.62349, 0.22252, -0.22252, -0.62349, -0.90097]
>>> w
[0.084489, 0.27433, 0.42658, 0.42658, 0.27433, 0.084489]
```

`sympy.integrals.quadrature.gauss_jacobi(n, alpha, beta, n_digits)`

Computes the Gauss-Jacobi quadrature [R329] (page 1244) points and weights.

The Gauss-Jacobi quadrature of the first kind approximates the integral:

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes  $x_i$  of an order  $n$  quadrature rule are the roots of  $P_n^{(\alpha, \beta)}$  and the weights  $w_i$  are given by:

$$w_i = -\frac{2n + \alpha + \beta + 2}{n + \alpha + \beta + 1} \frac{\Gamma(n + \alpha + 1)\Gamma(n + \beta + 1)}{\Gamma(n + \alpha + \beta + 1)(n + 1)!} \frac{2^{\alpha+\beta}}{P'_n(x_i)P_{n+1}^{(\alpha, \beta)}(x_i)}$$

**Parameters** `n` : the order of quadrature

`alpha` : the first parameter of the Jacobi Polynomial,  $\alpha > -1$

`beta` : the second parameter of the Jacobi Polynomial,  $\beta > -1$

`n_digits` : number of significant digits of the points and weights to return

**Returns** (`x`, `w`) : the `x` and `w` are lists of points and weights as Floats.

The points  $x_i$  and weights  $w_i$  are returned as (`x`, `w`) tuple of lists.

**See also:**

`sympy.integrals.quadrature.gauss_legendre` (page 557), `sympy.integrals.quadrature.gauss_laguerre` (page 557), `sympy.integrals.quadrature.gauss_hermite` (page 558), `sympy.integrals.quadrature.gauss_gen_laguerre` (page 559), `sympy.integrals.quadrature.gauss_chebyshev_t` (page 560), `sympy.integrals.quadrature.gauss_chebyshev_u` (page 561)

## References

[R329] (page 1244), [R330] (page 1244), [R331] (page 1244)

## Examples

```
>>> from sympy import S
>>> from sympy.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]

>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> x
[-0.87174, -0.5917, -0.2093, 0.2093, 0.5917, 0.87174]
>>> w
[0.050584, 0.22169, 0.39439, 0.39439, 0.22169, 0.050584]
```

## 3.12 Logic Module

### 3.12.1 Introduction

The logic module for SymPy allows to form and manipulate logic expressions using symbolic and Boolean values.

### 3.12.2 Forming logical expressions

You can build Boolean expressions with the standard python operators `&` ([And](#) (page 565)), `|` ([Or](#) (page 565)), `~` ([sympy.logic.boolalg.Not](#) (page 566)):

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
Or(And(x, y), y)
>>> x | y
Or(x, y)
>>> ~x
Not(x)
```

You can also form implications with `>>` and `<<`:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in SymPy, Boolean expressions inherit from [Basic](#) (page 62):

```
>>> (y & x).subs({x: True, y: True})
True
>>> (x | y).atoms()
set([x, y])
```

The logic module also includes the following functions to derive boolean expressions from their truth tables-

`sympy.logic.boolalg.SOPform(variables, minterms, dontcares=None)`

The SOPform function uses simplified\_pairs and a redundant group- eliminating algorithm to convert the list of all input combos that generate ‘1’ (the minterms) into the smallest Sum of Products form.

The variables must be given as the first argument.

Return a logical Or function (i.e., the “sum of products” or “SOP” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

### References

[R332] (page 1244)

## Examples

```
>>> from sympy.logic import SOPform
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
...              [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform(['w','x','y','z'], minterms, dontcares)
Or(And(Not(w), z), And(y, z))
```

`sympy.logic.boolalg.POSform(variables, minterms, dontcares=None)`

The POSform function uses simplified\_pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate ‘1’ (the minterms) into the smallest Product of Sums form.

The variables must be given as the first argument.

Return a logical And function (i.e., the “product of sums” or “POS” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

## References

[R333] (page 1244)

## Examples

```
>>> from sympy.logic import POSform
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1],
...              [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform(['w','x','y','z'], minterms, dontcares)
And(Or(Not(w), y), z)
```

### 3.12.3 Boolean functions

`class sympy.logic.boolalg.BooleanTrue`

SymPy version of True, a singleton that can be accessed via `S.true`.

This is the SymPy version of True, for use in the logic module. The primary advantage of using `true` instead of `True` is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with `True` they act bitwise on 1. Functions in the logic module will return this class when they evaluate to true.

See also:

`sympy.logic.boolalg.BooleanFalse` (page 565)

## Examples

```
>>> from sympy import sympify, true, Or
>>> sympify(True)
True
```

```
>>> ~true
False
>>> ~True
-2
>>> Or(True, False)
True
```

```
class sympy.logic.boolalg.BooleanFalse
```

SymPy version of False, a singleton that can be accessed via S.false.

This is the SymPy version of False, for use in the logic module. The primary advantage of using false instead of False is that shorthand boolean operations like `~` and `>>` will work as expected on this class, whereas with False they act bitwise on 0. Functions in the logic module will return this class when they evaluate to false.

See also:

[sympy.logic.boolalg.BooleanTrue](#) (page 564)

### Examples

```
>>> from sympy import sympify, false, Or, true
>>> sympify(False)
False
>>> false >> false
True
>>> False >> False
0
>>> Or(True, False)
True
```

```
class sympy.logic.boolalg.And
```

Logical AND function.

It evaluates its arguments in order, giving False immediately if any of them are False, and True if they are all True.

### Notes

The `&` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, `And(a, b)` and `a & b` will return different things if `a` and `b` are integers.

```
>>> And(x, y).subs(x, 1)
y
```

### Examples

```
>>> from sympy.core import symbols
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import And
>>> x & y
And(x, y)
```

```
class sympy.logic.boolalg.Or
Logical OR function
```

It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.

### Notes

The `|` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, `Or(a, b)` and `a | b` will return different things if `a` and `b` are integers.

```
>>> Or(x, y).subs(x, 0)
y
```

### Examples

```
>>> from sympy.core import symbols
>>> from sympy.abc import x, y
>>> from sympy.logic.boolalg import Or
>>> x | y
Or(x, y)
```

```
class sympy.logic.boolalg.Not
Logical Not function (negation)
```

Returns True if the statement is False Returns False if the statement is True

### Notes

- The `~` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise not. In particular, `~a` and `Not(a)` will be different if `a` is an integer. Furthermore, since bools in Python subclass from `int`, `~True` is the same as `~1` which is `-2`, which has a boolean value of True. To avoid this issue, use the SymPy boolean types `true` and `false`.

```
>>> from sympy import true
>>> ~True
-2
>>> ~true
False
```

### Examples

```
>>> from sympy.logic.boolalg import Not, And, Or
>>> from sympy.abc import x, A, B
>>> Not(True)
False
>>> Not(False)
True
>>> Not(And(True, False))
True
```

```
>>> Not(Or(True, False))
False
>>> Not(And(And(True, x), Or(x, False)))
Not(x)
>>> ~x
Not(x)
>>> Not(And(Or(A, B), Or(~A, ~B)))
Not(And(Or(A, B), Or(Not(A), Not(B))))
```

```
class sympy.logic.boolalg.Xor
    Logical XOR (exclusive OR) function.
```

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

### Notes

The `~` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, `a ^ b` and `Xor(a, b)` will be different if `a` and `b` are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

### Examples

```
>>> from sympy.logic.boolalg import Xor
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Xor(True, False)
True
>>> Xor(True, True)
False
>>> Xor(True, False, True, True, False)
True
>>> Xor(True, False, True, False)
False
>>> x ^ y
Xor(x, y)
```

```
class sympy.logic.boolalg.Nand
    Logical NAND function.
```

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

### Examples

```
>>> from sympy.logic.boolalg import Nand
>>> from sympy import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
```

```
True
>>> Nand(True, True)
False
>>> Nand(x, y)
Not(And(x, y))
```

```
class sympy.logic.boolalg.Nor
    Logical NOR function.
```

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

### Examples

```
>>> from sympy.logic.boolalg import Nor
>>> from sympy import symbols
>>> x, y = symbols('x y')

>>> Nor(True, False)
False
>>> Nor(True, True)
False
>>> Nor(False, True)
False
>>> Nor(False, False)
True
>>> Nor(x, y)
Not(Or(x, y))
```

```
class sympy.logic.boolalg.Implies
    Logical implication.
```

A implies B is equivalent to !A v B

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

### Notes

The `>>` and `<<` operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a, b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python considers `True` and `False` to be integers, `True >> True` will be the same as `1 >> 1`, i.e., 0, which has a truth value of False. To avoid this issue, use the SymPy objects `true` and `false`.

```
>>> from sympy import true, false
>>> True >> False
1
>>> true >> false
False
```

## Examples

```
>>> from sympy.logic.boolalg import Implies
>>> from sympy import symbols
>>> x, y = symbols('x y')

>>> Implies(True, False)
False
>>> Implies(False, False)
True
>>> Implies(True, True)
True
>>> Implies(False, True)
True
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

class sympy.logic.boolalg.Equivalent  
Equivalence relation.

Equivalent(A, B) is True iff A and B are both True or both False

Returns True if all of the arguments are logically equivalent. Returns False otherwise.

## Examples

```
>>> from sympy.logic.boolalg import Equivalent, And
>>> from sympy.abc import x, y
>>> Equivalent(False, False, False)
True
>>> Equivalent(True, False, False)
False
>>> Equivalent(x, And(x, True))
True
```

class sympy.logic.boolalg.ITE  
If then else clause.

ITE(A, B, C) evaluates and returns the result of B if A is true else it returns the result of C

## Examples

```
>>> from sympy.logic.boolalg import ITE, And, Xor, Or
>>> from sympy.abc import x, y, z
>>> ITE(True, False, True)
False
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
True
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
```

```
>>> ITE(x, y, y)
y
```

The following functions can be used to handle Conjunctive and Disjunctive Normal forms-

`sympy.logic.boolalg.to_cnf(expr, simplify=False)`

Convert a propositional logical sentence  $s$  to conjunctive normal form. That is, of the form  $((A \mid \neg B \mid \dots) \ \& \ (B \mid C \mid \dots) \ \& \ \dots)$  If `simplify` is True, the `expr` is evaluated to its simplest CNF form.

### Examples

```
>>> from sympy.logic.boolalg import to_cnf
>>> from sympy.abc import A, B, D
>>> to_cnf(~(A | B) | D)
And(Or(D, Not(A)), Or(D, Not(B)))
>>> to_cnf((A | B) & (A | ~A), True)
Or(A, B)
```

`sympy.logic.boolalg.to_dnf(expr, simplify=False)`

Convert a propositional logical sentence  $s$  to disjunctive normal form. That is, of the form  $((A \ \& \ \neg B \ \& \ \dots) \mid (B \ \& \ C \ \& \ \dots) \mid \dots)$  If `simplify` is True, the `expr` is evaluated to its simplest DNF form.

### Examples

```
>>> from sympy.logic.boolalg import to_dnf
>>> from sympy.abc import A, B, C
>>> to_dnf(B & (A | C))
Or(And(A, B), And(B, C))
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
Or(A, C)
```

`sympy.logic.boolalg.is_cnf(expr)`

Test whether or not an expression is in conjunctive normal form.

### Examples

```
>>> from sympy.logic.boolalg import is_cnf
>>> from sympy.abc import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

`sympy.logic.boolalg.is_dnf(expr)`

Test whether or not an expression is in disjunctive normal form.

### Examples

```
>>> from sympy.logic.boolalg import is_dnf
>>> from sympy.abc import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

### 3.12.4 Simplification and equivalence-testing

`sympy.logic.boolalg.simplify_logic(expr, form=None, deep=True)`

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an Or or And object in SymPy.

**Parameters** `expr` : string or boolean expression

`form` : string ('cnf' or 'dnf') or None (default).

If 'cnf' or 'dnf', the simplest expression in the corresponding normal form is returned; if None, the answer is returned according to the form with fewest args (in CNF by default).

`deep` : boolean (default True)

indicates whether to recursively simplify any non-boolean functions contained within the input.

#### Examples

```
>>> from sympy.logic import simplify_logic
>>> from sympy.abc import x, y, z
>>> from sympy import S
>>> b = '(~x & ~y & ~z) | (~x & ~y & z)'
>>> simplify_logic(b)
And(Not(x), Not(y))

>>> S(b)
Or(And(Not(x), Not(y), Not(z)), And(Not(x), Not(y), z))
>>> simplify_logic(_)
And(Not(x), Not(y))
```

SymPy's `simplify()` function can also be used to simplify logic expressions to their simplest forms.

`sympy.logic.boolalg.bool_map(bool1, bool2)`

Return the simplified version of `bool1`, and the mapping of variables that makes the two expressions `bool1` and `bool2` represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned. For example, `And(x, y)` is logically equivalent to `And(a, b)` for the mapping {`x: a, y:b`} or {`x: b, y:a`}. If no such mapping exists, return `False`.

### Examples

```
>>> from sympy import SOPform, bool_map, Or, And, Not, Xor
>>> from sympy.abc import w, x, y, z, a, b, c, d
>>> function1 = SOPform(['x','z','y'], [[1, 0, 1], [0, 0, 1]])
>>> function2 = SOPform(['a','b','c'], [[1, 0, 1], [1, 0, 0]])
>>> bool_map(function1, function2)
(And(Not(z), y), {y: a, z: b})
```

The results are not necessarily unique, but they are canonical. Here, ( $w, z$ ) could be ( $a, d$ ) or ( $d, a$ ):

```
>>> eq = Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))
>>> bool_map(eq, eq2)
(Or(And(Not(y), w), And(Not(y), z), And(x, y)), {w: a, x: b, y: c, z: d})
>>> eq = And(Xor(a, b), c, And(c,d))
>>> bool_map(eq, eq.subs(c, x))
(And(Or(Not(a), Not(b)), Or(a, b), c, d), {a: a, b: b, c: d, d: x})
```

## 3.12.5 Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence *True*.

For example, the expression  $x \& \neg x$  is not satisfiable, since there are no values for  $x$  that make this sentence *True*. On the other hand,  $(x \mid y) \& (x \mid \neg y) \& (\neg x \mid y)$  is satisfiable with both  $x$  and  $y$  being *True*.

```
>>> from sympy.logic.inference import satisfiable
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> satisfiable(x & ~x)
False
>>> satisfiable((x | y) & (x | ~y) & (~x | y))
{x: True, y: True}
```

As you see, when a sentence is satisfiable, it returns a model that makes that sentence *True*. If it is not satisfiable it will return `False`.

`sympy.logic.inference.satisfiable(expr, algorithm='dpll2', all_models=False)`

Check satisfiability of a propositional sentence. Returns a model when it succeeds. Returns `{true: true}` for trivially true expressions.

On setting `all_models` to `True`, if given `expr` is satisfiable then returns a generator of models. However, if `expr` is unsatisfiable then returns a generator containing the single element `False`.

### Examples

```
>>> from sympy.abc import A, B
>>> from sympy.logic.inference import satisfiable
>>> satisfiable(A & ~B)
{A: True, B: False}
>>> satisfiable(A & ~A)
```

```
False
>>> satisfiable(True)
{True: True}
>>> next(satisfiable(A & ~A, all_models=True))
False
>>> models = satisfiable((A >> B) & B, all_models=True)
>>> next(models)
{A: False, B: True}
>>> next(models)
{A: True, B: True}
>>> def use_models(models):
...     for model in models:
...         if model:
...             # Do something with the model.
...             print(model)
...         else:
...             # Given expr is unsatisfiable.
...             print("UNSAT")
>>> use_models(satisfiable(A >> ~A, all_models=True))
{A: False}
>>> use_models(satisfiable(A ^ A, all_models=True))
UNSAT
```

## 3.13 Matrices

A module that handles matrices.

Includes functions for fast creating matrices like zero, one/eye, random matrix, etc.

Contents:

### 3.13.1 Matrices (linear algebra)

#### Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```
>>> from sympy.interactive.printing import init_printing
>>> init_printing(use_unicode=False, wrap_line=False, no_global=True)
>>> from sympy.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1  0  0]
[]
[0  0  0]
>>> Matrix([M, (0, 0, -1)])
[1  0  0]
[]
[0  0  0]
[]
[0  0  -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
```

```
[ ]  
[2]  
[ ]  
[3]
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:

```
>>> Matrix(2, 3, [1, 2, 3, 4, 5, 6])  
[1 2 3]  
[      ]  
[4 5 6]
```

More interesting (and useful), is the ability to use a 2-variable function (or `lambda`) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):  
...     if i == j:  
...         return 1  
...     else:  
...         return 0  
...  
>>> Matrix(4, 4, f)  
[1 0 0 0]  
[      ]  
[0 1 0 0]  
[      ]  
[0 0 1 0]  
[      ]  
[0 0 0 1]
```

Finally let's use `lambda` to create a 1-line matrix with 1's in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)  
[1 0 1 0]  
[      ]  
[0 1 0 1]  
[      ]  
[1 0 1 0]
```

There are also a couple of special constructors for quick matrix construction: `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively, and `diag` to put matrices or elements along the diagonal:

```
>>> eye(4)  
[1 0 0 0]  
[      ]  
[0 1 0 0]  
[      ]  
[0 0 1 0]  
[      ]  
[0 0 0 1]  
>>> zeros(2)  
[0 0]  
[      ]  
[0 0]  
>>> zeros(2, 5)  
[0 0 0 0 0]  
[      ]
```

```
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[ ]
[1  1  1]
[ ]
[1  1  1]
>>> ones(1, 3)
[1  1  1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1  0  0]
[ ]
[0  1  2]
[ ]
[0  3  4]
```

## Basic Manipulation

While learning to work with matrices, let's choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```
>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5
```

Since this is Python we're also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```
>>> M[0:2, 0:2]
[1  2]
[ ]
[4  5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1 2 3]
[]
[4 5 6]
```

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing `M2` didn't change `M`. Since we can slice, we can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1 2 3 4]
[]
[5 6 7 8]
[]
[9 10 11 12]
[]
[13 14 15 16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1 2 3 0]
[]
[5 6 7 8]
[]
[9 10 0 12]
[]
[13 14 15 16]
```

as well as assign slices:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1 2 3 4]
[]
[5 6 7 8]
[]
[9 10 0 0]
[]
[13 14 0 0]
```

All the standard arithmetic operations are supported:

```
>>> M = Matrix(([1,2,3],[4,5,6],[7,8,9]))
>>> M - M
[0  0  0]
[      ]
[0  0  0]
[      ]
[0  0  0]
>>> M + M
[2  4  6 ]
[      ]
[8  10 12]
[      ]
[14 16 18]
>>> M * M
[30  36  42 ]
[      ]
[66  81  96 ]
[      ]
[102 126 150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[  ]
[29]
[  ]
[47]
>>> M**2
[30  36  42 ]
[      ]
[66  81  96 ]
[      ]
[102 126 150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)
>>> M
[4  5  6]
[      ]
[7  8  9]
>>> M.col_del(1)
>>> M
[4  6]
[      ]
[7  9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall that the `row_del()` and `col_del()` operations don't return a value - they simply change the matrix object. We can also "glue" together matrices of the appropriate size:

```
>>> M1 = eye(3)
>>> M2 = zeros(3, 4)
>>> M1.row_join(M2)
[[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

## Operations on entries

We are not restricted to having multiplication between two matrices:

```
>>> M = eye(3)
>>> 2*M
[[2 0 0]
 [0 2 0]
 [0 0 2]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[[2 0 0]
 [0 2 0]
 [0 0 2]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[[x, 0, 0],
 [0, x, 0],
 [0, 0, x]]
>>> M.subs(x, 4)
[[4, 0, 0],
 [0, 4, 0],
 [0, 0, 4]]
>>> y = Symbol('y')
>>> M.subs(x, y)
[[y, 0, 0],
 [0, y, 0],
 [0, 0, y]]
```

## Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```
>>> M = Matrix(([1, 2, 3], [3, 6, 2], [2, 0, 1]))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(([1, 0, 0], [1, 0, 0], [1, 0, 0]))
>>> M3.det()
0
```

Another common operation is the inverse: In SymPy, this is computed by Gaussian elimination by default (for dense matrices) but we can specify it be done by *LU* decomposition as well:

```
>>> M2.inv()
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]
>>> M2.inv(method="LU")
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1]]
>>> M.inv(method="LU")
[[-3/14, 1/14, 1/2],
 [-1/28, 5/28, -1/4]]
```

```
[  ]
[ 3/7   -1/7   0  ]
>>> M * M.inv(method="LU")
[1  0  0]
[  ]
[0  1  0]
[  ]
[0  0  1]
```

We can perform a  $QR$  factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[  ---  - ---  - --- ]
[ \sqrt{6}  -\sqrt{3}  -\sqrt{2} ]
[----- ----- -----]
[ 6       3       2   ]
[  ]
[  ]
[  ---  - ---  - --- ]
[ \sqrt{6}  -\sqrt{3}  \sqrt{2} ]
[----- ----- -----]
[ 6       3       2   ]
[  ]
[  ]
[  ---  - ---  - --- ]
[ \sqrt{6}  \sqrt{3}  0   ]
[ 3       3       0   ]
>>> R
[  ]
[  ---  4*\sqrt{6}  - --- ]
[ \sqrt{6}  -----  2*\sqrt{6} ]
[ 3       ]
[  ]
[  ]
[  ---  \sqrt{3}  ]
[ 0       -----  0   ]
[ 3       ]
[  ]
[  ]
[ 0       0       \sqrt{2} ]
>>> Q*R
[1  1  1]
[  ]
[1  1  3]
[  ]
[2  3  4]
```

In addition to the solvers in the `solver.py` file, we can solve the system  $Ax=b$  by passing the  $b$  vector to the matrix  $A$ 's `LUsolve` function. Here we'll cheat a little choose  $A$  and  $x$  then multiply to get  $b$ . Then we can solve for  $x$  and check that it's correct:

```
>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
```

```
[7]
[ ]
[5]
```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```
>>> L = [Matrix([2,3,5]), Matrix([3,6,2]), Matrix([8,3,6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)
```

Let's take a look at the vectors:

```
>>> for i in out1:
...     print(i)
...
Matrix([[2], [3], [5]])
Matrix([[23/19], [63/19], [-47/19]])
Matrix([[1692/353], [-1551/706], [-423/706]])
>>> for i in out2:
...     print(i)
...
Matrix([[sqrt(38)/19], [3*sqrt(38)/38], [5*sqrt(38)/38]])
Matrix([[23*sqrt(6707)/6707], [63*sqrt(6707)/6707], [-47*sqrt(6707)/6707]])
Matrix([[12*sqrt(706)/353], [-11*sqrt(706)/706], [-3*sqrt(706)/706]])
```

We can spot-check their orthogonality with `dot()` and their normality with `norm()`:

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the `matrices.py` file for all functionality.

## MatrixBase Class Reference

```
class sympy.matrices.matrices.MatrixBase
```

C

By-element conjugation.

D

Return Dirac conjugate (if `self.rows == 4`).

**See also:**

[conjugate](#) ([page 590](#)) By-element conjugation

[H](#) ([page 582](#)) Hermite conjugation

### Examples

```
>>> from sympy import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([
[1 - I, 0, 0, 0],
[0, 1 - I, 0, 0],
[0, 0, -1 + I, 0],
[2, 0, 0, -1 + I]])
```

If the matrix does not have 4 rows an `AttributeError` will be raised because this property is only defined for matrices with 4 rows.

```
>>> Matrix(eye(2)).D
Traceback (most recent call last):
...
AttributeError: Matrix has no attribute D.
```

[H](#)

Return Hermite conjugate.

### See also:

[conjugate](#) ([page 590](#)) By-element conjugation

[D](#) ([page 581](#)) Dirac conjugation

### Examples

```
>>> from sympy import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[0],
[1 + I],
[2],
[3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

[LDLdecomposition\(\)](#)

Returns the LDL Decomposition ( $L$ ,  $D$ ) of matrix  $A$ , such that  $L * D * L.T == A$  This method eliminates the use of square root. Further this ensures that all the diagonal entries of  $L$  are 1.  $A$  must be a square, symmetric, positive-definite and non-singular matrix.

### See also:

[cholesky](#) ([page 588](#)), [LUdecomposition](#) ([page 583](#)), [QRdecomposition](#) ([page 584](#))

## Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

### LDLsolve(*rhs*)

Solves  $Ax = B$  using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.

#### See also:

[LDLdecomposition](#) (page 582), [lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [diagonal\\_solve](#) (page 590), [LUsolve](#) (page 584), [QRsolve](#) (page 585), [pinv\\_solve](#) (page 608)

## Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

### LUdecomposition(*iszerofunc*=<*function \_iszero at 0x7fb9a6f2ed8*>)

Returns the decomposition LU and the row swaps p.

#### See also:

[cholesky](#) (page 588), [LDLdecomposition](#) (page 582), [QRdecomposition](#) (page 584), [LUdecomposition\\_Simple](#) (page 584), [LUdecompositionFF](#) (page 584), [LUsolve](#) (page 584)

## Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([
[ 1, 0],
[3/2, 1]])
>>> U
Matrix([
```

```
[4,      3],  
[0, -3/2]])
```

### LUdecompositionFF()

Compute a fraction-free LU decomposition.

Returns 4 matrices P, L, D, U such that PA = L D\*\*-1 U. If the elements of the matrix belong to some integral domain I, then all elements of L, D and U are guaranteed to belong to I.

#### Reference

- W. Zhou & D.J. Jeffrey, “Fraction-free matrix factors: new forms for LU and QR factors”. Frontiers in Computer Science in China, Vol 2, no. 1, pp. 67-80, 2008.

#### See also:

[LUdecomposition](#) (page 583), [LUdecomposition\\_Simple](#) (page 584), [LUsolve](#) (page 584)

### LUdecomposition\_Simple(iszerofunc=<function \_iszero at 0x7fb9a6f2ed8>)

Returns A comprised of L, U (L's diag entries are 1) and p which is the list of the row swaps (in order).

#### See also:

[LUdecomposition](#) (page 583), [LUdecompositionFF](#) (page 584), [LUsolve](#) (page 584)

### LUsolve(rhs, iszerofunc=<function \_iszero at 0x7fb9a6f2ed8>)

Solve the linear system Ax = rhs for x where A = self.

This is for symbolic matrices, for real or complex ones use mpmath.lu\_solve or mpmath.qr\_solve.

#### See also:

[lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [diagonal\\_solve](#) (page 590), [LDLsolve](#) (page 583), [QRsolve](#) (page 585), [pinv\\_solve](#) (page 608), [LUdecomposition](#) (page 583)

### QRdecomposition()

Return Q, R where A = Q\*R, Q is orthogonal and R is upper triangular.

#### See also:

[cholesky](#) (page 588), [LDLdecomposition](#) (page 582), [LUdecomposition](#) (page 583), [QRsolve](#) (page 585)

## Examples

This is the example from wikipedia:

```
>>> from sympy import Matrix  
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])  
>>> Q, R = A.QRdecomposition()  
>>> Q  
Matrix([  
[ 6/7, -69/175, -58/175],  
[ 3/7, 158/175, 6/175],  
[-2/7, 6/35, -33/35]])  
>>> R  
Matrix([  
[14, 21, -14],  
[ 0, 175, -70],  
[ 0, 0, 35]])
```

```
>>> A == Q*R
True

QR factorization of an identity matrix:

>>> A = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> R
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])

QRsolve(b)
Solve the linear system ‘Ax = b’.
```

‘self’ is the matrix ‘A’, the method argument is the vector ‘b’. The method returns the solution vector ‘x’. If ‘b’ is a matrix, the system is solved for each column of ‘b’ and the return value is a matrix of the same shape as ‘b’.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you don’t need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use mpmath.qr\_solve.

#### See also:

[lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [diagonal\\_solve](#) (page 590), [LDLsolve](#) (page 583), [LUsolve](#) (page 584), [pinv\\_solve](#) (page 608), [QRdecomposition](#) (page 584)

T

Matrix transposition.

add(b)

Return self + b

adjoint()

Conjugate transpose or Hermitian conjugation.

adjugate(*method='berkowitz'*)

Returns the adjugate matrix.

Adjugate matrix is the transpose of the cofactor matrix.

<http://en.wikipedia.org/wiki/Adjugate>

#### See also:

[cofactorMatrix](#) (page 589), [transpose](#) (page 614), [berkowitz](#) (page 586)

atoms(\**types*)

Returns the atoms that form the current object.

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy.matrices import Matrix
>>> Matrix([[x]])
Matrix([[x]])
>>> _.atoms()
set([x])
```

### berkowitz()

The Berkowitz algorithm.

Given  $N \times N$  matrix with symbolic content, compute efficiently coefficients of characteristic polynomials of ‘self’ and all its square sub-matrices composed by removing both  $i$ -th row and column, without division in the ground domain.

This method is particularly useful for computing determinant, principal minors and characteristic polynomial, when ‘self’ has complicated coefficients e.g. polynomials. Semi-direct usage of this algorithm is also important in computing efficiently sub-resultant PRS.

Assuming that  $M$  is a square matrix of dimension  $N \times N$  and  $I$  is  $N \times N$  identity matrix, then the following definition of characteristic polynomial is begin used:

$$\text{charpoly}(M) = \det(t^*I - M)$$

As a consequence, all polynomials generated by Berkowitz algorithm are monic.

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

>>> M = Matrix([[x, y, z], [1, 0, 0], [y, z, x]])

>>> p, q, r = M.berkowitz()

>>> p # 1 x 1 M's sub-matrix
(1, -x)

>>> q # 2 x 2 M's sub-matrix
(1, -x, -y)

>>> r # 3 x 3 M's sub-matrix
(1, -2*x, x**2 - y*z - y, x*y - z**2)
```

For more information on the implemented algorithm refer to:

- [1] **S.J. Berkowitz**, **On computing the determinant in small parallel time** using a small number of processors, ACM, Information Processing Letters 18, 1984, pp. 147-150
- [2] **M. Keber**, **Division-Free computation of sub-resultants** using Bezout matrices, Tech. Report MPI-I-2006-1-006, Saarbrucken, 2006

See also:

`berkowitz_det` (page 587), `berkowitz_minors` (page 587), `berkowitz_charpoly` (page 586), `berkowitz_eigenvals` (page 587)

`berkowitz_charpoly(x=_lambda, simplify=<function simplify at 0x7fb9a71f938>)`

Computes characteristic polynomial minors using Berkowitz method.

A PurePoly is returned so using different variables for `x` does not affect the comparison or the polynomials:

**See also:**

`berkowitz` (page 586)

### Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying `x` is optional; a Dummy with name `_lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that `x` doesn't clash with an existing symbol, so using the default (`_lambda`) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
_lambda**2 - _lambda - 2*x
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

`berkowitz_det()`

Computes determinant using Berkowitz method.

**See also:**

`det` (page 590), `berkowitz` (page 586)

`berkowitz_eigenvals(**flags)`

Computes eigenvalues of a Matrix using Berkowitz method.

**See also:**

`berkowitz` (page 586)

`berkowitz_minors()`

Computes principal minors using Berkowitz method.

**See also:**

`berkowitz` (page 586)

`charpoly(x=_lambda, simplify=<function simplify at 0x7fb9a71f938>)`

Computes characteristic polynomial minors using Berkowitz method.

A PurePoly is returned so using different variables for `x` does not affect the comparison or the polynomials:

**See also:**

`berkowitz` (page 586)

## Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying `x` is optional; a Dummy with name `lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that `x` doesn't clash with an existing symbol, so using the default (`lambda`) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
_lambda**2 - _lambda - 2*x
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

### cholesky()

Returns the Cholesky decomposition L of a matrix A such that  $L \cdot L.T = A$

A must be a square, symmetric, positive-definite and non-singular matrix.

**See also:**

[LDLdecomposition](#) (page 582), [LUdecomposition](#) (page 583), [QRdecomposition](#) (page 584)

## Examples

```
>>> from sympy.matrices import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5,  0,  0],
[ 3,  3,  0],
[-1,  1,  3]])
>>> A.cholesky() * A.cholesky().T
Matrix([
[25, 15, -5],
[15, 18,  0],
[-5,  0, 11]])
```

### cholesky\_solve(rhs)

Solves  $Ax = B$  using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows > cols, the least squares solution is returned.

**See also:**

[lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [diagonal\\_solve](#) (page 590), [LDLsolve](#) (page 583), [LUsolve](#) (page 584), [QRsolve](#) (page 585), [pinv\\_solve](#) (page 608)

### cofactor(*i, j, method='berkowitz'*)

Calculate the cofactor of an element.

**See also:**

`cofactorMatrix` (page 589), `minorEntry` (page 605), `minorMatrix` (page 605)

`cofactorMatrix(method='berkowitz')`

Return a matrix containing the cofactor of each element.

**See also:**

`cofactor` (page 588), `minorEntry` (page 605), `minorMatrix` (page 605), `adjugate` (page 585)

`col_insert(pos, mti)`

Insert one or more columns at the given column position.

**See also:**

`sympy.matrices.dense.DenseMatrix.col` (page 624), `sympy.matrices.sparse.SparseMatrix.col` (page 634), `row_insert` (page 610)

### Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([
[0, 1, 0],
[0, 1, 0],
[0, 1, 0]])
```

`col_join(bott)`

Concatenates two matrices along self's last and bott's first row

**See also:**

`sympy.matrices.dense.DenseMatrix.col` (page 624), `sympy.matrices.sparse.SparseMatrix.col` (page 634), `row_join` (page 610)

### Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

`condition_number()`

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

**See also:**

`singular_values` (page 611)

## Examples

```
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

### conjugate()

By-element conjugation.

### cross(*b*)

Return the cross product of *self* and *b* relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as *self* will be returned. If *b* has the same shape as *self* then common identities for the cross product (like  $axb = -bxa$ ) will hold.

#### See also:

[dot](#) (page 592), [multiply](#) (page 605), [multiply\\_elementwise](#) (page 605)

### det(*method='bareis'*)

Computes the matrix determinant using the method “*method*”.

**Possible values for “*method*”:** bareis ... det\_bareis berkowitz ... berkowitz\_det det\_LU ... det\_LU\_decomposition

#### See also:

[det\\_bareis](#) (page 590), [berkowitz\\_det](#) (page 587), [det\\_LU\\_decomposition](#) (page 590)

### det\_LU\_decomposition()

Compute matrix determinant using LU decomposition

Note that this method fails if the LU decomposition itself fails. In particular, if the matrix has no inverse this method will fail.

TODO: Implement algorithm for sparse matrices (SFF), <http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps>.

#### See also:

[det](#) (page 590), [det\\_bareis](#) (page 590), [berkowitz\\_det](#) (page 587)

### det\_bareis()

Compute matrix determinant using Bareis’ fraction-free algorithm which is an extension of the well known Gaussian elimination method. This approach is best suited for dense symbolic matrices and will result in a determinant with minimal number of fractions. It means that less term rewriting is needed on resulting formulae.

TODO: Implement algorithm for sparse matrices (SFF), <http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps>.

#### See also:

[det](#) (page 590), [berkowitz\\_det](#) (page 587)

### diagonal\_solve(*rhs*)

Solves  $Ax = B$  efficiently, where *A* is a diagonal Matrix, with non-zero diagonal entries.

#### See also:

[lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [LDLsolve](#) (page 583), [LUsolve](#) (page 584), [QRsolve](#) (page 585), [pinv\\_solve](#) (page 608)

## Examples

```
>>> from sympy.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True

diagonalize(real_only=False, sort=False, normalize=False)
Return (P, D), where D is diagonal and
```

$$D = P^{-1} * M * P$$

where M is current matrix.

See also:

[is\\_diagonal](#) (page 597), [is\\_diagonalizable](#) (page 597)

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
Matrix([
[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> (P, D) = m.diagonalize()
>>> D
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> P
Matrix([
[-1, 0, -1],
[ 0, 0, -1],
[ 2, 1,  2]])
>>> P.inv() * m * P
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])

diff(*args)
Calculate the derivative of each element in the matrix.
```

See also:

[integrate](#) (page 594), [limit](#) (page 605)

## Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
```

```
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

### dot(b)

Return the dot product of Matrix self and b relaxing the condition of compatible dimensions: if either the number of rows or columns are the same as the length of b then the dot product is returned. If self is a row or column vector, a scalar is returned. Otherwise, a list of results is returned (and in that case the number of columns in self must match the length of b).

**See also:**

[cross](#) (page 590), [multiply](#) (page 605), [multiply\\_elementwise](#) (page 605)

### Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = [1, 1, 1]
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> M.dot(v)
[6, 15, 24]
```

### dual()

Returns the dual of a matrix, which is:

$(1/2) * \text{levicivita}(i, j, k, l) * M(k, l)$  summed over indices  $k$  and  $l$

Since the levicivita method is anti\_symmetric for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the ‘matrix’  $M$  is a contravariant anti\_symmetric second rank tensor, so that the dual is a covariant second rank tensor.

### eigenvals(\*\*flags)

Return eigen values using the berkowitz\_eigenvals routine.

Since the roots routine doesn’t always work well with Floats, they will be replaced with Rationals before calling that routine. If this is not desired, set flag `rational` to False.

### eigenvecs(\*\*flags)

Return list of triples (eigenval, multiplicity, basis).

**The flag simplify has two effects:** 1) if `bool(simplify)` is True, `as_content_primitive()` will be used to tidy up normalization artifacts; 2) if `nullspace` needs simplification to compute the basis, the `simplify` flag will be passed on to the `nullspace` routine which will interpret it there.

If the matrix contains any Floats, they will be changed to Rationals for computation purposes, but the answers will be returned after being evaluated with `evalf`. If it is desired to removed small imaginary portions during the `evalf` step, pass a value for the `chop` flag.

### evalf(prec=None, \*\*options)

Apply `evalf()` to each element of self.

### exp()

Return the exponentiation of a square matrix.

```
expand(deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True,
       multinomial=True, basic=True, **hints)
Apply core.function.expand to each entry of the matrix.
```

### Examples

```
>>> from sympy.abc import x
>>> from sympy.matrices import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)]])
>>> _.expand()
Matrix([[x**2 + x]])
```

### extract(rowsList, colsList)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range  $-n \leq i < n$  where  $n$  is the number of rows or columns.

### Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0, 1],
[3, 4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[2],
[8]]))
```

### free\_symbols

Returns the free symbols within the matrix.

### Examples

```
>>> from sympy.abc import x
>>> from sympy.matrices import Matrix
>>> Matrix([[x], [1]]).free_symbols
set([x])
```

### get\_diag\_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

### Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([
[1, 3],
[y, z**2]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

### has(\*patterns)

Test whether any subexpression matches any of the patterns.

### Examples

```
>>> from sympy import Matrix, Float
>>> from sympy.abc import x, y
>>> A = Matrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

### classmethod hstack(\*args)

Return a matrix formed by joining args horizontally (i.e. by repeated application of row\_join).

### Examples

```
>>> from sympy.matrices import Matrix, eye
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

### integrate(\*args)

Integrate each element of the matrix.

See also:

[limit](#) (page 605), [diff](#) (page 591)

### Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([
[x**2/2, x*y],
[x, 0]])
```

```
>>> M.integrate((x, 0, 2))
Matrix([
[2, 2*y],
[2, 0]])
```

`inv(method=None, **kwargs)`  
Returns the inverse of the matrix

`inv_mod(m)`

Returns the inverse of the matrix  $K$  (mod  $m$ ), if it exists.

Method to find the matrix inverse of  $K$  (mod  $m$ ) implemented in this function:

- Compute  $\text{adj}(K) = \text{cof}(K)^t$ , the adjoint matrix of  $K$ .
- Compute  $r = 1/\det(K)$  (mod  $m$ ).
- $K^{-1} = r \cdot \text{adj}(K)$  (mod  $m$ ).

### Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
```

```
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

`inverse_ADJ(iszerofunc=<function _iszero at 0x7fb9a6f2ed8>)`

Calculates the inverse using the adjugate matrix and a determinant.

See also:

[sympy.matrices.matrices.MatrixBase.inv](#) (page 595), [inverse\\_LU](#) (page 596), [inverse\\_GE](#) (page 595)

`inverse_GE(iszerofunc=<function _iszero at 0x7fb9a6f2ed8>)`

Calculates the inverse using Gaussian elimination.

See also:

[sympy.matrices.matrices.MatrixBase.inv](#) (page 595), [inverse\\_LU](#) (page 596), [inverse\\_ADJ](#) (page 595)

`inverse_LU(iszerofunc=<function _iszero at 0x7fb9a6f2ed8>)`

Calculates the inverse using LU decomposition.

See also:

`sympy.matrices.matrices.MatrixBase.inv` (page 595), `inverse_GE` (page 595), `inverse_ADJ` (page 595)

`is_anti_symmetric(simplify=True)`

Check if matrix M is an antisymmetric matrix, that is, M is a square matrix with all  $M[i, j] == -M[j, i]$ .

When `simplify=True` (default), the sum  $M[i, j] + M[j, i]$  is simplified before testing to see if it is zero. By default, the SymPy simplify function is used. To use a custom function set `simplify` to a function that accepts a single argument which returns a simplified expression. To skip simplification, set `simplify` to `False` but note that although this will be faster, it may induce false negatives.

## Examples

```
>>> from sympy import Matrix, symbols
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[0, 1],
[-1, 0]])
>>> m.is_anti_symmetric()
True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[0, 0, x],
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [0, x**2 + 2*x + 1, y,
...                   -(x + 1)**2, 0, x*y,
...                   -y, -x*y, 0])
```

Simplification of matrix elements is done by default so even though two elements which should be equal and opposite wouldn't pass an equality test, the matrix is still reported as anti-symmetric:

```
>>> m[0, 1] == -m[1, 0]
False
>>> m.is_anti_symmetric()
True
```

If ‘`simplify=False`’ is used for the case when a Matrix is already simplified, this will speed things up. Here, we see that without simplification the matrix does not appear anti-symmetric:

```
>>> m.is_anti_symmetric(simplify=False)
False
```

But if the matrix were already expanded, then it would appear anti-symmetric and simplification in the `is_anti_symmetric` routine is not needed:

```
>>> m = m.expand()
>>> m.is_anti_symmetric(simplify=False)
True
```

### is\_diagonal()

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

**See also:**

[is\\_lower](#) (page 598), [is\\_upper](#) (page 601), [is\\_diagonalizable](#) (page 597), [diagonalize](#) (page 591)

### Examples

```
>>> from sympy import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([
[1, 0],
[0, 2]])
>>> m.is_diagonal()
True

>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([
[1, 1],
[0, 2]])
>>> m.is_diagonal()
False

>>> m = diag(1, 2, 3)
>>> m
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> m.is_diagonal()
True
```

### is\_diagonalizable(*reals\_only=False, clear\_subproducts=True*)

Check if matrix is diagonalizable.

If *reals\_only==True* then check that diagonalized matrix consists of the only not complex values.

Some subproducts could be used further in other methods to avoid double calculations, By default (if *clear\_subproducts==True*) they will be deleted.

**See also:**

[is\\_diagonal](#) (page 597), [diagonalize](#) (page 591)

### Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
```

```
>>> m
Matrix([
[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> m.is_diagonalizable()
True
>>> m = Matrix(2, 2, [0, 1, 0, 0])
>>> m
Matrix([
[0, 1],
[0, 0]])
>>> m.is_diagonalizable()
False
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[0, 1],
[-1, 0]])
>>> m.is_diagonalizable()
True
>>> m.is_diagonalizable(True)
False
```

### is\_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element  $i,j$  is the complex conjugate of element  $j,i$ .

### Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy import I
>>> from sympy.abc import x
>>> a = Matrix([[1, I], [-I, 1]])
>>> a
Matrix([
[1, I],
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

### is\_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

See also:

[is\\_upper](#) (page 601), [is\\_diagonal](#) (page 597), [is\\_lower\\_hessenberg](#) (page 599)

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_lower
True

>>> m = Matrix(4, 3, [0, 0, 0, 2, 0, 0, 1, 4, 0, 6, 6, 5])
>>> m
Matrix([
[0, 0, 0],
[2, 0, 0],
[1, 4, 0],
[6, 6, 5]])
>>> m.is_lower
True

>>> from sympy.abc import x, y
>>> m = Matrix(2, 2, [x**2 + y, y**2 + x, 0, x + y])
>>> m
Matrix([
[x**2 + y, x + y**2],
[0, x + y]])
>>> m.is_lower
False
```

### is\_lower\_hessenberg

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

See also:

[is\\_upper\\_hessenberg](#) (page 602), [is\\_lower](#) (page 598)

## Examples

```
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a
Matrix([
[1, 2, 0, 0],
[5, 2, 3, 0],
[3, 4, 3, 7],
[5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

### is\_nilpotent()

Checks if a matrix is nilpotent.

A matrix B is nilpotent if for some integer k,  $B^{**k}$  is a zero matrix.

## Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[0, 0, 0], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
True

>>> a = Matrix([[1, 0, 1], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
False
```

### is\_square

Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

## Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
False
>>> b.is_square
True
>>> c.is_square
True
```

### is\_symbolic()

Checks if any elements contain Symbols.

## Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.is_symbolic()
True
```

### is\_symmetric(*simplify=True*)

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using ‘simplify=False’; while speeding things a bit, this may however induce false negatives.

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m
Matrix([
[0, 1],
```

```
[1, 2]])
>>> m.is_symmetric()
True

>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m
Matrix([
[0, 1],
[2, 0]])
>>> m.is_symmetric()
False

>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m
Matrix([
[0, 0, 0],
[0, 0, 0]])
>>> m.is_symmetric()
False

>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0, 3])
>>> m
Matrix([
[1, x**2 + 2*x + 1, y],
[(x + 1)**2, 2, 0],
[y, 0, 3]])
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up `is_symmetric()` test by using ‘`simplify=False`’.

```
>>> m.is_symmetric(simplify=False)
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

### `is_upper`

Check if matrix is an upper triangular matrix. `True` can be returned even if the matrix is not square.

**See also:**

`is_lower` (page 598), `is_diagonal` (page 597), `is_upper_hessenberg` (page 602)

### Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_upper
True
```

```
>>> m = Matrix(4, 3, [5, 1, 9, 0, 4, 6, 0, 0, 5, 0, 0, 0])
>>> m
Matrix([
[5, 1, 9],
[0, 4, 6],
[0, 0, 5],
[0, 0, 0]])
>>> m.is_upper
True

>>> m = Matrix(2, 3, [4, 2, 5, 6, 1, 1])
>>> m
Matrix([
[4, 2, 5],
[6, 1, 1]])
>>> m.is_upper
False
```

### is\_upper\_hessenberg

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

See also:

[is\\_lower\\_hessenberg](#) (page 599), [is\\_upper](#) (page 601)

### Examples

```
>>> from sympy.matrices import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1, 3]])
>>> a
Matrix([
[1, 4, 2, 3],
[3, 4, 1, 7],
[0, 2, 3, 4],
[0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

### is\_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

### Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
```

```

True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero

```

**jacobian(*X*)**

Calculates the Jacobian matrix (derivative of a vectorial function).

**Parameters** *self* : vector of expressions representing functions  $f_i(x_1, \dots, x_n)$ .

**X** : set of  $x_i$ 's in order, it can be a list or a Matrix

**Both self and X can be a row or a column matrix in any order**

**(i.e., jacobian() should always work).**

**See also:**

[sympy.matrices.dense.hessian](#) (page 618), [sympy.matrices.dense.wronskian](#) (page 619)

**Examples**

```

>>> from sympy import sin, cos, Matrix
>>> from sympy.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi), rho*cos(phi)],
[2*rho, 0]])
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi), rho*cos(phi)]])

```

**jordan\_cells(*calc\_transformation=True*)**

Return a list of Jordan cells of current matrix. This list shape Jordan matrix J.

If *calc\_transformation* is specified as False, then transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

will not be calculated.

**See also:**

[jordan\\_form](#) (page 604)

**Notes**

Calculation of transformation P is not implemented yet.

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 4, [
...     6, 5, -2, -3,
...     -3, -1, 3, 3,
...     2, 1, -2, -3,
...     -1, 1, 5, 5])

>>> P, Jcells = m.jordan_cells()
>>> Jcells[0]
Matrix([
[2, 1],
[0, 2]])
>>> Jcells[1]
Matrix([
[2, 1],
[0, 2]]))

jordan_form(calc_transformation=True)
```

Return Jordan form J of current matrix.

Also the transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

and the jordan blocks forming J will be calculated.

See also:

[jordan\\_cells](#) (page 603)

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix([
...     [ 6, 5, -2, -3],
...     [-3, -1, 3, 3],
...     [ 2, 1, -2, -3],
...     [-1, 1, 5, 5]])
>>> P, J = m.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

`key2bounds(keys)`

Converts a key with potentially mixed types of keys (integer and slice) into a tuple of ranges and raises an error if any index is out of self's range.

See also:

[key2ij](#) (page 604)

`key2ij(key)`

Converts key into canonical form, converting integers or indexable items into valid integers for self's range or returning slices unchanged.

See also:

[key2bounds](#) (page 604)

`limit(*args)`

Calculate the limit of each element in the matrix.

See also:

[integrate](#) (page 594), [diff](#) (page 591)

### Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

`lower_triangular_solve(rhs)`

Solves  $Ax = B$ , where  $A$  is a lower triangular matrix.

See also:

[upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [diagonal\\_solve](#) (page 590), [LDLsolve](#) (page 583), [LUsolve](#) (page 584), [QRsolve](#) (page 585), [pinv\\_solve](#) (page 608)

`minorEntry(i, j, method='berkowitz')`

Calculate the minor of an element.

See also:

[minorMatrix](#) (page 605), [cofactor](#) (page 588), [cofactorMatrix](#) (page 589)

`minorMatrix(i, j)`

Creates the minor matrix of a given element.

See also:

[minorEntry](#) (page 605), [cofactor](#) (page 588), [cofactorMatrix](#) (page 589)

`multiply(b)`

Returns `self*b`

See also:

[dot](#) (page 592), [cross](#) (page 590), [multiply\\_elementwise](#) (page 605)

`multiply_elementwise(b)`

Return the Hadamard product (elementwise product) of A and B

See also:

[cross](#) (page 590), [dot](#) (page 592), [multiply](#) (page 605)

### Examples

```
>>> from sympy.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([
[ 0, 10, 200],
[300, 40,   5]])
```

`n(prec=None, **options)`

Apply `evalf()` to each element of `self`.

`norm(ord=None)`

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the `ord` parameter

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	•does not exist
inf	—	<code>max(abs(x))</code>
-inf	—	<code>min(abs(x))</code>
1	—	as below
-1	—	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	•does not exist	<code>sum(abs(x)**ord)**(1./ord)</code>

See also:

`normalized` (page 606)

### Examples

```
>>> from sympy import Matrix, Symbol, trigsimp, cos, sin, oo
>>> x = Symbol('x', extended_real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp(v.norm())
1
>>> v.norm(10)
(sin(x)**10 + cos(x)**10)**(1/10)
>>> A = Matrix([[1, 1], [1, 1]])
>>> A.norm(2)# Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1
```

`normalized()`

Return the normalized version of `self`.

See also:

[norm](#) (page 606)

`nullspace(simplify=False)`

Returns list of vectors (Matrix objects) that span nullspace of self

`permuteBkwd(perm)`

Permute the rows of the matrix with the given permutation in reverse.

See also:

[permuteFwd](#) (page 607)

### Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permuteBkwd([[0, 1], [0, 2]])
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

`permuteFwd(perm)`

Permute the rows of the matrix with the given permutation.

See also:

[permuteBkwd](#) (page 607)

### Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.permuteFwd([[0, 1], [0, 2]])
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

`pinv()`

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

See also:

[sympy.matrices.matrices.MatrixBase.inv](#) (page 595), [pinv\\_solve](#) (page 608)

### References

[R334] (page 1244)

## Examples

```
>>> from sympy import Matrix
>>> Matrix([[1, 2, 3], [4, 5, 6]]).pinv()
Matrix([
[-17/18, 4/9],
[-1/9, 1/9],
[13/18, -2/9]])

pinv_solve(B, arbitrary_matrix=None)
Solve Ax = B using the Moore-Penrose pseudoinverse.
```

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of `arbitrary_matrix`. If no solutions exist, the least-squares solution is returned.

### Parameters `B` : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

### `arbitrary_matrix` : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of `wn_m` will be used, with n and m being row and column position of each symbol.

### Returns `x` : Matrix

The matrix that will satisfy  $Ax = B$ . Will have as many rows as matrix A has columns, and as many columns as matrix B.

### See also:

[lower\\_triangular\\_solve](#) (page 605), [upper\\_triangular\\_solve](#) (page 614), [cholesky\\_solve](#) (page 588), [diagonal\\_solve](#) (page 590), [LDLsolve](#) (page 583), [LUsolve](#) (page 584), [QRsolve](#) (page 585), [pinv](#) (page 607)

## Notes

This may return either exact solutions or least squares solutions. To determine which, check `A * A.pinv() * B == B`. It will be True if exact solutions exist, and False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

## References

[R335] (page 1245)

### Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
[_w0_0/6 - _w1_0/3 + _w2_0/6 - 55/18],
[-_w0_0/3 + 2*_w1_0/3 - _w2_0/3 + 1/9],
[_w0_0/6 - _w1_0/3 + _w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
[-55/18],
[ 1/9],
[ 59/18]])
```

`print_nonzero(symb='X')`

Shows location of non-zero entries for fast shape lookup.

### Examples

```
>>> from sympy.matrices import Matrix, eye
>>> m = Matrix(2, 3, lambda i, j: i*3+j)
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5]])
>>> m.print_nonzero()
[ XX]
[XXX]
>>> m = eye(4)
>>> m.print_nonzero("x")
[x  ]
[ x  ]
[  x ]
[   x]
```

`project(v)`

Return the projection of `self` onto the line containing `v`.

### Examples

```
>>> from sympy import Matrix, S, sqrt
>>> V = Matrix([sqrt(3)/2, S.Half])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])
```

`rank(iszerofunc=<function _iszero at 0x7fb9a6f2ed8>, simplify=False)`

Returns the rank of a matrix

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

```
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2

replace(F, G, map=False)
    Replaces Function F in Matrix entries with Function G.
```

### Examples

```
>>> from sympy import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([
[F(0), F(1)],
[F(1), F(2)]])
>>> N = M.replace(F,G)
>>> N
Matrix([
[G(0), G(1)],
[G(1), G(2)]])
```

`row_insert(pos, mti)`  
Insert one or more rows at the given row position.

See also:

[sympy.matrices.dense.DenseMatrix.row](#) (page 625), [sympy.matrices.sparse.SparseMatrix.row](#) (page 637), `col_insert` (page 589)

### Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
[1, 1, 1],
[0, 0, 0],
[0, 0, 0]])
```

`row_join(rhs)`  
Concatenates two matrices along self's last and rhs's first column

See also:

[sympy.matrices.dense.DenseMatrix.row](#) (page 625), [sympy.matrices.sparse.SparseMatrix.row](#) (page 637), `col_join` (page 589)

## Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

`rref(iszerofunc=<function _iszero at 0x7fb9a6f2ed8>, simplify=False)`

Return reduced row-echelon form of matrix and indices of pivot vars.

To simplify elements before finding nonzero pivots set `simplify=True` (to use the default SymPy `simplify` function) or pass a custom `simplify` function.

## Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), [0, 1])
```

`shape`

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

## Examples

```
>>> from sympy.matrices import zeros
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

`simplify(ratio=1.7, measure=<function count_ops at 0x7fb9aef6e60>)`

Apply `simplify` to each element of the matrix.

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy import sin, cos
>>> from sympy.matrices import SparseMatrix
>>> SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])
Matrix([[x*sin(y)**2 + x*cos(y)**2]])
>>> _.simplify()
Matrix([[x]])
```

```
singular_values()
Compute the singular values of a Matrix
```

See also:

`condition_number` (page 589)

### Examples

```
>>> from sympy import Matrix, Symbol
>>> x = Symbol('x', extended_real=True)
>>> A = Matrix([[0, 1, 0], [0, x, 0], [-1, 0, 0]])
>>> A.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

`solve(rhs, method='GE')`  
Return solution to self\*soln = rhs using given inversion method.

For a list of possible inversion methods, see the `.inv()` docstring.

`solve_least_squares(rhs, method='CH')`  
Return the least-square fit to the data.

By default the cholesky\_solve routine is used (method='CH'); other methods of matrix inversion can be used. To find out which are available, see the docstring of the `.inv()` method.

### Examples

```
>>> from sympy.matrices import Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of S represent coefficients of Ax + By and x and y are [2, 3] then S\*xy is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by S\*xy - r:

```
>>> S*xy - r
Matrix([
[1/3],
```

---

```
[1/3],
[1/3])
>>> _ .norm() .n(2)
0.58
```

If a different `xy` is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm() .n(2)
1.5
```

```
subs(*args, **kwargs)
```

Return a new matrix with `subs` applied to each entry.

### Examples

```
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _ .subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

```
table(printer, rowstart='[', rowend=']', rowsep='\n', colsep=', ', align='right')
```

String form of Matrix as a table.

`printer` is the printer to use for on the elements (generally something like `StrPrinter()`)

`rowstart` is the string used to start each row (by default '[').

`rowend` is the string used to end each row (by default ']').

`rowsep` is the string used to separate rows (by default a newline).

`colsep` is the string used to separate columns (by default ', ').

`align` defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

### Examples

```
>>> from sympy import Matrix
>>> from sympy.printing.str import StrPrinter
>>> M = Matrix([[1, 2], [-33, 4]])
>>> printer = StrPrinter()
>>> M.table(printer)
'[ 1, 2]\n[-33, 4]'
>>> print(M.table(printer))
[ 1, 2]
[-33, 4]
>>> print(M.table(printer, rowsep=',\n'))
[ 1, 2],
[-33, 4]
>>> print('[%s]' % M.table(printer, rowsep=',\n'))
```

```
[[ 1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1 , 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}

transpose()
Matrix transposition.

upper_triangular_solve(rhs)
Solves Ax = B, where A is an upper triangular matrix.

See also:
lower_triangular_solve (page 605), cholesky_solve (page 588), diagonal_solve (page 590),
LDLsolve (page 583), LUsolve (page 584), QRsolve (page 585), pinv_solve (page 608)

values()
Return non-zero values of self.

vec()
Return the Matrix converted into a one column matrix by stacking columns

See also:
vech (page 614)

Examples

>>> from sympy import Matrix
>>> m=Matrix([[1, 3], [2, 4]])
>>> m
Matrix([
[1, 3],
[2, 4]])
>>> m.vec()
Matrix([
[1],
[2],
[3],
[4]])

vech(diagonal=True, check_symmetry=True)
Return the unique elements of a symmetric Matrix as a one column matrix by stacking the
elements in the lower triangle.

Arguments: diagonal – include the diagonal cells of self or not check_symmetry – checks symmetry
of self but not completely reliably

See also:
vec (page 614)
```

## Examples

```
>>> from sympy import Matrix
>>> m=Matrix([[1, 2], [2, 3]])
>>> m
Matrix([
[1, 2],
[2, 3]])
>>> m.vech()
Matrix([
[1],
[2],
[3]])
>>> m.vech(diagonal=False)
Matrix([[2]])

classmethod vstack(*args)
Return a matrix formed by joining args vertically (i.e. by repeated application of col_join).
```

## Examples

```
>>> from sympy.matrices import Matrix, eye
>>> Matrix.vstack(eye(2), 2*eye(2))
Matrix([
[1, 0],
[0, 1],
[2, 0],
[0, 2]])
```

## Matrix Exceptions Reference

```
class sympy.matrices.matrices.MatrixError
class sympy.matrices.matrices.ShapeError
    Wrong matrix shape
class sympy.matrices.matrices.NonSquareMatrixError
```

## Matrix Functions Reference

```
sympy.matrices.matrices.classof(A, B)
Get the type of the result when combining matrices of different types.

Currently the strategy is that immutability is contagious.
```

## Examples

```
>>> from sympy import Matrix, ImmutableMatrix
>>> from sympy.matrices.matrices import classof
>>> M = Matrix([[1, 2], [3, 4]]) # a Mutable Matrix
>>> IM = ImmutableMatrix([[1, 2], [3, 4]])
>>> classof(M, IM)
<class 'sympy.matrices.immutable.ImmutableMatrix'>
```

`sympy.matrices.dense.matrix_multiply_elementwise(A, B)`  
Return the Hadamard product (elementwise product) of A and B

```
>>> from sympy.matrices import matrix_multiply_elementwise
>>> from sympy.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([
[ 0, 10, 200],
[300, 40,   5]])
```

See also:

`sympy.matrices.dense.DenseMatrix.__mul__` (page 624)

`sympy.matrices.dense.zeros(r, c=None, cls=None)`

Returns a matrix of zeros with r rows and c columns; if c is omitted a square matrix will be returned.

See also:

`sympy.matrices.dense.ones` (page 616), `sympy.matrices.dense.eye` (page 616),  
`sympy.matrices.dense.diag` (page 616)

`sympy.matrices.dense.ones(r, c=None)`

Returns a matrix of ones with r rows and c columns; if c is omitted a square matrix will be returned.

See also:

`sympy.matrices.dense.zeros` (page 616), `sympy.matrices.dense.eye` (page 616),  
`sympy.matrices.dense.diag` (page 616)

`sympy.matrices.dense.eye(n, cls=None)`

Create square identity matrix n x n

See also:

`sympy.matrices.dense.diag` (page 616), `sympy.matrices.dense.zeros` (page 616),  
`sympy.matrices.dense.ones` (page 616)

`sympy.matrices.dense.diag(*values, **kwargs)`

Create a sparse, diagonal matrix from a list of diagonal values.

See also:

`sympy.matrices.dense.eye` (page 616)

## Notes

When arguments are matrices they are fitted in resultant matrix.

The returned matrix is a mutable, dense matrix. To make it a different type, send the desired class for keyword `cls`.

## Examples

```
>>> from sympy.matrices import diag, Matrix, ones
>>> diag(1, 2, 3)
Matrix([
[1, 0, 0],
```

```
[0, 2, 0],
[0, 0, 3])
>>> diag(*[1, 2, 3])
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The diagonal elements can be matrices; diagonal filling will continue on the diagonal from the last element of the matrix:

```
>>> from sympy.abc import x, y, z
>>> a = Matrix([x, y, z])
>>> b = Matrix([[1, 2], [3, 4]])
>>> c = Matrix([[5, 6]])
>>> diag(a, 7, b, c)
Matrix([
[x, 0, 0, 0, 0, 0, 0],
[y, 0, 0, 0, 0, 0, 0],
[z, 0, 0, 0, 0, 0, 0],
[0, 7, 0, 0, 0, 0, 0],
[0, 0, 1, 2, 0, 0, 0],
[0, 0, 3, 4, 0, 0, 0],
[0, 0, 0, 0, 5, 6]])
```

When diagonal elements are lists, they will be treated as arguments to Matrix:

```
>>> diag([1, 2, 3], 4)
Matrix([
[1, 0],
[2, 0],
[3, 0],
[0, 4]])
>>> diag([[1, 2, 3]], 4)
Matrix([
[1, 2, 3, 0],
[0, 0, 0, 4]])
```

A given band off the diagonal can be made by padding with a vertical or horizontal “kerning” vector:

```
>>> hpad = ones(0, 2)
>>> vpad = ones(2, 0)
>>> diag(vpad, 1, 2, 3, hpad) + diag(hpad, 4, 5, 6, vpad)
Matrix([
[0, 0, 4, 0, 0],
[0, 0, 0, 5, 0],
[1, 0, 0, 0, 6],
[0, 2, 0, 0, 0],
[0, 0, 3, 0, 0]])
```

The type is mutable by default but can be made immutable by setting the `mutable` flag to False:

```
>>> type(diag(1))
<class 'sympy.matrices.dense.MutableDenseMatrix'>
>>> from sympy.matrices import ImmutableMatrix
>>> type(diag(1, cls=ImmutableMatrix))
<class 'sympy.matrices.immutable.ImmutableMatrix'>
```

```
sympy.matrices.dense.jordan_cell(eigenval, n)
```

Create matrix of Jordan cell kind:

### Examples

```
>>> from sympy.matrices import jordan_cell
>>> from sympy.abc import x
>>> jordan_cell(x, 4)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

```
sympy.matrices.dense.hessian(f, varlist, constraints=[])

```

Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector. A list of constraints may optionally be given.

See also:

`sympy.matrices.matrices.MatrixBase.jacobian` (page 603), `sympy.matrices.dense.wronskian` (page 619)

### References

[http://en.wikipedia.org/wiki/Hessian\\_matrix](http://en.wikipedia.org/wiki/Hessian_matrix)

### Examples

```
>>> from sympy import Function, hessian, pprint
>>> from sympy.abc import x, y
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
[          d           d      ]
[   0       0  --(g(x, y))  --(g(x, y))  ]
[           dx           dy      ]
[           ]           ]
[   0       0       2*x           3      ]
[           ]           ]
[           2           2      ]
[d           d           d      ]
[--(g(x, y))  2*x  ---(f(x, y))  ----- (f(x, y))  ]
[dx           2           dy dx           ]
[           dx           ]           ]
[           ]           ]
[           2           2      ]
[d           d           d      ]
[--(g(x, y))  3  ----- (f(x, y))  --- (f(x, y))  ]
[dy           dy dx           2           ]
[           dy           ]           ]
```

```
sympy.matrices.dense.GramSchmidt(vlist, orthog=False)
```

Apply the Gram-Schmidt process to a set of vectors.

see: [http://en.wikipedia.org/wiki/Gram%20-%20Schmidt\\_process](http://en.wikipedia.org/wiki/Gram%20-%20Schmidt_process)

`sympy.matrices.dense.wronskian(functions, var, method='bareis')`

Compute Wronskian for  $\{f_1, \dots, f_n\}$  of functions

$$W(f_1, \dots, f_n) = \begin{vmatrix} f_1 & f_2 & \dots & f_n \\ f_1' & f_2' & \dots & f_n' \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ (n) & (n) & \dots & (n) \\ D & (f_1) D & (f_2) D & \dots & (f_n) D \end{vmatrix}$$

see: <http://en.wikipedia.org/wiki/Wronskian>

See also:

`sympy.matrices.matrices.MatrixBase.jacobian` (page 603), `sympy.matrices.dense.hessian` (page 618)

`sympy.matrices.dense.casoratian(seqs, n, zero=True)`

Given linear difference operator  $L$  of order ' $k$ ' and homogeneous equation  $Ly = 0$  we want to compute kernel of  $L$ , which is a set of ' $k$ ' sequences:  $a(n), b(n), \dots, z(n)$ .

Solutions of  $L$  are linearly independent iff their Casoratian, denoted as  $C(a, b, \dots, z)$ , do not vanish for  $n = 0$ .

Casoratian is defined by  $k \times k$  determinant:

$$+ a(n) \quad b(n) \quad \dots \quad z(n) \quad + \\ | a(n+1) \quad b(n+1) \quad \dots \quad z(n+1) | \\ | \vdots \quad \vdots \quad \ddots \quad \vdots | \\ | \vdots \quad \vdots \quad \ddots \quad \vdots | \\ + a(n+k-1) \quad b(n+k-1) \quad \dots \quad z(n+k-1) +$$

It proves very useful in `rsolve.hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> from sympy import Symbol, casoratian, factorial
>>> n = Symbol('n', integer=True)
```

Exponential and factorial are linearly independent:

```
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

`sympy.matrices.dense.randMatrix(r, c=None, min=0, max=99, seed=None, symmetric=False, percent=100)`

Create random matrix with dimensions  $r \times c$ . If  $c$  is omitted the matrix will be square. If `symmetric` is True the matrix must be square. If `percent` is less than 100 then only approximately the given percentage of elements will be non-zero.

## Examples

```
>>> from sympy.matrices import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54, 9]
[23, 96, 46]
>>> randMatrix(3, 2)
```

```
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[0, 68, 43]
[0, 68, 0]
[0, 91, 34]
```

## Numpy Utility Functions Reference

`sympy.matrices.dense.list2numpy(l, dtype=<type 'object'>)`

Converts python list of SymPy expressions to a NumPy array.

See also:

`sympy.matrices.dense.matrix2numpy` (page 620)

`sympy.matrices.dense.matrix2numpy(m, dtype=<type 'object'>)`

Converts SymPy's matrix to a NumPy array.

See also:

`sympy.matrices.dense.list2numpy` (page 620)

`sympy.matrices.dense.symarray(prefix, shape)`

Create a numpy ndarray of symbols (as an object array).

The created symbols are named `prefix_i1_i2...`. You should thus provide a non-empty prefix if you want your symbols to be unique for different output arrays, as SymPy symbols with identical names are the same object.

**Parameters** `prefix` : string

A prefix prepended to the name of every symbol.

`shape` : int or tuple

Shape of the created array. If an int, the array is one-dimensional; for more than one dimension the shape must be a tuple.

## Examples

These doctests require numpy.

```
>>> from sympy import symarray
>>> symarray(' ', 3)
[_0 _1 _2]
```

If you want multiple symarrays to contain distinct symbols, you *must* provide unique prefixes:

```
>>> a = symarray(' ', 3)
>>> b = symarray(' ', 3)
>>> a[0] == b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] == b[0]
False
```

Creating symarrays with a prefix:

```
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
  [a_0_1_0 a_0_1_1]
  [a_0_2_0 a_0_2_1]]

 [[a_1_0_0 a_1_0_1]
  [a_1_1_0 a_1_1_1]
  [a_1_2_0 a_1_2_1]]]
```

`sympy.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.

See also:

`sympy.matrices.dense.rot_axis2` ([page 622](#)) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

`sympy.matrices.dense.rot_axis3` ([page 622](#)) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

## Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis1
```

A rotation of  $\pi/3$  (60 degrees):

```
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([
 [1, 0, 0],
 [0, 1/2, sqrt(3)/2],
 [0, -sqrt(3)/2, 1/2]])
```

If we rotate by  $\pi/2$  (90 degrees):

```
>>> rot_axis1(pi/2)
Matrix([
[1, 0, 0],
[0, 0, 1],
[0, -1, 0]])
```

```
sympy.matrices.dense.rot_axis2(theta)
```

Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.

See also:

[sympy.matrices.dense.rot\\_axis1 \(page 621\)](#) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

[sympy.matrices.dense.rot\\_axis3 \(page 622\)](#) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

## Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis2
```

A rotation of  $\pi/3$  (60 degrees):

```
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([
[1/2, 0, -sqrt(3)/2],
[0, 1, 0],
[sqrt(3)/2, 0, 1/2]])
```

If we rotate by  $\pi/2$  (90 degrees):

```
>>> rot_axis2(pi/2)
Matrix([
[0, 0, -1],
[0, 1, 0],
[1, 0, 0]])
```

```
sympy.matrices.dense.rot_axis3(theta)
```

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.

See also:

[sympy.matrices.dense.rot\\_axis1 \(page 621\)](#) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

[sympy.matrices.dense.rot\\_axis2 \(page 622\)](#) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

## Examples

```
>>> from sympy import pi
>>> from sympy.matrices import rot_axis3
```

A rotation of  $\pi/3$  (60 degrees):

```
>>> theta = pi/3
>>> rot_axis3(theta)
Matrix([
[ 1/2, sqrt(3)/2, 0],
[-sqrt(3)/2, 1/2, 0],
[ 0, 0, 1]])
```

If we rotate by  $\pi/2$  (90 degrees):

```
>>> rot_axis3(pi/2)
Matrix([
[ 0, 1, 0],
[-1, 0, 0],
[ 0, 0, 1]])
```

`sympy.matrices.matrices.a2idx(j, n=None)`

Return integer after making positive and validating against n.

### 3.13.2 Dense Matrices

#### Matrix Class Reference

`class sympy.matrices.dense.DenseMatrix`

`--getitem__(key)`

Return portion of self defined by key. If the key involves a slice then a list will be returned (if key is a single slice) or a matrix (if key was a tuple involving a slice).

#### Examples

```
>>> from sympy import Matrix, I
>>> m = Matrix([
... [1, 2 + I],
... [3, 4]])
```

If the key is a tuple that doesn't involve a slice then that element is returned:

```
>>> m[1, 0]
3
```

When a tuple key involves a slice, a matrix is returned. Here, the first column is selected (all rows, column 0):

```
>>> m[:, 0]
Matrix([
[1],
[3]])
```

If the slice is not a tuple then it selects from the underlying list of elements that are arranged in row order and a list is returned if a slice is involved:

```
>>> m[0]
1
```

```
>>> m[::2]
[1, 3]

__mul__(other)
    Return self*other

applyfunc(f)
    Apply a function to each element of the matrix.
```

### Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])

as_immutable()
    Returns an Immutable version of this Matrix

asMutable()
    Returns a mutable version of this matrix
```

### Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.asMutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

`col(j)`  
Elementary column selector.

#### See also:

`sympy.matrices.dense.DenseMatrix.row` (page 625), `sympy.matrices.dense.MutableDenseMatrix.col_op` (page 627), `sympy.matrices.dense.MutableDenseMatrix.col_swap` (page 627), `sympy.matrices.dense.MutableDenseMatrix.col_del` (page 626), `sympy.matrices.matrices.MatrixBase.col_join` (page 589),  
`sympy.matrices.matrices.MatrixBase.col_insert` (page 589)

### Examples

```
>>> from sympy import eye
>>> eye(2).col(0)
Matrix([
```

```
[1],  
[0]])
```

```
equals(other, failing_expression=False)
```

Applies `equals` to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if failing\_expression is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

[sympy.core.expr.Expr.equals](#) (page 85)

### Examples

```
>>> from sympy.matrices import Matrix  
>>> from sympy.abc import x  
>>> from sympy import cos  
>>> A = Matrix([x*(x - 1), 0])  
>>> B = Matrix([x**2 - x, 0])  
>>> A == B  
False  
>>> A.simplify() == B.simplify()  
True  
>>> A.equals(B)  
True  
>>> A.equals(2)  
False
```

```
classmethod eye(n)
```

Return an n x n identity matrix.

```
reshape(rows, cols)
```

Reshape the matrix. Total number of elements must remain the same.

### Examples

```
>>> from sympy import Matrix  
>>> m = Matrix(2, 3, lambda i, j: 1)  
>>> m  
Matrix([  
[1, 1, 1],  
[1, 1, 1]])  
>>> m.reshape(1, 6)  
Matrix([[1, 1, 1, 1, 1, 1]])  
>>> m.reshape(3, 2)  
Matrix([  
[1, 1],  
[1, 1],  
[1, 1]])
```

```
row(i)
```

Elementary row selector.

See also:

```
sympy.matrices.dense.DenseMatrix.col (page 624), sympy.matrices.dense.MutableDenseMatrix.row_op
(page 629), sympy.matrices.dense.MutableDenseMatrix.row_swap
(page 629), sympy.matrices.dense.MutableDenseMatrix.row_del
(page 628), sympy.matrices.matrices.MatrixBase.row_join (page 610),
sympy.matrices.matrices.MatrixBase.row_insert (page 610)
```

### Examples

```
>>> from sympy import eye
>>> eye(2).row(0)
Matrix([[1, 0]])

tolist()
Return the Matrix as a nested Python list.
```

### Examples

```
>>> from sympy import Matrix, ones
>>> m = Matrix(3, 3, range(9))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> ones(3, 0).tolist()
[], [], []]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> ones(0, 3).tolist()
[]
```

### classmethod zeros(*r*, *c=None*)

Return an *r* x *c* matrix of zeros, square if *c* is omitted.

```
class sympy.matrices.dense.MutableDenseMatrix
```

### col\_del(*i*)

Delete the given column.

#### See also:

```
sympy.matrices.dense.DenseMatrix.col (page 624), sympy.matrices.dense.MutableDenseMatrix.row_del
(page 628)
```

### Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.col_del(1)
>>> M
```

```
Matrix([
[1, 0],
[0, 0],
[0, 1]])
```

`col_op(j, f)`

In-place operation on col j using two-arg functor whose args are interpreted as (self[i, j], i).

**See also:**

[sympy.matrices.dense.DenseMatrix.col](#) (page 624), [sympy.matrices.dense.MutableDenseMatrix.row\\_op](#) (page 629)

### Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([
[1, 2, 0],
[0, 1, 0],
[0, 0, 1]])
```

`col_swap(i, j)`

Swap the two given columns of the matrix in-place.

**See also:**

[sympy.matrices.dense.DenseMatrix.col](#) (page 624), [sympy.matrices.dense.MutableDenseMatrix.row\\_swap](#) (page 629)

### Examples

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[1, 0], [1, 0]])
>>> M
Matrix([
[1, 0],
[1, 0]])
>>> M.col_swap(0, 1)
>>> M
Matrix([
[0, 1],
[0, 1]])
```

`copyin_list(key, value)`

Copy in elements from a list.

**Parameters** `key` : slice

The section of this matrix to replace.

`value` : iterable

The iterable to copy values from.

**See also:**

[sympy.matrices.dense.MutableDenseMatrix.copyin\\_matrix](#) (page 628)

## Examples

```
>>> from sympy.matrices import eye
>>> I = eye(3)
>>> I[:2, 0] = [1, 2] # col
>>> I
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
>>> I[1, :2] = [[3, 4]]
>>> I
Matrix([
[1, 0, 0],
[3, 4, 0],
[0, 0, 1]]))

copyin_matrix(key, value)
Copy in values from a matrix into the given bounds.
```

**Parameters** `key` : slice

The section of this matrix to replace.

`value` : Matrix

The matrix to copy values from.

**See also:**

[sympy.matrices.dense.MutableDenseMatrix.copyin\\_list](#) (page 627)

## Examples

```
>>> from sympy.matrices import Matrix, eye
>>> M = Matrix([[0, 1], [2, 3], [4, 5]])
>>> I = eye(3)
>>> I[:3, :2] = M
>>> I
Matrix([
[0, 1, 0],
[2, 3, 0],
[4, 5, 1]])
>>> I[0, 1] = M
>>> I
Matrix([
[0, 0, 1],
[2, 2, 3],
[4, 4, 5]]))
```

`fill(value)`

Fill the matrix with the scalar value.

**See also:**

[sympy.matrices.dense.zeros](#) (page 616), [sympy.matrices.dense.ones](#) (page 616)

`row_del(i)`

Delete the given row.

See also:

`sympy.matrices.dense.DenseMatrix.row` (page 625), `sympy.matrices.dense.MutableDenseMatrix.col_del` (page 626)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.row_del(1)
>>> M
Matrix([
[1, 0, 0],
[0, 0, 1]])
```

`row_op(i, f)`  
In-place operation on row  $i$  using two-arg functor whose args are interpreted as `(self[i, j], j)`.

See also:

`sympy.matrices.dense.DenseMatrix.row` (page 625), `sympy.matrices.dense.MutableDenseMatrix.zip_row_op` (page 630), `sympy.matrices.dense.MutableDenseMatrix.col_op` (page 627)

Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

`row_swap(i, j)`  
Swap the two given rows of the matrix in-place.

See also:

`sympy.matrices.dense.DenseMatrix.row` (page 625), `sympy.matrices.dense.MutableDenseMatrix.col_swap` (page 627)

Examples

```
>>> from sympy.matrices import Matrix
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([
[0, 1],
[1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([
[1, 0],
[0, 1]])
```

```
simplify(ratio=1.7, measure=<function count_ops at 0x7fb9aef6e60>)
```

Applies simplify to the elements of a matrix in place.

This is a shortcut for M.applyfunc(lambda x: simplify(x, ratio, measure))

See also:

[sympy.simplify.simplify](#) (page 916)

```
zip_row_op(i, k, f)
```

In-place operation on row i using two-arg functor whose args are interpreted as (self[i, j], self[k, j]).

See also:

[sympy.matrices.dense.DenseMatrix.row](#) (page 625), [sympy.matrices.dense.MutableDenseMatrix.row\\_op](#) (page 629), [sympy.matrices.dense.MutableDenseMatrix.col\\_op](#) (page 627)

## Examples

```
>>> from sympy.matrices import eye
>>> M = eye(3)
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

## ImmutableMatrix Class Reference

```
class sympy.matrices.immutable.ImmutableMatrix
```

Create an immutable version of a matrix.

## Examples

```
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

C

By-element conjugation.

`adjoint()`

Conjugate transpose or Hermitian conjugation.

`as mutable()`

Returns a mutable version of this matrix

## Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.asMutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

### conjugate()

By-element conjugation.

### equals(other, failing\_expression=False)

Applies `equals` to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if failing\_expression is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

`sympy.core.expr.Expr.equals` (page 85)

## Examples

```
>>> from sympy.matrices import Matrix
>>> from sympy.abc import x
>>> from sympy import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
True
>>> A.equals(2)
False
```

### is\_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

## Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
```

```
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero
```

### 3.13.3 Sparse Matrices

#### SparseMatrix Class Reference

```
class sympy.matrices.sparse.SparseMatrix(*args)
A sparse matrix (a matrix with a large number of zero elements).
```

See also:

[sympy.matrices.dense.DenseMatrix](#) (page 623)

#### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> SparseMatrix(2, 2, range(4))
Matrix([
[0, 1],
[2, 3]])
>>> SparseMatrix(2, 2, {(1, 1): 2})
Matrix([
[0, 0],
[0, 2]])
```

CL

Alternate faster representation

`LDLdecomposition()`

Returns the LDL Decomposition (matrices L and D) of matrix A, such that  $L * D * L.T == A$ . A must be a square, symmetric, positive-definite and non-singular.

This method eliminates the use of square root and ensures that all the diagonal entries of L are 1.

#### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1,  0,  0],
[ 3/5,  1,  0],
[-1/5,  1/3,  1]])
>>> D
```

```
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T == A
True

RL
Alternate faster representation

add(other)
Add two sparse matrices with dictionary representation.

See also:
multiply (page 637)
```

### Examples

```
>>> from sympy.matrices import SparseMatrix, eye, ones
>>> SparseMatrix(eye(3)).add(SparseMatrix(ones(3)))
Matrix([
[2, 1, 1],
[1, 2, 1],
[1, 1, 2]])
>>> SparseMatrix(eye(3)).add(-SparseMatrix(eye(3)))
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

Only the non-zero elements are stored, so the resulting dictionary that is used to represent the sparse matrix is empty:

```
>>> _._smat
{}
```

```
applyfunc(f)
Apply a function to each element of the matrix.
```

### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> m = SparseMatrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])

as_immutable()
Returns an Immutable version of this Matrix.
```

`asMutable()`  
Returns a mutable version of this matrix.

#### Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.asMutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])

cholesky()
Returns the Cholesky decomposition L of a matrix A such that L * L.T = A
A must be a square, symmetric, positive-definite and non-singular matrix
```

#### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> A = SparseMatrix(((25,15,-5),(15,18,0),(-5,0,11)))
>>> A.cholesky()
Matrix([
[ 5,  0,  0],
[ 3,  3,  0],
[-1,  1,  3]])
>>> A.cholesky() * A.cholesky().T == A
True
```

`col(j)`  
Returns column j from self as a column vector.

See also:

`row` (page 637), `col_list` (page 634)

#### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix((1, 2), (3, 4))
>>> a.col(0)
Matrix([
[1],
[1],
[3]])

col_list()
Returns a column-sorted list of non-zero elements of the matrix.
```

See also:

`sympy.matrices.sparse.MutableSparseMatrix.col_op` (page 641), `row_list` (page 638)

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a=SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.CL
[(0, 0, 1), (1, 0, 3), (0, 1, 2), (1, 1, 4)]
```

### extract(rowsList, colsList)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range  $-n \leq i < n$  where  $n$  is the number of rows or columns.

## Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0, 1],
[3, 4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

### classmethod eye(n)

Return an  $n \times n$  identity matrix.

### has(\*patterns)

Test whether any subexpression matches any of the patterns.

## Examples

```
>>> from sympy import SparseMatrix, Float
>>> from sympy.abc import x, y
>>> A = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

#### is\_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element  $i,j$  is the complex conjugate of element  $j,i$ .

#### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> from sympy import I
>>> from sympy.abc import x
>>> a = SparseMatrix([[1, I], [-I, 1]])
>>> a
Matrix([
[1, I],
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

#### is\_symmetric(*simplify=True*)

Return True if self is symmetric.

#### Examples

```
>>> from sympy.matrices import SparseMatrix, eye
>>> M = SparseMatrix(eye(3))
>>> M.is_symmetric()
True
>>> M[0, 2] = 1
>>> M.is_symmetric()
False
```

#### liupc()

Liu's algorithm, for pre-determination of the Elimination Tree of the given matrix, used in row-based symbolic Cholesky factorization.

## References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999)  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.liupc()
([[0], [], [0], [1, 2]], [4, 3, 4, 4])

multiply(other)
Fast multiplication exploiting the sparsity of the matrix.
```

See also:

[add](#) (page 633)

## Examples

```
>>> from sympy.matrices import SparseMatrix, ones
>>> A, B = SparseMatrix(ones(4, 3)), SparseMatrix(ones(3, 4))
>>> A.multiply(B) == 3*ones(4)
True

nnz()
Returns the number of non-zero elements in Matrix.

reshape(rows, cols)
Reshape matrix while retaining original size.
```

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix(4, 2, range(8))
>>> S.reshape(2, 4)
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7]])

row(i)
Returns column i from self as a row vector.

See also:
col (page 634), row_list (page 638)
```

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.row(0)
Matrix([[1, 2]])
```

### row\_list()

Returns a row-sorted list of non-zero elements of the matrix.

#### See also:

`sympy.matrices.sparse.MutableSparseMatrix.row_op` (page 643), `col_list` (page 634)

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.RL
[(0, 0, 1), (0, 1, 2), (1, 0, 3), (1, 1, 4)]
```

### row\_structure\_symbolic\_cholesky()

Symbolic cholesky factorization, for pre-determination of the non-zero structure of the Cholesky factorization.

## References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999)  
<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.row_structure_symbolic_cholesky()
[[0], [], [0], [1, 2]]
```

### scalar\_multiply(scalar)

Scalar element-wise multiplication

### solve(rhs, method='LDL')

Return solution to self\*soln = rhs using given inversion method.

For a list of possible inversion methods, see the .inv() docstring.

### solve\_least\_squares(rhs, method='LDL')

Return the least-square fit to the data.

By default the cholesky\_solve routine is used (method='CH'); other methods of matrix inversion can be used. To find out which are available, see the docstring of the .inv() method.

### Examples

```
>>> from sympy.matrices import SparseMatrix, Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = SparseMatrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of S represent coefficients of Ax + By and x and y are [2, 3] then S\*xy is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by S\*xy - r:

```
>>> S*xy - r
Matrix([
[1/3],
[1/3],
[1/3]])
>>> _ .norm() .n(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm() .n(2)
1.5
```

`tolist()`

Convert this sparse matrix into a list of nested Python lists.

### Examples

```
>>> from sympy.matrices import SparseMatrix, ones
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.tolist()
[[1, 2], [3, 4]]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> SparseMatrix(ones(0, 3)).tolist()
[]
```

### classmethod zeros(*r*, *c=None*)

Return an *r* x *c* matrix of zeros, square if *c* is omitted.

```
class sympy.matrices.sparse.MutableSparseMatrix(*args)
```

### col\_del(*k*)

Delete the given column of the matrix.

**See also:**

[row\\_del](#) (page 642)

### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([
[0, 0],
[0, 1]])
>>> M.col_del(0)
>>> M
Matrix([
[0],
[1]])
```

### col\_join(*other*)

Returns *B* augmented beneath *A* (row-wise joining):

```
[A]
[B]
```

### Examples

```
>>> from sympy import SparseMatrix, Matrix, ones
>>> A = SparseMatrix(ones(3))
>>> A
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
>>> B = SparseMatrix.eye(3)
>>> B
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.col_join(B); C
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
```

```
[1, 1, 1],  
[1, 1, 1],  
[1, 0, 0],  
[0, 1, 0],  
[0, 0, 1])  
>>> C == A.col_join(Matrix(B))  
True
```

Joining along columns is the same as appending rows at the end of the matrix:

```
>>> C == A.row_insert(A.rows, Matrix(B))  
True
```

### col\_op(*j, f*)

In-place operation on col *j* using two-arg functor whose args are interpreted as (self[i, *j*], *i*) for *i* in range(self.rows).

### Examples

```
>>> from sympy.matrices import SparseMatrix  
>>> M = SparseMatrix.eye(3)*2  
>>> M[1, 0] = -1  
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M  
Matrix([  
    [2, 4, 0],  
    [-1, 0, 0],  
    [0, 0, 2]])
```

### col\_swap(*i, j*)

Swap, in place, columns *i* and *j*.

### Examples

```
>>> from sympy.matrices import SparseMatrix  
>>> S = SparseMatrix.eye(3); S[2, 1] = 2  
>>> S.col_swap(1, 0); S  
Matrix([  
    [0, 1, 0],  
    [1, 0, 0],  
    [2, 0, 1]])
```

### fill(*value*)

Fill self with the given value.

### Notes

Unless many values are going to be deleted (i.e. set to zero) this will create a matrix that is slower than a dense matrix in operations.

### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.zeros(3); M
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
>>> M.fill(1); M
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
```

### row\_del(*k*)

Delete the given row of the matrix.

See also:

[col\\_del](#) (page 640)

### Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([
[0, 0],
[0, 1]])
>>> M.row_del(0)
>>> M
Matrix([[0, 1]])
```

### row\_join(*other*)

Returns B appended after A (column-wise augmenting):

[A B]

### Examples

```
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix(((1, 0, 1), (0, 1, 0), (1, 1, 0)))
>>> A
Matrix([
[1, 0, 1],
[0, 1, 0],
[1, 1, 0]])
>>> B = SparseMatrix(((1, 0, 0), (0, 1, 0), (0, 0, 1)))
>>> B
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.row_join(B); C
Matrix([
[1, 0, 1, 0, 1, 0],
[0, 1, 0, 0, 1, 0],
[1, 1, 0, 0, 0, 1]])
```

```
>>> C == A.row_join(Matrix(B))
True
```

Joining at row ends is the same as appending columns at the end of the matrix:

```
>>> C == A.col_insert(A.cols, B)
True
```

`row_op(i, f)`

In-place operation on row  $i$  using two-arg functor whose args are interpreted as `(self[i, j], j)`.

**See also:**

[sympy.matrices.sparse.SparseMatrix.row](#) (page 637), [zip\\_row\\_op](#) (page 643), [col\\_op](#) (page 641)

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[2, -1, 0],
[4, 0, 0],
[0, 0, 2]])
```

`row_swap(i, j)`

Swap, in place, columns  $i$  and  $j$ .

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.row_swap(1, 0); S
Matrix([
[0, 1, 0],
[1, 0, 0],
[0, 2, 1]])
```

`zip_row_op(i, k, f)`

In-place operation on row  $i$  using two-arg functor whose args are interpreted as `(self[i, j], self[k, j])`.

**See also:**

[sympy.matrices.sparse.SparseMatrix.row](#) (page 637), [row\\_op](#) (page 643), [col\\_op](#) (page 641)

## Examples

```
>>> from sympy.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
```

```
Matrix([
[2, -1, 0],
[4, 0, 0],
[0, 0, 2]])
```

## ImmutableSparseMatrix Class Reference

```
class sympy.matrices.immutable.ImmutableSparseMatrix(*args)
Create an immutable version of a sparse matrix.
```

### Examples

```
>>> from sympy import eye
>>> from sympy.matrices.immutable import ImmutableSparseMatrix
>>> ImmutableSparseMatrix(1, 1, {})
Matrix([[0]])
>>> ImmutableSparseMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableSparseMatrix
>>> _.shape
(3, 3)

subs(*args, **kwargs)
Return a new matrix with subs applied to each entry.
```

### Examples

```
>>> from sympy.abc import x, y
>>> from sympy.matrices import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_.subs(y, x))
Matrix([[x]])
```

## 3.13.4 Immutable Matrices

The standard `Matrix` class in SymPy is mutable. This is important for performance reasons but means that standard matrices can not interact well with the rest of SymPy. This is because the `Basic` (page 62) object, from which most SymPy classes inherit, is immutable.

The mission of the `ImmutableMatrix` (page 645) class is to bridge the tension between performance/mutability and safety/immutability. Immutable matrices can do almost everything that normal matrices can do but they inherit from `Basic` (page 62) and can thus interact more naturally with the rest

of SymPy. `ImmutableMatrix` (page 645) also inherits from `MatrixExpr` (page 647), allowing it to interact freely with SymPy's Matrix Expression module.

You can turn any Matrix-like object into an `ImmutableMatrix` (page 645) by calling the constructor

```
>>> from sympy import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([
[1, 2, 3],
[4, 0, 6],
[7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```

## ImmutableMatrix Class Reference

```
class sympy.matrices.immutable.ImmutableMatrix
Create an immutable version of a matrix.
```

### Examples

```
>>> from sympy import eye
>>> from sympy.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

#### C

By-element conjugation.

#### adjoint()

Conjugate transpose or Hermitian conjugation.

#### asMutable()

Returns a mutable version of this matrix

### Examples

```
>>> from sympy import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.asMutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
```

```
[1, 2],  
[3, 5]])
```

### conjugate()

By-element conjugation.

### equals(*other*, *failing\_expression=False*)

Applies `equals` to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if *failing\_expression* is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

`sympy.core.expr.Expr.equals` (page 85)

### Examples

```
>>> from sympy.matrices import Matrix  
>>> from sympy.abc import x  
>>> from sympy import cos  
>>> A = Matrix([x*(x - 1), 0])  
>>> B = Matrix([x**2 - x, 0])  
>>> A == B  
False  
>>> A.simplify() == B.simplify()  
True  
>>> A.equals(B)  
True  
>>> A.equals(2)  
False
```

### is\_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

### Examples

```
>>> from sympy import Matrix, zeros  
>>> from sympy.abc import x  
>>> a = Matrix([[0, 0], [0, 0]])  
>>> b = zeros(3, 4)  
>>> c = Matrix([[0, 1], [0, 0]])  
>>> d = Matrix([])  
>>> e = Matrix([[x, 0], [0, 0]])  
>>> a.is_zero  
True  
>>> b.is_zero  
True  
>>> c.is_zero  
False  
>>> d.is_zero
```

```
True
>>> e.is_zero
```

### 3.13.5 Matrix Expressions

The Matrix expression module allows users to write down statements like

```
>>> from sympy import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
X^-1*X'^-1*Y

>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]])
```

```
>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]
```

where X and Y are `MatrixSymbol` (page 648)'s rather than scalar symbols.

#### Matrix Expressions Core Reference

```
class sympy.matrices.expressions.MatrixExpr
    Superclass for Matrix Expressions
```

`MatrixExprs` represent abstract matrices, linear transformations represented within a particular basis.

#### Examples

```
>>> from sympy import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y
```

`T`

Matrix transposition.

`as_explicit()`

Returns a dense Matrix with elements represented explicitly

Returns an object of type `ImmutableMatrix`.

`See also:`

`as mutable` (page 648) returns mutable Matrix type

#### Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

#### asMutable()

Returns a dense, mutable matrix with elements represented explicitly

See also:

[as\\_explicit](#) (page 647) returns ImmutableMatrix

#### Examples

```
>>> from sympy import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as Mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

#### equals(*other*)

Test elementwise equality between matrices, potentially of different types

```
>>> from sympy import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

#### class sympy.matrices.expressions.MatrixSymbol

Symbolic representation of a Matrix object

Creates a SymPy Symbol to represent a Matrix. This matrix has a shape and can be included in Matrix Expressions

```
>>> from sympy import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

#### class sympy.matrices.expressions.MatAdd

A Sum of Matrix Expressions

MatAdd inherits from and operates like SymPy Add

```
>>> from sympy import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

```
class sympy.matrices.expressions.MatMul
    A product of matrix expressions
```

### Examples

```
>>> from sympy import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

```
class sympy.matrices.expressions.MatPow
```

```
class sympy.matrices.expressions.Inverse
```

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the `.inverse()` method of matrices.

### Examples

```
>>> from sympy import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A^-1
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B^-1*A^-1
>>> Inverse(A*B)
(A*B)^-1
```

```
class sympy.matrices.expressions.Transpose
```

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the `transpose()` function, or the `.T` attribute of matrices.

### Examples

```
>>> from sympy.matrices import MatrixSymbol, Transpose
>>> from sympy.functions import transpose
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A'
```

```
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B)',
```

```
>>> transpose(A*B)
```

```
B'*A'
```

```
class sympy.matrices.expressions.Trace
Matrix Trace
```

Represents the trace of a matrix expression.

```
>>> from sympy import MatrixSymbol, Trace, eye
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
```

**See Also:** trace

```
class sympy.matrices.expressions.FunctionMatrix
Represents a Matrix using a function (Lambda)
```

This class is an alternative to SparseMatrix

```
>>> from sympy import FunctionMatrix, symbols, Lambda, MatMul, Matrix
>>> i, j = symbols('i,j')
>>> X = FunctionMatrix(3, 3, Lambda((i, j), i + j))
>>> Matrix(X)
Matrix([
[0, 1, 2],
[1, 2, 3],
[2, 3, 4]]))

>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))

>>> isinstance(Y*Y, MatMul) # this is an expression object
True

>>> (Y**2)[10,10] # So this is evaluated lazily
342923500
```

```
class sympy.matrices.expressions.Identity
The Matrix Identity I - multiplicative identity
```

```
>>> from sympy.matrices import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

```
class sympy.matrices.expressions.ZeroMatrix
The Matrix Zero 0 - additive identity
```

```
>>> from sympy import MatrixSymbol, ZeroMatrix
>>> A = MatrixSymbol('A', 3, 5)
>>> Z = ZeroMatrix(3, 5)
>>> A+Z
A
>>> Z*A.T
0
```

## Block Matrices

Block matrices allow you to construct larger matrices out of smaller sub-blocks. They can work with [MatrixExpr](#) (page 647) or [ImmutableMatrix](#) (page 645) objects.

```
class sympy.matrices.expressions.blockmatrix.BlockMatrix
A BlockMatrix is a Matrix composed of other smaller, submatrices
```

The submatrices are stored in a SymPy Matrix object but accessed as part of a Matrix Expression

```
>>> from sympy import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])

>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])

>>> print(block_collapse(C*B))
Matrix([[X, Z*Y + Z]])
```

`transpose()`

Return transpose of matrix.

### Examples

```
>>> from sympy import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from sympy.abc import l, m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([
[X, 0],
[Z, Y']])
>>> _.transpose()
Matrix([
[X, Z],
[0, Y]])
```

`class sympy.matrices.expressions.blockmatrix.BlockDiagMatrix`

A BlockDiagMatrix is a BlockMatrix with matrices only along the diagonal

```
>>> from sympy import MatrixSymbol, BlockDiagMatrix, symbols, Identity
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> BlockDiagMatrix(X, Y)
Matrix([
```

```
[X, 0],  
[0, Y])  
  
sympy.matrices.expressions.blockmatrix.block_collapse(expr)  
Evaluates a block matrix expression  
  
>>> from sympy import MatrixSymbol, BlockMatrix, symbols,  
>>> n,m,l = symbols('n m l')  
>>> X = MatrixSymbol('X', n, n)  
>>> Y = MatrixSymbol('Y', m ,m)  
>>> Z = MatrixSymbol('Z', n, m)  
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])  
>>> print(B)  
Matrix([  
    [X, Z],  
    [0, Y]])  
  
>>> C = BlockMatrix([[Identity(n), Z]])  
>>> print(C)  
Matrix([[I, Z]])  
  
>>> print(block_collapse(C*B))  
Matrix([[X, Z*Y + Z]])
```

## 3.14 Polynomials Manipulation Module

Computations with polynomials are at the core of computer algebra and having a fast and robust polynomials manipulation module is a key for building a powerful symbolic manipulation system. SymPy has a dedicated module `sympy.polys` (page 652) for computing in polynomial algebras over various coefficient domains.

There is a vast number of methods implemented, ranging from simple tools like polynomial division, to advanced concepts including Grbner bases and multivariate factorization over algebraic number domains.

### 3.14.1 Contents

#### Basic functionality of the module

Polynomial manipulation algorithms and algebraic objects.

#### Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within SymPy. All code examples assume:

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

#### Basic functionality

These functions provide different algorithms dealing with polynomials in the form of SymPy expression, like symbols, sums etc.

**Division** The function `div()` (page 665) provides division of polynomials with remainder. That is, for polynomials  $f$  and  $g$ , it computes  $q$  and  $r$ , such that  $f = g \cdot q + r$  and  $\deg(r) < \deg(g)$ . For polynomials in one variables with coefficients in a field, say, the rational numbers,  $q$  and  $r$  are uniquely defined this way:

```
>>> f = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x - 2
2
>>> r
2
5*x + 10*x + 3
```

As you can see,  $q$  has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```
>>> q, r = div(f, g, domain='ZZ')
>>> q
0
>>> r
2
5*x + 10*x + 3
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of  $f$ . Since 2 doesn't divide 5,  $2x$  doesn't divide  $5x^2$ , even if the degree is smaller. But:

```
>>> g = 5*x + 1

>>> q, r = div(f, g, domain='ZZ')
>>> q
x
>>> r
9*x + 3
>>> (q*g + r).expand()
2
5*x + 10*x + 3
```

This also works for polynomials with multiple variables:

```
>>> f = x*y + y*z
>>> g = 3*x + 3*z

>>> q, r = div(f, g, domain='QQ')
>>> q
y
-
3
>>> r
0
```

In the last examples, all of the three variables  $x$ ,  $y$  and  $z$  are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```
>>> a, b, c = symbols('a,b,c')
>>> f = a*x**2 + b*x + c
```

```
>>> g = 3*x + 2
>>> q, r = div(f, g, domain='QQ')
>>> q
a*x - 2*a/b
--- - --- + -
3         9      3

>>> r
4*a - 2*b
--- - --- + c
9         3
```

**GCD and LCM** With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents' gcd is also considered:

```
>>> f = (12*x + 12)*x
>>> g = 16*x**2
>>> gcd(f, g)
4*x
```

But if the polynomials have rational coefficients, then the returned polynomial is monic:

```
>>> f = 3*x**2/2
>>> g = 9*x/4
>>> gcd(f, g)
x
```

It also works with multiple variables. In this case, the variables are ordered alphabetically, by default, which has influence on the leading coefficient:

```
>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y

>>> gcd(f, g)
x + 2*y
```

The lcm is connected with the gcd and one can be computed using the other:

```
>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
3 2    2 3
x *y + x *y
>>> (f*g).expand()
4 3    3 4
x *y + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
4 3    3 4
x *y + x *y
```

**Square-free factorization** The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5
>>> sqf_list(f)
(1, [(x + 2, 1), (x, 2), (x + 1, 2)])
>>> sqf(f)

$$x^5 + 4x^4 + 5x^3 + 2x^2$$

```

**Factorization** This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)

$$\frac{x^4 + 5x^3 - x^2}{12}$$

-----
```

```
>>> factor(x**2 + 4*x*y + 4*y**2)

$$(x + 2y)^2$$

```

**Groebner bases** Buchberger's algorithm is implemented, supporting various monomial orders:

```
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')

$$\text{GroebnerBasis}\left[\left[x^2 + 1, y^4x + x^3\right], \{x, y\}, \text{order}=lex\right]$$

>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')

$$\text{GroebnerBasis}\left[\left[y^4x + x^3 + x^2y^3z^3 + 1, x^2 + 1\right], \{x, y, z\}, \text{order}=grevlex\right]$$

```

**Solving Equations** We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```
>>> from sympy import roots, solve_poly_system
```

```
>>> solve(x**3 + 2*x + 3, x)

$$\left[-1, -\frac{1 + \sqrt{-11}i}{2}, -\frac{1 - \sqrt{-11}i}{2}\right]$$

>>> p = Symbol('p')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)

$$\left[-\frac{p + \sqrt{p^2 - 4q}}{2}, -\frac{p - \sqrt{p^2 - 4q}}{2}\right]$$

>>> solve_poly_system([y - x, x - 5], x, y)
```

$[(5, 5)]$

```
>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)
[(0, -I), (0, I), (1, 0), (- - - -----, 0), (- - + -----, 0)]
              1   \ / 3 *I      1   \ / 3 *I
              2       2      2       2
```

## Examples from Wester's Article

## Introduction

In this tutorial we present examples from Wester's article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999] (page 1245)). All the examples are related to polynomial and algebraic computations and SymPy specific remarks were added to all of them.

## Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```
>>> from sympy import *
>>> init_printing(use_unicode=True, wrap_line=False, no_global=True)
>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)
```

**Simple univariate polynomial factorization** To obtain a factorization of a polynomial use `factor()` (page 675) function. By default `factor()` (page 675) returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```
>>> factor(6*x - 10)  
2(3x - 5)
```

To achieve the same effect in a more systematic way use `primitive()` (page 672) function, which returns the content and the primitive part of the input polynomial:

```
>>> primitive(6*x - 10)
(2, 3x - 5)
```

**Note:** The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use `monic()` (page 672).

**Univariate GCD, resultant and factorization** Consider univariate polynomials  $f$ ,  $g$  and  $h$  over integers:

```
>>> f = 64*x**34 - 21*x**47 - 126*x**8 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using `gcd()` (page 671) function:

```
>>> gcd(f, g)
1
```

We see that  $f$  and  $g$  have no common factors. However,  $f*h$  and  $g*h$  have an obvious factor  $h$ :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```
>>> factor(expand(f*g))
 60      47      34      8      5      60      52      39      25      23      10
-16x    + 21x    - 64x    + 126x   + 46x   + 8172x   - 83x   - 22x   - 25x   - 19x   + 54x   + 81
```

**Multivariate GCD and factorization** What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials  $f$ ,  $g$  and  $h$  in  $\mathbb{Z}[x, y, z]$ :

```
>>> f = 24*x*y**19*z**8 - 47*x**17*y**5*z**8 + 6*x**15*y**9*z**2 - 3*x**22 + 5
>>> g = 34*x**5*y**8*z**13 + 20*x**7*y**7*z**7 + 12*x**9*y**16*z**4 + 80*y**14*z
>>> h = 11*x**12*y**7*z**13 - 23*x**2*y**8*z**10 + 47*x**17*y**5*z**8
```

As previously, we can verify that  $f$  and  $g$  have no common factors:

```
>>> gcd(f, g)
1
```

However,  $f*h$  and  $g*h$  have an obvious factor  $h$ :

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

Multivariate factorization of large polynomials is also possible:

```
>>> factor(expand(f*g))
 7      9      9      3      7      6      5      12      7      22      17      5      8      15      9      2      19      8
-2y z6x y z + 10x z + 17x yz + 40y 3x + 47x y z - 6x y z - 24xy z - 5
```

**Support for symbols in exponents** Polynomial manipulation functions provided by `sympy.polys` (page 652) are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

```
>>> gcd(2*x**(n + 4) - x**(n + 2), 4*x**(n + 1) + 3*x**n)
n
x
```

**Testing if polynomials have common zeros** To test if two polynomials have a root in common we can use `resultant()` (page 667) function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

```
>>> resultant(3*x**4 + 3*x**3 + x**2 - x - 2, x**3 - 3*x**2 + x + 5)
0
```

We can visualize this fact by factoring the polynomials:

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

>>> factor(x**3 - 3*x**2 + x + 5)
      2
(x + 1)x  - 4x + 5
```

In both cases we obtained the factor  $x + 1$  which tells us that the common root is  $x = -1$ .

**Normalizing simple rational functions** To remove common factors from the numerator and the denominator of a rational function the elegant way, use [cancel\(\)](#) (page 678) function. For example:

```
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
x - 2
-----
x + 2
```

**Expanding expressions and factoring back** One can work easily with expressions in both expanded and factored forms. Consider a polynomial  $f$  in expanded form. We differentiate it and factor the result back:

```
>>> f = expand((x + 1)**20)

>>> g = diff(f, x)

>>> factor(g)
      19
20(x + 1)
```

The same can be achieved in factored form:

```
>>> diff((x + 1)**20, x)
      19
20(x + 1)
```

**Factoring in terms of cyclotomic polynomials** SymPy can very efficiently decompose polynomials of the form  $x^n \pm 1$  in terms of cyclotomic polynomials:

```
>>> factor(x**15 - 1)
      2      4      3      2      8      7      5      4      3
(x - 1)x  + x + 1x  + x + x + x + 1x  - x + x - x + x - x + 1
```

The original Wester's example was  $x^{100} - 1$ , but was truncated for readability purpose. Note that this is not a big struggle for [factor\(\)](#) (page 675) to decompose polynomials of degree 1000 or greater.

**Univariate factoring over Gaussian numbers** Consider a univariate polynomial  $f$  with integer coefficients:

```
>>> f = 4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153
```

We want to obtain a factorization of  $f$  over Gaussian numbers. To do this we use [factor\(\)](#) (page 675) as previously, but this time we set `gaussian` keyword to `True`:

```
>>> factor(f, gaussian=True)
      3i      3i
4x - ---x + ---(x + 1 - 4i)(x + 1 + 4i)
      2          2
```

As the result we got a splitting factorization of  $f$  with monic factors (this is a general rule when computing in a field with SymPy). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

```
>>> factor(f, extension=I)
      3i      3i
4x - ---x + ---(x + 1 - 4i)(x + 1 + 4i)
      2          2
```

**Computing with automatic field extensions** Consider two univariate polynomials  $f$  and  $g$ :

```
>>> f = x**3 + (sqrt(2) - 2)*x**2 - (2*sqrt(2) + 3)*x - 3*sqrt(2)
>>> g = x**2 - 2
```

We would like to reduce degrees of the numerator and the denominator of a rational function  $f/g$ . Do do this we employ `cancel()` (page 678) function:

```
>>> cancel(f/g)
      3      2      --- 2
x  - 2x  + \ 2 x  - 3x - 2\ 2 x - 3\ 2
-----
      2
x  - 2
```

Unfortunately nothing interesting happened. This is because by default SymPy treats  $\sqrt{2}$  as a generator, obtaining a bivariate polynomial for the numerator. To make `cancel()` (page 678) recognize algebraic properties of  $\sqrt{2}$ , one needs to use `extension` keyword:

```
>>> cancel(f/g, extension=True)
      2
x  - 2x - 3
-----
      2
x - \ 2
```

Setting `extension=True` tells `cancel()` (page 678) to find minimal algebraic number domain for the coefficients of  $f/g$ . The automatically inferred domain is  $\mathbb{Q}(\sqrt{2})$ . If one doesn't want to rely on automatic inference, the same result can be obtained by setting the `extension` keyword with an explicit algebraic number:

```
>>> cancel(f/g, extension=sqrt(2))
      2
x  - 2x - 3
-----
      2
x - \ 2
```

**Univariate factoring over various domains** Consider a univariate polynomial  $f$  with integer coefficients:

```
>>> f = x**4 - 3*x**2 + 1
```

With `sympy.polys` (page 652) we can obtain factorizations of `f` over different domains, which includes:

- rationals:

```
>>> factor(f)
      2           2
x  - x - 1x  + x - 1
```

- finite fields:

```
>>> factor(f, modulus=5)
      2           2
(x - 2) (x + 2)
```

- algebraic numbers:

```
>>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')
>>> factor(f, extension=alg)
(x - alpha)(x + alpha)(x - 1 - alpha)(x + alpha + 1)
```

**Factoring polynomials into linear factors** Currently SymPy can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the latter only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using `solve()` (page 1045) function.

**Advanced factoring over finite fields** Consider a univariate polynomial `f` with integer coefficients:

```
>>> f = x**11 + x + 1
```

We can factor `f` over a large finite field  $F_{65537}$ :

```
>>> factor(f, modulus=65537)
      2           9           8           6           5           3           2
x  + x + 1x  - x + x - x + x - x + 1
```

and expand the resulting factorization back:

```
>>> expand(_)
11
x  + x + 1
```

obtaining polynomial `f`. This was done using symmetric polynomial representation over finite fields. The same thing can be done using non-symmetric representation:

```
>>> factor(f, modulus=65537, symmetric=False)
      2           9           8           6           5           3           2
x  + x + 1x  + 65536x  + x + 65536x  + x + 65536x  + 1
```

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

```
>>> trunc(expand(_), 65537)
11
x + x + 1
```

**Working with expressions as polynomials** Consider a multivariate polynomial  $f$  in  $\mathbb{Z}[x, y, z]$ :

```
>>> f = expand((x - 2*y**2 + 3*z**3)**20)
```

We want to compute factorization of  $f$ . To do this we use `factor` as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding  $f$ :

```
>>> factor(f, expand=False)
20
      2      3
x - 2y + 3z
```

The default in `sympy.polys` (page 652) is to expand all expressions given as arguments to polynomial manipulation functions and `Poly` (page 679) class. If we know that expanding is unnecessary, then by setting `expand=False` we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:

```
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)

>>> factor(g, expand=False)
20
      2      3
-sin(x) + 2cos (y) - 3tan (z)
```

**Computing reduced Grbner bases** To compute a reduced Grbner basis for a set of polynomials use `groebner()` (page 678) function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The `lex` ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to `groebner()` (page 678) is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)

>>> groebner([f, c**2 + s**2 - 1])
      2      2      20      18      16      14      12      10
GroebnerBasis[c + s - 1, c - 5c + 10c - 10c + 5c - c , s, c, domain=, order=lex]
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```
>>> list(map(factor, _))
      2      2      10      5      5
c + s - 1, c (c - 1) (c + 1)
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use `solve()` (page 1045) to achieve this in a more systematic way:

```
>>> solve([f, s**2 + c**2 - 1], c, s)
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

**Multivariate factoring over algebraic numbers** Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over  $\mathbb{Q}(\sqrt{-3})$ :

```
>>> factor(x**3 + y**3, extension=sqrt(-3))
      1   \ 3 i      1   \ 3 i
(x + y)x + y - - -----x + y - - + -----
```

---

**Note:** Currently multivariate polynomials over finite fields aren't supported.

---

**Partial fraction decomposition** Consider a univariate rational function  $f$  with integer coefficients:

```
>>> f = (x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2)
```

To decompose  $f$  into partial fractions use `apart()` (page 725) function:

```
>>> apart(f)
      3      2      2
----- - ----- + -----
x + 2    x + 1      2
                  (x + 1)
```

To return from partial fractions to the rational function use a composition of `together()` (page 724) and `cancel()` (page 678):

```
>>> cancel(together(_))
      2
x  + 2x + 3
-----
      3      2
x  + 4x  + 5x + 2
```

## Literature

## Polynomials Manipulation Module Reference

### Basic polynomial manipulation functions

`sympy.polys.polytools.poly(expr, *gens, **args)`  
Efficiently transform an expression into a polynomial.

### Examples

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

>>> poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

```
sympy.polys.polytools.poly_from_expr(expr, *gens, **args)
    Construct a polynomial from an expression.

sympy.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)
    Construct polynomials from expressions.

sympy.polys.polytools.degree(f, *gens, **args)
    Return the degree of f in the given variable.
```

The degree of 0 is negative infinity.

### Examples

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

>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-oo
```

```
sympy.polys.polytools.degree_list(f, *gens, **args)
    Return a list of degrees of f in all variables.
```

### Examples

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

>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

```
sympy.polys.polytools.LC(f, *gens, **args)
    Return the leading coefficient of f.
```

### Examples

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

>>> LC(4*x**2 + 2*x*y**2 + x*y + 3*y)
4
```

```
sympy.polys.polytools.LM(f, *gens, **args)
    Return the leading monomial of f.
```

### Examples

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

```
>>> LM(4*x**2 + 2*x*y**2 + x*y + 3*y)
x**2
```

```
sympy.polys.polytools.LT(f, *gens, **args)
Return the leading term of f.
```

### Examples

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

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

```
sympy.polys.polytools.pdiv(f, g, *gens, **args)
Compute polynomial pseudo-division of f and g.
```

### Examples

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

```
>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

```
sympy.polys.polytools.prem(f, g, *gens, **args)
Compute polynomial pseudo-remainder of f and g.
```

### Examples

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

```
>>> prem(x**2 + 1, 2*x - 4)
20
```

```
sympy.polys.polytools.pquo(f, g, *gens, **args)
Compute polynomial pseudo-quotient of f and g.
```

### Examples

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

```
>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

```
sympy.polys.polytools.pexquo(f, g, *gens, **args)
Compute polynomial exact pseudo-quotient of f and g.
```

## Examples

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

>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2

>>> pexquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

sympy.polys.polytools.div(f, g, \*gens, \*\*args)  
Compute polynomial division of f and g.

## Examples

```
>>> from sympy import div, ZZ, QQ
>>> from sympy.abc import x

>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)

sympy.polys.polytools.rem(f, g, *gens, **args)  
Compute polynomial remainder of f and g.
```

## Examples

```
>>> from sympy import rem, ZZ, QQ
>>> from sympy.abc import x

>>> rem(x**2 + 1, 2*x - 4, domain=ZZ)
x**2 + 1
>>> rem(x**2 + 1, 2*x - 4, domain=QQ)
5

sympy.polys.polytools.quo(f, g, *gens, **args)  
Compute polynomial quotient of f and g.
```

## Examples

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

>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1

sympy.polys.polytools.exquo(f, g, *gens, **args)  
Compute polynomial exact quotient of f and g.
```

## Examples

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

>>> exquo(x**2 - 1, x - 1)
x + 1

>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`sympy.polys.polytools.half_gcdex(f, g, *gens, **args)`  
Half extended Euclidean algorithm of f and g.  
Returns (s, h) such that  $h = \gcd(f, g)$  and  $s*f = h \pmod{g}$ .

## Examples

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

>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x + 1)

sympy.polys.polytools.gcdex(f, g, *gens, **args)
Extended Euclidean algorithm of f and g.  
Returns (s, t, h) such that  $h = \gcd(f, g)$  and  $s*f + t*g = h$ .
```

## Examples

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

>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x**2/5 - 6*x/5 + 2, x + 1)

sympy.polys.polytools.invert(f, g, *gens, **args)
Invert f modulo g when possible.
```

## Examples

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

>>> invert(x**2 - 1, 2*x - 1)
-4/3

>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

---

```
sympy.polys.polytools.subresultants(f, g, *gens, **args)
Compute subresultant PRS of f and g.
```

### Examples

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

>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

```
sympy.polys.polytools.resultant(f, g, *gens, **args)
Compute resultant of f and g.
```

### Examples

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

>>> resultant(x**2 + 1, x**2 - 1)
4
```

```
sympy.polys.polytools.discriminant(f, *gens, **args)
Compute discriminant of f.
```

### Examples

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

>>> discriminant(x**2 + 2*x + 3)
-8
```

```
sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)
Compute the dispersion of polynomials.
```

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion  $\text{dis}(f, g)$  is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial  $\text{dis}(f) := \text{dis}(f, f)$ . Note that we make the definition  $\max\{\} := -\infty$ .

**See also:**

[sympy.polys.dispersion.dispersionset](#) (page 668)

### References

1. [ManWright94] (page 1246)
2. [Koepf98] (page 1246)
3. [Abramov71] (page 1246)

4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

### Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be  $-\infty$  as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)`

Compute the *dispersion set* of two polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion set  $J(f, g)$  is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines  $J(f) := J(f, f)$ .

See also:

`sympy.polys.dispersion.dispersion` (page 667)

## References

- 1.[ManWright94] (page 1246)
- 2.[Koepf98] (page 1246)
- 3.[Abramov71] (page 1246)
- 4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

```
sympy.polys.polytools.terms_gcd(f, *gens, **args)
Remove GCD of terms from f.
```

If the `deep` flag is True, then the arguments of `f` will have `terms.gcd` applied to them.

If a fraction is factored out of `f` and `f` is an `Add`, then an unevaluated `Mul` will be returned so that automatic simplification does not redistribute it. The hint `clear`, when set to `False`, can be used to prevent such factoring when all coefficients are not fractions.

See also:

[sympy.core.exprtools.gcd\\_terms](#) (page 158), [sympy.core.exprtools.factor\\_terms](#) (page 159)

## Examples

```
>>> from sympy import terms_gcd, cos
>>> from sympy.abc import x, y
>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of `polys` routines is to expand the expression given to them. `terms_gcd` follows this behavior:

```
>>> terms_gcd((3+3*x)*(x+x*y))
3*x*(x*y + x + y + 1)
```

If this is not desired then the hint `expand` can be set to `False`. In this case the expression will be treated as though it were comprised of one or more terms:

```
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a `Mul` or the arguments of other functions, the `deep` hint can be used:

```
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

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

Only the `y`-term had a coefficient that was a fraction; if one does not want to factor out the  $1/2$  in cases like this, the flag `clear` can be set to `False`:

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

The `clear` flag is ignored if all coefficients are fractions:

```
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

`sympy.polys.polytools.cofactors(f, g, *gens, **args)`  
Compute GCD and cofactors of  $f$  and  $g$ .

Returns polynomials ( $h$ ,  $cff$ ,  $cfg$ ) such that  $h = \text{gcd}(f, g)$ , and  $cff = \text{quo}(f, h)$  and  $cfg = \text{quo}(g, h)$  are, so called, cofactors of  $f$  and  $g$ .

### Examples

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

>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

`sympy.polys.polytools.gcd(f, g=None, *gens, **args)`  
Compute GCD of  $f$  and  $g$ .

### Examples

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

>>> gcd(x**2 - 1, x**2 - 3*x + 2)
x - 1
```

`sympy.polys.polytools.gcd_list(seq, *gens, **args)`  
Compute GCD of a list of polynomials.

### Examples

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

>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

`sympy.polys.polytools.lcm(f, g=None, *gens, **args)`  
Compute LCM of  $f$  and  $g$ .

### Examples

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

>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`sympy.polys.polytools.lcm_list(seq, *gens, **args)`  
Compute LCM of a list of polynomials.

**Examples**

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

>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2

sympy.polys.polytools.trunc(f, p, *gens, **args)
Reduce f modulo a constant p.
```

**Examples**

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

>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1

sympy.polys.polytools.monic(f, *gens, **args)
Divide all coefficients of f by LC(f).
```

**Examples**

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

>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3

sympy.polys.polytools.content(f, *gens, **args)
Compute GCD of coefficients of f.
```

**Examples**

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

>>> content(6*x**2 + 8*x + 12)
2

sympy.polys.polytools.primitive(f, *gens, **args)
Compute content and the primitive form of f.
```

**Examples**

```
>>> from sympy.polys.polytools import primitive
>>> from sympy.abc import x

>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)
```

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

Expansion is performed by default:

```
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set `expand` to `False` to shut this off. Note that the extraction will not be recursive; use the `as_content_primitive` method for recursive, non-destructive Rational extraction.

```
>>> primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)
```

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

```
sympy.polys.polytools.compose(f, g, *gens, **args)
Compute functional composition f(g).
```

### Examples

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

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

```
sympy.polys.polytools.decompose(f, *gens, **args)
Compute functional decomposition of f.
```

### Examples

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

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

```
sympy.polys.polytools.sturm(f, *gens, **args)
Compute Sturm sequence of f.
```

### Examples

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

```
>>> sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]
```

```
sympy.polys.polytools.gff_list(f, *gens, **args)
Compute a list of greatest factorial factors of f.
```

## Examples

```
>>> from sympy import gff_list, ff
>>> from sympy.abc import x

>>> f = x**5 + 2*x**4 - x**3 - 2*x**2

>>> gff_list(f)
[(x, 1), (x + 2, 4)]

>>> (ff(x, 1)*ff(x + 2, 4)).expand() == f
True

sympy.polys.polytools.gff(f, *gens, **args)
Compute greatest factorial factorization of f.

sympy.polys.polytools.sqf_norm(f, *gens, **args)
Compute square-free norm of f.

Returns s, f, r, such that g(x) = f(x-sa) and r(x) = Norm(g(x)) is a square-free polynomial over K, where a is the algebraic extension of the ground domain.
```

## Examples

```
>>> from sympy import sqf_norm, sqrt
>>> from sympy.abc import x

>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)

sympy.polys.polytools.sqf_part(f, *gens, **args)
Compute square-free part of f.
```

## Examples

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

>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2

sympy.polys.polytools.sqf_list(f, *gens, **args)
Compute a list of square-free factors of f.
```

## Examples

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

>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])

sympy.polys.polytools.sqf(f, *gens, **args)
Compute square-free factorization of f.
```

## Examples

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

>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

`sympy.polys.polytools.factor_list(f, *gens, **args)`

Compute a list of irreducible factors of  $f$ .

## Examples

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

>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

`sympy.polys.polytools.factor(f, *gens, **args)`

Compute the factorization of expression,  $f$ , into irreducibles. (To factor an integer into primes, use `factorint`.)

There two modes implemented: symbolic and formal. If  $f$  is not an instance of `Poly` (page 679) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, `factor()` (page 675) will traverse the expression tree and factor its components without any prior expansion, unless an instance of `Add` (page 115) is encountered (in this case formal factorization is used). This way `factor()` (page 675) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: `extension`, `modulus` or `domain`.

## See also:

`sympy.nttheory.factor_.factorint` (page 255)

## Examples

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

>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2

>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)

>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
```

```
>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**10000000*(x**2 + 1))
(x + 2)**20000000*(x**2 + 1)
```

By default, factor deals with an expression as a whole:

```
>>> eq = 2** (x**2 + 2*x + 1)
>>> factor(eq)
2** (x**2 + 2*x + 1)
```

If the `deep` flag is True then subexpressions will be factored:

```
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

```
sympy.polys.polytools.intervals(F, all=False, eps=None, inf=None, sup=None, strict=False,
                                 fast=False, sqf=False)
```

Compute isolating intervals for roots of  $f$ .

### Examples

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

>>> intervals(x**2 - 3)
[(-2, -1), (1, 2), (1)]
>>> intervals(x**2 - 3, eps=1e-2)
[(-26/15, -19/11), (19/11, 26/15), (1)]
```

```
sympy.polys.polytools.refine_root(f, s, t, eps=None, steps=None, fast=False, check_sqf=False)
```

Refine an isolating interval of a root to the given precision.

### Examples

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

>>> refine_root(x**2 - 3, 1, 2, eps=1e-2)
(19/11, 26/15)
```

```
sympy.polys.polytools.count_roots(f, inf=None, sup=None)
```

Return the number of roots of  $f$  in  $[inf, sup]$  interval.

If one of `inf` or `sup` is complex, it will return the number of roots in the complex rectangle with corners at `inf` and `sup`.

### Examples

```
>>> from sympy import count_roots, I
>>> from sympy.abc import x
```

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1

sympy.polys.polytools.real_roots(f, multiple=True)
    Return a list of real roots with multiplicities of f.
```

### Examples

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

>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]

sympy.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)
    Compute numerical approximations of roots of f.
```

### Examples

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

>>> nroots(x**2 - 3, n=15)
[-1.73205080756888, 1.73205080756888]
>>> nroots(x**2 - 3, n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]

sympy.polys.polytools.ground_roots(f, *gens, **args)
    Compute roots of f by factorization in the ground domain.
```

### Examples

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

>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}

sympy.polys.polytools.nth_power_roots_poly(f, n, *gens, **args)
    Construct a polynomial with n-th powers of roots of f.
```

### Examples

```
>>> from sympy import nth_power_roots_poly, factor, roots
>>> from sympy.abc import x

>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))
```

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

>>> R_f = [ (r**2).expand() for r in roots(f) ]
>>> R_g = roots(g).keys()

>>> set(R_f) == set(R_g)
True

sympy.polys.polytools.cancel(f, *gens, **args)
Cancel common factors in a rational function f.
```

### Examples

```
>>> from sympy import cancel, sqrt, Symbol
>>> from sympy.abc import x
>>> A = Symbol('A', commutative=False)

>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

```
sympy.polys.polytools.reduced(f, G, *gens, **args)
Reduces a polynomial f modulo a set of polynomials G.
```

Given a polynomial  $f$  and a set of polynomials  $G = (g_1, \dots, g_n)$ , computes a set of quotients  $q = (q_1, \dots, q_n)$  and the remainder  $r$  such that  $f = q_1*g_1 + \dots + q_n*g_n + r$ , where  $r$  vanishes or  $r$  is a completely reduced polynomial with respect to  $G$ .

### Examples

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

>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

```
sympy.polys.polytools.groebner(F, *gens, **args)
Computes the reduced Groebner basis for a set of polynomials.
```

Use the `order` argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of `solve_poly_system()`.

### References

1. [Buchberger01] (page 1245)

2. [Cox97] (page 1245)

[Buchberger01] (page 1245), [Cox97] (page 1245)

## Examples

Example taken from [1].

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

>>> F = [x*y - 2*y, 2*y**2 - x**2]

>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y, y**3 - 2*y], x, y,
              domain='ZZ', order='lex')
>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grlex')
>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using `method` flag or with the function `setup()` (page 849):

```
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]

>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

`sympy.polys.polytools.is_zero_dimensional(F, *gens, **args)`

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of `F` is bounded.

## References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

`class sympy.polys.polytools.Poly`

Generic class for representing polynomial expressions.

`EC(f, order=None)`

Returns the last non-zero coefficient of `f`.

## Examples

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

>>> Poly(x**3 + 2*x**2 + 3*x, x).EC()
3
```

`EM(f, order=None)`

Returns the last non-zero monomial of `f`.

### Examples

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

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

`ET(f, order=None)`

Returns the last non-zero term of `f`.

### Examples

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

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

`LC(f, order=None)`

Returns the leading coefficient of `f`.

### Examples

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

>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

`LM(f, order=None)`

Returns the leading monomial of `f`.

### Examples

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

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

`LT(f, order=None)`

Returns the leading term of `f`.

### Examples

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

>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

`TC(f)`

Returns the trailing coefficient of `f`.

#### Examples

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

>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

`abs(f)`

Make all coefficients in `f` positive.

#### Examples

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

>>> Poly(x**2 - 1, x).abs()
Poly(x**2 + 1, x, domain='ZZ')
```

`add(f, g)`

Add two polynomials `f` and `g`.

#### Examples

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

>>> Poly(x**2 + 1, x).add(Poly(x - 2, x))
Poly(x**2 + x - 1, x, domain='ZZ')

>>> Poly(x**2 + 1, x) + Poly(x - 2, x)
Poly(x**2 + x - 1, x, domain='ZZ')
```

`add_ground(f, coeff)`

Add an element of the ground domain to `f`.

#### Examples

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

>>> Poly(x + 1).add_ground(2)
Poly(x + 3, x, domain='ZZ')
```

`all_coeffs(f)`

Returns all coefficients from a univariate polynomial `f`.

**Examples**

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

>>> Poly(x**3 + 2*x - 1, x).all_coeffs()
[1, 0, 2, -1]

all_monomoms(f)
Returns all monomials from a univariate polynomial f.
```

See also:

[all\\_terms](#) (page 682)

**Examples**

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

>>> Poly(x**3 + 2*x - 1, x).all_monomoms()
[(3,), (2,), (1,), (0,)]
```

[all\\_roots](#)(*f*, *multiple=True*, *radicals=True*)

Return a list of real and complex roots with multiplicities.

**Examples**

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

>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).all_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).all_roots()
[RootOf(x**3 + x + 1, 0),
 RootOf(x**3 + x + 1, 1),
 RootOf(x**3 + x + 1, 2)]
```

[all\\_terms](#)(*f*)

Returns all terms from a univariate polynomial f.

**Examples**

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

>>> Poly(x**3 + 2*x - 1, x).all_terms()
[((3,), 1), ((2,), 0), ((1,), 2), ((0,), -1)]
```

**args**

Don't mess up with the core.

## Examples

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

>>> Poly(x**2 + 1, x).args
(x**2 + 1,)
```

`as_dict(f, native=False, zero=False)`  
Switch to a dict representation.

## Examples

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

>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}

as_expr(f, *gens)
Convert a Poly instance to an Expr instance.
```

## Examples

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

>>> f = Poly(x**2 + 2*x*y**2 - y, x, y)

>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379

as_list(f, native=False)
Switch to a list representation.
```

`cancel(f, g, include=False)`  
Cancel common factors in a rational function  $f/g$ .

## Examples

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

>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x))
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))

>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

```
clear_denoms(convert=False)
Clear denominators, but keep the ground domain.
```

### Examples

```
>>> from sympy import Poly, S, QQ
>>> from sympy.abc import x

>>> f = Poly(x/2 + S(1)/3, x, domain=QQ)

>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

`coeff_monomial(f, monom)`

Returns the coefficient of `monom` in `f` if there, else `None`.

**See also:**

`nth` ([page 703](#)) more efficient query using exponents of the monomial's generators

### Examples

```
>>> from sympy import Poly, exp
>>> from sympy.abc import x, y

>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)

>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*exp(8)
```

Note that `Expr.coeff()` behaves differently, collecting terms if possible; the `Poly` must be converted to an `Expr` to use that method, however:

```
>>> p.as_expr().coeff(x)
24*y*exp(8) + 23
>>> p.as_expr().coeff(y)
24*x*exp(8)
>>> p.as_expr().coeff(x*y)
24*exp(8)
```

`coeffs(f, order=None)`

Returns all non-zero coefficients from `f` in lex order.

**See also:**

`all_coeffs` ([page 681](#)), `coeff_monomial` ([page 684](#)), `nth` ([page 703](#))

## Examples

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

```
>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

`cofactors(f, g)`

Returns the GCD of  $f$  and  $g$  and their cofactors.

Returns polynomials ( $h$ ,  $cff$ ,  $cfg$ ) such that  $h = \text{gcd}(f, g)$ , and  $cff = \text{quo}(f, h)$  and  $cfg = \text{quo}(g, h)$  are, so called, cofactors of  $f$  and  $g$ .

## Examples

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

```
>>> Poly(x**2 - 1, x).cofactors(Poly(x**2 - 3*x + 2, x))
(Poly(x - 1, x, domain='ZZ'),
 Poly(x + 1, x, domain='ZZ'),
 Poly(x - 2, x, domain='ZZ'))
```

`compose(f, g)`

Computes the functional composition of  $f$  and  $g$ .

## Examples

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

```
>>> Poly(x**2 + x, x).compose(Poly(x - 1, x))
Poly(x**2 - x, x, domain='ZZ')
```

`content(f)`

Returns the GCD of polynomial coefficients.

## Examples

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

```
>>> Poly(6*x**2 + 8*x + 12, x).content()
2
```

`count_roots(f, inf=None, sup=None)`

Return the number of roots of  $f$  in  $[inf, sup]$  interval.

**Examples**

```
>>> from sympy import Poly, I
>>> from sympy.abc import x

>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1

decompose(f)
Computes a functional decomposition of f.
```

**Examples**

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

>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]

deflate(f)
Reduce degree of f by mapping x_i**m to y_i.
```

**Examples**

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

>>> Poly(x**6*y**2 + x**3 + 1, x, y).deflate()
((3, 2), Poly(x**2*y + x + 1, x, y, domain='ZZ'))

degree(f, gen=0)
Returns degree of f in x_j.

The degree of 0 is negative infinity.
```

**Examples**

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

>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
>>> Poly(0, x).degree()
-oo

degree_list(f)
Returns a list of degrees of f.
```

## Examples

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

>>> Poly(x**2 + y*x + 1, x, y).degree_list()
(2, 1)

diff(f, *specs)
Computes partial derivative of f.
```

## Examples

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

>>> Poly(x**2 + 2*x + 1, x).diff()
Poly(2*x + 2, x, domain='ZZ')

>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')

discriminant(f)
Computes the discriminant of f.
```

## Examples

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

>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

`dispersion(f, g=None)`  
Compute the *dispersion* of polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion  $\text{dis}(f, g)$  is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial  $\text{dis}(f) := \text{dis}(f, f)$ .

**See also:**

`sympy.polys.polytools.Poly.dispersionset` (page 688)

## References

- 1.[ManWright94] (page 1246)
- 2.[Koepf98] (page 1246)
- 3.[Abramov71] (page 1246)

4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`dispersionset(f, g=None)`

Compute the *dispersion set* of two polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion set  $J(f, g)$  is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines  $J(f) := J(f, f)$ .

See also:

`sympy.polys.polytools.Poly.dispersion` (page 687)

## References

- 1.[ManWright94] (page 1246)
- 2.[Koepf98] (page 1246)
- 3.[Abramov71] (page 1246)
- 4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]

div(f, g, auto=True)
Polynomial division with remainder of f by g.
```

### Examples

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

>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))

>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

### domain

Get the ground domain of `self`.

```
eject(f, *gens)
Eject selected generators into the ground domain.
```

### Examples

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

>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x, y)

>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')
>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')

eval(x, a=None, auto=True)
Evaluate f at a in the given variable.
```

### Examples

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

>>> Poly(x**2 + 2*x + 3, x).eval(2)
11

>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')

>>> f = Poly(2*x*y + 3*x + y + 2*z, x, y, z)
```

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45

>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')

exclude(f)
Remove unnecessary generators from f.
```

### Examples

```
>>> from sympy import Poly
>>> from sympy.abc import a, b, c, d, x

>>> Poly(a + x, a, b, c, d, x).exclude()
Poly(a + x, a, x, domain='ZZ')

exquo(f, g, auto=True)
Computes polynomial exact quotient of f by g.
```

### Examples

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

>>> Poly(x**2 - 1, x).exquo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')

>>> Poly(x**2 + 1, x).exquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

```
exquo_ground(f, coeff)
Exact quotient of f by a an element of the ground domain.
```

### Examples

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

>>> Poly(2*x + 4).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')

>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
```

```
...  
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

### `factor_list(f)`

Returns a list of irreducible factors of `f`.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x, y  
  
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y  
  
>>> Poly(f).factor_list()  
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),  
      (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

### `factor_list_include(f)`

Returns a list of irreducible factors of `f`.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x, y  
  
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y  
  
>>> Poly(f).factor_list_include()  
[(Poly(2*x + 2*y, x, y, domain='ZZ'), 1),  
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)]
```

### `free_symbols`

Free symbols of a polynomial expression.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x, y  
  
>>> Poly(x**2 + 1).free_symbols  
set([x])  
>>> Poly(x**2 + y).free_symbols  
set([x, y])  
>>> Poly(x**2 + y, x).free_symbols  
set([x, y])
```

### `free_symbols_in_domain`

Free symbols of the domain of `self`.

## Examples

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

>>> Poly(x**2 + 1).free_symbols_in_domain
set()
>>> Poly(x**2 + y).free_symbols_in_domain
set()
>>> Poly(x**2 + y, x).free_symbols_in_domain
set([y])

classmethod from_dict(rep, *gens, **args)
Construct a polynomial from a dict.

classmethod from_expr(rep, *gens, **args)
Construct a polynomial from an expression.

classmethod from_list(rep, *gens, **args)
Construct a polynomial from a list.

classmethod from_poly(rep, *gens, **args)
Construct a polynomial from a polynomial.

gcd(f, g)
Returns the polynomial GCD of f and g.
```

## Examples

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

>>> Poly(x**2 - 1, x).gcd(Poly(x**2 - 3*x + 2, x))
Poly(x - 1, x, domain='ZZ')

gcdex(f, g, auto=True)
Extended Euclidean algorithm of f and g.

Returns (s, t, h) such that h = gcd(f, g) and s*f + t*g = h.
```

## Examples

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

>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4

>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))

gen
Return the principal generator.
```

**Examples**

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

>>> Poly(x**2 + 1, x).gen
x

get_domain(f)
Get the ground domain of f.

get_modulus(f)
Get the modulus of f.
```

**Examples**

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

>>> Poly(x**2 + 1, modulus=2).get_modulus()
2

gff_list(f)
Computes greatest factorial factorization of f.
```

**Examples**

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

>>> f = x**5 + 2*x**4 - x**3 - 2*x**2

>>> Poly(f).gff_list()
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]

ground_roots(f)
Compute roots of f by factorization in the ground domain.
```

**Examples**

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

>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()
{0: 2, 1: 2}

half_gcdex(f, g, auto=True)
Half extended Euclidean algorithm of f and g.

Returns (s, h) such that h = gcd(f, g) and s*f = h (mod g).
```

## Examples

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

>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4

>>> Poly(f).half_gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

`has_only_gens(f, *gens)`  
Return True if `Poly(f, *gens)` retains ground domain.

## Examples

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

>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```

`homogeneous_order(f)`  
Returns the homogeneous order of f.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of f. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 697).

## Examples

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

>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

`homogenize(f, s)`  
Returns the homogeneous polynomial of f.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use `Poly.is_homogeneous()` (page 697). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 695).

## Examples

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

```
>>> f = Poly(x**5 + 2*x**2*y**2 + 9*x*y**3)
>>> f.homogenize(z)
Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

**inject(*f*, *front=False*)**  
Inject ground domain generators into *f*.

#### Examples

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

>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)

>>> f.inject()
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')
>>> f.inject(front=True)
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

**integrate(\*specs, \*\*args)**  
Computes indefinite integral of *f*.

#### Examples

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

>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')

>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

**intervals(*f*, *all=False*, *eps=None*, *inf=None*, *sup=None*, *fast=False*, *sqf=False*)**  
Compute isolating intervals for roots of *f*.

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

#### Examples

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

>>> Poly(x**2 - 3, x).intervals()
[(-2, -1), (1, 2)]
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[(-26/15, -19/11), (19/11, 26/15)]
```

**invert(*f*, *g*, *auto=True*)**  
Invert *f* modulo *g* when possible.

## Examples

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

>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')

>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

### is\_cyclotomic

Returns True if  $f$  is a cyclotomic polynomial.

## Examples

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

>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> Poly(f).is_cyclotomic
False

>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> Poly(g).is_cyclotomic
True
```

### is\_ground

Returns True if  $f$  is an element of the ground domain.

## Examples

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

>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

### is\_homogeneous

Returns True if  $f$  is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use `Poly.homogeneous_order()` (page 695).

**Examples**

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

>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

**is\_irreducible**

Returns **True** if  $f$  has no factors over its domain.

**Examples**

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

>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

**is\_linear**

Returns **True** if  $f$  is linear in all its variables.

**Examples**

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

>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

**is\_monic**

Returns **True** if the leading coefficient of  $f$  is one.

**Examples**

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

>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

**is\_monomial**

Returns **True** if  $f$  is zero or has only one term.

### Examples

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

>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

#### is\_multivariate

Returns True if  $f$  is a multivariate polynomial.

### Examples

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

>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

#### is\_one

Returns True if  $f$  is a unit polynomial.

### Examples

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

>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

#### is\_primitive

Returns True if GCD of the coefficients of  $f$  is one.

### Examples

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

>>> Poly(2*x**2 + 6*x + 12, x).is_primitive
False
>>> Poly(x**2 + 3*x + 6, x).is_primitive
True
```

`is_quadratic`  
Returns True if  $f$  is quadratic in all its variables.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x, y  
  
>>> Poly(x*y + 2, x, y).is_quadratic  
True  
>>> Poly(x*y**2 + 2, x, y).is_quadratic  
False
```

`is_sqf`  
Returns True if  $f$  is a square-free polynomial.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x  
  
>>> Poly(x**2 - 2*x + 1, x).is_sqf  
False  
>>> Poly(x**2 - 1, x).is_sqf  
True
```

`is_univariate`  
Returns True if  $f$  is a univariate polynomial.

#### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x, y  
  
>>> Poly(x**2 + x + 1, x).is_univariate  
True  
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate  
False  
>>> Poly(x*y**2 + x*y + 1, x).is_univariate  
True  
>>> Poly(x**2 + x + 1, x, y).is_univariate  
False
```

`is_zero`  
Returns True if  $f$  is a zero polynomial.

#### Examples

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

```
>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False
```

`l1_norm(f)`  
Returns l1 norm of f.

#### Examples

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

>>> Poly(-x**2 + 2*x - 3, x).l1_norm()
6
```

`lcm(f, g)`  
Returns polynomial LCM of f and g.

#### Examples

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

>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

`length(f)`  
Returns the number of non-zero terms in f.

#### Examples

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

>>> Poly(x**2 + 2*x - 1).length()
3
```

`lift(f)`  
Convert algebraic coefficients to rationals.

#### Examples

```
>>> from sympy import Poly, I
>>> from sympy.abc import x

>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

`ltrim(f, gen)`  
Remove dummy generators from the “left” of f.

## Examples

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

>>> Poly(y**2 + y*z**2, x, y, z).ltrim(y)
Poly(y**2 + y*z**2, y, z, domain='ZZ')

max_norm(f)
Returns maximum norm of f.
```

## Examples

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

>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3

monic(auto=True)
Divides all coefficients by LC(f).
```

## Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x

>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')

>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')

monoms(f, order=None)
Returns all non-zero monomials from f in lex order.
```

### See also:

[all\\_monomials](#) (page 682)

## Examples

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

>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]

mul(f, g)
Multiply two polynomials f and g.
```

## Examples

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

>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')

>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')

mul_ground(f, coeff)
Multiply f by a an element of the ground domain.
```

## Examples

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

>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')

neg(f)
Negate all coefficients in f.
```

## Examples

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

>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')

>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')

classmethod new(rep, *gens)
Construct Poly (page 679) instance from raw representation.

nroots(f, n=15, maxsteps=50, cleanup=True)
Compute numerical approximations of roots of f.
```

**Parameters** n ... the number of digits to calculate  
maxsteps ... the maximum number of iterations to do  
If the accuracy ‘n’ cannot be reached in ‘maxsteps’, it will raise an exception. You need to rerun with higher maxsteps.

## Examples

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

```
>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`nth(f, *N)`

Returns the  $n$ -th coefficient of  $f$  where  $N$  are the exponents of the generators in the term of interest.

See also:

`coeff_monomial` (page 684)

### Examples

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

>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*sqrt(x), y, sqrt(x), domain='ZZ')
>>> _.nth(1, 1)
4
```

`nth_power_roots_poly(f, n)`

Construct a polynomial with  $n$ -th powers of roots of  $f$ .

### Examples

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

>>> f = Poly(x**4 - x**2 + 1)

>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```

`one`

Return one polynomial with `self`'s properties.

`pdiv(f, g)`

Polynomial pseudo-division of  $f$  by  $g$ .

## Examples

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

>>> Poly(x**2 + 1, x).pdiv(Poly(2*x - 4, x))
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

`per(f, rep, gens=None, remove=None)`  
Create a Poly out of the given representation.

## Examples

```
>>> from sympy import Poly, ZZ
>>> from sympy.abc import x, y

>>> from sympy.polys.polyclasses import DMP

>>> a = Poly(x**2 + 1)

>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')

pexquo(f, g)
Polynomial exact pseudo-quotient of f by g.
```

## Examples

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

>>> Poly(x**2 - 1, x).pexquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')

>>> Poly(x**2 + 1, x).pexquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`pow(f, n)`  
Raise f to a non-negative power n.

## Examples

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

>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')

>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

`pquo(f, g)`  
Polynomial pseudo-quotient of  $f$  by  $g$ .

#### Examples

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

>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')

>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

`prem(f, g)`  
Polynomial pseudo-remainder of  $f$  by  $g$ .

#### Examples

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

>>> Poly(x**2 + 1, x).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

`primitive(f)`  
Returns the content and a primitive form of  $f$ .

#### Examples

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

>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

`quo(f, g, auto=True)`  
Computes polynomial quotient of  $f$  by  $g$ .

#### Examples

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

>>> Poly(x**2 + 1, x).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')

>>> Poly(x**2 - 1, x).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')

quo_ground(f, coeff)
Quotient of  $f$  by a an element of the ground domain.
```

### Examples

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

>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')

>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')

rat_clear_denoms(g)
Clear denominators in a rational function f/g.
```

### Examples

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

>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)

>>> p, q = f.rat_clear_denoms(g)

>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')

real_roots(f, multiple=True, radicals=True)
Return a list of real roots with multiplicities.
```

### Examples

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

>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).real_roots()
[RootOf(x**3 + x + 1, 0)]

refine_root(f, s, t, eps=None, steps=None, fast=False, check_sgf=False)
Refine an isolating interval of a root to the given precision.
```

### Examples

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

>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

`rem(f, g, auto=True)`  
Computes the polynomial remainder of  $f$  by  $g$ .

#### Examples

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

>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')

>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

`reorder(f, *gens, **args)`  
Efficiently apply new order of generators.

#### Examples

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

>>> Poly(x**2 + x*y**2, x, y).reorder(y, x)
Poly(y**2*x + x**2, y, x, domain='ZZ')
```

`replace(f, x, y=None)`  
Replace  $x$  with  $y$  in generators list.

#### Examples

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

>>> Poly(x**2 + 1, x).replace(x, y)
Poly(y**2 + 1, y, domain='ZZ')
```

`resultant(f, g, includePRS=False)`  
Computes the resultant of  $f$  and  $g$  via PRS.

If `includePRS=True`, it includes the subresultant PRS in the result. Because the PRS is used to calculate the resultant, this is more efficient than calling `subresultants()` (page 666) separately.

#### Examples

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

>>> f = Poly(x**2 + 1, x)
```

```
>>> f.resultant(Poly(x**2 - 1, x))
4
>>> f.resultant(Poly(x**2 - 1, x), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
      Poly(-2, x, domain='ZZ')])
```

**retract(*f*, *field=None*)**  
Recalculate the ground domain of a polynomial.

#### Examples

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

>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')

>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

**revert(*f*, *n*)**  
Compute  $f^{**(-1)} \bmod x^{**n}$ .

**root(*f*, *index*, *radicals=True*)**  
Get an indexed root of a polynomial.

#### Examples

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

>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)

>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3

>>> Poly(x**5 + x + 1).root(0)
RootOf(x**3 - x**2 + 1, 0)

set_domain(f, domain)
Set the ground domain of f.

set_modulus(f, modulus)
Set the modulus of f.
```

## Examples

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

>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)

shift(f, a)
Efficiently compute Taylor shift f(x + a).
```

## Examples

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

>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')

slice(f, x, m, n=None)
Take a continuous subsequence of terms of f.

sqf_list(f, all=False)
Returns a list of square-free factors of f.
```

## Examples

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

>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16

>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])

>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
      (Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])

sqf_list_include(f, all=False)
Returns a list of square-free factors of f.
```

## Examples

```
>>> from sympy import Poly, expand
>>> from sympy.abc import x

>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4
```

```
>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]

>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

`sqf_norm(f)`  
Computes square-free norm of  $f$ .

Returns  $s, f, r$ , such that  $g(x) = f(x-sa)$  and  $r(x) = \text{Norm}(g(x))$  is a square-free polynomial over  $K$ , where  $a$  is the algebraic extension of the ground domain.

### Examples

```
>>> from sympy import Poly, sqrt
>>> from sympy.abc import x

>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()

>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

`sqf_part(f)`  
Computes square-free part of  $f$ .

### Examples

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

>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

`sqr(f)`  
Square a polynomial  $f$ .

### Examples

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

>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

**sturm(*auto=True*)**

Computes the Sturm sequence of *f*.

**Examples**

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

>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

**sub(*f, g*)**

Subtract two polynomials *f* and *g*.

**Examples**

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

>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')

>>> Poly(x**2 + 1, x) - Poly(x - 2, x)
Poly(x**2 - x + 3, x, domain='ZZ')
```

**sub\_ground(*f, coeff*)**

Subtract an element of the ground domain from *f*.

**Examples**

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

>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```

**subresultants(*f, g*)**

Computes the subresultant PRS of *f* and *g*.

**Examples**

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

```
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

**terms(*f*, *order=None*)**  
Returns all non-zero terms from *f* in lex order.

See also:

[all\\_terms](#) (page 682)

#### Examples

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

>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[((2, 0), 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

**terms\_gcd(*f*)**  
Remove GCD of terms from the polynomial *f*.

#### Examples

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

>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

**termwise(*f*, *func*, \**gens*, \*\**args*)**  
Apply a function to all terms of *f*.

#### Examples

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

>>> def func(k, coeff):
...     k = k[0]
...     return coeff//10**(2-k)

>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

**to\_exact(*f*)**  
Make the ground domain exact.

#### Examples

```
>>> from sympy import Poly, RR  
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()  
Poly(x**2 + 1, x, domain='QQ')
```

**to\_field(*f*)**

Make the ground domain a field.

**Examples**

```
>>> from sympy import Poly, ZZ  
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, x, domain=ZZ).to_field()  
Poly(x**2 + 1, x, domain='QQ')
```

**to\_ring(*f*)**

Make the ground domain a ring.

**Examples**

```
>>> from sympy import Poly, QQ  
>>> from sympy.abc import x
```

```
>>> Poly(x**2 + 1, domain=QQ).to_ring()  
Poly(x**2 + 1, x, domain='ZZ')
```

**total\_degree(*f*)**

Returns the total degree of *f*.

**Examples**

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

```
>>> Poly(x**2 + y*x + 1, x, y).total_degree()  
2  
>>> Poly(x + y**5, x, y).total_degree()  
5
```

**trunc(*f*, *p*)**

Reduce *f* modulo a constant *p*.

**Examples**

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

```
>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)  
Poly(-x**3 - x + 1, x, domain='ZZ')
```

**unify(*f*, *g*)**  
Make *f* and *g* belong to the same domain.

#### Examples

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

>>> f, g = Poly(x/2 + 1), Poly(2*x + 1)

>>> f
Poly(1/2*x + 1, x, domain='QQ')
>>> g
Poly(2*x + 1, x, domain='ZZ')

>>> F, G = f.unify(g)

>>> F
Poly(1/2*x + 1, x, domain='QQ')
>>> G
Poly(2*x + 1, x, domain='QQ')
```

**unit**  
Return unit polynomial with `self`'s properties.

**zero**  
Return zero polynomial with `self`'s properties.

**class sympy.polys.polytools.PurePoly**  
Class for representing pure polynomials.

**free\_symbols**  
Free symbols of a polynomial.

#### Examples

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

>>> PurePoly(x**2 + 1).free_symbols
set()
>>> PurePoly(x**2 + y).free_symbols
set()
>>> PurePoly(x**2 + y, x).free_symbols
set([y])
```

**class sympy.polys.polytools.GroebnerBasis**  
Represents a reduced Groebner basis.

**contains(*poly*)**  
Check if *poly* belongs the ideal generated by `self`.

## Examples

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

>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([x**2 + y**2 - 1, x*y - 2])

>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

### `fglm(order)`

Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

## References

J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering

## Examples

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

>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')

>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
```

### `is_zero_dimensional`

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of  $F$  is bounded.

## References

David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230

### `reduce(expr, auto=True)`

Reduces a polynomial modulo a Groebner basis.

Given a polynomial  $f$  and a set of polynomials  $G = (g_1, \dots, g_n)$ , computes a set of quotients  $q = (q_1, \dots, q_n)$  and the remainder  $r$  such that  $f = q_1*f_1 + \dots + q_n*f_n + r$ , where  $r$  vanishes or  $r$  is a completely reduced polynomial with respect to  $G$ .

## Examples

```
>>> from sympy import groebner, expand
>>> from sympy.abc import x, y

>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])

>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = -

>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

## Extra polynomial manipulation functions

`sympy.polys.polyfuncs.symmetrize(F, *gens, **args)`

Rewrite a polynomial in terms of elementary symmetric polynomials.

A symmetric polynomial is a multivariate polynomial that remains invariant under any variable permutation, i.e., if  $f = f(x_1, x_2, \dots, x_n)$ , then  $f = f(x_{\{i\_1\}}, x_{\{i\_2\}}, \dots, x_{\{i\_n\}})$ , where  $(i\_1, i\_2, \dots, i\_n)$  is a permutation of  $(1, 2, \dots, n)$  (an element of the group  $S_n$ ).

Returns a tuple of symmetric polynomials  $(f_1, f_2, \dots, f_n)$  such that  $f = f_1 + f_2 + \dots + f_n$ .

## Examples

```
>>> from sympy.polys.polyfuncs import symmetrize
>>> from sympy.abc import x, y

>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)

>>> symmetrize(x**2 + y**2, formal=True)
(s1**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])

>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)

>>> symmetrize(x**2 - y**2, formal=True)
(s1**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

`sympy.polys.polyfuncs.horner(f, *gens, **args)`

Rewrite a polynomial in Horner form.

Among other applications, evaluation of a polynomial at a point is optimal when it is applied using the Horner scheme ([1]).

## References

[1] - [http://en.wikipedia.org/wiki/Horner\\_scheme](http://en.wikipedia.org/wiki/Horner_scheme)

## Examples

```
>>> from sympy.polys.polyfuncs import horner
>>> from sympy.abc import x, y, a, b, c, d, e

>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5

>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))

>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y

>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))

>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```

`sympy.polys.polyfuncs.interpolate(data, x)`  
Construct an interpolating polynomial for the data points.

## Examples

```
>>> from sympy.polys.polyfuncs import interpolate
>>> from sympy.abc import x
```

A list is interpreted as though it were paired with a range starting from 1:

```
>>> interpolate([1, 4, 9, 16], x)
x**2
```

This can be made explicit by giving a list of coordinates:

```
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
x**2
```

The (x, y) coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```
>>> interpolate([-1, 2), (1, 2), (2, 5)], x)
x**2 + 1
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
x**2 + 1
```

`sympy.polys.polyfuncs.viete(f, roots=None, *gens, **args)`  
Generate Viete's formulas for `f`.

## Examples

```
>>> from sympy.polys.polyfuncs import viete
>>> from sympy import symbols

>>> x, a, b, c, r1, r2 = symbols('x,a:c,r1:3')
```

---

```
>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

## Domain constructors

`sympy.polys.constructor.construct_domain(obj, **args)`  
Construct a minimal domain for the list of coefficients.

## Algebraic number fields

`sympy.polys.numberfields.minimal_polynomial(ex, x=None, **args)`  
Computes the minimal polynomial of an algebraic element.

**Parameters** `ex` : algebraic element expression  
`x` : independent variable of the minimal polynomial

### Notes

By default `compose=True`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `compose=False`, a bottom-up algorithm is used with `groebner`. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

### Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y

>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`sympy.polys.numberfields.minpoly(ex, x=None, **args)`  
Computes the minimal polynomial of an algebraic element.

**Parameters** `ex` : algebraic element expression  
`x` : independent variable of the minimal polynomial

### Notes

By default `compose=True`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `compose=False`, a bottom-up algorithm is used with `groebner`. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

### Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y

>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y

sympy.polys.numberfields.primitive_element(extension, x=None, **args)
    Construct a common number field for all extensions.

sympy.polys.numberfields.field_isomorphism(a, b, **args)
    Construct an isomorphism between two number fields.

sympy.polys.numberfields.to_number_field(extension, theta=None, **args)
    Express extension in the field generated by theta.

sympy.polys.numberfields.isolate(alg, eps=None, fast=False)
    Give a rational isolating interval for an algebraic number.

class sympy.polys.numberfields.AlgebraicNumber
    Class for representing algebraic numbers in SymPy.

    as_expr(x=None)
        Create a Basic expression from self.

    as_poly(x=None)
        Create a Poly instance from self.

    coeffs()
        Returns all SymPy coefficients of an algebraic number.

    is_aliased
        Returns True if alias was set.

    native_coeffs()
        Returns all native coefficients of an algebraic number.

    to_algebraic_integer()
        Convert self to an algebraic integer.
```

### Monomials encoded as tuples

```
class sympy.polys.monomials.Monomial(monom, gens=None)
    Class representing a monomial, i.e. a product of powers.

sympy.polys.monomials.terminomials(variables, degree)
    Generate a set of monomials of the given total degree or less.
```

Given a set of variables  $V$  and a total degree  $N$  generate a set of monomials of degree at most  $N$ . The total number of monomials is huge and is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree  $N = 50$  in 5 variables, assuming that exponents and all of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of memory! Fortunately most polynomials, that we will encounter, are sparse.

## Examples

Consider monomials in variables  $x$  and  $y$ :

```
>>> from sympy.polys.monomials import itermonomials
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]

>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]
```

`sympy.polys.monomials.monomial_count(V, N)`

Computes the number of monomials.

The number of monomials is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

where  $N$  is a total degree and  $V$  is a set of variables.

## Examples

```
>>> from sympy.polys.monomials import itermonomials, monomial_count
>>> from sympy.polys.orderings import monomial_key
>>> from sympy.abc import x, y

>>> monomial_count(2, 2)
6

>>> M = itermonomials([x, y], 2)

>>> sorted(M, key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
>>> len(M)
6
```

## Orderings of monomials

```
class sympy.polys.orderings.LexOrder
    Lexicographic order of monomials.
```

```
class sympy.polys.orderings.GradedLexOrder
    Graded lexicographic order of monomials.

class sympy.polys.orderings.ReversedGradedLexOrder
    Reversed graded lexicographic order of monomials.
```

### Formal manipulation of roots of polynomials

```
class sympy.polys.rootoftools.RootOf
    Represents k-th root of a univariate polynomial.

class sympy.polys.rootoftools.RootSum
    Represents a sum of all roots of a univariate polynomial.
```

### Symbolic root-finding algorithms

```
sympy.polys.polyroots.roots(f, *gens, **flags)
    Computes symbolic roots of a univariate polynomial.
```

Given a univariate polynomial  $f$  with symbolic coefficients (or a list of the polynomial's coefficients), returns a dictionary with its roots and their multiplicities.

Only roots expressible via radicals will be returned. To get a complete set of roots use `RootOf` class or numerical methods instead. By default cubic and quartic formulas are used in the algorithm. To disable them because of unreadable output set `cubics=False` or `quartics=False` respectively. If cubic roots are real but are expressed in terms of complex numbers (casus irreducibilis [1]) the `trig` flag can be set to True to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the `filter` flag with one of the following specifiers: Z, Q, R, I, C. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to True; the list will have identical roots appearing next to each other in the result. (For a given `Poly`, the `all_roots` method will give the roots in sorted numerical order.)

### References

1.[http://en.wikipedia.org/wiki/Cubic\\_function#Trigonometric\\_and\\_hyperbolic\\_method](http://en.wikipedia.org/wiki/Cubic_function#Trigonometric_and_hyperbolic_method)

### Examples

```
>>> from sympy import Poly, roots
>>> from sympy.abc import x, y

>>> roots(x**2 - 1, x)
{-1: 1, 1: 1}

>>> p = Poly(x**2-1, x)
>>> roots(p)
{-1: 1, 1: 1}
```

```
>>> p = Poly(x**2-y, x, y)

>>> roots(Poly(p, x))
{-sqrt(y): 1, sqrt(y): 1}

>>> roots(x**2 - y, x)
{-sqrt(y): 1, sqrt(y): 1}

>>> roots([1, 0, -1])
{-1: 1, 1: 1}
```

## Special polynomials

`sympy.polys.specialpolys.swinnerton_dyter_poly(n, x=None, **args)`

Generates n-th Swinnerton-Dyer polynomial in  $x$ .

`sympy.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')`

Construct Lagrange interpolating polynomial for  $n$  data points.

`sympy.polys.specialpolys.cyclotomic_poly(n, x=None, **args)`

Generates cyclotomic polynomial of order  $n$  in  $x$ .

`sympy.polys.specialpolys.symmetric_poly(n, *gens, **args)`

Generates symmetric polynomial of order  $n$ .

`sympy.polys.specialpolys.random_poly(x, n, inf, sup, domain=ZZ, polys=False)`

Return a polynomial of degree  $n$  with coefficients in  $[inf, sup]$ .

## Orthogonal polynomials

`sympy.polys.orthopolys.chebyshev_t_poly(n, x=None, **args)`

Generates Chebyshev polynomial of the first kind of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.chebyshev_u_poly(n, x=None, **args)`

Generates Chebyshev polynomial of the second kind of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.gegenbauer_poly(n, a, x=None, **args)`

Generates Gegenbauer polynomial of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.hermite_poly(n, x=None, **args)`

Generates Hermite polynomial of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.jacobi_poly(n, a, b, x=None, **args)`

Generates Jacobi polynomial of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.legendre_poly(n, x=None, **args)`

Generates Legendre polynomial of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.laguerre_poly(n, x=None, alpha=None, **args)`

Generates Laguerre polynomial of degree  $n$  in  $x$ .

`sympy.polys.orthopolys.spherical_bessel_fn(n, x=None, **args)`

Coefficients for the spherical Bessel functions.

Those are only needed in the `jn()` function.

The coefficients are calculated from:

$$fn(0, z) = 1/z \quad fn(1, z) = 1/z^{**}2 \quad fn(n-1, z) + fn(n+1, z) == (2*n+1)/z * fn(n, z)$$

## Examples

```
>>> from sympy.polys.orthopolys import spherical_bessel_fn as fn
>>> from sympy import Symbol
>>> z = Symbol("z")
>>> fn(1, z)
z**(-2)
>>> fn(2, z)
-1/z + 3/z**3
>>> fn(3, z)
-6/z**2 + 15/z**4
>>> fn(4, z)
1/z - 45/z**3 + 105/z**5
```

## Manipulation of rational functions

`sympy.polys.rationaltools.together(expr, deep=False)`

Denest and combine rational expressions using symbolic methods.

This function takes an expression or a container of expressions and puts it (them) together by denesting and combining rational subexpressions. No heroic measures are taken to minimize degree of the resulting numerator and denominator. To obtain completely reduced expression use `cancel()` (page 678). However, `together()` (page 724) can preserve as much as possible of the structure of the input expression in the output (no expansion is performed).

A wide variety of objects can be put together including lists, tuples, sets, relational objects, integrals and others. It is also possible to transform interior of function applications, by setting `deep` flag to `True`.

By definition, `together()` (page 724) is a complement to `apart()` (page 725), so `apart(together(expr))` should return `expr` unchanged. Note however, that `together()` (page 724) uses only symbolic methods, so it might be necessary to use `cancel()` (page 678) to perform algebraic simplification and minimise degree of the numerator and denominator.

## Examples

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

>>> together(1/x + 1/y)
(x + y)/(x*y)
>>> together(1/x + 1/y + 1/z)
(x*y + x*z + y*z)/(x*y*z)

>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)

>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))

>>> together(exp(1/x + 1/y))
exp(1/y + 1/x)
>>> together(exp(1/x + 1/y), deep=True)
exp((x + y)/(x*y))
```

```
>>> together(1/exp(x) + 1/(x*exp(x)))
(x + 1)*exp(-x)/x

>>> together(1/exp(2*x) + 1/(x*exp(3*x)))
(x*exp(x) + 1)*exp(-3*x)/x
```

### Partial fraction decomposition

`sympy.polys.partfrac.apart(expr, *args, **kwargs)`

Compute partial fraction decomposition of a rational function.

Given a rational function `f`, computes the partial fraction decomposition of `f`. Two algorithms are available: One is based on the undetermined coefficients method, the other is Bronstein's full partial fraction decomposition algorithm.

The undetermined coefficients method (selected by `full=False`) uses polynomial factorization (and therefore accepts the same options as `factor`) for the denominator. Per default it works over the rational numbers, therefore decomposition of denominators with non-rational roots (e.g. irrational, complex roots) is not supported by default (see options of `factor`).

Bronstein's algorithm can be selected by using `full=True` and allows a decomposition of denominators with non-rational roots. A human-readable result can be obtained via `doit()` (see examples below).

See also:

`apart_list` (page 725), `assemble_partfrac_list` (page 727)

### Examples

```
>>> from sympy.polys.partfrac import apart
>>> from sympy.abc import x, y
```

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*_a*y/3 - y/3)/(-_a + x)))
```

Calling `doit()` yields a human-readable result:

```
>>> apart(y/(x**2 + x + 1), x, full=True).doit()
(-y/3 - 2*y*(-1/2 - sqrt(3)*I/2)/3)/(x + 1/2 + sqrt(3)*I/2) + (-y/3 -
2*y*(-1/2 + sqrt(3)*I/2)/3)/(x + 1/2 - sqrt(3)*I/2)
```

`sympy.polys.partfrac.apart_list(f, x=None, dummies=None, **options)`

Compute partial fraction decomposition of a rational function and return the result in structured form.

Given a rational function  $f$  compute the partial fraction decomposition of  $f$ . Only Bronstein's full partial fraction decomposition algorithm is supported by this method. The return value is highly structured and perfectly suited for further algorithmic treatment rather than being human-readable. The function returns a tuple holding three elements:

- The first item is the common coefficient, free of the variable  $x$  used for decomposition. (It is an element of the base field  $K$ .)
- The second item is the polynomial part of the decomposition. This can be the zero polynomial. (It is an element of  $K[x]$ .)
- The third part itself is a list of quadruples. Each quadruple has the following elements in this order:
  - The (not necessarily irreducible) polynomial  $D$  whose roots  $w_i$  appear in the linear denominator of a bunch of related fraction terms. (This item can also be a list of explicit roots. However, at the moment `apart_list` never returns a result this way, but the related `assemble_partfrac_list` function accepts this format as input.)
  - The numerator of the fraction, written as a function of the root  $w$
  - The linear denominator of the fraction *excluding its power exponent*, written as a function of the root  $w$ .
  - The power to which the denominator has to be raised.

One can always rebuild a plain expression by using the function `assemble_partfrac_list`.

#### See also:

`apart` (page 725), `assemble_partfrac_list` (page 727)

#### References

1. [Bronstein93] (page 1245)

[Bronstein93] (page 1245)

#### Examples

A first example:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, t

>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
 Poly(2*x + 4, x, domain='ZZ'),
 [(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1)])

>>> assemble_partfrac_list(pfd)
2*x + 4 + 4/(x - 1)
```

Second example:

```
>>> f = (-2*x - 2*x**2) / (3*x**2 - 6*x)
>>> pfd = apart_list(f)
>>> pfd
(-1,
Poly(2/3, x, domain='QQ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 2), Lambda(_a, -_a + x), 1)])]

>>> assemble_partfrac_list(pfd)
-2/3 - 2/(x - 2)
```

Another example, showing symbolic parameters:

```
>>> pfd = apart_list(t/(x**2 + x + t), x)
>>> pfd
(1,
Poly(0, x, domain='ZZ[t']),
[(Poly(_w**2 + _w + t, _w, domain='ZZ[t]'),
Lambda(_a, -2*_a*t/(4*t - 1) - t/(4*t - 1)),
Lambda(_a, -_a + x),
1)])]

>>> assemble_partfrac_list(pfd)
RootSum(_w**2 + _w + t, Lambda(_a, (-2*_a*t/(4*t - 1) - t/(4*t - 1))/(-_a + x)))
```

This example is taken from Bronstein's original paper:

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
(Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*_a - 6), Lambda(_a, -_a + x), 2),
(Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1)])]

>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

`sympy.polys.partfrac.assemble_partfrac_list(partial_list)`

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

See also:

`apart` (page 725), `apart_list` (page 725)

## Examples

This example is taken from Bronstein's original paper:

```
>>> from sympy.polys.partfrac import apart_list, assemble_partfrac_list
>>> from sympy.abc import x, y

>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
```

```
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, -_a + x), 1),
 (Poly(_w**2 - 1, _w, domain='ZZ'), Lambda(_a, -3*a - 6), Lambda(_a, -_a + x), 2),
 (Poly(_w + 1, _w, domain='ZZ'), Lambda(_a, -4), Lambda(_a, -_a + x), 1))]

>>> assemble_partfrac_list(pfd)
-4/(x + 1) - 3/(x + 1)**2 - 9/(x - 1)**2 + 4/(x - 2)
```

If we happen to know some roots we can provide them easily inside the structure:

```
>>> pfd = apart_list(2/(x**2-2))
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w**2 - 2, _w, domain='ZZ'),
Lambda(_a, _a/2),
Lambda(_a, -_a + x),
1)])]

>>> pfda = assemble_partfrac_list(pfd)
>>> pfda
RootSum(_w**2 - 2, Lambda(_a, _a/(-_a + x)))/2

>>> pfda.doit()
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))

>>> from sympy import Dummy, Poly, Lambda, sqrt
>>> a = Dummy("a")
>>> pfd = (1, Poly(0, x, domain='ZZ'), [[sqrt(2), -sqrt(2)], Lambda(a, a/2), Lambda(a, -a + x), 1]])

>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

## Dispersion of Polynomials

```
sympy.polys.dispersion.dispersionset(p, q=None, *gens, **args)
Compute the dispersion set of two polynomials.
```

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion set  $J(f, g)$  is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines  $J(f) := J(f, f)$ .

See also:

`sympy.polys.dispersion.dispersion` (page 667)

## References

1. [ManWright94] (page 1246)
2. [Koepf98] (page 1246)
3. [Abramov71] (page 1246)

4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

### Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`sympy.polys.dispersion.dispersion(p, q=None, *gens, **args)`

Compute the *dispersion* of polynomials.

For two polynomials  $f(x)$  and  $g(x)$  with  $\deg f > 0$  and  $\deg g > 0$  the dispersion  $\text{dis}(f, g)$  is defined as:

$$\begin{aligned}\text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\}\end{aligned}$$

and for a single polynomial  $\text{dis}(f) := \text{dis}(f, f)$ . Note that we make the definition  $\max\{\} := -\infty$ .

See also:

`sympy.polys.dispersion.dispersionset` (page 668)

## References

- 1.[ManWright94] (page 1246)
- 2.[Koepf98] (page 1246)
- 3.[Abramov71] (page 1246)
- 4.[Man93] (page 1246)

[ManWright94] (page 1246), [Koepf98] (page 1246), [Abramov71] (page 1246), [Man93] (page 1246)

## Examples

```
>>> from sympy import poly
>>> from sympy.polys.dispersion import dispersion, dispersionset
>>> from sympy.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be  $-\infty$  as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from sympy import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from sympy.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

## AGCA - Algebraic Geometry and Commutative Algebra Module

### Introduction

Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightening calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.

George R. Kempf 1944 – 2002

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and Dedekind, it was realized that there is an intimate connection between properties of the set of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69] (page 1245). An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008] (page 1245).

This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

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

### Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

**Base Rings** Almost all computations in commutative algebra are relative to a “base ring”. (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys “domains” can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic  $y^2 = x^3$  at the origin, over  $\mathbb{Q}$ , you do:

```
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
>>> lr
[x, y, order=ilex]
-----
      3   2 \
\ - x + y
```

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary sympy objects into objects understood by the AGCA module (although in many cases this will be done automatically – for example the list was automatically turned into an ideal, and in the process the symbols  $x$  and  $y$  were automatically converted into other representations). For example:

```
>>> X, Y = lr.convert(x), lr.convert(y) ; X
      3   2 \
x + \ - x + y

>>> x**3 == y**2
False

>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space  $\mathbb{A}^3$ :

```
>>> ar = QQ.old_poly_ring(x, y, z); ar
[x, y, z]
```

For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

```
class sympy.polys.domains.ring.Ring
    Represents a ring domain.

    free_module(rank)
        Generate a free module of rank rank over self.
        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).free_module(2)
        QQ[x]**2

    ideal(*gens)
        Generate an ideal of self.
        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).ideal(x**2)
        <x**2>

    quotient_ring(e)
        Form a quotient ring of self.
        Here e can be an ideal or an iterable.
        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
```

```
>>> QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

```
sympy.polys.domains.polynomialring.PolynomialRing(domain_or_ring, symbols=None, order=None)
```

A class for representing multivariate polynomial rings.

```
class sympy.polys.domains.quotientring.QuotientRing(ring, ideal)
```

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>

>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- `ring` - the base ring
- `base_ideal` - the ideal used to form the quotient

**Modules, Ideals and their Elementary Properties** Let  $A$  be a ring. An  $A$ -module is a set  $M$ , together with two binary operations  $+$ :  $M \times M \rightarrow M$  and  $\times$ :  $R \times M \rightarrow M$  called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69] (page 1245). In this way modules are a direct generalisation of both vector spaces ( $A$  being a field) and abelian groups ( $A = \mathbb{Z}$ ). A *submodule* of the  $A$ -module  $M$  is a subset  $N \subset M$ , such that the binary operations restrict to  $N$ , and  $N$  becomes an  $A$ -module with these operations.

The ring  $A$  itself has a natural  $A$ -module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This  $A$ -module is also written as  $A$ . An  $A$ -submodule of  $A$  is called an *ideal* of  $A$ . Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient “elbow room” beyond talking only about ideals.

If  $M, N$  are  $A$ -modules, then there is a natural (componentwise)  $A$ -module structure on  $M \times N$ . Similarly there are  $A$ -module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many  $A$ -modules, with this  $A$ -module structure, is the finite biproduct in the category of all  $A$ -modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the  $A$ -module  $M$  is denoted  $M, M^2, M^3 \dots$ , or  $M^I$  for arbitrary index sets  $I$ .

An  $A$ -module  $M$  is called *free* if it is isomorphic to the  $A$ -module  $A^I$  for some (not necessarily finite) index set  $I$  (refer to the next section for a definition of isomorphism). The cardinality of  $I$  is called the *rank* of

$M$ ; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of  $\mathbb{A}^2$  we created above, together with a submodule:

```
>>> F = ar.free_module(4) ; F
        4
[x, y, z]

>>> S = F.submodule([1, x, x**2, x**3], [0, 1, 0, y]) ; S
      2   3
      \_
[1, x, x , x , [0, 1, 0, y]
```

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the `convert` method can be used to convert sympy/python objects into the internal AGCA representation (see detailed reference below).

Here is the detailed documentation of the classes for modules, free modules, and submodules:

**class sympy.polys.agca.modules.Module(ring)**  
Abstract base class for modules.

Do not instantiate - use ring explicit constructors instead:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

Attributes:

- `dtype` - type of elements
- `ring` - containing ring

Non-implemented methods:

- `submodule`
- `quotient_module`
- `is_zero`
- `is_submodule`
- `multiply_ideal`

The method `convert` likely needs to be changed in subclasses.

**contains(elem)**  
Return True if `elem` is an element of this module.

**convert(elem, M=None)**  
Convert `elem` into internal representation of this module.  
If `M` is not None, it should be a module containing it.

**identity\_hom()**  
Return the identity homomorphism on `self`.

**is\_submodule(other)**  
Returns True if `other` is a submodule of `self`.

**is\_zero()**  
Returns True if `self` is a zero module.

```
multiply_ideal(other)
    Multiply self by the ideal other.

quotient_module(other)
    Generate a quotient module.

submodule(*gens)
    Generate a submodule.

subset(other)
    Returns True if other is a subset of self.
    >>> from sympy.abc import x
    >>> from sympy import QQ
    >>> F = QQ.old_poly_ring(x).free_module(2)
    >>> F.subset([(1, x), (x, 2)])
    True
    >>> F.subset([(1/x, x), (x, 2)])
    False

class sympy.polys.agca.modules.FreeModule(ring, rank)
    Abstract base class for free modules.

Additional attributes:
    • rank - rank of the free module

Non-implemented methods:
    • submodule

basis()
    Return a set of basis elements.
    >>> from sympy.abc import x
    >>> from sympy import QQ
    >>> QQ.old_poly_ring(x).free_module(3).basis()
    ([1, 0, 0], [0, 1, 0], [0, 0, 1])

convert(elem, M=None)
    Convert elem into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.
    >>> from sympy.abc import x
    >>> from sympy import QQ
    >>> F = QQ.old_poly_ring(x).free_module(2)
    >>> F.convert([1, 0])
    [1, 0]

dtype
    alias of FreeModuleElement (page 740)

identity_hom()
    Return the identity homomorphism on self.
    >>> from sympy.abc import x
    >>> from sympy import QQ
    >>> QQ.old_poly_ring(x).free_module(2).identity_hom()
    Matrix([
        [1, 0], : QQ[x]**2 -> QQ[x]**2
        [0, 1]])
```

**is\_submodule(*other*)**

Returns True if *other* is a submodule of *self*.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> F.is_submodule(F)
True
>>> F.is_submodule(M)
True
>>> M.is_submodule(F)
False
```

**is\_zero()**

Returns True if *self* is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

**multiply\_ideal(*other*)**

Multiply *self* by the ideal *other*.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

**quotient\_module(*submodule*)**

Return a quotient module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more concisely, using the overloaded division operator:

```
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

**class sympy.polys.agca.modules.SubModule(*gens*, *container*)**

Base class for submodules.

Attributes:

- container* - containing module
- gens* - generators (subset of containing module)
- rank* - rank of containing module

Non-implemented methods:

- \_contains
- \_syzygies
- \_in\_terms\_of\_generators
- \_intersect
- \_module\_quotient

Methods that likely need change in subclasses:

- reduce\_element

`convert(elem, M=None)`

Convert `elem` into the internal representation.

Mostly called implicitly.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

`identity_hom()`

Return the identity homomorphism on `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_hom()
Matrix([
[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

`in_terms_of_generators(e)`

Express element `e` of `self` in terms of the generators.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([1, 0], [1, 1])
>>> M.in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

`inclusion_hom()`

Return a homomorphism representing the inclusion map of `self`.

That is, the natural map from `self` to `self.container`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_hom()
Matrix([
[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

`intersect(other, **options)`

Returns the intersection of `self` with submodule `other`.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is `relations=True`, in which case the function will return a triple (`res`, `rela`, `relb`), where `res` is the intersection module, and `rela` and `relb` are lists of coefficient vectors, expressing the generators of `res` in terms of the generators of `self` (`rela`) and `other` (`relb`).

```
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
(<[x*y, x*y]>, [(y,)], [(x,)])
```

The above result says: the intersection module is generated by the single element  $(-xy, -xy) = -y(x, x) = -x(y, y)$ , where  $(x, x)$  and  $(y, y)$  respectively are the unique generators of the two modules being intersected.

#### `is_full_module()`

Return True if `self` is the entire free module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

#### `is_submodule(other)`

Returns True if `other` is a submodule of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

#### `is_zero()`

Return True if `self` is a zero module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_zero()
False
>>> F.submodule([0, 0]).is_zero()
True
```

#### `module_quotient(other, **options)`

Returns the module quotient of `self` by submodule `other`.

That is, if `self` is the module  $M$  and `other` is  $N$ , then return the ideal  $\{f \in R | fN \subset M\}$ .

```
>>> from sympy import QQ
>>> from sympy.abc import x, y
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is `relations=True`, which may only be passed if `other` is principal. In this case the function will return a pair (`res, rel`) where `res` is the ideal, and `rel` is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of `other` in terms of generators of `self`.

```
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element  $y$ , and that  $y(x, x) = 1(xy, xy)$ ,  $(x, x)$  and  $(xy, xy)$  being the generators of  $T$  and  $S$ , respectively.

### `multiply_ideal(I)`

Multiply `self` by the ideal `I`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

### `quotient_module(other, **opts)`

Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<[x, 1] + <[x**2, x]>>
```

Or more concisely, using the overloaded division operator:

```
>>> F.submodule([x, 1]) / [(x**2, x)]
<[x, 1] + <[x**2, x]>>
```

### `reduce_element(x)`

Reduce the element `x` of our ring modulo the ideal `self`.

Here “reduce” has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

### `submodule(*gens)`

Generate a submodule.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([x, 1])
```

```
>>> M.submodule([x**2, x])
<[x**2, x]>
```

### syzygy\_module(\*\*opts)

Compute the syzygy module of the generators of `self`.

Suppose  $M$  is generated by  $f_1, \dots, f_n$  over the ring  $R$ . Consider the homomorphism  $\phi : R^n \rightarrow M$ , given by sending  $(r_1, \dots, r_n) \mapsto r_1 f_1 + \dots + r_n f_n$ . The syzygy module is defined to be the kernel of  $\phi$ .

The syzygy module is zero iff the generators generate freely a free submodule:

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([1, 0], [1, 1]).syzygy_module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y, 2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```

### union(other)

Returns the module generated by the union of `self` and `other`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True
```

## class sympy.polys.agca.modules.FreeModuleElement(module, data)

Element of a free module. Data stored as a tuple.

Ideals are created very similarly to modules. For example, let's verify that the nodal cubic is indeed singular at the origin:

```
>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False

>>> I == lr.ideal(y)
False
```

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of  $x$  and  $y$  must be generators.

This is the detailed documentation of the class `Ideal`. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

## class sympy.polys.agca.ideals.Ideal(ring)

Abstract base class for ideals.

Do not instantiate - use explicit constructors in the ring class instead:

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

```
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>
```

#### Attributes

- `ring` - the ring this ideal belongs to

#### Non-implemented methods:

- `_contains_elem`
- `_contains_ideal`
- `_quotient`
- `_intersect`
- `_union`
- `_product`
- `is_whole_ring`
- `is_zero`
- `is_prime, is_maximal, is_primary, is_radical`
- `is_principal`
- `height, depth`
- `radical`

#### Methods that likely should be overridden in subclasses:

- `reduce_element`

##### `contains(elem)`

Return True if `elem` is an element of this ideal.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

##### `depth()`

Compute the depth of `self`.

##### `height()`

Compute the height of `self`.

##### `intersect(J)`

Compute the intersection of `self` with ideal `J`.

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x).intersect(R.ideal(y))
<x*y>
```

##### `is_maximal()`

Return True if `self` is a maximal ideal.

`is_primary()`  
Return True if `self` is a primary ideal.

`is_prime()`  
Return True if `self` is a prime ideal.

`is_principal()`  
Return True if `self` is a principal ideal.

`is_radical()`  
Return True if `self` is a radical ideal.

`is_whole_ring()`  
Return True if `self` is the whole ring.

`is_zero()`  
Return True if `self` is the zero ideal.

`product(J)`  
Compute the ideal product of `self` and `J`.

That is, compute the ideal generated by products  $xy$ , for  $x$  an element of `self` and  $y \in J$ .

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).ideal(y))
<x*y>
```

`quotient(J, **opts)`  
Compute the ideal quotient of `self` by `J`.

That is, if `self` is the ideal  $I$ , compute the set  $I : J = \{x \in R | xJ \subset I\}$ .

```
>>> from sympy.abc import x, y
>>> from sympy import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

`radical()`  
Compute the radical of `self`.

`reduce_element(x)`  
Reduce the element `x` of our ring modulo the ideal `self`.

Here “reduce” has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

`saturate(J)`  
Compute the ideal saturation of `self` by `J`.

That is, if `self` is the ideal  $I$ , compute the set  $I : J^\infty = \{x \in R | xJ^n \subset I \text{ for some } n\}$ .

`subset(other)`  
Returns True if `other` is a subset of `self`.

Here `other` may be an ideal.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
```

```

>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True

union(J)
Compute the ideal generated by the union of self and J.

>>> from sympy.abc import x
>>> from sympy import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).ideal((x+1)**2)) == QQ.old_poly_ring(x)
True

```

If  $M$  is an  $A$ -module and  $N$  is an  $A$ -submodule, we can define two elements  $x$  and  $y$  of  $M$  to be equivalent if  $x - y \in N$ . The set of equivalence classes is written  $M/N$ , and has a natural  $A$ -module structure. This is called the quotient module of  $M$  by  $N$ . If  $K$  is a submodule of  $M$  containing  $N$ , then  $K/N$  is in a natural way a submodule of  $M/N$ . Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

`class sympy.polys.agca.modules.QuotientModule(ring, base, submodule)`  
 Class for quotient modules.

Do not instantiate this directly. For subquotients, see the `SubQuotientModule` class.

Attributes:

- `base` - the base module we are a quotient of
- `killed_module` - the submodule used to form the quotient
- `rank` of the base

`convert(elem, M=None)`  
 Convert `elem` into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

```

>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> F.convert([1, 0])
[1, 0] + <[1, 2], [1, x]>

```

`dtype`  
 alias of `QuotientModuleElement` (page 745)

`identity_hom()`  
 Return the identity homomorphism on `self`.

```

>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.identity_hom()
Matrix([
[1, 0], : QQ[x]**2<[1, 2], [1, x]> -> QQ[x]**2<[1, 2], [1, x]>
[0, 1]])

```

`is_submodule(other)`  
 Return True if `other` is a submodule of `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> S = Q.submodule([1, 0])
>>> Q.is_submodule(S)
True
>>> S.is_submodule(Q)
False
```

### is\_zero()

Return True if `self` is a zero module.

This happens if and only if the base module is the same as the submodule being killed.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> (F/[(1, 0)]).is_zero()
False
>>> (F/[(1, 0), (0, 1)]).is_zero()
True
```

### quotient\_hom()

Return the quotient homomorphism to `self`.

That is, return a homomorphism representing the natural map from `self.base` to `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.quotient_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2<[1, 2], [1, x]>
[0, 1]])
```

### submodule(\*gens, \*\*opts)

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> Q.submodule([x, 0])
<[x, 0] + <x, x>>
```

## class sympy.polys.agca.modules.SubQuotientModule(gens, container, \*\*opts)

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient\_module constructing methods:

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[(1, 0)]
>>> S/[(1, 0)] == Q.submodule([5, x])
True
```

Attributes:

- `base` - base module we are quotient of
- `killed_module` - submodule used to form the quotient

`is_full_module()`

Return True if `self` is the entire free module.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

`quotient_hom()`

Return the quotient homomorphism to `self`.

That is, return the natural map from `self.base` to `self`.

```
>>> from sympy.abc import x
>>> from sympy import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [(1, x)]).submodule([1, 0])
>>> M.quotient_hom()
Matrix([
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>>
[0, 1]])
```

`class sympy.polys.agca.modules.QuotientModuleElement(module, data)`  
Element of a quotient module.

`eq(d1, d2)`  
Equality comparison.

**Module Homomorphisms and Syzygies** Let  $M$  and  $N$  be  $A$ -modules. A mapping  $f : M \rightarrow N$  satisfying various obvious properties (see [Atiyah69] (page 1245)) is called an  $A$ -module homomorphism. In this case  $M$  is called the *domain* and  $N$  the *codomain*. The set  $\{x \in M | f(x) = 0\}$  is called the *kernel*  $\ker(f)$ , whereas the set  $\{f(x) | x \in M\}$  is called the *image*  $\text{im}(f)$ . The kernel is a submodule of  $M$ , the image is a submodule of  $N$ . The homomorphism  $f$  is injective if and only if  $\ker(f) = 0$  and surjective if and only if  $\text{im}(f) = N$ . A bijective homomorphism is called an *isomorphism*. Equivalently,  $\ker(f) = 0$  and  $\text{im}(f) = N$ . (A related notion, which currently has no special name in the AGCA module, is that of the *cokernel*,  $\text{coker}(f) = N/\text{im}(f)$ .)

Suppose now  $M$  is an  $A$ -module.  $M$  is called *finitely generated* if there exists a surjective homomorphism  $A^n \rightarrow M$  for some  $n$ . If such a morphism  $f$  is chosen, the images of the standard basis of  $A^n$  are called the *generators* of  $M$ . The module  $\ker(f)$  is called *syzygy module* with respect to the generators. A module is called *finitely presented* if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that  $f : M \rightarrow N$  is an arbitrary morphism of  $A$ -modules. Then if  $K$  is a submodule of  $M$ ,  $f$  naturally defines a new homomorphism  $g : K \rightarrow N$  (via  $g(x) = f(x)$ ), called the *restriction* of  $f$  to  $K$ . If now  $K$  contained in the kernel of  $f$ , then moreover  $f$  defines in a natural homomorphism  $g : M/K \rightarrow N$  (same formula as above!).

and we say that  $f$  descends to  $M/K$ . Similarly, if  $L$  is a submodule of  $N$ , there is a natural homomorphism  $g : M \rightarrow N/L$ , we say that  $g$  factors through  $f$ . Finally, if now  $L$  contains the image of  $f$ , then there is a natural homomorphism  $g : M \rightarrow L$  (defined, again, by the same formula), and we say  $g$  is obtained from  $f$  by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given  $g$ , one can always (non-uniquely) find  $f$  such that  $g$  is obtained from  $f$  in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module  $A^n$  to any module  $M$  is precisely the same as giving  $n$  elements of  $M$  (the images of the standard basis), and giving an element of a free module  $A^m$  is precisely the same as giving  $m$  elements of  $A$ . Hence a homomorphism of free modules  $A^n \rightarrow A^m$  can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions `restrict_domain` etc. of the class `Homomorphism` can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function `homomorphism` to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

```
sympy.polys.agca.homomorphisms.homomorphism(domain, codomain, matrix)
```

Create a homomorphism object.

This function tries to build a homomorphism from `domain` to `codomain` via the matrix `matrix`.

## Examples

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)
```

If `domain` is a free module generated by  $e_1, \dots, e_n$ , then `matrix` should be an  $n$ -element iterable  $(b_1, \dots, b_n)$  where the  $b_i$  are elements of `codomain`. The constructed homomorphism is the unique homomorphism sending  $e_i$  to  $b_i$ .

```
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([
[1, x**2], : QQ[x]**2 -> QQ[x]**2
[x, 0]])
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If `domain` is a submodule of a free module, them `matrix` determines a homomoprism from the containing free module to `codomain`, and the homomorphism returned is obtained by restriction to `domain`.

```
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([
```

```
[1, x**2], : <[1, 0], [0, x]> -> QQ[x]**2
[x,      0]])
```

If `domain` is a (sub)quotient  $N/K$ , then `matrix` determines a homomorphism from  $N$  to `codomain`. If the kernel contains  $K$ , this homomorphism descends to `domain` and is returned; otherwise an exception is raised.

```
>>> homomorphism(S/[(1, 0)], T, [0, [x**2, 0]])
Matrix([
[0, x**2], : <[1, 0] + <[1, 0]>, [0, x] + <[1, 0]>, [1, 0] + <[1, 0]>> -> QQ[x]**2
[0,      0]])
>>> homomorphism(S/[(0, x)], T, [0, [x**2, 0]])
Traceback (most recent call last):
...
ValueError: kernel <[1, 0], [0, 0]> must contain sm, got <[0,x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

```
class sympy.polys.agca.homomorphisms.ModuleHomomorphism(domain, codomain)
Abstract base class for module homomorphisms. Do not instantiate.
```

Instead, use the `homomorphism` function:

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

Attributes:

- `ring` - the ring over which we are considering modules
- `domain` - the domain module
- `codomain` - the codomain module
- `_ker` - cached kernel
- `_img` - cached image

Non-implemented methods:

- `_kernel`
- `_image`
- `_restrict_domain`
- `_restrict_codomain`
- `_quotient_domain`
- `_quotient_codomain`
- `_apply`
- `_mul_scalar`
- `_compose`

• `add`

`image()`

Compute the image of `self`.

That is, if `self` is the homomorphism  $\phi : M \rightarrow N$ , then compute  $im(\phi) = \{\phi(x) | x \in M\}$ . This is a submodule of  $N$ .

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1, 0])
True
```

`is_injective()`

Return True if `self` is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

`is_isomorphism()`

Return True if `self` is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, `self` is both injective and surjective.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

`is_surjective()`

Return True if `self` is surjective.

That is, check if every element of the codomain has at least one preimage.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
```

```
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True

is_zero()
Return True if self is a zero morphism.

That is, check if every element of the domain is mapped to zero under self.

>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
>>> h.quotient_codomain(h.image()).is_zero()
True

kernel()
Compute the kernel of self.

That is, if self is the homomorphism  $\phi : M \rightarrow N$ , then compute  $\ker(\phi) = \{x \in M | \phi(x) = 0\}$ . This is a submodule of  $M$ .

>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>

quotient_codomain(sm)
Return self with codomain replaced by codomain/sm.

Here sm must be a submodule of self.codomain.

>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]]))

This is the same as composing with the quotient map on the left:
```

```
>>> (F/[(1, 1)]).quotient_hom() * h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2<[1, 1]>
[0, 0]])
```

**quotient\_domain(*sm*)**  
Return `self` with domain replaced by `domain/sm`.

Here `sm` must be a submodule of `self.kernel()`.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_domain(F.submodule([-x, 1]))
Matrix([
[1, x], : QQ[x]**2<[-x, 1]> -> QQ[x]**2
[0, 0]])
```

**restrict\_codomain(*sm*)**  
Return `self`, with codomain restricted to to `sm`.

Here `sm` has to be a submodule of `self.codomain` containing the image.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_codomain(F.submodule([1, 0]))
Matrix([
[1, x], : QQ[x]**2 -> <[1, 0]>
[0, 0]])
```

**restrict\_domain(*sm*)**  
Return `self`, with the domain restricted to `sm`.

Here `sm` has to be a submodule of `self.domain`.

```
>>> from sympy import QQ
>>> from sympy.abc import x
>>> from sympy.polys.agca import homomorphism

>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
```

---

```
[0, 0]])
>>> h.restrict_domain(F.submodule([1, 0]))
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

## Internals of the Polynomial Manipulation Module

The implementation of the polynomials module is structured internally in “levels”. There are four levels, called L0, L1, L2 and L3. The levels three and four contain the user-facing functionality and were described in the previous section. This section focuses on levels zero and one.

Level zero provides core polynomial manipulation functionality with C-like, low-level interfaces. Level one wraps this low-level functionality into object oriented structures. These are *not* the classes seen by the user, but rather classes used internally throughout the polys module.

There is one additional complication in the implementation. This comes from the fact that all polynomial manipulations are relative to a *ground domain*. For example, when factoring a polynomial like  $x^{10} - 1$ , one has to decide what ring the coefficients are supposed to belong to, or less trivially, what coefficients are allowed to appear in the factorization. This choice of coefficients is called a ground domain. Typical choices include the integers  $\mathbb{Z}$ , the rational numbers  $\mathbb{Q}$  or various related rings and fields. But it is perfectly legitimate (although in this case uninteresting) to factorize over polynomial rings such as  $k[Y]$ , where  $k$  is some fixed field.

Thus the polynomial manipulation algorithms (both complicated ones like factoring, and simpler ones like addition or multiplication) have to rely on other code to manipulate the coefficients. In the polynomial manipulation module, such code is encapsulated in so-called “domains”. A domain is basically a factory object: it takes various representations of data, and converts them into objects with unified interface. Every object created by a domain has to implement the arithmetic operations  $+$ ,  $-$  and  $\times$ . Other operations are accessed through the domain, e.g. as in `ZZ.quo(ZZ(4), ZZ(2))`.

Note that there is some amount of *circularity*: the polynomial ring domains use the level one classes, the level one classes use the level zero functions, and level zero functions use domains. It is possible, in principle, but not in the current implementation, to work in rings like  $k[X][Y]$ . This would create even more layers. For this reason, working in the isomorphic ring  $k[X, Y]$  is preferred.

### Domains

Here we document the various implemented ground domains. There are three types: abstract domains, concrete domains, and “implementation domains”. Abstract domains cannot be (usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. `ZZ` always refers to the most efficient implementation of the integer ring available.

## Abstract Domains

```
class sympy.polys.domains.domain.Domain
    Represents an abstract domain.

    abs(a)
        Absolute value of a, implies __abs__.

    add(a, b)
        Sum of a and b, implies __add__.

    algebraic_field(*extension)
        Returns an algebraic field, i.e. K(α, ...).

    almosteq(a, b, tolerance=None)
        Check if a and b are almost equal.

    characteristic()
        Return the characteristic of this domain.

    cofactors(a, b)
        Returns GCD and cofactors of a and b.

    convert(element, base=None)
        Convert element to self.dtype.

    convert_from(element, base)
        Convert element to self.dtype given the base domain.

    denom(a)
        Returns denominator of a.

    div(a, b)
        Division of a and b, implies something.

    evalf(prec=None, **options)
        Returns numerical approximation of a.

    exquo(a, b)
        Exact quotient of a and b, implies something.

    frac_field(*symbols, **kwargs)
        Returns a fraction field, i.e. K(X).

    from_AlgebraicField(K1, a, K0)
        Convert an algebraic number to dtype.

    from_ComplexField(K1, a, K0)
        Convert a complex element to dtype.

    from_ExpressionDomain(K1, a, K0)
        Convert a EX object to dtype.

    from_FF_gmpy(K1, a, K0)
        Convert ModularInteger(mpz) to dtype.

    from_FF_python(K1, a, K0)
        Convert ModularInteger(int) to dtype.

    from_FractionField(K1, a, K0)
        Convert a rational function to dtype.

    from_GlobalPolynomialRing(K1, a, K0)
        Convert a polynomial to dtype.
```

```
from_PolynomialRing(K1, a, K0)
    Convert a polynomial to dtype.

from_QQ_gmpy(K1, a, K0)
    Convert a GMPY mpq object to dtype.

from_QQ_python(K1, a, K0)
    Convert a Python Fraction object to dtype.

from_RealField(K1, a, K0)
    Convert a real element object to dtype.

from_ZZ_gmpy(K1, a, K0)
    Convert a GMPY mpz object to dtype.

from_ZZ_python(K1, a, K0)
    Convert a Python int object to dtype.

from_sympy(a)
    Convert a SymPy object to dtype.

gcd(a, b)
    Returns GCD of a and b.

gcdex(a, b)
    Extended GCD of a and b.

get_exact()
    Returns an exact domain associated with self.

get_field()
    Returns a field associated with self.

get_ring()
    Returns a ring associated with self.

half_gcdex(a, b)
    Half extended GCD of a and b.

inject(*symbols)
    Inject generators into this domain.

invert(a, b)
    Returns inversion of a mod b, implies something.

is_negative(a)
    Returns True if a is negative.

is_nonnegative(a)
    Returns True if a is non-negative.

is_nonpositive(a)
    Returns True if a is non-positive.

is_one(a)
    Returns True if a is one.

is_positive(a)
    Returns True if a is positive.

is_zero(a)
    Returns True if a is zero.
```

**lcm(a, b)**  
Returns LCM of **a** and **b**.

**log(a, b)**  
Returns b-base logarithm of **a**.

**map(seq)**  
Rersively apply **self** to all elements of **seq**.

**mul(a, b)**  
Product of **a** and **b**, implies **\_\_mul\_\_**.

**n(a, prec=None, \*\*options)**  
Returns numerical approximation of **a**.

**neg(a)**  
Returns **a** negated, implies **\_\_neg\_\_**.

**numer(a)**  
Returns numerator of **a**.

**of\_type(element)**  
Check if **a** is of type **dtype**.

**old\_frac\_field(\*symbols, \*\*kwargs)**  
Returns a fraction field, i.e.  $K(X)$ .

**old\_poly\_ring(\*symbols, \*\*kwargs)**  
Returns a polynomial ring, i.e.  $K[X]$ .

**poly\_ring(\*symbols, \*\*kwargs)**  
Returns a polynomial ring, i.e.  $K[X]$ .

**pos(a)**  
Returns **a** positive, implies **\_\_pos\_\_**.

**pow(a, b)**  
Raise **a** to power **b**, implies **\_\_pow\_\_**.

**quo(a, b)**  
Quotient of **a** and **b**, implies something.

**rem(a, b)**  
Remainder of **a** and **b**, implies **\_\_mod\_\_**.

**revert(a)**  
Returns  $a^{**(-1)}$  if possible.

**sqrt(a)**  
Returns square root of **a**.

**sub(a, b)**  
Difference of **a** and **b**, implies **\_\_sub\_\_**.

**to\_sympy(a)**  
Convert **a** to a SymPy object.

**unify(K0, K1, symbols=None)**  
Construct a minimal domain that contains elements of **K0** and **K1**.  
Known domains (from smallest to largest):

- GF(p)
- ZZ

- QQ
- RR(prec, tol)
- CC(prec, tol)
- ALG(a, b, c)
- K[x, y, z]
- K(x, y, z)
- EX

class sympy.polys.domains.field.Field  
Represents a field domain.

div(a, b)  
Division of a and b, implies \_\_div\_\_.

exquo(a, b)  
Exact quotient of a and b, implies \_\_div\_\_.

gcd(a, b)  
Returns GCD of a and b.  
This definition of GCD over fields allows to clear denominators in primitive().

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, gcd, primitive
>>> from sympy.abc import x

>>> QQ.gcd(QQ(2, 3), QQ(4, 9))
2/9
>>> gcd(S(2)/3, S(4)/9)
2/9
>>> primitive(2*x/3 + S(4)/9)
(2/9, 3*x + 2)
```

get\_field()  
Returns a field associated with self.

get\_ring()  
Returns a ring associated with self.

lcm(a, b)  
Returns LCM of a and b.

```
>>> from sympy.polys.domains import QQ
>>> from sympy import S, lcm

>>> QQ.lcm(QQ(2, 3), QQ(4, 9))
4/3
>>> lcm(S(2)/3, S(4)/9)
4/3
```

quo(a, b)  
Quotient of a and b, implies \_\_div\_\_.

rem(a, b)  
Remainder of a and b, implies nothing.

revert(a)  
Returns a\*\*(-1) if possible.

```
class sympy.polys.domains.ring.Ring
    Represents a ring domain.

    denom(a)
        Returns denominator of a.

    div(a, b)
        Division of a and b, implies __divmod__.

    exquo(a, b)
        Exact quotient of a and b, implies __floordiv__.

    free_module(rank)
        Generate a free module of rank rank over self.

        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).free_module(2)
        QQ[x]**2

    get_ring()
        Returns a ring associated with self.

    ideal(*gens)
        Generate an ideal of self.

        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).ideal(x**2)
        <x**2>

    invert(a, b)
        Returns inversion of a mod b.

    numer(a)
        Returns numerator of a.

    quo(a, b)
        Quotient of a and b, implies __floordiv__.

    quotient_ring(e)
        Form a quotient ring of self.

        Here e can be an ideal or an iterable.

        >>> from sympy.abc import x
        >>> from sympy import QQ
        >>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
        QQ[x]/<x**2>
        >>> QQ.old_poly_ring(x).quotient_ring([x**2])
        QQ[x]/<x**2>

    The division operator has been overloaded for this:

    >>> QQ.old_poly_ring(x)/[x**2]
    QQ[x]/<x**2>

    rem(a, b)
        Remainder of a and b, implies __mod__.

    revert(a)
        Returns a $\star(-1)$  if possible.
```

```
class sympy.polys.domains.simpledomain.SimpleDomain
    Base class for simple domains, e.g. ZZ, QQ.

    inject(*gens)
        Inject generators into this domain.

class sympy.polys.domains.compositedomain.CompositeDomain
    Base class for composite domains, e.g. ZZ[x], ZZ(X).

    inject(*symbols)
        Inject generators into this domain.
```

### Concrete Domains

```
class sympy.polys.domains.FiniteField(mod, dom=None, symmetric=True)
    General class for finite fields.

    characteristic()
        Return the characteristic of this domain.

    from_FF_gmpy(K1, a, K0=None)
        Convert ModularInteger(mpz) to dtype.

    from_FF_python(K1, a, K0=None)
        Convert ModularInteger(int) to dtype.

    from_QQ_gmpy(K1, a, K0=None)
        Convert GMPY's mpq to dtype.

    from_QQ_python(K1, a, K0=None)
        Convert Python's Fraction to dtype.

    from_RealField(K1, a, K0)
        Convert mpmath's mpf to dtype.

    from_ZZ_gmpy(K1, a, K0=None)
        Convert GMPY's mpz to dtype.

    from_ZZ_python(K1, a, K0=None)
        Convert Python's int to dtype.

    from_sympy(a)
        Convert SymPy's Integer to SymPy's Integer.

    get_field()
        Returns a field associated with self.

    to_sympy(a)
        Convert a to a SymPy object.

class sympy.polys.domains.IntegerRing
    General class for integer rings.

    algebraic_field(*extension)
        Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .

    from_AlgebraicField(K1, a, K0)
        Convert a ANP object to dtype.

    get_field()
        Returns a field associated with self.

    log(a, b)
        Returns b-base logarithm of a.
```

```
class sympy.polys.domains.PolynomialRing(domain_or_ring, symbols=None, order=None)
    A class for representing multivariate polynomial rings.

    factorial(a)
        Returns factorial of  $a$ .

    from_AlgebraicField(K1, a, K0)
        Convert an algebraic number to  $\text{dtype}$ .

    from_FractionField(K1, a, K0)
        Convert a rational function to  $\text{dtype}$ .

    from_PolynomialRing(K1, a, K0)
        Convert a polynomial to  $\text{dtype}$ .

    from_QQ_gmpy(K1, a, K0)
        Convert a GMPY  $mpq$  object to  $\text{dtype}$ .

    from_QQ_python(K1, a, K0)
        Convert a Python  $Fraction$  object to  $\text{dtype}$ .

    from_RealField(K1, a, K0)
        Convert a mpmath  $mpf$  object to  $\text{dtype}$ .

    from_ZZ_gmpy(K1, a, K0)
        Convert a GMPY  $mpz$  object to  $\text{dtype}$ .

    from_ZZ_python(K1, a, K0)
        Convert a Python  $int$  object to  $\text{dtype}$ .

    from_sympy(a)
        Convert SymPy's expression to  $\text{dtype}$ .

    gcd(a, b)
        Returns GCD of  $a$  and  $b$ .

    gcdex(a, b)
        Extended GCD of  $a$  and  $b$ .

    get_field()
        Returns a field associated with  $self$ .

    is_negative(a)
        Returns True if  $LC(a)$  is negative.

    is_nonnegative(a)
        Returns True if  $LC(a)$  is non-negative.

    is_nonpositive(a)
        Returns True if  $LC(a)$  is non-positive.

    is_positive(a)
        Returns True if  $LC(a)$  is positive.

    lcm(a, b)
        Returns LCM of  $a$  and  $b$ .

    to_sympy(a)
        Convert  $a$  to a SymPy object.

class sympy.polys.domains.RationalField
    General class for rational fields.
```

```
algebraic_field(*extension)
    Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .
from_AlgebraicField(K1, a, K0)
    Convert a ANP object to dtype.

class sympy.polys.domains.AlgebraicField(dom, *ext)
    A class for representing algebraic number fields.

algebraic_field(*extension)
    Returns an algebraic field, i.e.  $\mathbb{Q}(\alpha, \dots)$ .
denom(a)
    Returns denominator of a.

dtype
    alias of ANP (page 768)

from_QQ_gmpy(K1, a, K0)
    Convert a GMPY mpq object to dtype.

from_QQ_python(K1, a, K0)
    Convert a Python Fraction object to dtype.

from_RealField(K1, a, K0)
    Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(K1, a, K0)
    Convert a GMPY mpz object to dtype.

from_ZZ_python(K1, a, K0)
    Convert a Python int object to dtype.

from_sympy(a)
    Convert SymPy's expression to dtype.

get_ring()
    Returns a ring associated with self.

is_negative(a)
    Returns True if a is negative.

is_nonnegative(a)
    Returns True if a is non-negative.

is_nonpositive(a)
    Returns True if a is non-positive.

is_positive(a)
    Returns True if a is positive.

numer(a)
    Returns numerator of a.

to_sympy(a)
    Convert a to a SymPy object.

class sympy.polys.domains.FractionField(domain_or_field, symbols=None, order=None)
    A class for representing multivariate rational function fields.

denom(a)
    Returns denominator of a.
```

```
factorial(a)
    Returns factorial of  $a$ .

from_AlgebraicField(K1, a, K0)
    Convert an algebraic number to dtype.

from_FractionField(K1, a, K0)
    Convert a rational function to dtype.

from_PolynomialRing(K1, a, K0)
    Convert a polynomial to dtype.

from_QQ_gmpy(K1, a, K0)
    Convert a GMPY mpq object to dtype.

from_QQ_python(K1, a, K0)
    Convert a Python Fraction object to dtype.

from_RealField(K1, a, K0)
    Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(K1, a, K0)
    Convert a GMPY mpz object to dtype.

from_ZZ_python(K1, a, K0)
    Convert a Python int object to dtype.

from_sympy(a)
    Convert SymPy's expression to dtype.

get_ring()
    Returns a field associated with self.

is_negative(a)
    Returns True if  $LC(a)$  is negative.

is_nonnegative(a)
    Returns True if  $LC(a)$  is non-negative.

is_nonpositive(a)
    Returns True if  $LC(a)$  is non-positive.

is_positive(a)
    Returns True if  $LC(a)$  is positive.

numer(a)
    Returns numerator of  $a$ .

to_sympy(a)
    Convert  $a$  to a SymPy object.

class sympy.polys.domains.RealField(prec=53, dps=None, tol=None)
    Real numbers up to the given precision.

almosteq(a, b, tolerance=None)
    Check if  $a$  and  $b$  are almost equal.

from_sympy(expr)
    Convert SymPy's number to dtype.

gcd(a, b)
    Returns GCD of  $a$  and  $b$ .
```

```
get_exact()
    Returns an exact domain associated with self.

get_ring()
    Returns a ring associated with self.

lcm(a, b)
    Returns LCM of a and b.

to_rational(element, limit=True)
    Convert a real number to rational number.

to_sympy(element)
    Convert element to SymPy number.

class sympy.polys.domains.ExpressionDomain
    A class for arbitrary expressions.

class Expression(ex)
    An arbitrary expression.

ExpressionDomain.denom(a)
    Returns denominator of a.

ExpressionDomain.dtype
    alias of Expression (page 761)

ExpressionDomain.from(ExpressionDomain(K1, a, K0)
    Convert a EX object to dtype.

ExpressionDomain.from(FractionField(K1, a, K0)
    Convert a DMF object to dtype.

ExpressionDomain.from(PolynomialRing(K1, a, K0)
    Convert a DMP object to dtype.

ExpressionDomain.from(QQ_gmpy(K1, a, K0)
    Convert a GMPY mpq object to dtype.

ExpressionDomain.from(QQ_python(K1, a, K0)
    Convert a Python Fraction object to dtype.

ExpressionDomain.from(RealField(K1, a, K0)
    Convert a mpmath mpf object to dtype.

ExpressionDomain.from(ZZ_gmpy(K1, a, K0)
    Convert a GMPY mpz object to dtype.

ExpressionDomain.from(ZZ_python(K1, a, K0)
    Convert a Python int object to dtype.

ExpressionDomain.from_sympy(a)
    Convert SymPy's expression to dtype.

ExpressionDomain.get_field()
    Returns a field associated with self.

ExpressionDomain.get_ring()
    Returns a ring associated with self.

ExpressionDomain.is_negative(a)
    Returns True if a is negative.
```

```
ExpressionDomain.is_nonnegative(a)
    Returns True if a is non-negative.

ExpressionDomain.is_nonpositive(a)
    Returns True if a is non-positive.

ExpressionDomain.is_positive(a)
    Returns True if a is positive.

ExpressionDomain.numer(a)
    Returns numerator of a.

ExpressionDomain.to_sympy(a)
    Convert a to a SymPy object.
```

## Implementation Domains

```
class sympy.polys.domains.PythonFiniteField(mod, symmetric=True)
    Finite field based on Python's integers.

class sympy.polys.domains.GMPYFiniteField(mod, symmetric=True)
    Finite field based on GMPY integers.

class sympy.polys.domains.PythonIntegerRing
    Integer ring based on Python's int type.

class sympy.polys.domains.GMPYIntegerRing
    Integer ring based on GMPY's mpz type.

class sympy.polys.domains.PythonRationalField
    Rational field based on Python rational number type.

class sympy.polys.domains.GMPYRationalField
    Rational field based on GMPY mpq class.
```

## Level One

```
class sympy.polys.polyclasses.DMP(rep, dom, lev=None, ring=None)
    Dense Multivariate Polynomials over K.

    LC(f)
        Returns the leading coefficient of f.

    TC(f)
        Returns the trailing coefficient of f.

    abs(f)
        Make all coefficients in f positive.

    add(f, g)
        Add two multivariate polynomials f and g.

    add_ground(f, c)
        Add an element of the ground domain to f.

    all_coeffs(f)
        Returns all coefficients from f.

    all_monoms(f)
        Returns all monomials from f.
```

**all\_terms(f)**  
Returns all terms from a  $f$ .

**cancel(f, g, include=True)**  
Cancel common factors in a rational function  $f/g$ .

**clear\_denoms(f)**  
Clear denominators, but keep the ground domain.

**coeffs(f, order=None)**  
Returns all non-zero coefficients from  $f$  in lex order.

**cofactors(f, g)**  
Returns GCD of  $f$  and  $g$  and their cofactors.

**compose(f, g)**  
Computes functional composition of  $f$  and  $g$ .

**content(f)**  
Returns GCD of polynomial coefficients.

**convert(f, dom)**  
Convert the ground domain of  $f$ .

**count\_complex\_roots(f, inf=None, sup=None)**  
Return the number of complex roots of  $f$  in  $[inf, sup]$ .

**count\_real\_roots(f, inf=None, sup=None)**  
Return the number of real roots of  $f$  in  $[inf, sup]$ .

**decompose(f)**  
Computes functional decomposition of  $f$ .

**deflate(f)**  
Reduce degree of  $f$  by mapping  $x_i^m$  to  $y_i$ .

**degree(f, j=0)**  
Returns the leading degree of  $f$  in  $x_j$ .

**degree\_list(f)**  
Returns a list of degrees of  $f$ .

**diff(f, m=1, j=0)**  
Computes the  $m$ -th order derivative of  $f$  in  $x_j$ .

**discriminant(f)**  
Computes discriminant of  $f$ .

**div(f, g)**  
Polynomial division with remainder of  $f$  and  $g$ .

**eject(f, dom, front=False)**  
Eject selected generators into the ground domain.

**eval(f, a, j=0)**  
Evaluates  $f$  at the given point  $a$  in  $x_j$ .

**exclude(f)**  
Remove useless generators from  $f$ .

Returns the removed generators and the new excluded  $f$ .

## Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.polys.domains import ZZ

>>> DMP([[ZZ(1)]], [[ZZ(1)], [ZZ(2)]], ZZ).exclude()
([2], DMP([[1], [1, 2]], ZZ, None))

exquo(f, g)
    Computes polynomial exact quotient of f and g.

exquo_ground(f, c)
    Exact quotient of f by a an element of the ground domain.

factor_list(f)
    Returns a list of irreducible factors of f.

factor_list_include(f)
    Returns a list of irreducible factors of f.

classmethod from_dict(rep, lev, dom)
    Construct and instance of cls from a dict representation.

classmethod from_list(rep, lev, dom)
    Create an instance of cls given a list of native coefficients.

classmethod from_sympy_list(rep, lev, dom)
    Create an instance of cls given a list of SymPy coefficients.

gcd(f, g)
    Returns polynomial GCD of f and g.

gcdex(f, g)
    Extended Euclidean algorithm, if univariate.

gff_list(f)
    Computes greatest factorial factorization of f.

half_gcdex(f, g)
    Half extended Euclidean algorithm, if univariate.

homogeneous_order(f)
    Returns the homogeneous order of f.

homogenize(f, s)
    Return homogeneous polynomial of f

inject(f, front=False)
    Inject ground domain generators into f.

integrate(f, m=1, j=0)
    Computes the m-th order indefinite integral of f in x_j.

intervals(f, all=False, eps=None, inf=None, sup=None, fast=False, sqf=False)
    Compute isolating intervals for roots of f.

invert(f, g)
    Invert f modulo g, if possible.

is_cyclotomic
    Returns True if f is a cyclotomic polynomial.
```

**is\_ground**  
Returns True if  $f$  is an element of the ground domain.

**is\_homogeneous**  
Returns True if  $f$  is a homogeneous polynomial.

**is\_irreducible**  
Returns True if  $f$  has no factors over its domain.

**is\_linear**  
Returns True if  $f$  is linear in all its variables.

**is\_monic**  
Returns True if the leading coefficient of  $f$  is one.

**is\_monomial**  
Returns True if  $f$  is zero or has only one term.

**is\_one**  
Returns True if  $f$  is a unit polynomial.

**is\_primitive**  
Returns True if the GCD of the coefficients of  $f$  is one.

**is\_quadratic**  
Returns True if  $f$  is quadratic in all its variables.

**is\_sqf**  
Returns True if  $f$  is a square-free polynomial.

**is\_zero**  
Returns True if  $f$  is a zero polynomial.

**l1\_norm( $f$ )**  
Returns l1 norm of  $f$ .

**lcm( $f, g$ )**  
Returns polynomial LCM of  $f$  and  $g$ .

**lift( $f$ )**  
Convert algebraic coefficients to rationals.

**max\_norm( $f$ )**  
Returns maximum norm of  $f$ .

**monic( $f$ )**  
Divides all coefficients by LC( $f$ ).

**monoms( $f$ ,  $order=None$ )**  
Returns all non-zero monomials from  $f$  in lex order.

**mul( $f, g$ )**  
Multiply two multivariate polynomials  $f$  and  $g$ .

**mul\_ground( $f, c$ )**  
Multiply  $f$  by a an element of the ground domain.

**neg( $f$ )**  
Negate all coefficients in  $f$ .

**nth( $f, *N$ )**  
Returns the n-th coefficient of  $f$ .

**pdiv(f, g)**  
Polynomial pseudo-division of  $f$  and  $g$ .

**per(f, rep, dom=None, kill=False, ring=None)**  
Create a DMP out of the given representation.

**permute(f, P)**  
Returns a polynomial in  $K[x_{P(1)}, \dots, x_{P(n)}]$ .

### Examples

```
>>> from sympy.polys.polyclasses import DMP
>>> from sympy.domains import ZZ

>>> DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [], ZZ).permute([1, 0, 2])
DMP([[2], [], [1, 0], []], ZZ, None)

>>> DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [], ZZ).permute([1, 2, 0])
DMP([[1], [], [2, 0], []], ZZ, None)
```

**pexquo(f, g)**  
Polynomial exact pseudo-quotient of  $f$  and  $g$ .

**pow(f, n)**  
Raise  $f$  to a non-negative power  $n$ .

**pquo(f, g)**  
Polynomial pseudo-quotient of  $f$  and  $g$ .

**prem(f, g)**  
Polynomial pseudo-remainder of  $f$  and  $g$ .

**primitive(f)**  
Returns content and a primitive form of  $f$ .

**quo(f, g)**  
Computes polynomial quotient of  $f$  and  $g$ .

**quo\_ground(f, c)**  
Quotient of  $f$  by a an element of the ground domain.

**refine\_root(f, s, t, eps=None, steps=None, fast=False)**  
Refine an isolating interval to the given precision.

`eps` should be a rational number.

**rem(f, g)**  
Computes polynomial remainder of  $f$  and  $g$ .

**resultant(f, g, includePRS=False)**  
Computes resultant of  $f$  and  $g$  via PRS.

**revert(f, n)**  
Compute  $f^{**(-1)} \bmod x^{**n}$ .

**shift(f, a)**  
Efficiently compute Taylor shift  $f(x + a)$ .

**slice(f, m, n, j=0)**  
Take a continuous subsequence of terms of  $f$ .

```
sqf_list(f, all=False)
    Returns a list of square-free factors of f.

sqf_list_include(f, all=False)
    Returns a list of square-free factors of f.

sqf_norm(f)
    Computes square-free norm of f.

sqf_part(f)
    Computes square-free part of f.

sqr(f)
    Square a multivariate polynomial f.

sturm(f)
    Computes the Sturm sequence of f.

sub(f, g)
    Subtract two multivariate polynomials f and g.

sub_ground(f, c)
    Subtract an element of the ground domain from f.

subresultants(f, g)
    Computes subresultant PRS sequence of f and g.

terms(f, order=None)
    Returns all non-zero terms from f in lex order.

terms_gcd(f)
    Remove GCD of terms from the polynomial f.

to_dict(f, zero=False)
    Convert f to a dict representation with native coefficients.

to_exact(f)
    Make the ground domain exact.

to_field(f)
    Make the ground domain a field.

to_ring(f)
    Make the ground domain a ring.

to_sympy_dict(f, zero=False)
    Convert f to a dict representation with SymPy coefficients.

to_tuple(f)
    Convert f to a tuple representation with native coefficients.

    This is needed for hashing.

total_degree(f)
    Returns the total degree of f.

trunc(f, p)
    Reduce f modulo a constant p.

unify(f, g)
    Unify representations of two multivariate polynomials.

class sympy.polys.polyclasses.DMF(rep, dom, lev=None, ring=None)
    Dense Multivariate Fractions over K.
```

```
add(f, g)
    Add two multivariate fractions f and g.

cancel(f)
    Remove common factors from f.num and f.den.

denom(f)
    Returns the denominator of f.

exquo(f, g)
    Computes quotient of fractions f and g.

frac_unify(f, g)
    Unify representations of two multivariate fractions.

half_per(f, rep, kill=False)
    Create a DMP out of the given representation.

invert(f, check=True)
    Computes inverse of a fraction f.

is_one
    Returns True if f is a unit fraction.

is_zero
    Returns True if f is a zero fraction.

mul(f, g)
    Multiply two multivariate fractions f and g.

neg(f)
    Negate all coefficients in f.

numer(f)
    Returns the numerator of f.

per(f, num, den, cancel=True, kill=False, ring=None)
    Create a DMF out of the given representation.

poly_unify(f, g)
    Unify a multivariate fraction and a polynomial.

pow(f, n)
    Raise f to a non-negative power n.

quo(f, g)
    Computes quotient of fractions f and g.

sub(f, g)
    Subtract two multivariate fractions f and g.

class sympy.polys.polyclasses.ANP(rep, mod, dom)
    Dense Algebraic Number Polynomials over a field.

LC(f)
    Returns the leading coefficient of f.

TC(f)
    Returns the trailing coefficient of f.

is_ground
    Returns True if f is an element of the ground domain.
```

```
is_one
    Returns True if f is a unit algebraic number.

is_zero
    Returns True if f is a zero algebraic number.

pow(f, n)
    Raise f to a non-negative power n.

to_dict(f)
    Convert f to a dict representation with native coefficients.

to_list(f)
    Convert f to a list representation with native coefficients.

to_sympy_dict(f)
    Convert f to a dict representation with SymPy coefficients.

to_sympy_list(f)
    Convert f to a list representation with SymPy coefficients.

to_tuple(f)
    Convert f to a tuple representation with native coefficients.

This is needed for hashing.

unify(f, g)
    Unify representations of two algebraic numbers.
```

## Level Zero

Level zero contains the bulk code of the polynomial manipulation module.

**Manipulation of dense, multivariate polynomials** These functions can be used to manipulate polynomials in  $K[X_0, \dots, X_u]$ . Functions for manipulating multivariate polynomials in the dense representation have the prefix `dmp_`. Functions which only apply to univariate polynomials (i.e.  $u = 0$ ) have the prefix `dup_`. The ground domain  $K$  has to be passed explicitly. For many multivariate polynomial manipulation functions also the level  $u$ , i.e. the number of generators minus one, has to be passed. (Note that, in many cases, `dup_` versions of functions are available, which may be slightly more efficient.)

### Basic manipulation:

```
sympy.polys.densebasic.dmp_LC(f, K)
    Return leading coefficient of f.
```

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_LC

>>> poly_LC([], ZZ)
0
>>> poly_LC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
1
```

```
sympy.polys.densebasic.dmp_TC(f, K)
    Return trailing coefficient of f.
```

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import poly_TC

>>> poly_TC([], ZZ)
0
>>> poly_TC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
3
```

`sympy.polys.densebasic.dmp_ground_LC(f, u, K)`

Return the ground leading coefficient.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_LC

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_LC(f, 2, ZZ)
1
```

`sympy.polys.densebasic.dmp_ground_TC(f, u, K)`

Return the ground trailing coefficient.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_TC

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_TC(f, 2, ZZ)
3
```

`sympy.polys.densebasic.dmp_true_LT(f, u, K)`

Return the leading term  $c * x_1^{n_1} \dots x_k^{n_k}$ .

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_true_LT

>>> f = ZZ.map([[4], [2, 0], [3, 0, 0]])

>>> dmp_true_LT(f, 1, ZZ)
((2, 0), 4)
```

`sympy.polys.densebasic.dmp_degree(f, u)`

Return the leading degree of  $f$  in  $x_0$  in  $K[X]$ .

Note that the degree of 0 is negative infinity (the SymPy object `-oo`).

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree

>>> dmp_degree([[[]]], 2)
-oo

>>> f = ZZ.map([[2], [1, 2, 3]])

>>> dmp_degree(f, 1)
1

sympy.polys.densebasic.dmp_degree_in(f, j, u)
Return the leading degree of f in x_j in K[X].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_in

>>> f = ZZ.map([[2], [1, 2, 3]])

>>> dmp_degree_in(f, 0, 1)
1
>>> dmp_degree_in(f, 1, 1)
2

sympy.polys.densebasic.dmp_degree_list(f, u)
Return a list of degrees of f in K[X].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_degree_list

>>> f = ZZ.map([[1], [1, 2, 3]])

>>> dmp_degree_list(f, 1)
(1, 2)

sympy.polys.densebasic.dmp_strip(f, u)
Remove leading zeros from f in K[X].
```

## Examples

```
>>> from sympy.polys.densebasic import dmp_strip

>>> dmp_strip([], [0, 1, 2], [1], 1)
[[0, 1, 2], [1]]

sympy.polys.densebasic.dmp_validate(f, K=None)
Return the number of levels in f and recursively strip it.
```

## Examples

```
>>> from sympy.polys.densebasic import dmp_validate

>>> dmp_validate([], [0, 1, 2], [1])
([[], 2], [1]), 1)

>>> dmp_validate([[1], 1])
Traceback (most recent call last):
...
ValueError: invalid data structure for a multivariate polynomial

sympy.polys.densebasic.dup_reverse(f)
Compute  $x^{**n} * f(1/x)$ , i.e.: reverse  $f$  in  $K[x]$ .
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_reverse

>>> f = ZZ.map([1, 2, 3, 0])

>>> dup_reverse(f)
[3, 2, 1]

sympy.polys.densebasic.dmp_copy(f, u)
Create a new copy of a polynomial  $f$  in  $K[X]$ .
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_copy

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_copy(f, 1)
[[1], [1, 2]]

sympy.polys.densebasic.dmp_to_tuple(f, u)
Convert  $f$  into a nested tuple of tuples.

This is needed for hashing. This is similar to dmp_copy().
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_to_tuple

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_to_tuple(f, 1)
((1,), (1, 2))
```

```
sympy.polys.densebasic.dmp_normal(f, u, K)
    Normalize a multivariate polynomial in the given domain.
```

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_normal

>>> dmp_normal([], [0, 1.5, 2], 1, ZZ)
[[1, 2]]
```

```
sympy.polys.densebasic.dmp_convert(f, u, K0, K1)
    Convert the ground domain of f from K0 to K1.
```

#### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_convert

>>> R, x = ring("x", ZZ)

>>> dmp_convert([[R(1)], [R(2)]], 1, R.to_domain(), ZZ)
[[1], [2]]
>>> dmp_convert([[ZZ(1)], [ZZ(2)]], 1, ZZ, R.to_domain())
[[1], [2]]
```

```
sympy.polys.densebasic.dmp_from_sympy(f, u, K)
    Convert the ground domain of f from SymPy to K.
```

#### Examples

```
>>> from sympy import S
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_sympy

>>> dmp_from_sympy([[S(1)], [S(2)]], 1, ZZ) == [[ZZ(1)], [ZZ(2)]]
True
```

```
sympy.polys.densebasic.dmp_nth(f, n, u, K)
    Return the n-th coefficient of f in K[x].
```

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nth

>>> f = ZZ.map([1, 2, 3])
```

```
>>> dmp_nth(f, 0, 1, ZZ)
[3]
>>> dmp_nth(f, 4, 1, ZZ)
[]
```

`sympy.polys.densebasic.dmp_ground_nth(f, N, u, K)`  
Return the ground n-th coefficient of f in K[x].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_ground_nth

>>> f = ZZ.map([[1], [2, 3]])

>>> dmp_ground_nth(f, (0, 1), 1, ZZ)
2
```

`sympy.polys.densebasic.dmp_zero_p(f, u)`  
Return True if f is zero in K[X].

### Examples

```
>>> from sympy.polys.densebasic import dmp_zero_p

>>> dmp_zero_p([[[[[[]]]]], 4)
True
>>> dmp_zero_p([[[[[1]]]]], 4)
False
```

`sympy.polys.densebasic.dmp_zero(u)`  
Return a multivariate zero.

### Examples

```
>>> from sympy.polys.densebasic import dmp_zero

>>> dmp_zero(4
[[[[[]]]]]
```

`sympy.polys.densebasic.dmp_one_p(f, u, K)`  
Return True if f is one in K[X].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one_p

>>> dmp_one_p([[[ZZ(1)]]], 2, ZZ)
True
```

`sympy.polys.densebasic.dmp_one(u, K)`  
Return a multivariate one over K.

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_one

>>> dmp_one(2, ZZ)
[[[1]]]

sympy.polys.densebasic.dmp_ground_p(f, c, u)
Return True if f is constant in K[X].
```

## Examples

```
>>> from sympy.polys.densebasic import dmp_ground_p

>>> dmp_ground_p([[3]], 3, 2)
True
>>> dmp_ground_p([[4]], None, 2)
True

sympy.polys.densebasic.dmp_ground(c, u)
Return a multivariate constant.
```

## Examples

```
>>> from sympy.polys.densebasic import dmp_ground

>>> dmp_ground(3, 5)
[[[[3]]]]
>>> dmp_ground(1, -1)
1

sympy.polys.densebasic.dmp_zeros(n, u, K)
Return a list of multivariate zeros.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_zeros

>>> dmp_zeros(3, 2, ZZ)
[[[], [], []], [[], []], [[[1]]]]
>>> dmp_zeros(3, -1, ZZ)
[0, 0, 0]

sympy.polys.densebasic.dmp_grounds(c, n, u)
Return a list of multivariate constants.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_grounds

>>> dmp_grounds(ZZ(4), 3, 2)
[[[4]], [[4]], [[4]]]
>>> dmp_grounds(ZZ(4), 3, -1)
[4, 4, 4]
```

`sympy.polys.densebasic.dmp_negative_p(f, u, K)`

Return True if LC(f) is negative.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_negative_p

>>> dmp_negative_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
False
>>> dmp_negative_p([-ZZ(1), [ZZ(1)]], 1, ZZ)
True
```

`sympy.polys.densebasic.dmp_positive_p(f, u, K)`

Return True if LC(f) is positive.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_positive_p

>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
True
>>> dmp_positive_p([-ZZ(1), [ZZ(1)]], 1, ZZ)
False
```

`sympy.polys.densebasic.dmp_from_dict(f, u, K)`

Create a K[X] polynomial from a dict.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_from_dict

>>> dmp_from_dict({(0, 0): ZZ(3), (0, 1): ZZ(2), (2, 1): ZZ(1)}, 1, ZZ)
[[1, 0], [], [2, 3]]
>>> dmp_from_dict({}, 0, ZZ)
[]
```

`sympy.polys.densebasic.dmp_to_dict(f, u, K=None, zero=False)`

Convert a K[X] polynomial to a dict<sup>“</sup>.

## Examples

```
>>> from sympy.polys.densebasic import dmp_to_dict

>>> dmp_to_dict([[1, 0], [], [2, 3]], 1)
{(0, 0): 3, (0, 1): 2, (2, 1): 1}
>>> dmp_to_dict([], 0)
{}

sympy.polys.densebasic.dmp_swap(f, i, j, u, K)
    Transform K[..x_i..x_j..] to K[..x_j..x_i..].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_swap

>>> f = ZZ.map([[[2], [1, 0]], []])

>>> dmp_swap(f, 0, 1, 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_swap(f, 1, 2, 2, ZZ)
[[[1], [2, 0]], [[]]]
>>> dmp_swap(f, 0, 2, 2, ZZ)
[[[1, 0]], [[2, 0], []]]

sympy.polys.densebasic.dmp_permute(f, P, u, K)
    Return a polynomial in K[x_{P(1)}, ..., x_{P(n)}].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_permute

>>> f = ZZ.map([[[2], [1, 0]], []])

>>> dmp_permute(f, [1, 0, 2], 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_permute(f, [1, 2, 0], 2, ZZ)
[[[1], []], [[2, 0], []]]

sympy.polys.densebasic.dmp_nest(f, l, K)
    Return a multivariate value nested 1-levels.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_nest

>>> dmp_nest([[ZZ(1)]], 2, ZZ)
[[[[1]]]]

sympy.polys.densebasic.dmp_raise(f, l, u, K)
    Return a multivariate polynomial raised 1-levels.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_raise

>>> f = ZZ.map([], [1, 2])

>>> dmp_raise(f, 2, 1, ZZ)
[[[], []], [[1]], [[2]]]

sympy.polys.densebasic.dmp_deflate(f, u, K)
Map x_i**m_i to y_i in a polynomial in K[X].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_deflate

>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])

>>> dmp_deflate(f, 1, ZZ)
((2, 3), [[1, 2], [3, 4]])

sympy.polys.densebasic.dmp_multi_deflate(polys, u, K)
Map x_i**m_i to y_i in a set of polynomials in K[X].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_multi_deflate

>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
>>> g = ZZ.map([[1, 0, 2], [], [3, 0, 4]])

>>> dmp_multi_deflate((f, g), 1, ZZ)
((2, 1), ([[1, 0, 0, 2], [3, 0, 0, 4]], [[1, 0, 2], [3, 0, 4]]))

sympy.polys.densebasic.dmp_inflate(f, M, u, K)
Map y_i to x_i**k_i in a polynomial in K[X].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inflate

>>> f = ZZ.map([[1, 2], [3, 4]])

>>> dmp_inflate(f, (2, 3), 1, ZZ)
[[1, 0, 0, 2], [], [3, 0, 0, 4]]

sympy.polys.densebasic.dmp_exclude(f, u, K)
Exclude useless levels from f.

Return the levels excluded, the new excluded f, and the new u.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_exclude

>>> f = ZZ.map([[1]], [[1], [2]])

>>> dmp_exclude(f, 2, ZZ)
([2], [[1], [1, 2]], 1)

sympy.polys.densebasic.dmp_include(f, J, u, K)
Include useless levels in f.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_include

>>> f = ZZ.map([[1], [1, 2]])

>>> dmp_include(f, [2], 1, ZZ)
[[[1]], [[1], [2]]]

sympy.polys.densebasic.dmp_inject(f, u, K, front=False)
Convert f from K[X][Y] to K[X,Y].
```

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_inject

>>> R, x,y = ring("x,y", ZZ)

>>> dmp_inject([R(1), x + 2], 0, R.to_domain())
([[[1]], [[1], [2]]], 2)
>>> dmp_inject([R(1), x + 2], 0, R.to_domain(), front=True)
([[[1]], [[1, 2]]], 2)

sympy.polys.densebasic.dmp_eject(f, u, K, front=False)
Convert f from K[X,Y] to K[X][Y].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_eject

>>> dmp_eject([[1]], [[1], [2]], 2, ZZ['x', 'y'])
[1, x + 2]

sympy.polys.densebasic.dmp_terms_gcd(f, u, K)
Remove GCD of terms from f in K[X].
```

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_terms_gcd
```

```
>>> f = ZZ.map([[1, 0], [1, 0, 0], [], []])
```

```
>>> dmp_terms_gcd(f, 1, ZZ)
((2, 1), [[1], [1, 0]])
```

`sympy.polys.densebasic.dmp_list_terms(f, u, K, order=None)`

List all non-zero terms from `f` in the given order `order`.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_list_terms
```

```
>>> f = ZZ.map([[1, 1], [2, 3]])
```

```
>>> dmp_list_terms(f, 1, ZZ)
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
>>> dmp_list_terms(f, 1, ZZ, order='grevlex')
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
```

`sympy.polys.densebasic.dmp_apply_pairs(f, g, h, args, u, K)`

Apply `h` to pairs of coefficients of `f` and `g`.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dmp_apply_pairs
```

```
>>> h = lambda x, y, z: 2*x + y - z
```

```
>>> dmp_apply_pairs([[1, [2, 3]], [[3], [2, 1]]], h, (1,), 1, ZZ)
[[4], [5, 6]]
```

`sympy.polys.densebasic.dmp_slice(f, m, n, u, K)`

Take a continuous subsequence of terms of `f` in  $K[X]$ .

`sympy.polys.densebasic.dup_random(n, a, b, K)`

Return a polynomial of degree `n` with coefficients in `[a, b]`.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.densebasic import dup_random
```

```
>>> dup_random(3, -10, 10, ZZ)
[-2, -8, 9, -4]
```

**Arithmetic operations:**

```
sympy.polys.densearith.dmp_add_term(f, c, i, u, K)
Add c(x_2..x_u)*x_0**i to f in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_add_term(x*y + 1, 2, 2)
2*x**2 + x*y + 1
```

```
sympy.polys.densearith.dmp_sub_term(f, c, i, u, K)
Subtract c(x_2..x_u)*x_0**i from f in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_sub_term(2*x**2 + x*y + 1, 2, 2)
x*y + 1
```

```
sympy.polys.densearith.dmp_mul_term(f, c, i, u, K)
Multiply f by c(x_2..x_u)*x_0**i in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_mul_term(x**2*y + x, 3*y, 2)
3*x**4*y**2 + 3*x**3*y
```

```
sympy.polys.densearith.dmp_add_ground(f, c, u, K)
Add an element of the ground domain to f.
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_add_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x + 8
```

```
sympy.polys.densearith.dmp_sub_ground(f, c, u, K)
Subtract an element of the ground domain from f.
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_sub_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x

sympy.polys.densearith.dmp_mul_ground(f, c, u, K)
    Multiply f by a constant value in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_mul_ground(2*x + 2*y, ZZ(3))
6*x + 6*y

sympy.polys.densearith.dmp_quo_ground(f, c, u, K)
    Quotient by a constant in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, ZZ(2))
x**2*y + x

>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, QQ(2))
x**2*y + 3/2*x

sympy.polys.densearith.dmp_exquo_ground(f, c, u, K)
    Exact quotient by a constant in K[X].
```

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> R.dmp_exquo_ground(x**2*y + 2*x, QQ(2))
1/2*x**2*y + x

sympy.polys.densearith.dup_lshift(f, n, K)
    Efficiently multiply f by x**n in K[x].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_lshift(x**2 + 1, 2)
x**4 + x**2

sympy.polys.densearith.dup_rshift(f, n, K)
    Efficiently divide f by x**n in K[x].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_rshift(x**4 + x**2, 2)
x**2 + 1
>>> R.dup_rshift(x**4 + x**2 + 2, 2)
x**2 + 1

sympy.polys.densearith.dmp_abs(f, u, K)
    Make all coefficients positive in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_abs(x**2*y - x)
x**2*y + x

sympy.polys.densearith.dmp_neg(f, u, K)
    Negate a polynomial in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_neg(x**2*y - x)
-x**2*y + x

sympy.polys.densearith.dmp_add(f, g, u, K)
    Add dense polynomials in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_add(x**2 + y, x**2*y + x)
x**2*y + x**2 + x + y

sympy.polys.densearith.dmp_sub(f, g, u, K)
    Subtract dense polynomials in K[X].
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub(x**2 + y, x**2*y + x)
-x**2*y + x**2 - x + y
```

sympy.polys.densearith.dmp\_add\_mul( $f, g, h, u, K$ )  
Returns  $f + g*h$  where  $f, g, h$  are in  $K[X]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_mul(x**2 + y, x, x + 2)
2*x**2 + 2*x + y
```

sympy.polys.densearith.dmp\_sub\_mul( $f, g, h, u, K$ )  
Returns  $f - g*h$  where  $f, g, h$  are in  $K[X]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_mul(x**2 + y, x, x + 2)
-2*x + y
```

sympy.polys.densearith.dmp\_mul( $f, g, u, K$ )  
Multiply dense polynomials in  $K[X]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul(x*y + 1, x)
x**2*y + x
```

sympy.polys.densearith.dmp\_sqr( $f, u, K$ )  
Square dense polynomials in  $K[X]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sqr(x**2 + x*y + y**2)
x**4 + 2*x**3*y + 3*x**2*y**2 + 2*x*y**3 + y**4
```

```
sympy.polys.densearith.dmp_pow(f, n, u, K)
    Raise f to the n-th power in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_pow(x*y + 1, 3)
x**3*y**3 + 3*x**2*y**2 + 3*x*y + 1
```

```
sympy.polys.densearith.dmp_pdiv(f, g, u, K)
    Polynomial pseudo-division in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_pdiv(x**2 + x*y, 2*x + 2)
(2*x + 2*y - 2, -4*y + 4)
```

```
sympy.polys.densearith.dmp_prem(f, g, u, K)
    Polynomial pseudo-remainder in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_prem(x**2 + x*y, 2*x + 2)
-4*y + 4
```

```
sympy.polys.densearith.dmp_pquo(f, g, u, K)
    Polynomial exact pseudo-quotient in K[X].
```

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2

>>> R.dmp_pquo(f, g)
2*x

>>> R.dmp_pquo(f, h)
2*x + 2*y - 2
```

```
sympy.polys.densearith.dmp_pexquo(f, g, u, K)
    Polynomial pseudo-quotient in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x**2 + x*y
>>> g = 2*x + 2*y
>>> h = 2*x + 2

>>> R.dmp_pexquo(f, g)
2*x

>>> R.dmp_pexquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]

sympy.polys.densearith.dmp_rr_div(f, g, u, K)
Multivariate division with remainder over a ring.
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_rr_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)

sympy.polys.densearith.dmp_ff_div(f, g, u, K)
Polynomial division with remainder over a field.
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> R.dmp_ff_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)

sympy.polys.densearith.dmp_div(f, g, u, K)
Polynomial division with remainder in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)

>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

```
sympy.polys.densearith.dmp_rem(f, g, u, K)
```

Returns polynomial remainder in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
x**2 + x*y

>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
-y + 1
```

```
sympy.polys.densearith.dmp_quo(f, g, u, K)
```

Returns exact polynomial quotient in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
0

>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
1/2*x + 1/2*y - 1/2
```

```
sympy.polys.densearith.dmp_exquo(f, g, u, K)
```

Returns polynomial quotient in  $K[X]$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x**2 + x*y
>>> g = x + y
>>> h = 2*x + 2

>>> R.dmp_exquo(f, g)
x

>>> R.dmp_exquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

```
sympy.polys.densearith.dmp_max_norm(f, u, K)
```

Returns maximum norm of a polynomial in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_max_norm(2*x*y - x - 3)
3
```

```
sympy.polys.densearith.dmp_l1_norm(f, u, K)
>Returns l1 norm of a polynomial in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_l1_norm(2*x*y - x - 3)
6
```

```
sympy.polys.densearith.dmp_expand(polys, u, K)
>Multiply together several polynomials in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_expand([x**2 + y**2, x + 1])
x**3 + x**2 + x*y**2 + y**2
```

## Further tools:

```
sympy.polys.denseTools.dmp_integrate(f, m, u, K)
>Computes the indefinite integral of f in x_0 in K[X].
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate(x + 2*y, 1)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate(x + 2*y, 2)
1/6*x**3 + x**2*y
```

```
sympy.polys.denseTools.dmp_integrate_in(f, m, j, u, K)
>Computes the indefinite integral of f in x_j in K[X].
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate_in(x + 2*y, 1, 0)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate_in(x + 2*y, 1, 1)
x*y + y**2
```

`sympy.polys.denseTools.dmp_diff(f, m, u, K)`  
m-th order derivative in  $x_0$  of a polynomial in  $K[X]$ .

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1

>>> R.dmp_diff(f, 1)
y**2 + 2*y + 3
>>> R.dmp_diff(f, 2)
0
```

`sympy.polys.denseTools.dmp_diff_in(f, m, j, u, K)`  
m-th order derivative in  $x_j$  of a polynomial in  $K[X]$ .

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1

>>> R.dmp_diff_in(f, 1, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_in(f, 1, 1)
2*x*y + 2*x + 4*y + 3
```

`sympy.polys.denseTools.dmp_eval(f, a, u, K)`  
Evaluate a polynomial at  $x_0 = a$  in  $K[X]$  using the Horner scheme.

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_eval(2*x*y + 3*x + y + 2, 2)
5*y + 8
```

`sympy.polys.denseTools.dmp_eval_in(f, a, j, u, K)`  
Evaluate a polynomial at  $x_j = a$  in  $K[X]$  using the Horner scheme.

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 2*x*y + 3*x + y + 2

>>> R.dmp_eval_in(f, 2, 0)
5*y + 8
>>> R.dmp_eval_in(f, 2, 1)
7*x + 4

sympy.polys.densetools.dmp_eval_tail(f, A, u, K)
Evaluate a polynomial at x_j = a_j, ... in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 2*x*y + 3*x + y + 2

>>> R.dmp_eval_tail(f, [2])
7*x + 4
>>> R.dmp_eval_tail(f, [2, 2])
18

sympy.polys.densetools.dmp_diff_eval_in(f, m, a, j, u, K)
Differentiate and evaluate a polynomial in x_j at a in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1

>>> R.dmp_diff_eval_in(f, 1, 2, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_eval_in(f, 1, 2, 1)
6*x + 11

sympy.polys.densetools.dmp_trunc(f, p, u, K)
Reduce a K[X] polynomial modulo a polynomial p in K[Y].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
>>> g = (y - 1).drop(x)
```

```
>>> R.dmp_trunc(f, g)
11*x**2 + 11*x + 5

sympy.polys.denseutils.dmp_ground_trunc(f, p, u, K)
Reduce a K[X] polynomial modulo a constant p in K.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3

>>> R.dmp_ground_trunc(f, ZZ(3))
-x**2 - x*y - y

sympy.polys.denseutils.dup_monic(f, K)
Divide all coefficients by LC(f) in K[x].
```

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)
>>> R.dup_monic(3*x**2 + 6*x + 9)
x**2 + 2*x + 3

>>> R, x = ring("x", QQ)
>>> R.dup_monic(3*x**2 + 4*x + 2)
x**2 + 4/3*x + 2/3

sympy.polys.denseutils.dmp_ground_monic(f, u, K)
Divide all coefficients by LC(f) in K[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> f = 3*x**2*y + 6*x**2 + 3*x*y + 9*y + 3

>>> R.dmp_ground_monic(f)
x**2*y + 2*x**2 + x*y + 3*y + 1

>>> R, x,y = ring("x,y", QQ)
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3

>>> R.dmp_ground_monic(f)
x**2*y + 8/3*x**2 + 5/3*x*y + 2*x + 2/3*y + 1

sympy.polys.denseutils.dup_content(f, K)
Compute the GCD of coefficients of f in K[x].
```

## Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_content(f)
2

>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_content(f)
2
```

`sympy.polys.densetools.dmp_ground_content(f, u, K)`  
Compute the GCD of coefficients of  $f$  in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_content(f)
2

>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_content(f)
2
```

`sympy.polys.densetools.dup_primitive(f, K)`  
Compute content and the primitive form of  $f$  in  $K[x]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)

>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12

>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

`sympy.polys.densetools.dmp_ground_primitive(f, u, K)`  
Compute content and the primitive form of  $f$  in  $K[X]$ .

## Examples

```
>>> from sympy.polys import ring, ZZ, QQ

>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)

>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12

>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)

sympy.polys.densetools.dup_extract(f, g, K)
Extract common content from a pair of polynomials in K[x].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_extract(6*x**2 + 12*x + 18, 4*x**2 + 8*x + 12)
(2, 3*x**2 + 6*x + 9, 2*x**2 + 4*x + 6)

sympy.polys.densetools.dmp_ground_extract(f, g, u, K)
Extract common content from a pair of polynomials in K[X].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_ground_extract(6*x*y + 12*x + 18, 4*x*y + 8*x + 12)
(2, 3*x*y + 6*x + 9, 2*x*y + 4*x + 6)

sympy.polys.densetools.dup_real_imag(f, K)
Return bivariate polynomials f1 and f2, such that f = f1 + f2*I.
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)

sympy.polys.densetools.dup_mirror(f, K)
Evaluate efficiently the composition f(-x) in K[x].
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

`sympy.polys.densetools.dup_scale(f, a, K)`  
Evaluate efficiently composition  $f(a \cdot x)$  in  $K[x]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

`sympy.polys.densetools.dup_shift(f, a, K)`  
Evaluate efficiently Taylor shift  $f(x + a)$  in  $K[x]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

`sympy.polys.densetools.dup_transform(f, p, q, K)`  
Evaluate functional transformation  $q^{**n} * f(p/q)$  in  $K[x]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

`sympy.polys.densetools.dmp_compose(f, g, u, K)`  
Evaluate functional composition  $f(g)$  in  $K[X]$ .

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

```
sympy.polys.denseTools.dup_decompose(f, K)
    Computes functional decomposition of f in K[x].
```

Given a univariate polynomial  $f$  with coefficients in a field of characteristic zero, returns list  $[f_1, f_2, \dots, f_n]$ , where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and  $f_2, \dots, f_n$  are monic and homogeneous polynomials of at least second degree.

Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

```
1.f o g = f(x + b) o (g - b)
```

```
2.x**n o x**m = x**m o x**n
```

```
3.T_n o T_m = T_m o T_n
```

where  $T_n$  and  $T_m$  are Chebyshev polynomials.

## References

1.[Kozen89] (page 1245)

[Kozen89] (page 1245)

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```

```
sympy.polys.denseTools.dmp_lift(f, u, K)
    Convert algebraic coefficients to integers in K[X].
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> from sympy import I

>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)

>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])
>>> R.dmp_lift(f)
x**8 + 2*x**6 + 9*x**4 - 8*x**2 + 16
```

```
sympy.polys.denseTools.dup_sign_variations(f, K)
    Compute the number of sign variations of f in K[x].
```

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2

sympy.polys.densetools.dmp_clear_denoms(f, u, K0, K1=None, convert=False)
Clear denominators, i.e. transform K_0 to K_1.
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> f = QQ(1,2)*x + QQ(1,3)*y + 1

>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)

sympy.polys.densetools.dmp_revert(f, g, u, K)
Compute f**(-1) mod x**n using Newton iteration.
```

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

**Manipulation of dense, univariate polynomials with finite field coefficients** Functions in this module carry the prefix `gf_`, referring to the classical name “Galois Fields” for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

```
sympy.polys.galoistools.gf_crt(U, M, K=None)
Chinese Remainder Theorem.
```

Given a set of integer residues  $u_0, \dots, u_n$  and a set of co-prime integer moduli  $m_0, \dots, m_n$ , returns an integer  $u$ , such that  $u = u_i \bmod m_i$  for  $i = 0, \dots, n$ .

As an example consider a set of residues  $U = [49, 76, 65]$  and a set of moduli  $M = [99, 97, 95]$ . Then we have:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt
>>> from sympy.nttheory.modular import solve_congruence

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]  
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

See also:

`sympy.nttheory.modular.crt` (page 260) a higher level crt routine

`sympy.nttheory.modular.solve_congruence` (page 261)

`sympy.polys.galoistools.gf_crt1(M, K)`

First part of the Chinese Remainder Theorem.

### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_crt1  
  
>>> gf_crt1([99, 97, 95], ZZ)  
(912285, [9215, 9405, 9603], [62, 24, 12])
```

`sympy.polys.galoistools.gf_crt2(U, M, p, E, S, K)`

Second part of the Chinese Remainder Theorem.

### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_crt2  
  
>>> U = [49, 76, 65]  
>>> M = [99, 97, 95]  
>>> p = 912285  
>>> E = [9215, 9405, 9603]  
>>> S = [62, 24, 12]  
  
>>> gf_crt2(U, M, p, E, S, ZZ)  
639985
```

`sympy.polys.galoistools.gf_int(a, p)`

Coerce  $a \bmod p$  to an integer in the range  $[-p/2, p/2]$ .

### Examples

```
>>> from sympy.polys.galoistools import gf_int  
  
>>> gf_int(2, 7)  
2  
>>> gf_int(5, 7)  
-2
```

`sympy.polys.galoistools.gf_degree(f)`

Return the leading degree of  $f$ .

**Examples**

```
>>> from sympy.polys.galoistools import gf_degree  
  
>>> gf_degree([1, 1, 2, 0])  
3  
>>> gf_degree([])  
-1
```

`sympy.polys.galoistools.gf_LC(f, K)`  
Return the leading coefficient of f.

**Examples**

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_LC  
  
>>> gf_LC([3, 0, 1], ZZ)  
3
```

`sympy.polys.galoistools.gf_TC(f, K)`  
Return the trailing coefficient of f.

**Examples**

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_TC  
  
>>> gf_TC([3, 0, 1], ZZ)  
1
```

`sympy.polys.galoistools.gf_strip(f)`  
Remove leading zeros from f.

**Examples**

```
>>> from sympy.polys.galoistools import gf_strip  
  
>>> gf_strip([0, 0, 0, 3, 0, 1])  
[3, 0, 1]
```

`sympy.polys.galoistools.gf_trunc(f, p)`  
Reduce all coefficients modulo p.

**Examples**

```
>>> from sympy.polys.galoistools import gf_trunc  
  
>>> gf_trunc([7, -2, 3], 5)  
[2, 3, 3]
```

`sympy.polys.galoistools.gf_normal(f, p, K)`  
Normalize all coefficients in K.

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_normal

>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]

sympy.polys.galoistools.gf_from_dict(f, p, K)
Create a GF(p)[x] polynomial from a dict.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_dict

>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 3, 0, 0, 0, 4]

sympy.polys.galoistools.gf_to_dict(f, p, symmetric=True)
Convert a GF(p)[x] polynomial to a dict.
```

## Examples

```
>>> from sympy.polys.galoistools import gf_to_dict

>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5)
{0: -1, 4: -2, 10: -1}
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False)
{0: 4, 4: 3, 10: 4}

sympy.polys.galoistools.gf_from_int_poly(f, p)
Create a GF(p)[x] polynomial from Z[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_int_poly

>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3, 3]

sympy.polys.galoistools.gf_to_int_poly(f, p, symmetric=True)
Convert a GF(p)[x] polynomial to Z[x].
```

## Examples

```
>>> from sympy.polys.galoistools import gf_to_int_poly
```

```
>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

`sympy.polys.galoistools.gf_neg(f, p, K)`

Negate a polynomial in GF(p)[x].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_neg

>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

`sympy.polys.galoistools.gf_add_ground(f, a, p, K)`

Compute f + a where f in GF(p)[x] and a in GF(p).

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_ground

>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

`sympy.polys.galoistools.gf_sub_ground(f, a, p, K)`

Compute f - a where f in GF(p)[x] and a in GF(p).

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_ground

>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

`sympy.polys.galoistools.gf_mul_ground(f, a, p, K)`

Compute f \* a where f in GF(p)[x] and a in GF(p).

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul_ground

>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

`sympy.polys.galoistools.gf_quo_ground(f, a, p, K)`

Compute f/a where f in GF(p)[x] and a in GF(p).

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo_ground

>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]

sympy.polys.galoistools.gf_add(f, g, p, K)
Add polynomials in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add

>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]

sympy.polys.galoistools.gf_sub(f, g, p, K)
Subtract polynomials in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub

>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]

sympy.polys.galoistools.gf_mul(f, g, p, K)
Multiply polynomials in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul

>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]

sympy.polys.galoistools.gf_sqr(f, p, K)
Square polynomials in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqr

>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

```
sympy.polys.galoistools.gf_add_mul(f, g, h, p, K)
```

Returns  $f + g \cdot h$  where  $f, g, h$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_mul
>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

```
sympy.polys.galoistools.gf_sub_mul(f, g, h, p, K)
```

Compute  $f - g \cdot h$  where  $f, g, h$  in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_mul
>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```

```
sympy.polys.galoistools.gf_expand(F, p, K)
```

Expand results of `factor()` (page 675) in  $\text{GF}(p)[x]$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_expand
>>> gf_expand([(3, 2, 4), 1], (2, 2, 2), (3, 1, 3)], 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

```
sympy.polys.galoistools.gf_div(f, g, p, K)
```

Division with remainder in  $\text{GF}(p)[x]$ .

Given univariate polynomials  $f$  and  $g$  with coefficients in a finite field with  $p$  elements, returns polynomials  $q$  and  $r$  (quotient and remainder) such that  $f = q \cdot g + r$ .

Consider polynomials  $x^{**}3 + x + 1$  and  $x^{**}2 + x$  in  $\text{GF}(2)$ :

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_div, gf_add_mul
>>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient  $x + 1$  and remainder 1, thus:

```
>>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```

### References

1. [Monagan93] (page 1245)

2.[Gathen99] (page 1245)

[Monagan93] (page 1245), [Gathen99] (page 1245)

`sympy.polys.galoistools.gf_rem(f, g, p, K)`  
Compute polynomial remainder in GF(p)[x].

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rem

>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```

`sympy.polys.galoistools.gf_quo(f, g, p, K)`  
Compute exact quotient in GF(p)[x].

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo

>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

`sympy.polys.galoistools.gf_exquo(f, g, p, K)`  
Compute polynomial quotient in GF(p)[x].

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_exquo

>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]

>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...
ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

`sympy.polys.galoistools.gf_lshift(f, n, K)`  
Efficiently multiply f by x\*\*n.

#### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lshift
```

```
>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0]
```

`sympy.polys.galoistools.gf_rshift(f, n, K)`  
Efficiently divide f by  $x^{**n}$ .

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rshift

>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2], [3, 4, 0])
```

`sympy.polys.galoistools.gf_pow(f, n, p, K)`  
Compute  $f^{**n}$  in  $\text{GF}(p)[x]$  using repeated squaring.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow

>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

`sympy.polys.galoistools.gf_pow_mod(f, n, g, p, K)`  
Compute  $f^{**n}$  in  $\text{GF}(p)[x]/(g)$  using repeated squaring.

Given polynomials  $f$  and  $g$  in  $\text{GF}(p)[x]$  and a non-negative integer  $n$ , efficiently computes  $f^{**n} \pmod{g}$  i.e. the remainder of  $f^{**n}$  from division by  $g$ , using the repeated squaring algorithm.

### References

1. [Gathen99] (page 1245)

[Gathen99] (page 1245)

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow_mod

>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

`sympy.polys.galoistools.gf_gcd(f, g, p, K)`  
Euclidean Algorithm in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcd

>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]

sympy.polys.galoistools.gf_lcm(f, g, p, K)
Compute polynomial LCM in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lcm

>>> gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 2, 0, 4]

sympy.polys.galoistools.gf_cofactors(f, g, p, K)
Compute polynomial GCD and cofactors in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_cofactors

>>> gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
([1, 3], [3, 3], [2, 1])

sympy.polys.galoistools.gf_gcdex(f, g, p, K)
Extended Euclidean Algorithm in GF(p)[x].
```

Given polynomials  $f$  and  $g$  in  $\text{GF}(p)[x]$ , computes polynomials  $s, t$  and  $h$ , such that  $h = \text{gcd}(f, g)$  and  $s*f + t*g = h$ . The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials  $f = (x + 7)(x + 1)$ ,  $g = (x + 7)(x^{**}2 + 1)$  in  $\text{GF}(11)[x]$ . Application of Extended Euclidean Algorithm gives:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcdex, gf_mul, gf_add

>>> s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> s, t, g
([5, 6], [6], [1, 7])
```

As result we obtained polynomials  $s = 5*x + 6$  and  $t = 6$ , and additionally  $\text{gcd}(f, g) = x + 7$ . This is correct because:

```
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)

>>> gf_add(S, T, 11, ZZ) == [1, 7]
True
```

## References

1.[Gathen99] (page 1245)

[Gathen99] (page 1245)

```
sympy.polys.galoistools.gf_monic(f, p, K)
Compute LC and a monic polynomial in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_monic

>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

```
sympy.polys.galoistools.gf_diff(f, p, K)
Differentiate polynomial in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_diff

>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

```
sympy.polys.galoistools.gf_eval(f, a, p, K)
Evaluate f(a) in GF(p) using Horner scheme.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_eval

>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

```
sympy.polys.galoistools.gf_multi_eval(f, A, p, K)
Evaluate f(a) for a in [a_1, ..., a_n].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_multi_eval

>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

```
sympy.polys.galoistools.gf_compose(f, g, p, K)
Compute polynomial composition f(g) in GF(p)[x].
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose

>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]

sympy.polys.galoistools.gf_compose_mod(g, h, f, p, K)
Compute polynomial composition g(h) in GF(p)[x]/(f).
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose_mod

>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]), 5, ZZ)
[4]

sympy.polys.galoistools.gf_trace_map(a, b, c, n, f, p, K)
Compute polynomial trace map in GF(p)[x]/(f).
```

Given a polynomial  $f$  in  $\text{GF}(p)[x]$ , polynomials  $a, b, c$  in the quotient ring  $\text{GF}(p)[x]/(f)$  such that  $b = c^{**t} \pmod{f}$  for some positive power  $t$  of  $p$ , and a positive integer  $n$ , returns a mapping:

$a \rightarrow a^{**t**n}, a + a^{**t} + a^{**t**2} + \dots + a^{**t**n} \pmod{f}$

In factorization context,  $b = x^{**p} \pmod{f}$  and  $c = x \pmod{f}$ . This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

## References

1.[Gathen92] (page 1245)

[Gathen92] (page 1245)

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_trace_map

>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])

sympy.polys.galoistools.gf_random(n, p, K)
Generate a random polynomial in GF(p)[x] of degree n.
```

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_random
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 1, 2, 0, 4, 2]
```

`sympy.polys.galoistools.gf_irreducible(n, p, K)`  
Generate random irreducible polynomial of degree n in GF(p)[x].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

`sympy.polys.galoistools.gf_irreducible_p(f, p, K)`  
Test irreducibility of a polynomial f in GF(p)[x].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible_p

>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_p(f, p, K)`  
Return True if f is square-free in GF(p)[x].

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_p

>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 2, 1, 4]), 5, ZZ)
False
```

`sympy.polys.galoistools.gf_sqf_part(f, p, K)`  
Return square-free part of a GF(p)[x] polynomial.

### Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_part

>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

---

```
sympy.polys.galoistools.gf_sqf_list(f, p, K, all=False)
```

Return the square-free decomposition of a  $\text{GF}(p)[x]$  polynomial.

Given a polynomial  $f$  in  $\text{GF}(p)[x]$ , returns the leading coefficient of  $f$  and a square-free decomposition  $f_1^{e_1} f_2^{e_2} \dots f_k^{e_k}$  such that all  $f_i$  are monic polynomials and  $(f_i, f_j)$  for  $i \neq j$  are co-prime and  $e_1 \dots e_k$  are given in increasing order. All trivial terms (i.e.  $f_i = 1$ ) aren't included in the output.

Consider polynomial  $f = x^{11} + 1$  over  $\text{GF}(11)[x]$ :

```
>>> from sympy.polys.domains import ZZ

>>> from sympy.polys.galoistools import (
...     gf_from_dict, gf_diff, gf_sqf_list, gf_pow,
... )
...
...
>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)
```

Note that  $f'(x) = 0$ :

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon doesn't happen in characteristic zero. However we can still compute square-free decomposition of  $f$  using `gf_sqf()`:

```
>>> gf_sqf_list(f, 11, ZZ)
(1, [[([1, 1], 11)])
```

We obtained factorization  $f = (x + 1)^{11}$ . This is correct because:

```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

## References

1. [Geddes92] (page 1245)

[Geddes92] (page 1245)

```
sympy.polys.galoistools.gf_Qmatrix(f, p, K)
```

Calculate Berlekamp's Q matrix.

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix

>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]

>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0],
 [0, 4, 0, 0],
```

```
[0, 0, 1, 0],  
[0, 0, 0, 4]]
```

`sympy.polys.galoistools.gf_Qbasis(Q, p, K)`  
Compute a basis of the kernel of  $\mathbb{Q}$ .

#### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_Qmatrix, gf_Qbasis  
  
>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)  
[[1, 0, 0, 0], [0, 0, 1, 0]]  
  
>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)  
[[1, 0]]
```

`sympy.polys.galoistools.gf_berlekamp(f, p, K)`  
Factor a square-free  $f$  in  $\text{GF}(p)[x]$  for small  $p$ .

#### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_berlekamp  
  
>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)  
[[1, 0, 2], [1, 0, 3]]
```

`sympy.polys.galoistools.gf_zassenhaus(f, p, K)`  
Factor a square-free  $f$  in  $\text{GF}(p)[x]$  for medium  $p$ .

#### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_zassenhaus  
  
>>> gf_zassenhaus(ZZ.map([1, 4, 3]), 5, ZZ)  
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_shoup(f, p, K)`  
Factor a square-free  $f$  in  $\text{GF}(p)[x]$  for large  $p$ .

#### Examples

```
>>> from sympy.polys.domains import ZZ  
>>> from sympy.polys.galoistools import gf_shoup  
  
>>> gf_shoup(ZZ.map([1, 4, 3]), 5, ZZ)  
[[1, 1], [1, 3]]
```

`sympy.polys.galoistools.gf_factor_sqf(f, p, K, method=None)`  
Factor a square-free polynomial  $f$  in  $\text{GF}(p)[x]$ .

## Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor_sqf

>>> gf_factor_sqf(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [[1, 1], [1, 3]])

sympy.polys.galoistools.gf_factor(f, p, K)
Factor (non square-free) polynomials in GF(p)[x].
```

Given a possibly non square-free polynomial  $f$  in  $\text{GF}(p)[x]$ , returns its complete factorization into irreducibles:

```
f_1(x)**e_1 f_2(x)**e_2 ... f_d(x)**e_d
```

where each  $f_i$  is a monic polynomial and  $\text{gcd}(f_i, f_j) == 1$ , for  $i \neq j$ . The result is given as a tuple consisting of the leading coefficient of  $f$  and a list of factors of  $f$  with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of  $f$  and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial  $f = (7*x + 1) (x + 2)^2$  in  $\text{GF}(11)[x]$ . We obtain its factorization into irreducibles as follows:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

>>> gf_factor(ZZ.map([5, 2, 7, 2]), 11, ZZ)
(5, [[(1, 2), 1], [(1, 8), 2]])
```

We arrived with factorization  $f = 5 (x + 2) (x + 8)^2$ . We didn't recover the exact form of the input polynomial because we requested to get monic factors of  $f$  and its leading coefficient separately.

Square-free factors of  $f$  can be factored into irreducibles over  $\text{GF}(p)$  using three very different methods:

**Berlekamp** efficient for very small values of  $p$  (usually  $p < 25$ )

**Cantor-Zassenhaus** efficient on average input and with “typical”  $p$

**Shoup-Kaltofen-Gathen** efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set `GF_FACTOR_METHOD` with one of `berlekamp`, `zassenhaus` or `shoup` values.

## References

1. [Gathen99] (page 1245)

[Gathen99] (page 1245)

```
sympy.polys.galoistools.gf_value(f, a)
Value of polynomial 'f' at 'a' in field R.
```

## Examples

```
>>> from sympy.polys.galoistools import gf_value
```

```
>>> gf_value([1, 7, 2, 4], 11)
2204

sympy.polys.galoistools.gf_csolve(f, n)
To solve f(x) congruent 0 mod(n).
```

n is divided into canonical factors and f(x) cong 0 mod( $p^{**}e$ ) will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

## References

- [1] ‘An introduction to the Theory of Numbers’ 5th Edition by Ivan Niven, Zuckerman and Montgomery.

## Examples

Solve [1, 1, 7] congruent 0 mod(189):

```
>>> from sympy.polys.galoistools import gf_csolve
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```

**Manipulation of sparse, distributed polynomials and vectors** Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a *sparse* representation.

Sparse polynomials are represented as dictionaries.

```
sympy.polys.rings.ring(symbols, domain, order=LexOrder())
Construct a polynomial ring returning (ring, x_1, ..., x_n).
```

**Parameters** `symbols` : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

`domain` : Domain (page 752) or coercible

`order` : Order (page 848) or coercible, optional, defaults to lex

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> R, x, y, z = ring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

```
sympy.polys.rings.xring(symbols, domain, order=LexOrder())
Construct a polynomial ring returning (ring, (x_1, ..., x_n)).
```

**Parameters** `symbols` : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)  
`domain` : Domain (page 752) or coercible  
`order` : Order (page 848) or coercible, optional, defaults to lex

### Examples

```
>>> from sympy.polys.rings import xring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> R, (x, y, z) = xring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.vring(symbols, domain, order=LexOrder())`

Construct a polynomial ring and inject `x_1`, ..., `x_n` into the global namespace.

**Parameters** `symbols` : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)  
`domain` : Domain (page 752) or coercible  
`order` : Order (page 848) or coercible, optional, defaults to lex

### Examples

```
>>> from sympy.polys.rings import vring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex

>>> vring("x,y,z", ZZ, lex)
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>
```

`sympy.polys.rings.sring(exprs, *symbols, **options)`

Construct a ring deriving generators and domain from options and input expressions.

**Parameters** `exprs` : Expr (page 74) or sequence of Expr (page 74) (sympifiable)  
`symbols` : sequence of Symbol (page 95)/Expr (page 74)  
`options` : keyword arguments understood by Options (page 848)

### Examples

```
>>> from sympy.core import symbols
>>> from sympy.polys.rings import sring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex
```

```
>>> x, y, z = symbols("x,y,z")
>>> R, f = ring(x + 2*y + 3*z)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> f
x + 2*y + 3*z
>>> type(_)
<class 'sympy.polys.rings.PolyElement'>

class sympy.polys.rings.PolyRing
    Multivariate distributed polynomial ring.

    add(*objs)
        Add a sequence of polynomials or containers of polynomials.
```

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> R, x = ring("x", ZZ)
>>> R.add([ x**2 + 2*i + 3 for i in range(4) ])
4*x**2 + 24
>>> _.factor_list()
(4, [(x**2 + 6, 1)])

drop(*gens)
    Remove specified generators from this ring.

drop_to_ground(*gens)
    Remove specified generators from the ring and inject them into its domain.

index(gen)
    Compute index of gen in self.gens.

monomial_basis(i)
    Return the ith-basis element.

mul(*objs)
    Multiply a sequence of polynomials or containers of polynomials.
```

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> R, x = ring("x", ZZ)
>>> R.mul([ x**2 + 2*i + 3 for i in range(4) ])
x**8 + 24*x**6 + 206*x**4 + 744*x**2 + 945
>>> _.factor_list()
(1, [(x**2 + 3, 1), (x**2 + 5, 1), (x**2 + 7, 1), (x**2 + 9, 1)])

class sympy.polys.rings.PolyElement
    Element of multivariate distributed polynomial ring.

    almosteq(p1, p2, tolerance=None)
        Approximate equality test for polynomials.
```

`cancel(g)`  
Cancel common factors in a rational function  $f/g$ .

#### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

`coeff(element)`  
Returns the coefficient that stands next to the given monomial.

**Parameters** `element` : PolyElement (with `is_monomial = True`) or 1

#### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = ring("x,y,z", ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23

>>> f.coeff(x**2*y)
3
>>> f.coeff(x*y)
0
>>> f.coeff(1)
23
```

`coeffs(order=None)`  
Ordered list of polynomial coefficients.

**Parameters** `order` : Order (page 848) or coercible, optional

#### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3

>>> f.coeffs()
[2, 1]
>>> f.coeffs(grlex)
[1, 2]
```

`compose(f, x, a=None)`  
Computes the functional composition.

`const()`  
Returns the constant coefficient.

**content(*f*)**

Returns GCD of polynomial's coefficients.

**copy()**

Return a copy of polynomial self.

Polynomials are mutable; if one is interested in preserving a polynomial, and one plans to use inplace operations, one can copy the polynomial. This method makes a shallow copy.

**Examples**

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.rings import ring

>>> R, x, y = ring('x, y', ZZ)
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
>>> p[R.zero_monom] = 3
>>> p
x**2 + 2*x*y + y**2 + 3
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + 2*x*y + y**2 + 3
```

**degree(*f*, *x=None*)**

The leading degree in *x* or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo).

**degrees(*f*)**

A tuple containing leading degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**diff(*f*, *x*)**

Computes partial derivative in *x*.

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring("x,y", ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1
```

**div(*fv*)**

Division algorithm, see [CLO] p64.

**fv array of polynomials** return qv, r such that self = sum(fv[i]\*qv[i]) + r

All polynomials are required not to be Laurent polynomials.

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> f = x**3
>>> f0 = x - y**2
>>> f1 = x - y
>>> qv, r = f.div((f0, f1))
>>> qv[0]
x**2 + x*y**2 + y**4
>>> qv[1]
0
>>> r
y**6
```

### `imul_num(p, c)`

multiply inplace the polynomial p by an element in the coefficient ring, provided p is not one of the generators; else multiply not inplace

## Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p1 = p.imul_num(3)
>>> p1
3*x + 3*y**2
>>> p1 is p
True
>>> p = x
>>> p1 = p.imul_num(3)
>>> p1
3*x
>>> p1 is p
False
```

### `itercoeffs()`

Iterator over coefficients of a polynomial.

### `itermonoms()`

Iterator over monomials of a polynomial.

### `iterterms()`

Iterator over terms of a polynomial.

### `leading_expv()`

Leading monomial tuple according to the monomial ordering.

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y, z = ring('x, y, z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)

leading_monom()
```

Leading monomial as a polynomial element.

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_monom()
x*y
```

```
leading_term()
```

Leading term as a polynomial element.

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ

>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

```
listcoeffs()
```

Unordered list of polynomial coefficients.

```
listmonoms()
```

Unordered list of polynomial monomials.

```
listterms()
```

Unordered list of polynomial terms.

```
monic(f)
```

Divides all coefficients by the leading coefficient.

```
monoms(order=None)
```

Ordered list of polynomial monomials.

**Parameters** `order` : `Order` (page 848) or coercible, optional

**Examples**

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3

>>> f.monoms()
[(2, 3), (1, 7)]
>>> f.monoms(grlex)
[(1, 7), (2, 3)]
```

**primitive(f)**  
Returns content and a primitive polynomial.

**square()**  
square of a polynomial

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p.square()
x**2 + 2*x*y**2 + y**4
```

**strip\_zero()**  
Eliminate monomials with zero coefficient.

**tail\_degree(f, x=None)**  
The tail degree in x or the main variable.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**tail\_degrees(f)**  
A tuple containing tail degrees in all variables.

Note that the degree of 0 is negative infinity (the SymPy object -oo)

**terms(order=None)**  
Ordered list of polynomial terms.

**Parameters** `order` : `Order` (page 848) or coercible, optional

### Examples

```
>>> from sympy.polys.rings import ring
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.orderings import lex, grlex

>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.terms()
[((2, 3), 2), ((1, 7), 1)]
>>> f.terms(grlex)
[((1, 7), 1), ((2, 3), 2)]
```

In commutative algebra, one often studies not only polynomials, but also *modules* over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called  $f_1, f_2, \dots$ .

`sympy.polys.distributedmodules.sdm_monomial_mul(M, X)`

Multiply tuple  $X$  representing a monomial of  $K[X]$  into the tuple  $M$  representing a monomial of  $F$ .

### Examples

Multiplying  $xy^3$  into  $x f_1$  yields  $x^2 y^3 f_1$ :

```
>>> from sympy.polys.distributedmodules import sdm_monomial_mul
>>> sdm_monomial_mul((1, 1, 0), (1, 3))
(1, 2, 3)
```

`sympy.polys.distributedmodules.sdm_monomial_deg(M)`

Return the total degree of  $M$ .

### Examples

For example, the total degree of  $x^2 y f_5$  is 3:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_deg
>>> sdm_monomial_deg((5, 2, 1))
3
```

`sympy.polys.distributedmodules.sdm_monomial_divides(A, B)`

Does there exist a (polynomial) monomial  $X$  such that  $XA = B$ ?

### Examples

Positive examples:

In the following examples, the monomial is given in terms of  $x, y$  and the generator(s),  $f_1, f_2$  etc. The tuple form of that monomial is used in the call to `sdm_monomial_divides`. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

$A = f_1$  divides  $B = f_1$

```
>>> from sympy.polys.distributedmodules import sdm_monomial_divides
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

$A = f_1$  divides  $B = x^2 y f_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$  divides  $B = x^2yf_5$

```
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

$A = f_1$  does not divide  $B = f_2$

```
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

$A = xf_1$  does not divide  $B = f_1$

```
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

$A = xy^2f_5$  does not divide  $B = yf_5$

```
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

`sympy.polys.distributedmodules.sdm_LC(f, K)`

Returns the leading coefficient of  $f$ .

`sympy.polys.distributedmodules.sdm_to_dict(f)`

Make a dictionary from a distributed polynomial.

`sympy.polys.distributedmodules.sdm_from_dict(d, O)`

Create an sdm from a dictionary.

Here  $O$  is the monomial order to use.

```
>>> from sympy.polys.distributedmodules import sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
>>> sdm_from_dict(dic, lex)
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

`sympy.polys.distributedmodules.sdm_add(f, g, O, K)`

Add two module elements  $f, g$ .

Addition is done over the ground field  $K$ , monomials are ordered according to  $O$ .

## Examples

All examples use lexicographic order.

$$(xyf_1) + (f_2) = f_2 + xyf_1$$

```
>>> from sympy.polys.distributedmodules import sdm_add
>>> from sympy.polys import lex, QQ
>>> sdm_add([(1, 1, 1), QQ(1)], [(2, 0, 0), QQ(1)], lex, QQ)
[((2, 0, 0), 1), ((1, 1, 1), 1)]
```

$$(xyf_1) + (-xyf_1) = 0^c$$

```
>>> sdm_add([(1, 1, 1), QQ(1)], [(-1, 1, 1), QQ(-1)], lex, QQ)
[]
```

$$(f_1) + (2f_1) = 3f_1$$

```
>>> sdm_add([(1, 0, 0), QQ(1)], [(1, 0, 0), QQ(2)], lex, QQ)
[((1, 0, 0), 3)]
```

$$(yf_1) + (xf_1) = xf_1 + yf_1$$

```
>>> sdm_add([(1, 0, 1), QQ(1)], [(1, 1, 0), QQ(1)], lex, QQ)
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

`sympy.polys.distributedmodules.sdm_LM(f)`

Returns the leading monomial of  $f$ .

Only valid if  $f \neq 0$ .

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_LM, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
>>> sdm_LM(sdm_from_dict(dic, lex))
(4, 0, 1)
```

`sympy.polys.distributedmodules.sdm_LT(f)`

Returns the leading term of  $f$ .

Only valid if  $f \neq 0$ .

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

`sympy.polys.distributedmodules.sdm_mul_term(f, term, O, K)`

Multiply a distributed module element  $f$  by a (polynomial) term  $\text{term}$ .

Multiplication of coefficients is done over the ground field  $K$ , and monomials are ordered according to  $O$ .

### Examples

$$0f_1 = 0$$

```
>>> from sympy.polys.distributedmodules import sdm_mul_term
>>> from sympy.polys import lex, QQ
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((0, 0), QQ(0)), lex, QQ)
[]
```

$$x0 = 0$$

```
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

$$(x)(f_1) = xf_1$$

```
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

$$(2xy)(3xf_1 + 4yf_2) = 8xy^2f_2 + 6x^2yf_1$$

```
>>> f = [(2, 0, 1), QQ(4)), ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

`sympy.polys.distributedmodules.sdm_zero()`

Return the zero module element.

`sympy.polys.distributedmodules.sdm_deg(f)`

Degree of  $f$ .

This is the maximum of the degrees of all its monomials. Invalid if  $f$  is zero.

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_deg
>>> sdm_deg([(1, 2, 3), 1], ((10, 0, 1), 1), ((2, 3, 4), 4)])
7
```

`sympy.polys.distributedmodules.sdm_from_vector(vec, O, K, **opts)`

Create an sdm from an iterable of expressions.

Coefficients are created in the ground field  $K$ , and terms are ordered according to monomial order  $O$ . Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> sdm_from_vector([x**2+y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

`sympy.polys.distributedmodules.sdm_to_vector(f, gens, K, n=None)`

Convert sdm  $f$  into a list of polynomial expressions.

The generators for the polynomial ring are specified via `gens`. The rank of the module is guessed, or passed via `n`. The ground field is assumed to be  $K$ .

### Examples

```
>>> from sympy.polys.distributedmodules import sdm_to_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> f = [(1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0), QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

**Polynomial factorization algorithms** Many variants of Euclid's algorithm:

**Classical remainder sequence** Let  $K$  be a field, and consider the ring  $K[X]$  of polynomials in a single indeterminate  $X$  with coefficients in  $K$ . Given two elements  $f$  and  $g$  of  $K[X]$  with  $g \neq 0$  there are unique polynomials  $q$  and  $r$  such that  $f = qg + r$  and  $\deg(r) < \deg(g)$  or  $r = 0$ . They are denoted by  $\text{quo}(f, g)$  (*quotient*) and  $\text{rem}(f, g)$  (*remainder*), so we have the *division identity*

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal  $I$  of  $K[X]$  is a principal ideal, generated by any element  $\neq 0$  of minimum degree (assuming  $I$  non-zero). In fact, if  $g$  is such a polynomial and  $f$  is any element of  $I$ ,  $\text{rem}(f, g)$  belongs to  $I$  as a linear combination of  $f$  and  $g$ , hence must be zero; therefore  $f$  is a multiple of  $g$ .

Using this result it is possible to find a [greatest common divisor](#) ( $\gcd$ ) of any polynomials  $f, g, \dots$  in  $K[X]$ . If  $I$  is the ideal formed by all linear combinations of the given polynomials with coefficients in  $K[X]$ , and  $d$  is its generator, then every common divisor of the polynomials also divides  $d$ . On the other hand, the given polynomials are multiples of the generator  $d$ ; hence  $d$  is a  $\gcd$  of the polynomials, denoted  $\gcd(f, g, \dots)$ .

An algorithm for the  $\gcd$  of two polynomials  $f$  and  $g$  in  $K[X]$  can now be obtained as follows. By the division identity,  $r = \text{rem}(f, g)$  is in the ideal generated by  $f$  and  $g$ , as well as  $f$  is in the ideal generated by  $g$  and  $r$ . Hence the ideals generated by the pairs  $(f, g)$  and  $(g, r)$  are the same. Set  $f_0 = f$ ,  $f_1 = g$ , and define recursively  $f_i = \text{rem}(f_{i-2}, f_{i-1})$  for  $i \geq 2$ . The recursion ends after a finite number of steps with  $f_{k+1} = 0$ , since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs  $(f_{i-1}, f_i)$  generate the same ideal. In particular, the ideal generated by  $f$  and  $g$  is generated by  $f_k$  alone as  $f_{k+1} = 0$ . Hence  $d = f_k$  is a  $\gcd$  of  $f$  and  $g$ .

The sequence of polynomials  $f_0, f_1, \dots, f_k$  is called the *Euclidean polynomial remainder sequence* determined by  $(f, g)$  because of the analogy with the classical [Euclidean algorithm](#) for the  $\gcd$  of natural numbers.

The algorithm may be extended to obtain an expression for  $d$  in terms of  $f$  and  $g$  by using the full division identities to write recursively each  $f_i$  as a linear combination of  $f$  and  $g$ . This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to [Bezout's identity](#) in the case of integers.

```
sympy.polys.euclidtools.dmp_half_gcdex(f, g, u, K)
Half extended Euclidean algorithm in F[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
sympy.polys.euclidtools.dmp_gcdex(f, g, u, K)
Extended Euclidean algorithm in F[X].
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
sympy.polys.euclidtools.dmp_invert(f, g, u, K)
Compute multiplicative inverse of f modulo g in F[X].
```

**Examples**

```
>>> from sympy.polys import ring, QQ
>>> R, x = ring("x", QQ)

sympy.polys.euclidtools.dmp_euclidean_prs(f, g, u, K)
    Euclidean polynomial remainder sequence (PRS) in  $K[X]$ .
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

**Simplified remainder sequences** Assume, as is usual, that the coefficient field  $K$  is the field of fractions of an integral domain  $A$ . In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If  $A$  is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in  $K[X]$  is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial  $f$  in  $K[X]$  can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation  $f = cF$  where  $c \in K$  is the *content* of  $f$  and  $F$  is a *primitive* polynomial, i.e., a polynomial in  $A[X]$  with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials  $f$  and  $g$  with their primitive parts. This will only modify  $\text{rem}(f, g)$  by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive  $\text{gcd}(f, g)$ .

This sequence is the *primitive polynomial remainder sequence*. It is an example of *general polynomial remainder sequences* where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

```
sympy.polys.euclidtools.dmp_primitive_prs(f, g, u, K)
    Primitive polynomial remainder sequence (PRS) in  $K[X]$ .
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

**Subresultant sequence** The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called *subresultant polynomials* of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain  $A$ , so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in  $A$ .

To see how subresultants are associated with remainder sequences recall that all polynomials  $h$  in the sequence are linear combinations of the given polynomials  $f$  and  $g$

$$h = uf + vg$$

with polynomials  $u$  and  $v$  in  $K[X]$ . Moreover, as is seen from the extended Euclidean algorithm, the degrees of  $u$  and  $v$  are relatively low, with limited growth from step to step.

Let  $n = \deg(f)$ , and  $m = \deg(g)$ , and assume  $n \geq m$ . If  $\deg(h) = j < m$ , the coefficients of the powers  $X^k$  ( $k > j$ ) in the products  $uf$  and  $vg$  cancel each other. In particular, the products must have the same degree, say,  $l$ . Then  $\deg(u) = l - n$  and  $\deg(v) = l - m$  with a total of  $2l - n - m + 2$  coefficients to be determined.

On the other hand, the equality  $h = uf + vg$  implies that  $l - j$  linear combinations of the coefficients are zero, those associated with the powers  $X^i$  ( $j < i \leq l$ ), and one has a given non-zero value, namely the leading coefficient of  $h$ .

To satisfy these  $l - j + 1$  linear equations the total number of coefficients to be determined cannot be lower than  $l - j + 1$ , in general. This leads to the inequality  $l \geq n + m - j - 1$ . Taking  $l = n + m - j - 1$ , we obtain  $\deg(u) = m - j - 1$  and  $\deg(v) = n - j - 1$ .

In the case  $j = 0$  the matrix of the resulting system of linear equations is the [Sylvester matrix](#)  $S(f, g)$  associated to  $f$  and  $g$ , an  $(n+m) \times (n+m)$  matrix with coefficients of  $f$  and  $g$  as entries. Its determinant is the [resultant](#)  $\text{res}(f, g)$  of the pair  $(f, g)$ . It is non-zero if and only if  $f$  and  $g$  are relatively prime.

For any  $j$  in the interval from 0 to  $m$  the matrix of the linear system is an  $(n+m-2j) \times (n+m-2j)$  submatrix of the Sylvester matrix. Its determinant  $s_j(f, g)$  is called the  $j$  th *scalar subresultant* of  $f$  and  $g$ .

If  $s_j(f, g)$  is not zero, the associated equation  $h = uf + vg$  has a unique solution where  $\deg(h) = j$  and the leading coefficient of  $h$  has any given value; the one with leading coefficient  $s_j(f, g)$  is the  $j$  th *subresultant polynomial* or, briefly, *subresultant* of the pair  $(f, g)$ , and denoted  $S_j(f, g)$ . This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if  $f$  and  $g$  are in  $A[X]$ , so is  $S_j(f, g)$  as well. This construction of subresultants applies to any  $j$  between 0 and  $m$  regardless of the value of  $s_j(f, g)$ ; if it is zero, then  $\deg(S_j(f, g)) < j$ .

The properties of subresultants are as follows. Let  $n_0 = \deg(f)$ ,  $n_1 = \deg(g)$ ,  $n_2, \dots, n_k$  be the decreasing sequence of degrees of polynomials in a remainder sequence. Let  $0 \leq j \leq n_1$ ; then

- $s_j(f, g) \neq 0$  if and only if  $j = n_i$  for some  $i$ .
- $S_j(f, g) \neq 0$  if and only if  $j = n_i$  or  $j = n_i - 1$  for some  $i$ .

Normally,  $n_{i-1} - n_i = 1$  for  $1 < i \leq k$ . If  $n_{i-1} - n_i > 1$  for some  $i$  (the *abnormal* case), then  $S_{n_{i-1}-1}(f, g)$  and  $S_{n_i}(f, g)$  are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the *subresultant remainder sequence* by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \leq i \leq k).$$

In the normal case, these are the same as the  $S_{n_i}(f, g)$ . He also derived expressions for the constants  $\gamma_i$  in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of  $f_1, \dots, f_{i-1}$ , working in the field  $K$ .

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients  $\gamma_i$ . Their algorithm deals with elements of the domain  $A$  exclusively (assuming  $f, g \in A[X]$ ). However, in the abnormal case there was a problem, a division in  $A$  which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of  $f_i$  is  $s_{n_{i-2}}(f, g)$

(Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but  $s_{n_k}(f, g)$  which is not needed after finding  $f_{k+1} = 0$ . Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

```
sympy.polys.euclidtools.dmp_inner_subresultants(f, g, u, K)
Subresultant PRS algorithm in  $K[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729, -216, 16]]

>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

```
sympy.polys.euclidtools.dmp_subresultants(f, g, u, K)
Computes subresultant PRS of two polynomials in  $K[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

```
sympy.polys.euclidtools.dmp_prs_resultant(f, g, u, K)
Resultant algorithm in  $K[X]$  using subresultant PRS.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16

>>> res, prs = R.dmp_prs_resultant(f, g)

>>> res == b                      # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True

sympy.polys.euclidtools.dmp_zz_modular_resultant(f, g, p, u, K)
    Compute resultant of f and g modulo a prime p.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x + y + 2
>>> g = 2*x*y + x + 3

>>> R.dmp_zz_modular_resultant(f, g, 5)
-2*y**2 + 1
```

```
sympy.polys.euclidtools.dmp_zz_collins_resultant(f, g, u, K)
    Collins's modular resultant algorithm in  $Z[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = x + y + 2
>>> g = 2*x*y + x + 3

>>> R.dmp_zz_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

```
sympy.polys.euclidtools.dmp_qq_collins_resultant(f, g, u, K0)
    Collins's modular resultant algorithm in  $Q[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)

>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_qq_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

```
sympy.polys.euclidtools.dmp_resultant(f, g, u, K, includePRS=False)
Computes resultant of two polynomials in  $K[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9

>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
sympy.polys.euclidtools.dmp_discriminant(f, u, K)
Computes discriminant of a polynomial in  $K[X]$ .
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)

>>> R.dmp_discriminant(x**2*y + x*z + t)
-4*y*t + z**2
```

```
sympy.polys.euclidtools.dmp_rr_prs_gcd(f, g, u, K)
Computes polynomial GCD using subresultants over a ring.
```

Returns ( $h$ ,  $cff$ ,  $cfg$ ) such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

```
sympy.polys.euclidtools.dmp_ff_prs_gcd(f, g, u, K)
Computes polynomial GCD using subresultants over a field.
```

Returns ( $h$ ,  $cff$ ,  $cfg$ ) such that  $a = \gcd(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)

>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y

>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

`sympy.polys.euclidtools.dmp_zz_heu_gcd(f, g, u, K)`

Heuristic polynomial GCD in  $Z[X]$ .

Given univariate polynomials  $f$  and  $g$  in  $Z[X]$ , returns their GCD and cofactors, i.e. polynomials  $h$ ,  $cff$  and  $cfg$  such that:

```
h = gcd(f, g), cff = quo(f, h) and cfg = quo(g, h)
```

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.

The algorithm computes the polynomial GCD by evaluating polynomials  $f$  and  $g$  at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation process reduces  $f$  and  $g$  variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.

## References

1. [Liao95] (page 1245)

[Liao95] (page 1245)

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

`sympy.polys.euclidtools.dmp_qq_heu_gcd(f, g, u, K0)`

Heuristic polynomial GCD in  $Q[X]$ .

Returns  $(h, cff, cfg)$  such that  $a = \text{gcd}(f, g)$ ,  $cff = \text{quo}(f, h)$ , and  $cfg = \text{quo}(g, h)$ .

## Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)

>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y

>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)

sympy.polys.euclidtools.dmp_inner_gcd(f, g, u, K)
Computes polynomial GCD and cofactors of  $f$  and  $g$  in  $K[X]$ .
Returns ( $h$ ,  $cff$ ,  $cfg$ ) such that  $a = \gcd(f, g)$ ,  $cff = quo(f, h)$ , and  $cfg = quo(g, h)$ .
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)

sympy.polys.euclidtools.dmp_gcd(f, g, u, K)
Computes polynomial GCD of  $f$  and  $g$  in  $K[X]$ .
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_gcd(f, g)
x + y

sympy.polys.euclidtools.dmp_lcm(f, g, u, K)
Computes polynomial LCM of  $f$  and  $g$  in  $K[X]$ .
```

**Examples**

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y

>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

```
sympy.polys.euclidtools.dmp_content(f, u, K)
    Returns GCD of multivariate coefficients.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

```
sympy.polys.euclidtools.dmp_primitive(f, u, K)
    Returns multivariate content and a primitive polynomial.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)

>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

```
sympy.polys.euclidtools.dmp_cancel(f, g, u, K, include=True)
    Cancel common factors in a rational function f/g.
```

### Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)

>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:

```
sympy.polys.factor.tools.dmp_trial_division(f, factors, u, K)
    Determine multiplicities of factors using trial division.
```

```
sympy.polys.factor.tools.dmp_zz_mignotte_bound(f, u, K)
    Mignotte bound for multivariate polynomials in  $K[X]$ .
```

```
sympy.polys.factor.tools.dup_zz_hensel_step(m, f, g, h, s, t, K)
    One step in Hensel lifting in  $Z[x]$ .
```

Given positive integer  $m$  and  $Z[x]$  polynomials  $f, g, h, s$  and  $t$  such that:

```
f == g*h (mod m)
s*g + t*h == 1 (mod m)

lc(f) is not a zero divisor (mod m)
lc(h) == 1

deg(f) == deg(g) + deg(h)
deg(s) < deg(h)
deg(t) < deg(g)
```

returns polynomials  $G$ ,  $H$ ,  $S$  and  $T$ , such that:

```
f == G*H (mod m**2)
S*G + T**H == 1 (mod m**2)
```

## References

1.[Gathen99] (page 1245)

[Gathen99] (page 1245)

`sympy.polys.factor.tools.dup_zz_hensel_lift(p, f, f_list, l, K)`

Multifactor Hensel lifting in  $Z[x]$ .

Given a prime  $p$ , polynomial  $f$  over  $Z[x]$  such that  $lc(f)$  is a unit modulo  $p$ , monic pair-wise coprime polynomials  $f_i$  over  $Z[x]$  satisfying:

```
f = lc(f) f_1 ... f_r (mod p)
```

and a positive integer  $l$ , returns a list of monic polynomials  $F_1, F_2, \dots, F_r$  satisfying:

```
f = lc(f) F_1 ... F_r (mod p**l)
```

```
F_i = f_i (mod p), i = 1..r
```

## References

1.[Gathen99] (page 1245)

[Gathen99] (page 1245)

`sympy.polys.factor.tools.dup_zz_zassenhaus(f, K)`

Factor primitive square-free polynomials in  $Z[x]$ .

`sympy.polys.factor.tools.dup_zz_irreducible_p(f, K)`

Test irreducibility using Eisenstein's criterion.

`sympy.polys.factor.tools.dup_cyclotomic_p(f, K, irreducible=False)`

Efficiently test if  $f$  is a cyclotomic polynomial.

## Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False

>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```

`sympy.polys.factor.tools.dup_zz_cyclotomic_poly(n, K)`

Efficiently generate n-th cyclotomic polynomial.

```
sympy.polys.factor.tools.dup_zz_cyclotomic_factor(f, K)
```

Efficiently factor polynomials  $x^{*n} - 1$  and  $x^{*n} + 1$  in  $Z[x]$ .

Given a univariate polynomial  $f$  in  $Z[x]$  returns a list of factors of  $f$ , provided that  $f$  is in the form  $x^{*n} - 1$  or  $x^{*n} + 1$  for  $n >= 1$ . Otherwise returns None.

Factorization is performed using cyclotomic decomposition of  $f$ , which makes this method much faster than any other direct factorization approach (e.g. Zassenhaus's).

## References

1.[Weisstein09] (page 1245)

[Weisstein09] (page 1245)

```
sympy.polys.factor.tools.dup_zz_factor_sqf(f, K)
```

Factor square-free (non-primitive) polynomials in  $Z[x]$ .

```
sympy.polys.factor.tools.dup_zz_factor(f, K)
```

Factor (non square-free) polynomials in  $Z[x]$ .

Given a univariate polynomial  $f$  in  $Z[x]$  computes its complete factorization  $f_1, \dots, f_n$  into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial  $f = 2 * x^{*4} - 2$ :

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)

>>> R.dup_zz_factor(2*x**4 - 2)
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
```

In result we got the following factorization:

```
f = 2 (x - 1) (x + 1) (x**2 + 1)
```

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials  $x^{*n} - 1$  and  $x^{*n} + 1$  are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set cyclotomic=False.

## References

1.[Gathen99] (page 1245)

[Gathen99] (page 1245)

```
sympy.polys.factor.tools.dmp_zz_wang_non_divisors(E, cs, ct, K)
```

Wang/EEZ: Compute a set of valid divisors.

```
sympy.polys.factor.tools.dmp_zz_wang_test_points(f, T, ct, A, u, K)
Wang/EEZ: Test evaluation points for suitability.

sympy.polys.factor.tools.dmp_zz_wang_lead_coeffs(f, T, cs, E, H, A, u, K)
Wang/EEZ: Compute correct leading coefficients.

sympy.polys.factor.tools.dmp_zz_diophantine(F, c, A, d, p, u, K)
Wang/EEZ: Solve multivariate Diophantine equations.

sympy.polys.factor.tools.dmp_zz_wang_hensel_lifting(f, H, LC, A, p, u, K)
Wang/EEZ: Parallel Hensel lifting algorithm.
```

`sympy.polys.factor.tools.dmp_zz_wang(f, u, K, mod=None, seed=None)`  
Factor primitive square-free polynomials in  $Z[X]$ .

Given a multivariate polynomial  $f$  in  $Z[x_1, \dots, x_n]$ , which is primitive and square-free in  $x_1$ , computes factorization of  $f$  into irreducibles over integers.

The procedure is based on Wang's Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing  $f$  as a univariate polynomial in  $Z[x_2, \dots, x_n][x_1]$ , for which an evaluation mapping is computed:

```
x_2 -> a_2, ..., x_n -> a_n
```

where  $a_i$ , for  $i = 2, \dots, n$ , are carefully chosen integers. The mapping is used to transform  $f$  into a univariate polynomial in  $Z[x_1]$ , which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter `seed` is passed to `_randint` and can be used to seed `randint` (when an integer) or (for testing purposes) can be a sequence of numbers.

## References

1. [Wang78] (page 1245)
2. [Geddes92] (page 1245)

[Wang78] (page 1245), [Geddes92] (page 1245)

```
sympy.polys.factor.tools.dmp_zz_factor(f, u, K)
Factor (non square-free) polynomials in  $Z[X]$ .
```

Given a multivariate polynomial  $f$  in  $Z[x]$  computes its complete factorization  $f_1, \dots, f_n$  into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial  $f = 2 * (x ** 2 - y ** 2)$ :

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

```
f = 2 (x - y) (x + y)
```

## References

1.[Gathen99] (page 1245)

[Gathen99] (page 1245)

```
sympy.polys.factor.tools.dmp_ext_factor(f, u, K)
    Factor multivariate polynomials over algebraic number fields.

sympy.polys.factor.tools.dup_gf_factor(f, K)
    Factor univariate polynomials over finite fields.

sympy.polys.factor.tools.dmp_factor_list(f, u, K0)
    Factor polynomials into irreducibles in  $K[X]$ .

sympy.polys.factor.tools.dmp_factor_list_include(f, u, K)
    Factor polynomials into irreducibles in  $K[X]$ .

sympy.polys.factor.tools.dmp_irreducible_p(f, u, K)
    Returns True if f has no factors over its domain.
```

**Groebner basis algorithms** Groebner bases can be used to answer many problems in computational commutative algebra. Their computation is rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

```
sympy.polys.groebnertools.groebner(seq, ring, method=None)
    Computes Groebner basis for a set of polynomials in  $K[X]$ .
```

Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via `method` argument or `setup()` (page 849), where `method` can be either `buchberger` or `f5b`.

```
sympy.polys.groebnertools.spoly(p1, p2, ring)
    Compute LCM(LM(p1), LM(p2))/LM(p1)*p1 - LCM(LM(p1), LM(p2))/LM(p2)*p2 This is the S-poly
    provided p1 and p2 are monic
```

```
sympy.polys.groebnertools.red_groebner(G, ring)
    Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216
```

Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

```
sympy.polys.groebnertools.is_groebner(G, ring)
    Check if G is a Groebner basis.
```

```
sympy.polys.groebnertools.is_minimal(G, ring)
    Checks if G is a minimal Groebner basis.
```

```
sympy.polys.groebnertools.is_reduced(G, ring)
    Checks if G is a reduced Groebner basis.
```

---

```
sympy.polys.fglmtools.matrix_fglm(F, ring, O_to)
```

Converts the reduced Groebner basis F of a zero-dimensional ideal w.r.t. `O_from` to a reduced Groebner basis w.r.t. `O_to`.

## References

J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering

Groebner basis algorithms for modules are also provided:

```
sympy.polys.distributedmodules.sdm_spoly(f, g, O, K, phantom=None)
```

Compute the generalized s-polynomial of f and g.

The ground field is assumed to be K, and monomials ordered according to O.

This is invalid if either of f or g is zero.

If the leading terms of f and g involve different basis elements of F, their s-poly is defined to be zero. Otherwise it is a certain linear combination of f and g in which the leading terms cancel. See [SCA, defn 2.3.6] for details.

If `phantom` is not `None`, it should be a pair of module elements on which to perform the same operation(s) as on f and g. The in this case both results are returned.

## Examples

```
>>> from sympy.polys.distributedmodules import sdm_spoly
>>> from sympy.polys import QQ, lex
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

```
sympy.polys.distributedmodules.sdm_ecart(f)
```

Compute the ecart of f.

This is defined to be the difference of the total degree of f and the total degree of the leading monomial of f [SCA, defn 2.3.7].

Invalid if f is zero.

## Examples

```
>>> from sympy.polys.distributedmodules import sdm_ecart
>>> sdm_ecart([(1, 2, 3), 1], ((1, 0, 1), 1))
0
>>> sdm_ecart([(2, 2, 1), 1], ((1, 5, 1), 1))
3
```

```
sympy.polys.distributedmodules.sdm_nf_mora(f, G, O, K, phantom=None)
```

Compute a weak normal form of f with respect to G and order O.

The ground field is assumed to be K, and monomials ordered according to O.

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of  $G$ .

The most important property of a weak normal form is the following: if  $R$  is the ring associated with the monomial ordering (if the ordering is global, we just have  $R = K[x_1, \dots, x_n]$ , otherwise it is a certain localization thereof),  $I$  any ideal of  $R$  and  $G$  a standard basis for  $I$ , then for any  $f \in R$ , we have  $f \in I$  if and only if  $NF(f|G) = 0$ .

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If `phantom` is not `None`, it should be a pair of “phantom” arguments on which to perform the same computations as on `f`, `G`, both results are then returned.

```
sympy.polys.distributedmodules.sdm_groebner(G, NF, O, K, extended=False)
```

Compute a minimal standard basis of  $G$  with respect to order  $O$ .

The algorithm uses a normal form `NF`, for example `sdm_nf_mora`. The ground field is assumed to be  $K$ , and monomials ordered according to  $O$ .

Let  $N$  denote the submodule generated by elements of  $G$ . A standard basis for  $N$  is a subset  $S$  of  $N$ , such that  $in(S) = in(N)$ , where for any subset  $X$  of  $F$ ,  $in(X)$  denotes the submodule generated by the initial forms of elements of  $X$ . [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of  $G$ .

If `extended=True`, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of  $G$ .

This functions implements the “sugar” strategy, see

Giovini et al: “One sugar cube, please” OR Selection strategies in Buchberger algorithm.

## Exceptions

These are exceptions defined by the polynomials module.

TODO sort and explain

```
class sympy.polys.polyerrors.BasePolynomialError
    Base class for polynomial related exceptions.
```

```
class sympy.polys.polyerrors.ExactQuotientFailed(f, g, dom=None)
```

```
class sympy.polys.polyerrors.OperationNotSupported(poly, func)
```

```
class sympy.polys.polyerrors.HeuristicGCDFailed
```

```
class sympy.polys.polyerrors.HomomorphismFailed
```

```
class sympy.polys.polyerrors.IsomorphismFailed
```

```
class sympy.polys.polyerrors.ExtraneousFactors
```

```
class sympy.polys.polyerrors.EvaluationFailed
```

```
class sympy.polys.polyerrors.RefinementFailed
```

```

class sympy.polys.polyerrors.CoercionFailed
class sympy.polys.polyerrors.NotInvertible
class sympy.polys.polyerrors.NotReversible
class sympy.polys.polyerrors.NotAlgebraic
class sympy.polys.polyerrors.DomainError
class sympy.polys.polyerrors.PolynomialError
class sympy.polys.polyerrors.UnificationFailed
class sympy.polys.polyerrors.GeneratorsNeeded
class sympy.polys.polyerrors.ComputationFailed(func, nargs, exc)
class sympy.polys.polyerrors.GeneratorsError
class sympy.polys.polyerrors.UnivariatePolynomialError
class sympy.polys.polyerrors.MultivariatePolynomialError
class sympy.polys.polyerrors.PolificationFailed(opt, origs, exprs, seq=False)
class sympy.polys.polyerrors.OptionError
class sympy.polys.polyerrors.FlagError

```

## Reference

### Modular GCD

`sympy.polys.modulargcd.modgcd_univariate(f, g)`

Computes the GCD of two polynomials in  $\mathbb{Z}[x]$  using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials  $f$  and  $g$  by computing the GCD in  $\mathbb{Z}_p[x]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

**Parameters** `f` : PolyElement

univariate integer polynomial

`g` : PolyElement

univariate integer polynomial

**Returns** `h` : PolyElement

GCD of the polynomials  $f$  and  $g$

`cff` : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

`cfg` : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

## References

1. [Monagan00] (page 1246)

[Monagan00] (page 1246)

## Examples

```
>>> from sympy.polys.modulargcd import modgcd_univariate
>>> from sympy.polys import ring, ZZ

>>> R, x = ring("x", ZZ)

>>> f = x**5 - 1
>>> g = x - 1

>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)

>>> cff * h == f
True
>>> cfg * h == g
True

>>> f = 6*x**2 - 6
>>> g = 2*x**2 + 4*x + 2

>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(2*x + 2, 3*x - 3, x + 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.modgcd_bivariate(f, g)`

Computes the GCD of two polynomials in  $\mathbb{Z}[x, y]$  using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials  $f$  and  $g$  by calculating the GCD in  $\mathbb{Z}_p[x, y]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the bivariate GCD over  $\mathbb{Z}_p$ , the polynomials  $f \bmod p$  and  $g \bmod p$  are evaluated at  $y = a$  for certain  $a \in \mathbb{Z}_p$  and then their univariate GCD in  $\mathbb{Z}_p[x]$  is computed. Interpolating those yields the bivariate GCD in  $\mathbb{Z}_p[x, y]$ . To verify the result in  $\mathbb{Z}[x, y]$ , trial division is done, but only for candidates which are very likely the desired GCD.

**Parameters** `f` : PolyElement

bivariate integer polynomial

`g` : PolyElement

bivariate integer polynomial

**Returns** `h` : PolyElement

GCD of the polynomials  $f$  and  $g$

`cff` : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

`cfg` : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

## References

1.[Monagan00] (page 1246)

[Monagan00] (page 1246)

## Examples

```
>>> from sympy.polys.modulargcd import modgcd_bivariate
>>> from sympy.polys import ring, ZZ

>>> R, x, y = ring("x, y", ZZ)

>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2

>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)

>>> cff * h == f
True
>>> cfg * h == g
True

>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2

>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.modgcd_multivariate(f, g)`

Compute the GCD of two polynomials in  $\mathbb{Z}[x_0, \dots, x_{k-1}]$  using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials  $f$  and  $g$  by calculating the GCD in  $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over  $\mathbb{Z}_p$  the recursive subroutine `_modgcd_multivariate_p` is used. To verify the result in  $\mathbb{Z}[x_0, \dots, x_{k-1}]$ , trial division is done, but only for candidates which are very likely the desired GCD.

**Parameters** `f` : PolyElement

multivariate integer polynomial

`g` : PolyElement

multivariate integer polynomial

**Returns** `h` : PolyElement

GCD of the polynomials  $f$  and  $g$

`cff` : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

**cfg** : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

See also:

[\\_modgcd\\_multivariate\\_p](#) (page 844)

## References

1. [Monagan00] (page 1246)

2. [Brown71] (page 1246)

[Monagan00] (page 1246), [Brown71] (page 1246)

## Examples

```
>>> from sympy.polys.modulargcd import modgcd_multivariate
>>> from sympy.polys import ring, ZZ

>>> R, x, y = ring("x, y", ZZ)

>>> f = x**2 - y**2
>>> g = x**2 + 2*x*y + y**2

>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)

>>> cff * h == f
True
>>> cfg * h == g
True

>>> R, x, y, z = ring("x, y, z", ZZ)

>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z

>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)

>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd.func_field_modgcd(f, g)`

Compute the GCD of two polynomials  $f$  and  $g$  in  $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$  using a modular algorithm.

The algorithm first computes the primitive associate  $\check{m}_\alpha(z)$  of the minimal polynomial  $m_\alpha$  in  $\mathbb{Z}[z]$  and the primitive associates of  $f$  and  $g$  in  $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\check{m}_\alpha)[x_0]$ . Then it computes the GCD in  $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$ . This is done by calculating the GCD

in  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$  for suitable primes  $p$  and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$  is computed with a recursive subroutine, which evaluates the polynomials at  $x_{n-1} = a$  for suitable evaluation points  $a \in \mathbb{Z}_p$  and then calls itself recursively until the ground domain does no longer contain any parameters. For  $\mathbb{Z}_p[z]/(\check{m}_\alpha(z))[x_0]$  the Euclidean Algorithm is used. The results of those recursive calls are then interpolated and Rational Function Reconstruction is used to obtain the correct coefficients. The results, both in  $\mathbb{Q}(x_1, \dots, x_{n-1})[z]/(m_\alpha(z))[x_0]$  and  $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\check{m}_\alpha(z))[x_0]$ , are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in  $\mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}]$  have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this `func_field_modgcd` is called recursively.

**Parameters** `f, g` : PolyElement

polynomials in  $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

**Returns** `h` : PolyElement

monic GCD of the polynomials  $f$  and  $g$

`cff` : PolyElement

cofactor of  $f$ , i.e.  $\frac{f}{h}$

`cfg` : PolyElement

cofactor of  $g$ , i.e.  $\frac{g}{h}$

## References

1. [Hoeij04] (page 1246)

[Hoeij04] (page 1246)

## Examples

```
>>> from sympy.polys.modulargcd import func_field_modgcd
>>> from sympy.polys import AlgebraicField, QQ, ring
>>> from sympy import sqrt

>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)

>>> f = x**2 - 2
>>> g = x + sqrt(2)

>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == x + sqrt(2)
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y = ring('x, y', A)
>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y

>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == x + sqrt(2)*y
True
>>> cff * h == f
True
>>> cfg * h == g
True

>>> f = x + sqrt(2)*y
>>> g = x + y

>>> h, cff, cfg = func_field_modgcd(f, g)

>>> h == R.one
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

`sympy.polys.modulargcd._modgcd_multivariate_p(f, g, p, degbound, contbound)`

Compute the GCD of two polynomials in  $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ .

The algorithm reduces the problem step by step by evaluating the polynomials  $f$  and  $g$  at  $x_{k-1} = a$  for suitable  $a \in \mathbb{Z}_p$  and then calls itself recursively to compute the GCD in  $\mathbb{Z}_p[x_0, \dots, x_{k-2}]$ . If these recursive calls are successful for enough evaluation points, the GCD in  $k$  variables is interpolated, otherwise the algorithm returns `None`. Every time a GCD or a content is computed, their degrees are compared with the bounds. If a degree greater than the bound is encountered, then the current call returns `None` and a new evaluation point has to be chosen. If at some point the degree is smaller, the correspondent bound is updated and the algorithm fails.

**Parameters** `f` : PolyElement

multivariate integer polynomial with coefficients in  $\mathbb{Z}_p$

`g` : PolyElement

multivariate integer polynomial with coefficients in  $\mathbb{Z}_p$

`p` : Integer

prime number, modulus of  $f$  and  $g$

`degbound` : list of Integer objects

`degbound[i]` is an upper bound for the degree of the GCD of  $f$  and  $g$  in the variable  $x_i$

`contbound` : list of Integer objects

`contbound[i]` is an upper bound for the degree of the content of the GCD in  $\mathbb{Z}_p[x_i][x_0, \dots, x_{i-1}]$ , `contbound[0]` is not used and can therefore be chosen arbitrarily.

**Returns** `h` : PolyElement

GCD of the polynomials  $f$  and  $g$  or None

## References

1. [Monagan00] (page 1246)
2. [Brown71] (page 1246)

[Monagan00] (page 1246), [Brown71] (page 1246)

**Manipulation of power series** Functions in this module carry the prefix `rs_`, standing for “ring series”. They manipulate finite power series in the sparse representation provided by `polys.ring.ring`.

`sympy.polys.ring_series.rs_trunc(p1, x, prec)`  
truncate the series in the `x` variable with precision `prec`, that is modulo  $O(x^{**prec})$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

`sympy.polys.ring_series.rs_mul(p1, p2, x, prec)`  
product of series modulo  $O(x^{**prec})$   
`x` is the series variable or its position in the generators.

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_mul
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```

`sympy.polys.ring_series.rs_square(p1, x, prec)`  
square modulo  $O(x^{**prec})$

## Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_square
>>> R, x = ring('x', QQ)
```

```
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1

sympy.polys.ring_series.rs_pow(p1, n, x, prec)
    return p1**n modulo O(x**prec)
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_pow
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1

sympy.polys.ring_series.rs_series_inversion(p, x, prec)
    multivariate series inversion 1/p modulo O(x**prec)
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_inversion
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
-x**3*y**6 + x**2*y**4 - x*y**2 + 1
>>> rs_series_inversion(1 + x*y**2, y, 4)
-x*y**2 + 1

sympy.polys.ring_series.rs_series_from_list(p, c, x, prec, concur=1)
    series sum c[n]*p**n modulo O(x**prec)

reduce the number of multiplication summing concurrently ax = [1, p, p**2, ..., p**((J - 1))] s
= sum(c[i]*ax[i] for i in range(r, (r + 1)*J))*p**((K - 1)*J) with K >= (n + 1)/J
```

See also:

[sympy.polys.rings.PolyElement.compose](#) (page 815)

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_series_from_list, rs_trunc
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> pc = R.from_list(list(reversed(c)))
```

```
>>> rs_trunc(pc.compose(x, p), x, 4)
6*x**3 + 11*x**2 + 8*x + 6

sympy.polys.ring_series.rs_integrate(self, x)
    integrate p with respect to x
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_integrate
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2

sympy.polys.ring_series.rs_log(p, x, prec)
logarithm of p modulo O(x**prec)
```

### Notes

truncation of integral  $\int p \, dx$   $p^{**-1}*d \, p/dx$  is used.

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_log
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x

sympy.polys.ring_series.rs_exp(p, x, prec)
exponentiation of a series modulo O(x**prec)
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_exp
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1

sympy.polys.ring_series.rs_newton(p, x, prec)
compute the truncated Newton sum of the polynomial p
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_newton
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2

sympy.polys.ring_series.rs_hadamard_exp(p1, inverse=False)
    return sum f_i/i!*x**i from sum f_i*x**i, where x is the first variable.

If invers=True return sum f_i*i!*x**i
```

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_hadamard_exp
>>> R, x = ring('x', QQ)
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1

sympy.polys.ring_series.rs_compose_add(p1, p2)
    compute the composed sum prod(p2(x - beta) for beta root of p1)
```

### References

A. Bostan, P. Flajolet, B. Salvy and E. Schost “Fast Computation with Two Algebraic Numbers”, (2002) Research Report 4579, Institut National de Recherche en Informatique et en Automatique

### Examples

```
>>> from sympy.polys.domains import QQ
>>> from sympy.polys.rings import ring
>>> from sympy.polys.ring_series import rs_compose_add
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
x**4 - 10*x**2 + 1
```

### Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

```
class sympy.polys.polyoptions.Order
    order option to polynomial manipulation functions.

class sympy.polys.polyoptions.Options(gens, args, flags=None, strict=False)
    Options manager for polynomial manipulation module.
```

## Examples

```
>>> from sympy.polys.polyoptions import Options
>>> from sympy.polys.polyoptions import build_options

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

>>> Options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}

>>> build_options((x, y, z), {'domain': 'ZZ'})
{'auto': False, 'domain': ZZ, 'gens': (x, y, z)}
```

## Options

- Expand — boolean option
- Gens — option
- Wrt — option
- Sort — option
- Order — option
- Field — boolean option
- Greedy — boolean option
- Domain — option
- Split — boolean option
- Gaussian — boolean option
- Extension — option
- Modulus — option
- Symmetric — boolean option
- Strict — boolean option

## Flags

- Auto — boolean flag
- Frac — boolean flag
- Formal — boolean flag
- Polys — boolean flag
- Include — boolean flag
- All — boolean flag
- Gen — flag

`sympy.polys.polyconfig.setup(key, value=None)`

Assign a value to (or reset) a configuration item.

## Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

## 3.15 Printing System

See the *Printing* (page 18) section in Tutorial for introduction into printing.

This guide documents the printing system in SymPy and how it works internally.

### 3.15.1 Printer Class

Printing subsystem driver

SymPy's printing system works the following way: Any expression can be passed to a designated Printer who then is responsible to return an adequate representation of that expression.

**The basic concept is the following:**

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

Some more information how the single concepts work and who should use which:

1. The object prints itself

This was the original way of doing printing in sympy. Every class had its own latex, mathml, str and repr methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefor all printing code was combined into the different printers, which works great for built-in sympy objects, but not that good for user defined classes where it is inconvenient to patch the printers.

Nevertheless, to get a fitting representation, the printers look for a specific method in every object, that will be called if it's available and is then responsible for the representation. The name of that method depends on the specific printer and is defined under Printer.printmethod.

2. Take the best fitting method defined in the printer.

The printer loops through expr classes (class + its bases), and tries to dispatch the work to `_print_<EXPR_CLASS>`

e.g., suppose we have the following class hierarchy:

```
Basic
|
Atom
|
Number
|
Rational
```

then, for `expr=Rational(...)`, in order to dispatch, we will try calling printer methods as shown in the figure below:

```
p._print(expr)
|
|-- p._print_Rational(expr)
|
|-- p._print_Number(expr)
|
|-- p._print_Atom(expr)
|
'-- p._print_Basic(expr)
```

if `_print_Rational` method exists in the printer, then it is called, and the result is returned back.

otherwise, we proceed with trying Rational bases in the inheritance order.

3. As fall-back use the `emptyPrinter` method for the printer.

As fall-back `self.emptyPrinter` will be called with the expression. If not defined in the Printer subclass this will be the same as `str(expr)`.

The main class responsible for printing is `Printer` (see also its [source code](#)):

```
class sympy.printing.printer.Printer(settings=None)
    Generic printer
```

Its job is to provide infrastructure for implementing new printers easily.

Basically, if you want to implement a printer, all you have to do is:

1. Subclass `Printer`.
2. Define `Printer.printmethod` in your subclass. If a object has a method with that name, this method will be used for printing.
3. In your subclass, define `_print_<CLASS>` methods

For each class you want to provide printing to, define an appropriate method how to do it. For example if you want a class `FOO` to be printed in its own way, define `_print_FOO`:

```
def _print_FOO(self, e):
    ...
```

this should return how `FOO` instance `e` is printed

Also, if `BAR` is a subclass of `FOO`, `_print_FOO(bar)` will be called for instance of `BAR`, if no `_print_BAR` is provided. Thus, usually, we don't need to provide printing routines for every class we want to support – only generic routine has to be provided for a set of classes.

A good example for this are functions - for example `PrettyPrinter` only defines `_print_Function`, and there is no `_print_sin`, `_print_tan`, etc...

On the other hand, a good printer will probably have to define separate routines for `Symbol`, `Atom`, `Number`, `Integral`, `Limit`, etc...

4. If convenient, override `self.emptyPrinter`

This callable will be called to obtain printing result as a last resort, that is when no appropriate print method was found for an expression.

Examples of overloading `StrPrinter`:

```
from sympy import Basic, Function, Symbol
from sympy.printing.str import StrPrinter
```

```
class CustomStrPrinter(StrPrinter):
    """
    Examples of how to customize the StrPrinter for both a SymPy class and a
    user defined class subclassed from the SymPy Basic class.
    """

    def _print_Derivative(self, expr):
        """
        Custom printing of the SymPy Derivative class.

        Instead of:

        D(x(t), t) or D(x(t), t, t)

        We will print:

        x'      or      x''

        In this example, expr.args == (x(t), t), and expr.args[0] == x(t), and
        expr.args[0].func == x
        """
        return str(expr.args[0].func) + "''*len(expr.args[1:])

    def _print_MyClass(self, expr):
        """
        Print the characters of MyClass.s alternatively lower case and upper
        case
        """
        s = ""
        i = 0
        for char in expr.s:
            if i % 2 == 0:
                s += char.lower()
            else:
                s += char.upper()
            i += 1
        return s

    # Override the __str__ method of to use CustomStrPrinter
    Basic.__str__ = lambda self: CustomStrPrinter().doprint(self)
    # Demonstration of CustomStrPrinter:
    t = Symbol('t')
    x = Function('x')(t)
    dxdt = x.diff(t)           # dxdt is a Derivative instance
    d2xdt2 = dxdt.diff(t)     # d2xdt2 is a Derivative instance
    ex = MyClass('I like both lowercase and uppercase')

    print dxdt
    print d2xdt2
    print ex
```

The output of the above code is:

```
x'
x''
I like BoTh l0wErCaSe aNd uPpEr cAsE
```

By overriding Basic.\_\_str\_\_, we can customize the printing of anything that is subclassed from Basic.

```
printmethod = None
_print(expr, *args, **kwargs)
    Internal dispatcher
```

Tries the following concepts to print an expression:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

```
doprint(expr)
    Returns printer's representation for expr (as a string)
```

```
classmethod set_global_settings(**settings)
    Set system-wide printing settings.
```

### 3.15.2 PrettyPrinter Class

The pretty printing subsystem is implemented in `sympy.printing.pretty.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `sympy.printing.pretty.stringPict`, and `sympy.printing.pretty.symbology` for rendering nice-looking formulas.

The module `stringPict` provides a base class `stringPict` and a derived class `prettyForm` that ease the creation and manipulation of formulas that span across multiple lines.

The module `pretty_symbology` provides primitives to construct 2D shapes (`hline`, `vline`, etc) together with a technique to use unicode automatically when possible.

```
class sympy.printing.pretty.pretty.PrettyPrinter(settings=None)
    Printer, which converts an expression into 2D ASCII-art figure.
```

`printmethod = '_pretty'`

```
sympy.printing.pretty.pretty.pretty(expr, **settings)
    Returns a string containing the prettified form of expr.
```

For information on keyword arguments see `pretty_print` function.

```
sympy.printing.pretty.pretty.pretty_print(expr, **settings)
    Prints expr in pretty form.
```

`pprint` is just a shortcut for this function.

**Parameters** `expr` : expression

the expression to print

`wrap_line` : bool, optional

line wrapping enabled/disabled, defaults to True

`num_columns` : int or None, optional

number of columns before line breaking (default to None which reads the terminal width), useful when using SymPy without terminal.

`use_unicode` : bool or None, optional

use unicode characters, such as the Greek letter pi instead of the string pi.

`full_prec` : bool or string, optional

use full precision. Default to “auto”

**order** : bool or string, optional  
set to ‘none’ for long expressions if slow; default is None

### 3.15.3 CCodePrinter

This class implements C code printing (i.e. it converts Python expressions to strings of C code).

## Usage:

```
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2)
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2))
fabs(pow(x, 2))
```

```
sympy.printing.ccode.known_functions = {'sinh': 'sinh', 'asinh': 'asinh'}
```

class sympy.printing.ccode.CCodePrinter(settings={})

A printer to convert python expressions to strings of c code

printmethod = '\_ccode'

indent\_code(code)

Accepts a string of code or a list of code lines

```
sympy.printing.ccode.ccode(expr, assign_to=None, **settings)
```

Converts an expr to a string of c code

Parameters expr : Expr

A sympy expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as pi [default=15].

**user\_functions** : dict, optional

A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. `[(argument_test, cfunction_string)]`. See below for examples.

**dereference** : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract: bool, optional**

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

### Examples

```
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> ccode((2*tau)**Rational(7, 2))
'8*sqrt(2)*pow(tau, 7.0L/2.0L)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {“type” : “function”} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")],
...     "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by (`expr, True`) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau))
if (x > 0) {
tau = x + 1;
}
else {
tau = x;
}
```

Support for loops is provided through `Indexed` types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
```

```
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A))
A[0] = pow(x, 2);
if (x > 0) {
    A[1] = x + 1;
}
else {
    A[1] = x;
}
A[2] = sin(x);

sympy.printing.ccode.print_ccode(expr, **settings)
Prints C representation of the given expression.
```

### 3.15.4 Fortran Printing

The `fcode` function translates a `sympy` expression into Fortran code. The main purpose is to take away the burden of manually translating long mathematical expressions. Therefore the resulting expression should also require no (or very little) manual tweaking to make it compilable. The optional arguments of `fcode` can be used to fine-tune the behavior of `fcode` in such a way that manual changes in the result are no longer needed.

```
sympy.printing.fcode.fcode(expr, assign_to=None, **settings)
Converts an expr to a string of c code
```

**Parameters** `expr` : `Expr`

A `sympy` expression to be converted.

**assign\_to** : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

**precision** : integer, optional

The precision for numbers such as `pi` [default=15].

**user\_functions** : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. `[(argument_test, cfunction_string)]`. See below for examples.

**human** : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols\_to\_declare, not\_supported\_functions, code\_text). [default=True].

**contract**: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

**source\_format** : optional

The source format can be either ‘fixed’ or ‘free’. [default=’fixed’]

**standard** : integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

## Examples

```
>>> from sympy import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
',     8*sqrt(2.0d0)*tau**(7.0d0/2.0d0),
>>> fcode(sin(x), assign_to="s")
',     s = sin(x),
```

Custom printing can be defined for certain types by passing a dictionary of “type” : “function” to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument\_test, cfunction\_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "floor": [(lambda x: not x.is_integer, "FLOOR1"),
...               (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
',     CEIL(x) + FLOOR1(x),
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by (`expr, True`) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
    if (x > 0) then
        tau = x + 1
    else
        tau = x
    end if
```

Support for loops is provided through `Indexed` types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be

looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
'    Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i)),
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
      A(1, 1) = x**2
      if (x > 0) then
        A(2, 1) = x + 1
      else
        A(2, 1) = x
      end if
      A(3, 1) = sin(x)
```

`sympy.printing.fcode.print_fcode(expr, **settings)`

Prints the Fortran representation of the given expression.

See `fcode` for the meaning of the optional arguments.

`class sympy.printing.fcode.FCodePrinter(settings={})`

A printer to convert `sympy` expressions to strings of Fortran code

`printmethod = '_fcode'`

`indent_code(code)`

Accepts a string of code or a list of code lines

Two basic examples:

```
>>> from sympy import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
'    sqrt(-x**2 + 1),
>>> fcode((3 + 4*I)/(1 - conjugate(x)))
'    (cmplx(3,4))/(-conjg(x) + 1),
```

An example where line wrapping is required:

```
>>> expr = sqrt(1-x**2).series(x,n=20).remove0()
>>> print(fcode(expr))
-715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
@ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
@ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
@ /2.0d0*x**2 + 1
```

In case of line wrapping, it is handy to include the assignment so that lines are wrapped properly when the assignment part is added.

```
>>> print(fcode(expr, assign_to="var"))
    var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
    @ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
    @ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
    @ /2.0d0*x**2 + 1
```

For piecewise functions, the `assign_to` option is mandatory:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), assign_to="var"))
    if (x < 1) then
        var = x
    else
        var = x**2
    end if
```

Note that by default only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran 77. Inline conditionals can be supported using the `merge` function introduced in Fortran 95 by setting of the kwarg `standard=95`:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), standard=95))
    merge(x, x**2, x < 1)
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the `assign_to` option.

```
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
    do i = 1, m
        A(i) = 2*B(i)
    end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```
>>> print(fcode(A[i, i], assign_to=x))
    x = 0
    do i = 1, m
        x = x + A(i, i)
    end do
```

By default, number symbols such as `pi` and `E` are detected and defined as Fortran parameters. The precision of the constants can be tuned with the `precision` argument. Parameter definitions are easily avoided using the `N` function.

```
>>> print(fcode(x - pi**2 - E))
    parameter (E = 2.71828182845905d0)
    parameter (pi = 3.14159265358979d0)
    x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
    parameter (E = 2.718281828459045235360287d0)
    parameter (pi = 3.141592653589793238462643d0)
    x - pi**2 - E
>>> print(fcode(N(x - pi**2, 25)))
    x - 9.869604401089358618834491d0
```

When some functions are not part of the Fortran standard, it might be desirable to introduce the names of user-defined functions in the Fortran expression.

```
>>> print(fcode(1 - gamma(x)**2, user_functions={‘gamma’: ‘mygamma’}))  
-mygamma(x)**2 + 1
```

However, when the `user_functions` argument is not provided, `fcode` attempts to use a reasonable default and adds a comment to inform the user of the issue.

```
>>> print(fcode(1 - gamma(x)**2))  
C      Not supported in Fortran:  
C      gamma  
      -gamma(x)**2 + 1
```

By default the output is human readable code, ready for copy and paste. With the option `human=False`, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing: (i) a set of number symbols that must be defined as ‘Fortran parameters’, (ii) a list functions that can not be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False)  
(set(), set([gamma(x)]), ‘      -gamma(x)**2 + 1’)  
>>> fcode(1 - sin(x)**2, human=False)  
(set(), set(), ‘      -sin(x)**2 + 1’)  
>>> fcode(x - pi**2, human=False)  
(set([(pi, ‘3.14159265358979d0’)]), set(), ‘      x - pi**2’)
```

### 3.15.5 Mathematica code printing

```
sympy.printing.mathematica.known_functions = {‘atan’: [(<function <lambda> at 0x7fb9d1578c0>, ‘ArcTan’),  
class sympy.printing.mathematica.MCodePrinter(settings={})  
    A printer to convert python expressions to strings of the Wolfram’s Mathematica code  
    printmethod = ‘_mcode’  
    doprint(expr)  
        Returns printer’s representation for expr (as a string)  
sympy.printing.mathematica.mathematica_code(expr, **settings)  
    Converts an expr to a string of the Wolfram Mathematica code
```

#### Examples

```
>>> from sympy import mathematica_code as mcode, symbols, sin  
>>> x = symbols(‘x’)  
>>> mcode(sin(x).series(x).remove0())  
‘(1/120)*x^5 - 1/6*x^3 + x’
```

### 3.15.6 Gtk

You can print to a grkmathview widget using the function `print_gtk` located in `sympy.printing.gtk` (it requires to have installed gtkmatmatview and libggtkmathview-bin in some systems).

GtkMathView accepts MathML, so this rendering depends on the MathML representation of the expression.

Usage:

```
from sympy import *
print_gtk(x**2 + 2*exp(x**3))

sympy.printing gtk.print_gtk(x, start_viewer=True)
Print to Gtkmathview, a gtk widget capable of rendering MathML.

Needs libgtkmathview-bin
```

### 3.15.7 LambdaPrinter

This classes implements printing to strings that can be used by the `sympy.utilities.lambdify.lambdify()` (page 1149) function.

```
class sympy.printing.lambdarepr.LambdaPrinter(settings=None)
    This printer converts expressions into strings that can be used by lambdify.

    printmethod = '_sympystr'

sympy.printing.lambdarepr.lambdarepr(expr, **settings)
    Returns a string usable for lambdifying.
```

### 3.15.8 LatexPrinter

This class implements LaTeX printing. See `sympy.printing.latex`.

```
sympy.printing.latex.accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 'sinh', 'cosh',
list() -> new empty list list(iterable) -> new list initialized from iterable's items
```

```
class sympy.printing.latex.LatexPrinter(settings=None)

    printmethod = '_latex'

sympy.printing.latex.latex(expr, **settings)
    Convert the given expression to LaTeX representation.

>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational
>>> from sympy.abc import x, y, mu, r, tau

>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

order: Any of the supported monomial orderings (currently “lex”, “grlex”, or “grevlex”), “old”, and “none”. This parameter does nothing for Mul objects. Setting order to “old” uses the compatibility ordering for Add defined in Printer. For very large expressions, set the ‘order’ keyword to ‘none’ if speed is a concern.

mode: Specifies how the generated code will be delimited. ‘mode’ can be one of ‘plain’, ‘inline’, ‘equation’ or ‘equation\*’. If ‘mode’ is set to ‘plain’, then the resulting code will not be delimited at all (this is the default). If ‘mode’ is set to ‘inline’ then inline LaTeX \$ \$ will be used. If ‘mode’ is set to ‘equation’ or ‘equation\*’, the resulting code will be enclosed in the ‘equation’ or ‘equation\*’ environment (remember to import ‘amsmath’ for ‘equation\*’), unless the ‘itex’ option is set. In the latter case, the \$\$ \$\$ syntax is used.

```
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
```

```
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{\frac{7}{2}}
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
```

itex: Specifies if itex-specific syntax is used, including emitting \$\$ \$\$.

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$8 \sqrt{2} \mu^{\frac{7}{2}}
```

fold\_frac\_powers: Emit " $\left\{ p/q \right\}$ " instead of " $\left\{ \frac{p}{q} \right\}$ " for fractional powers.

```
>>> print(latex((2*tau)**Rational(7,2), fold_frac_powers=True))
8 \sqrt{2} \tau^{7/2}
```

fold\_func\_brackets: Fold function brackets where applicable.

```
>>> print(latex((2*tau)**sin(Rational(7,2))))
\left(2 \tau\right)^{\sin\left(\frac{7}{2}\right)}
>>> print(latex((2*tau)**sin(Rational(7,2)), fold_func_brackets = True))
\left(2 \tau\right)^{\sin\left(\frac{7}{2}\right)}
```

fold\_short\_frac: Emit "p / q" instead of "frac{p}{q}" when the denominator is simple enough (at most two terms and no powers). The default value is *True* for inline mode, False otherwise.

```
>>> print(latex(3*x**2/y))
\frac{3 x^2}{y}
>>> print(latex(3*x**2/y, fold_short_frac=True))
3 x^2 / y
```

long\_frac\_ratio: The allowed ratio of the width of the numerator to the width of the denominator before we start breaking off long fractions. The default value is 2.

```
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=2))
\frac{\int r \, dr}{2 \pi}
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=0))
\frac{\int r \, dr}{2 \pi}
```

mul\_symbol: The symbol to use for multiplication. Can be one of None, "idot", "dot", or "times".

```
>>> print(latex((2*tau)**sin(Rational(7,2)), mul_symbol="times"))
\left(2 \times \tau\right)^{\sin\left(\frac{7}{2}\right)}
```

inv\_trig\_style: How inverse trig functions should be displayed. Can be one of "abbreviated", "full", or "power". Defaults to "abbreviated".

```
>>> print(latex(asin(Rational(7,2))))
\operatorname{asin}\left(\frac{7}{2}\right)
>>> print(latex(asin(Rational(7,2)), inv_trig_style="full"))
\arcsin\left(\frac{7}{2}\right)
>>> print(latex(asin(Rational(7,2)), inv_trig_style="power"))
\sin^{-1}\left(\frac{7}{2}\right)
```

mat\_str: Which matrix environment string to emit. "smallmatrix", "matrix", "array", etc. Defaults to "smallmatrix" for inline mode, "matrix" for matrices of no more than 10 columns, and "array"

otherwise.

```
>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\\y\end{matrix}\right]

>>> print(latex(Matrix(2, 1, [x, y]), mat_str = "array"))
\left[\begin{array}{c}x\\y\end{array}\right]
```

`mat_delim`: The delimiter to wrap around matrices. Can be one of “[”, “(”, or the empty string. Defaults to “[”.

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim="("))
\left(\begin{matrix}x\\y\end{matrix}\right)
```

`symbol_names`: Dictionary of symbols and the custom strings they should be emitted as.

```
>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^{2}
```

`latex` also supports the builtin container types list, tuple, and dictionary.

```
>>> print(latex([2/x, y], mode='inline'))
$ \left[ 2 / x, \quad y \right] $
```

`sympy.printing.latex.print_latex(expr, **settings)`  
Prints LaTeX representation of the given expression.

### 3.15.9 MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml content: <http://www.w3.org/TR/MathML2/chapter4.html>

`class sympy.printing.mathml.MathMLPrinter(settings=None)`

Prints an expression to the MathML markup language

Whenever possible tries to use Content markup and not Presentation markup.

References: <http://www.w3.org/TR/MathML2/>

`printmethod = 'mathml'`

`doprint(expr)`

Prints the expression as MathML.

`mathml_tag(e)`

Returns the MathML tag for an expression.

`sympy.printing.mathml.mathml(expr, **settings)`

Returns the MathML representation of expr

`sympy.printing.mathml.print_mathml(expr, **settings)`

Prints a pretty representation of the MathML code for expr

#### Examples

```
>>> ##
>>> from sympy.printing.mathml import print_mathml
>>> from sympy.abc import x
```

```
>>> print_mathml(x+1)
<apply>
  <plus/>
  <ci>x</ci>
  <cn>1</cn>
</apply>
```

### 3.15.10 PythonPrinter

This class implements Python printing. Usage:

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

>>> print_python(5*x**3 + sin(x))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

### 3.15.11 ReprPrinter

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

```
class sympy.printing.repr.ReprPrinter(settings=None)

printmethod = '_sympyrepr'
emptyPrinter(expr)
    The fallback printer.

reify(args, sep)
    Prints each item in args and joins them with sep.

sympy.printing.repr.srepr(expr, **settings)
    return expr in repr form
```

### 3.15.12 StrPrinter

This module generates readable representations of SymPy expressions.

```
class sympy.printing.str.StrPrinter(settings=None)

printmethod = '_sympystr'
sympy.printing.str.sstrrepr(expr, **settings)
    return expr in mixed str/repr form

i.e. strings are returned in repr form with quotes, and everything else is returned in str form.

This function could be useful for hooking into sys.displayhook
```

### 3.15.13 Tree Printing

The functions in this module create a representation of an expression as a tree.

```
sympy.printing.tree pprint_nodes(subtrees)
Prettyprints systems of nodes.
```

### Examples

```
>>> from sympy.printing.tree import pprint_nodes
>>> print(pprint_nodes(["a", "b1\nb2", "c"]))
+-a
+-b1
| b2
+-c
```

```
sympy.printing.tree print_node(node)
Returns information about the “node”.
```

This includes class name, string representation and assumptions.

```
sympy.printing.tree tree(node)
Returns a tree representation of “node” as a string.
```

It uses print\_node() together with pprint\_nodes() on node.args recursively.

See also: print\_tree()

```
sympy.printing.tree print_tree(node)
Prints a tree representation of “node”.
```

### Examples

```
>>> from sympy.printing import print_tree
>>> from sympy import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)
>>> print_tree(y**x)
Pow: y**x
+-Symbol: y
| algebraic: True
| commutative: True
| complex: True
| even: True
| extended_real: True
| finite: True
| hermitian: True
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False
+-Symbol: x
    algebraic: True
    commutative: True
    complex: True
    even: False
    extended_real: True
```

```
finite: True
hermitian: True
imaginary: False
infinite: False
integer: True
irrational: False
noninteger: False
nonzero: True
odd: True
rational: True
real: True
transcendental: False
zero: False
```

See also: `tree()`

### 3.15.14 Preview

A useful function is `preview`:

```
sympy.printing.preview(expr, output='png', viewer=None, euler=True, packages=(),
                       filename=None, outputbuffer=None, preamble=None, dvioptions=None,
                       outputTexFile=None, **latex_settings)
```

View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.

If the `expr` argument is an expression, it will be exported to LaTeX and then compiled using the available TeX distribution. The first argument, ‘`expr`’, may also be a LaTeX string. The function will then run the appropriate viewer for the given output format or use the user defined one. By default `png` output is generated.

By default pretty Euler fonts are used for typesetting (they were used to typeset the well known “Concrete Mathematics” book). For that to work, you need the ‘`eulervm.sty`’ LaTeX style (in Debian/Ubuntu, install the `texlive-fonts-extra` package). If you prefer default AMS fonts or your system lacks ‘`eulervm`’ LaTeX package then unset the ‘`euler`’ keyword argument.

To use viewer auto-detection, lets say for ‘`png`’ output, issue

```
>>> from sympy import symbols, preview, Symbol
>>> x, y = symbols("x,y")

>>> preview(x + y, output='png')
```

This will choose ‘`pyglet`’ by default. To select a different one, do

```
>>> preview(x + y, output='png', viewer='gimp')
```

The ‘`png`’ format is considered special. For all other formats the rules are slightly different. As an example we will take ‘`dvi`’ output format. If you would run

```
>>> preview(x + y, output='dvi')
```

then ‘`view`’ will look for available ‘`dvi`’ viewers on your system (predefined in the function, so it will try `evince`, first, then `kdv` and `xdv`). If nothing is found you will need to set the viewer explicitly.

```
>>> preview(x + y, output='dvi', viewer='superior-dvi-viewer')
```

This will skip auto-detection and will run user specified ‘`superior-dvi-viewer`’. If ‘`view`’ fails to find it on your system it will gracefully raise an exception.

You may also enter ‘file’ for the viewer argument. Doing so will cause this function to return a file object in read-only mode, if ‘filename’ is unset. However, if it was set, then ‘preview’ writes the generated file to this filename instead.

There is also support for writing to a BytesIO like object, which needs to be passed to the ‘outputbuffer’ argument.

```
>>> from io import BytesIO
>>> obj = BytesIO()
>>> preview(x + y, output='png', viewer='BytesIO',
...           outputbuffer=obj)
```

The LaTeX preamble can be customized by setting the ‘preamble’ keyword argument. This can be used, e.g., to set a different font size, use a custom documentclass or import certain set of LaTeX packages.

```
>>> preamble = "\\\documentclass[10pt]{article}\n" \
...           "\\\usepackage{amsmath,amsfonts}\\begin{document}"
>>> preview(x + y, output='png', preamble=preamble)
```

If the value of ‘output’ is different from ‘dvi’ then command line options can be set (‘dvioptions’ argument) for the execution of the ‘dvi’+output conversion tool. These options have to be in the form of a list of strings (see subprocess.Popen).

Additional keyword args will be passed to the latex call, e.g., the symbol\_names flag.

```
>>> phidd = Symbol('phidd')
>>> preview(phidd, symbol_names={phidd:r'\ddot{\varphi}'})
```

For post-processing the generated TeX File can be written to a file by passing the desired filename to the ‘outputTexFile’ keyword argument. To write the TeX code to a file named “sample.tex” and run the default png viewer to display the resulting bitmap, do

```
>>> preview(x + y, outputTexFile="sample.tex")
```

### 3.15.15 Implementation - Helper Classes/Functions

`sympy.printing.conventions.split_super_sub(text)`  
Split a symbol name into a name, superscripts and subscripts

The first part of the symbol name is considered to be its actual ‘name’, followed by super- and subscripts. Each superscript is preceded with a “^” character or by “\_”. Each subscript is preceded by a “\_” character. The three return values are the actual name, a list with superscripts and a list with subscripts.

```
>>> from sympy.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

#### CodePrinter

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

```
class sympy.printing.codeprinter.CodePrinter(settings=None)
    The base class for code-printing subclasses.

    printmethod = '_sympystr'

exception sympy.printing.codeprinter.AssignmentError
    Raised if an assignment variable for a loop is missing.
```

## Precedence

```
sympy.printing.precedence.PRECEDENCE = {'Xor': 10, 'Add': 40, 'Relational': 35, 'Mul': 50, 'Or': 20, 'Lambda': 1}
    Default precedence values for some basic types.

sympy.printing.precedence.PRECEDENCE_VALUES = {'And': 30, 'Xor': 10, 'Sub': 40, 'factorial': 60, 'Pow': 60, 'Erf': 50}
    A dictionary assigning precedence values to certain classes. These values are treated like they were
    inherited, so not every single class has to be named here.

sympy.printing.precedence.PRECEDENCE_FUNCTIONS = {'FracElement': <function precedence_FracElement at 0x...>}
    Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be
    inserted in this dictionary that takes an instance of this class as argument and returns the appropriate
    precedence value.

sympy.printing.precedence.precedence(item)
    Returns the precedence of a given object.
```

### 3.15.16 Pretty-Printing Implementation Helpers

```
sympy.printing.pretty.pretty_symbolology.U(name)
    unicode character by name or None if not found

sympy.printing.pretty.pretty_symbolology.pretty_use_unicode(flag=None)
    Set whether pretty-printer should use unicode by default

sympy.printing.pretty.pretty_symbolology.pretty_try_use_unicode()
    See if unicode output is available and leverage it if possible

sympy.printing.pretty.pretty_symbolology.xstr(*args)
    call str or unicode depending on current mode
```

The following two functions return the Unicode version of the inputted Greek letter.

```
sympy.printing.pretty.pretty_symbolology.g(l)
```

```
sympy.printing.pretty.pretty_symbolology.G(l)
```

```
sympy.printing.pretty.pretty_symbolology.greek_letters = ['alpha', 'beta', 'gamma', 'delta', 'epsilon', 'zeta', 'epsilon_bar', 'pi', 'rho', 'tau', 'phi', 'psi', 'omega', 'theta', 'nu', 'xi', 'mu', 'sigma', 'tau_bar', 'phi_bar', 'psi_bar', 'omega_bar', 'theta_bar', 'nu_bar', 'xi_bar', 'mu_bar', 'sigma_bar']
list() -> new empty list list(iterable) -> new list initialized from iterable's items
```

```
sympy.printing.pretty.pretty_symbolology.digit_2txt = {'1': 'ONE', '0': 'ZERO', '3': 'THREE', '2': 'TWO', '4': 'FOUR', '5': 'FIVE', '6': 'SIX', '7': 'SEVEN', '8': 'EIGHT', '9': 'NINE'}
```

```
sympy.printing.pretty.pretty_symbolology.symb_2txt = {'int': 'INTEGRAL', '{}': 'CURLY BRACKET', '[': 'LEFT CURLY BRACKET', ']': 'RIGHT CURLY BRACKET'}
```

The following functions return the Unicode subscript/superscript version of the character.

```
sympy.printing.pretty.pretty_symbolology.sub = {')': u'\u208e', 'chi': u'\u1d6a', '+': u'\u208a', '-': u'\u208b'}
```

```
sympy.printing.pretty.pretty_symbolology.sup = {')': u'\u207e', 'i': u'\u2071', '(': u'\u207d', '+': u'\u207a', '-'': u'\u207b'}
```

The following functions return Unicode vertical objects.

```
sympy.printing.pretty.pretty_symbology.xobj(symb, length)
Construct spatial object of given length.
```

return: [] of equal-length strings

```
sympy.printing.pretty.pretty_symbology.vobj(symb, height)
Construct vertical object of a given height
```

see: xobj

```
sympy.printing.pretty.pretty_symbology.hobj(symb, width)
Construct horizontal object of a given width
```

see: xobj

The following constants are for rendering roots and fractions.

```
sympy.printing.pretty.pretty_symbology.root = {2: u'\u221a', 3: u'\u221b', 4: u'\u221c'}
```

```
sympy.printing.pretty.pretty_symbology.VF(txt)
```

```
sympy.printing.pretty.pretty_symbology.frac = {(1, 3): u'\u2153', (5, 6): u'\u215a', (1, 4): u'\xbc', (2, 3):
```

The following constants/functions are for rendering atoms and symbols.

```
sympy.printing.pretty.pretty_symbology.xsym(sym)
get symbology for a 'character'
```

```
sympy.printing.pretty.pretty_symbology.atoms_table = {'Integers': u'\u2124', 'NegativeInfinity': u'-\u221e',
```

```
sympy.printing.pretty.pretty_symbology.pretty_atom(atom_name, default=None)
return pretty representation of an atom
```

```
sympy.printing.pretty.pretty_symbology.pretty_symbol(symb_name)
return pretty representation of a symbol
```

```
sympy.printing.pretty.pretty_symbology.annotated(letter)
```

Return a stylised drawing of the letter *letter*, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See pretty.py functions \_print\_meijerg, \_print\_hyper on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at pietjepuk314 at the reverse of ku.oc.oohay). All objects have a method that create a "stringPict", that can be used in the str method for pretty printing.

### Updates by Jason Gedge (email <my last name> at cs mun ca)

- terminal\_string() method
- minor fixes and changes (mostly to prettyForm)

### TODO:

- Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

```
class sympy.printing.pretty.stringpict.stringPict(s, baseline=0)
```

An ASCII picture. The pictures are represented as a list of equal length strings.

```
above(*args)
```

Put pictures above this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of bottom picture.

```
below(*args)
```

Put pictures under this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of top picture

### Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("x+3").below(
...     stringPict.LINE, '3')[0])
x+3
---
3

height()
The height of the picture in characters.

left(*args)
Put pictures (left to right) at left. Returns string, baseline arguments for stringPict.

leftslash()
Precede object by a slash of the proper size.

static next(*args)
Put a string of stringPicts next to each other. Returns string, baseline arguments for stringPict.

parens(left='(', right=')', ifascii_nougly=False)
Put parentheses around self. Returns string, baseline arguments for stringPict.

left or right can be None or empty string which means 'no paren from that side'

render(*args, **kwargs)
Return the string form of self.

Unless the argument line_break is set to False, it will break the expression in a form that can be
printed on the terminal without being broken up.

right(*args)
Put pictures next to this one. Returns string, baseline arguments for stringPict. (Multiline)
strings are allowed, and are given a baseline of 0.
```

### Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("10").right(" + ",stringPict("1\r-\r2",1))[0])
1
10 + -
2

root(n=None)
Produce a nice root symbol. Produces ugly results for big n inserts.

static stack(*args)
Put pictures on top of each other, from top to bottom. Returns string, baseline arguments for
stringPict. The baseline is the baseline of the second picture. Everything is centered. Baseline
is the baseline of the second picture. Strings are allowed. The special value stringPict.LINE is a
row of '-' extended to the width.

terminal_width()
Return the terminal width if possible, otherwise return 0.

width()
The width of the picture in characters.
```

---

```
class sympy.printing.pretty.stringpict.prettyForm(s, baseline=0, binding=0, unicode=None)
Extension of the stringPict class that knows about basic math applications, optimizing double minus signs.
```

“Binding” is interpreted as follows:

```
ATOM this is an atom: never needs to be parenthesized
FUNC this is a function application: parenthesize if added (?)
DIV this is a division: make wider division if divided
POW this is a power: only parenthesize if exponent
MUL this is a multiplication: parenthesize if powered
ADD this is an addition: parenthesize if multiplied or powered
NEG this is a negative number: optimize if added, parenthesize if
multiplied or powered
OPEN this is an open object: parenthesize if added, multiplied, or
powered (example: Piecewise)
```

```
static apply(function, *args)
Functions of one or more variables.
```

### 3.15.17 dotprint

```
sympy.printing.dot.dotprint(expr, styles=[(<class 'sympy.core.basic.Basic'>, {'color': 'blue',
                                             'shape': 'ellipse'}), (<class 'sympy.core.expr.Expr'>, {'color':
                                             'black'})], atom=<function <lambda> at 0x7fb8ae7c848>,
maxdepth=None, repeat=True, labelfunc=<type 'str'>, **kwargs)
```

DOT description of a SymPy expression tree

Options are

**styles:** Styles for different classes. The default is:

```
[(Basic, {'color': 'blue', 'shape': 'ellipse'}),
(Expr, {'color': 'black'})]]'
```

**atom:** Function used to determine if an arg is an atom. The default is `lambda x: not isinstance(x, Basic)`. Another good choice is `lambda x: not x.args`.

**maxdepth:** The maximum depth. The default is None, meaning no limit.

**repeat:** Whether to different nodes for separate common subexpressions. The default is True. For example, for `x + x*y` with `repeat=True`, it will have two nodes for `x` and with `repeat=False`, it will have one (warning: even if it appears twice in the same object, like `Pow(x, x)`, it will still only appear once. Hence, with `repeat=False`, the number of arrows out of an object might not equal the number of args it has).

**labelfunc:** How to label leaf nodes. The default is `str`. Another good option is `srepr`. For example with `str`, the leaf nodes of `x + 1` are labeled, `x` and `1`. With `srepr`, they are labeled `Symbol('x')` and `Integer(1)`.

Additional keyword arguments are included as styles for the graph.

#### Examples

```
>>> 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 #
#####

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

## 3.16 Plotting Module

### 3.16.1 Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using `matplotlib` as a backend.

The plotting module has the following functions:

- `plot`: Plots 2D line plots.
- `plot_parametric`: Plots 2D parametric plots.
- `plot_implicit`: Plots 2D implicit and region plots.
- `plot3d`: Plots 3D plots of functions in two variables.
- `plot3d_parametric_line`: Plots 3D line plots, defined by a parameter.
- `plot3d_parametric_surface`: Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding `Series` class to `Plot` as argument.

### 3.16.2 Plot Class

```
class sympy.plotting.plot.Plot(*args, **kwargs)
```

The central class of the plotting module.

For interactive work the function `plot` is better suited.

This class permits the plotting of sympy expressions using numerous backends (`matplotlib`, Google charts api, etc).

The figure can contain an arbitrary number of plots of sympy expressions, lists of coordinates of points, etc. `Plot` has a private attribute `_series` that contains all data series to be plotted (expressions for lines

or surfaces, lists of points, etc (all subclasses of `BaseSeries`)). Those data series are instances of classes not imported by `from sympy import *`.

The customization of the figure is on two levels. Global options that concern the figure as a whole (eg title, xlabel, scale, etc) and per-data series options (eg name) and aesthetics (eg. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are: - None (the backend uses default values) - a constant - a function of one variable (the first coordinate or parameter) - a function of two variables (the first and second coordinate or parameters) - a function of three variables (only in nonparametric 3D plots) Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.

The arguments for the constructor `Plot` must be subclasses of `BaseSeries`.

Any global option can be specified as a keyword argument.

The global options for a figure are:

- `title` : str
- `xlabel` : str
- `ylabel` : str
- `legend` : bool
- `xscale` : {‘linear’, ‘log’}
- `yscale` : {‘linear’, ‘log’}
- `axis` : bool
- `axis_center` : tuple of two floats or {‘center’, ‘auto’}
- `xlim` : tuple of two floats
- `ylim` : tuple of two floats
- `aspect_ratio` : tuple of two floats or {‘auto’}
- `autoscale` : bool
- `margin` : float in [0, 1]

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

`ListSeries`, `LineOver1DRangeSeries`, `Parametric2DLineSeries`, `Parametric3DLineSeries` support the following:

Aesthetics:

- `line_color` : function which returns a float.

options:

- label : str
- steps : bool
- integers\_only : bool

SurfaceOver2DRangeSeries, ParametricSurfaceSeries support the following:

aesthetics:

- surface\_color : function which returns a float.

`append(arg)`

Adds an element from a plot's series to an existing plot.

**See also:**

`extend` (page 874)

### Examples

Consider two Plot objects, p1 and p2. To add the second plot's first series object to the first, use the `append` method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.append(p2[0])
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

`extend(arg)`

Adds all series from another plot.

### Examples

Consider two Plot objects, p1 and p2. To add the second plot to the first, use the `extend` method, like so:

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.extend(p2)
>>> p1
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

## 3.16.3 Plotting Function Reference

`sympy.plotting.plot.plot(*args, **kwargs)`

Plots a function of a single variable and returns an instance of the Plot class (also, see the description

of the `show` keyword argument below).

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

**See also:**

[Plot](#) (page 872), `sympy.plotting.plot.LineOver1DRangeSeries` (page 878)

### Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot
>>> x = symbols('x')
```

Single Plot

```
>>> plot(x**2, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x**2 for x over (-5.0, 5.0)
```

Multiple plots with single range.

```
>>> plot(x, x**2, x**3, (x, -5, 5))
Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

```
>>> plot(x**2, adaptive=False, nb_of_points=400)
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

`sympy.plotting.plot.plot_parametric(*args, **kwargs)`

Plots a 2D parametric plot.

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

**See also:**

[Plot](#) (page 872), `Parametric2DLineSeries` (page 879)

### Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot_parametric
>>> u = symbols('u')
```

Single Parametric plot

```
>>> plot_parametric(cos(u), sin(u), (u, -5, 5))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

Multiple parametric plot with single range.

```
>>> plot_parametric((cos(u), sin(u)), (u, cos(u)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

Multiple parametric plots.

```
>>> plot_parametric((cos(u), sin(u), (u, -5, 5)),
...                   (cos(u), u, (u, -5, 5)))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```

`sympy.plotting.plot.plot3d(*args, **kwargs)`

Plots a 3D surface plot.

See also:

[Plot](#) (page 872), [SurfaceOver2DRangeSeries](#) (page 879)

## Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```
>>> plot3d(x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```
>>> plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)),
...           (x*y, (x, -3, 3), (y, -3, 3)))
Plot object containing:
[0]: cartesian surface: x**2 + y**2 for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: x*y for x over (-3.0, 3.0) and y over (-3.0, 3.0)
```

`sympy.plotting.plot.plot3d_parametric_line(*args, **kwargs)`

Plots a 3D parametric line plot.

See also:

[Plot](#) (page 872), [Parametric3DLineSeries](#) (page 879)

### Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

Single plot.

```
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
```

Multiple plots.

```
>>> plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)),
...     (sin(u), u**2, u, (u, -5, 5)))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.0)
```

`sympy.plotting.plot.plot3d_parametric_surface(*args, **kwargs)`

Plots a 3D parametric surface plot.

See also:

[Plot](#) (page 872), [ParametricSurfaceSeries](#) (page 879)

### Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```
>>> plot3d_parametric_surface(cos(u + v), sin(u - v), u - v,
...     (u, -5, 5), (v, -5, 5))
Plot object containing:
[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u over (-5.0, 5.0) and v over (-5.0, 5.0)
```

`sympy.plotting.plot_implicit.plot_implicit(expr, x_var=None, y_var=None, **kwargs)`

A plot function to plot implicit equations / inequalities.

### Examples

Plot expressions:

```
>>> from sympy import plot_implicit, cos, sin, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression

```
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols

```
>>> p2 = plot_implicit(Eq(x**2 + y**2, 3),
...                      (x, -3, 3), (y, -3, 3))
```

With depth of recursion as argument.

```
>>> p3 = plot_implicit(Eq(x**2 + y**2, 5),
...                      (x, -4, 4), (y, -4, 4), depth = 2)
```

Using mesh grid and not using adaptive meshing.

```
>>> p4 = plot_implicit(Eq(x**2 + y**2, 5),
...                      (x, -5, 5), (y, -2, 2), adaptive=False)
```

Using mesh grid with number of points as input.

```
>>> p5 = plot_implicit(Eq(x**2 + y**2, 5),
...                      (x, -5, 5), (y, -2, 2),
...                      adaptive=False, points=400)
```

Plotting regions.

```
>>> p6 = plot_implicit(y > x**2)
```

Plotting Using boolean conjunctions.

```
>>> p7 = plot_implicit(And(y > x, y > -x))
```

When plotting an expression with a single variable ( $y - 1$ , for example), specify the  $x$  or the  $y$  variable explicitly:

```
>>> p8 = plot_implicit(y - 1, y_var=y)
>>> p9 = plot_implicit(x - 1, x_var=x)
```

### 3.16.4 Series Classes

```
class sympy.plotting.plot.BaseSeries
```

Base class for the data objects containing stuff to be plotted.

The backend should check if it supports the data series that it's given. (eg TextBackend supports only LineOver1DRange). It's the backend responsibility to know how to use the class of data series that it's given.

Some data series classes are grouped (using a class attribute like `is_2Dline`) according to the api they present (based only on convention). The backend is not obliged to use that api (eg. The LineOver1DRange belongs to the `is_2Dline` group and presents the `get_points` method, but the TextBackend does not use the `get_points` method).

```
class sympy.plotting.plot.Line2DBaseSeries
```

A base class for 2D lines.

- adding the label, steps and only\_integers options
- making `is_2Dline` true
- defining `get_segments` and `get_color_array`

```
class sympy.plotting.plot.LineOver1DRangeSeries(expr, var_start_end, **kwargs)
```

Representation for a line consisting of a SymPy expression over a range.

**get\_segments()**

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

**References**

- [1] Adaptive polygonal approximation of parametric curves, Luiz Henrique de Figueiredo.

class `sympy.plotting.plot.Parametric2DLineSeries(expr_x, expr_y, var_start_end, **kwargs)`

Representation for a line consisting of two parametric sympy expressions over a range.

**get\_segments()**

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

**References**

- [1] Adaptive polygonal approximation of parametric curves, Luiz Henrique de Figueiredo.

class `sympy.plotting.plot.Line3DBaseSeries`

A base class for 3D lines.

Most of the stuff is derived from Line2DBaseSeries.

class `sympy.plotting.plot.Parametric3DLineSeries(expr_x, expr_y, expr_z, var_start_end, **kwargs)`

Representation for a 3D line consisting of two parametric sympy expressions and a range.

class `sympy.plotting.plot.SurfaceBaseSeries`

A base class for 3D surfaces.

class `sympy.plotting.plot.SurfaceOver2DRangeSeries(expr, var_start_end_x, var_start_end_y, **kwargs)`

Representation for a 3D surface consisting of a sympy expression and 2D range.

class `sympy.plotting.plot.ParametricSurfaceSeries(expr_x, expr_y, expr_z, var_start_end_u, var_start_end_v, **kwargs)`

Representation for a 3D surface consisting of three parametric sympy expressions and a range.

class `sympy.plotting.plot_implicit.ImplicitSeries(expr, var_start_end_x, var_start_end_y, has_equality, use_interval_math, depth, nb_of_points, line_color)`

Representation for Implicit plot

## 3.17 Assumptions module

### 3.17.1 Contents

#### Ask

Module for querying SymPy objects about assumptions.

```
class sympy.assumptions.ask.Q
    Supported ask keys.
```

```
sympy.assumptions.ask.ask(proposition, assumptions=True, context=AssumptionsContext([]))
    Method for inferring properties about objects.
```

#### Syntax

- ask(proposition)
- ask(proposition, assumptions)  
where proposition is any boolean expression

#### Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
>>> ask(Q.prime(x*y), Q.integer(x) & Q.integer(y))
False
```

**Remarks** Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), Q.is_true(x > 0))
```

It is however a work in progress.

```
sympy.assumptions.ask.ask_full_inference(proposition, assumptions, known_facts_cnf)
    Method for inferring properties about objects.
```

```
sympy.assumptions.ask.compute_known_facts(known_facts, known_facts_keys)
    Compute the various forms of knowledge compilation used by the assumptions system.
```

This function is typically applied to the variables `known_facts` and `known_facts_keys` defined at the bottom of this file.

```
sympy.assumptions.ask.register_handler(key, handler)
```

Register a handler in the ask system. key must be a string and handler a class inheriting from AskHandler:

```
>>> from sympy.assumptions import register_handler, ask, Q
>>> from sympy.assumptions.handlers import AskHandler
>>> class MersenneHandler(AskHandler):
...     # Mersenne numbers are in the form 2**n + 1, n integer
...     @staticmethod
```

```
...     def Integer(expr, assumptions):
...         import math
...         return ask(Q.integer(math.log(expr + 1, 2)))
>>> register_handler('mersenne', MersenneHandler)
>>> ask(Q.mersenne(7))
True

sympy.assumptions.ask.remove_handler(key, handler)
Removes a handler from the ask system. Same syntax as register_handler
```

## Assume

```
class sympy.assumptions.ask.AppliedPredicate
The class of expressions resulting from applying a Predicate.
```

### Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> Q.integer(x)
Q.integer(x)
>>> type(Q.integer(x))
<class 'sympy.assumptions.ask.AppliedPredicate'>
```

**arg**  
Return the expression used by this assumption.

### Examples

```
>>> from sympy import Q, Symbol
>>> x = Symbol('x')
>>> a = Q.integer(x + 1)
>>> a.arg
x + 1
```

```
class sympy.assumptions.ask.AssumptionsContext
Set representing assumptions.
```

This is used to represent global assumptions, but you can also use this class to create your own local assumptions contexts. It is basically a thin wrapper to Python's set, so see its documentation for advanced usage.

### Examples

```
>>> from sympy import AppliedPredicate, Q
>>> from sympy.assumptions.ask import global_assumptions
>>> global_assumptions
AssumptionsContext()
>>> from sympy.abc import x
>>> global_assumptions.add(Q.real(x))
>>> global_assumptions
AssumptionsContext([Q.real(x)])
>>> global_assumptions.remove(Q.real(x))
```

```
>>> global_assumptions
AssumptionsContext()
>>> global_assumptions.clear()
```

```
add(*assumptions)
    Add an assumption.
```

```
class sympy.assumptions.Assume
```

A predicate is a function that returns a boolean value.

Predicates merely wrap their argument and remain unevaluated:

```
>>> from sympy import Q, ask, Symbol, S
>>> x = Symbol('x')
>>> Q.prime(7)
Q.prime(7)
```

To obtain the truth value of an expression containing predicates, use the function *ask*:

```
>>> ask(Q.prime(7))
True
```

The tautological predicate *Q.isTrue* can be used to wrap other objects:

```
>>> Q.is_true(x > 1)
Q.is_true(x > 1)
>>> Q.is_true(S(1) < x)
Q.is_true(1 < x)

eval(expr, assumptions=True)
    Evaluate self(expr) under the given assumptions.
```

This uses only direct resolution methods, not logical inference.

```
sympy.assumptions.assume.assuming(*args, **kwds)
    Context manager for assumptions
```

## Examples

```
>>> from sympy.assumptions import assuming, Q, ask
>>> from sympy.abc import x, y

>>> print(ask(Q.integer(x + y)))
None

>>> with assuming(Q.integer(x), Q.integer(y)):
...     print(ask(Q.integer(x + y)))
True
```

## Refine

```
sympy.assumptions.refine.refine(expr, assumptions=True)
    Simplify an expression using assumptions.
```

Gives the form of expr that would be obtained if symbols in it were replaced by explicit numerical expressions satisfying the assumptions.

## Examples

```
>>> from sympy import refine, sqrt, Q
>>> from sympy.abc import x
>>> refine(sqrt(x**2), Q.real(x))
Abs(x)
>>> refine(sqrt(x**2), Q.positive(x))
x
```

`sympy.assumptions.refine.refine_Pow(expr, assumptions)`

Handler for instances of Pow.

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.refine import refine_Pow
>>> from sympy.abc import x,y,z
>>> refine_Pow((-1)**x, Q.real(x))
>>> refine_Pow((-1)**x, Q.even(x))
1
>>> refine_Pow((-1)**x, Q.odd(x))
-1
```

For powers of -1, even parts of the exponent can be simplified:

```
>>> refine_Pow((-1)**(x+y), Q.even(x))
(-1)**y
>>> refine_Pow((-1)**(x+y+z), Q.odd(x) & Q.odd(z))
(-1)**y
>>> refine_Pow((-1)**(x+y+2), Q.odd(x))
(-1)**(y + 1)
>>> refine_Pow((-1)**(x+3), True)
(-1)**(x + 1)
```

`sympy.assumptions.refine.refine_Relational(expr, assumptions)`

Handler for Relational

```
>>> from sympy.assumptions.refine import refine_Relational
>>> from sympy.assumptions.ask import Q
>>> from sympy.abc import x
>>> refine_Relational(x<0, ~Q.is_true(x<0))
False
```

`sympy.assumptions.refine.refine_abs(expr, assumptions)`

Handler for the absolute value.

## Examples

```
>>> from sympy import Symbol, Q, refine, Abs
>>> from sympy.assumptions.refine import refine_abs
>>> from sympy.abc import x
>>> refine_abs(Abs(x), Q.real(x))
>>> refine_abs(Abs(x), Q.positive(x))
x
>>> refine_abs(Abs(x), Q.negative(x))
-x
```

`sympy.assumptions.refine.refine_atan2(expr, assumptions)`

Handler for the atan2 function

## Examples

```
>>> from sympy import Symbol, Q, refine, atan2
>>> from sympy.assumptions.refine import refine_atan2
>>> from sympy.abc import x, y
>>> refine_atan2(atan2(y,x), Q.real(y) & Q.positive(x))
atan(y/x)
>>> refine_atan2(atan2(y,x), Q.negative(y) & Q.negative(x))
atan(y/x) - pi
>>> refine_atan2(atan2(y,x), Q.positive(y) & Q.negative(x))
atan(y/x) + pi
```

`sympy.assumptions.refine.refine_exp(expr, assumptions)`

Handler for exponential function.

```
>>> from sympy import Symbol, Q, exp, I, pi
>>> from sympy.assumptions.refine import refine_exp
>>> from sympy.abc import x
>>> refine_exp(exp(pi*I*2*x), Q.real(x))
>>> refine_exp(exp(pi*I*2*x), Q.integer(x))
1
```

## Handlers

### Contents

**Calculus** This module contains query handlers responsible for calculus queries: infinitesimal, finite, etc.

`class sympy.assumptions.handlers.calculus.AskFiniteHandler`  
Handler for key ‘finite’.

Test that an expression is finite respect to all its variables.

## Examples

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.handlers.calculus import AskFiniteHandler
>>> from sympy.abc import x
>>> a = AskFiniteHandler()
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.finite(x))
True
```

`static Add(expr, assumptions)`

Return True if expr is finite, False if not and None if unknown.

Truth Table:

	B	U	?	‘+’	‘-’	‘x’
B	B	U	?	‘+’	‘-’	‘x’
U	‘+’	U	?	U	?	?
‘-’	?	U	?	?	U	?
‘x’	?	?	?	?	?	?
?	?	?	?	?	?	?

- ‘B’ = Bounded
- ‘U’ = Unbounded
- ‘?’ = unknown boundedness
- ‘+’ = positive sign
- ‘-’ = negative sign
- ‘x’ = sign unknown

- All Bounded -> True
- 1 Unbounded and the rest Bounded -> False
- >1 Unbounded, all with same known sign -> False
- Any Unknown and unknown sign -> None
- Else -> None

When the signs are not the same you can have an undefined result as in oo - oo, hence ‘finite’ is also undefined.

#### **static Mul(expr, assumptions)**

Return True if expr is finite, False if not and None if unknown.

Truth Table:

	B	U	?
B	B	U	s   /s
U	U	U	?
?			?

- B = Bounded
- U = Unbounded
- ? = unknown boundedness
- s = signed (hence nonzero)
- /s = not signed

#### **static Pow(expr, assumptions)**

Unbounded \*\* NonZero -> Unbounded Bounded \*\* Bounded -> Bounded Abs()<=1 \*\* Positive -> Bounded Abs()>=1 \*\* Negative -> Bounded Otherwise unknown

#### **static Symbol(expr, assumptions)**

Handles Symbol.

#### **Examples**

```
>>> from sympy import Symbol, Q
>>> from sympy.assumptions.handlers.calculus import AskFiniteHandler
>>> from sympy.abc import x
>>> a = AskFiniteHandler()
>>> a.Symbol(x, Q.positive(x)) == None
True
>>> a.Symbol(x, Q.finite(x))
True

class sympy.assumptions.handlers.calculus.AskInfinitesimalHandler
    Handler for key ‘infinitesimal’ Test that a given expression is equivalent to an infinitesimal number

    static Add(expr, assumptions)
        Infinitesimal*Bounded -> Infinitesimal

    static Mul(expr, assumptions)
        Infinitesimal*Bounded -> Infinitesimal

    static Pow(expr, assumptions)
        Infinitesimal*Bounded -> Infinitesimal
```

**nTheory** Handlers for keys related to number theory: prime, even, odd, etc.

```
class sympy.assumptions.handlers.nttheory.AskOddHandler
    Handler for key ‘odd’ Test that an expression represents an odd number

class sympy.assumptions.handlers.nttheory.AskPrimeHandler
    Handler for key ‘prime’ Test that an expression represents a prime number. When the expression is a number the result, when True, is subject to the limitations of isprime() which is used to return the result.

    static Pow(expr, assumptions)
        Integer**Integer -> !Prime
```

**Order** AskHandlers related to order relations: positive, negative, etc.

```
class sympy.assumptions.handlers.order.AskNegativeHandler
    This is called by ask() when key='negative'

    Test that an expression is less (strict) than zero.
```

## Examples

```
>>> from sympy import ask, Q, pi
>>> ask(Q.negative(pi+1)) # this calls AskNegativeHandler.Add
False
>>> ask(Q.negative(pi**2)) # this calls AskNegativeHandler.Pow
False

    static Add(expr, assumptions)
        Positive + Positive -> Positive, Negative + Negative -> Negative

    static Pow(expr, assumptions)
        Real ** Even -> NonNegative Real ** Odd -> same_as_base NonNegative ** Positive -> Non-Negative

class sympy.assumptions.handlers.order.AskNonZeroHandler
    Handler for key ‘zero’ Test that an expression is not identically zero
```

```

class sympy.assumptions.handlers.order.AskPositiveHandler
    Handler for key ‘positive’ Test that an expression is greater (strict) than zero

Sets Handlers for predicates related to set membership: integer, rational, etc.

class sympy.assumptions.handlers.sets.AskAlgebraicHandler
    Handler for Q.algebraic key.

class sympy.assumptions.handlers.sets.AskAntiHermitianHandler
    Handler for Q.antihermitian Test that an expression belongs to the field of anti-Hermitian operators,
    that is, operators in the form  $x^*I$ , where  $x$  is Hermitian

static Add(expr, assumptions)
    Antihermitian + Antihermitian -> Antihermitian Antihermitian + !Antihermitian -> !Antiher-
    mitian

static Mul(expr, assumptions)
    As long as there is at most only one noncommutative term: Hermitian*Hermitian -> !Antihermi-
    tian Hermitian*Antihermitian -> Antihermitian Antihermitian*Antihermitian -> !Antihermitian

static Pow(expr, assumptions)
    Hermitian**Integer -> !Antihermitian Antihermitian**Even -> !Antihermitian Antihermi-
    tian**Odd -> Antihermitian

class sympy.assumptions.handlers.sets.AskComplexHandler
    Handler for Q.complex Test that an expression belongs to the field of complex numbers

class sympy.assumptions.handlers.sets.AskExtendedRealHandler
    Handler for Q.extended_real Test that an expression belongs to the field of extended real numbers, that
    is real numbers union {Infinity, -Infinity}

class sympy.assumptions.handlers.sets.AskHermitianHandler
    Handler for Q.hermitian Test that an expression belongs to the field of Hermitian operators

static Add(expr, assumptions)
    Hermitian + Hermitian -> Hermitian Hermitian + !Hermitian -> !Hermitian

static Mul(expr, assumptions)
    As long as there is at most only one noncommutative term: Hermitian*Hermitian -> Hermitian
    Hermitian*Antihermitian -> !Hermitian Antihermitian*Antihermitian -> Hermitian

static Pow(expr, assumptions)
    Hermitian**Integer -> Hermitian

class sympy.assumptions.handlers.sets.AskImaginaryHandler
    Handler for Q.imaginary Test that an expression belongs to the field of imaginary numbers, that is,
    numbers in the form  $x^*I$ , where  $x$  is real (or zero)

static Add(expr, assumptions)
    Imaginary + Imaginary -> Imaginary Imaginary + Complex -> ? Imaginary + Nonzero Real ->
    !Imaginary

static Mul(expr, assumptions)
    Real*Imaginary -> Imaginary Imaginary*Imaginary -> Real

static Pow(expr, assumptions)
    Imaginary**Odd -> Imaginary Imaginary**Even -> Real b**Imaginary -> !Imaginary if ex-
    ponent is an integer multiple of  $I*pi/\log(b)$  Imaginary**Real -> ? Nonnegative**Real -> Real
    Negative**Integer -> Real Negative** $(\text{Integer}/2)$  -> Imaginary Negative**Real -> not Imaginary
    if exponent is not Rational

```

```
class sympy.assumptions.handlers.sets.AskIntegerHandler
    Handler for Q.integer Test that an expression belongs to the field of integer numbers

    static Add(expr, assumptions)
        Integer + Integer -> Integer Integer + !Integer -> !Integer !Integer + !Integer -> ?

    static Mul(expr, assumptions)
        Integer*Integer -> Integer Integer*Irrational -> !Integer Odd/Even -> !Integer Integer*Rational
        -> ?

    static Pow(expr, assumptions)
        Integer + Integer -> Integer Integer + !Integer -> !Integer !Integer + !Integer -> ?

class sympy.assumptions.handlers.sets.AskRationalHandler
    Handler for Q.rational Test that an expression belongs to the field of rational numbers

    static Add(expr, assumptions)
        Rational + Rational -> Rational Rational + !Rational -> !Rational !Rational + !Rational -> ?

    static Mul(expr, assumptions)
        Rational + Rational -> Rational Rational + !Rational -> !Rational !Rational + !Rational -> ?

    static Pow(expr, assumptions)
        Rational ** Integer -> Rational Irrational ** Rational -> Irrational Rational ** Irrational -> ?

class sympy.assumptions.handlers.sets.AskRealHandler
    Handler for Q.real Test that an expression belongs to the field of real numbers

    static Add(expr, assumptions)
        Real + Real -> Real Real + (Complex & !Real) -> !Real

    static Mul(expr, assumptions)
        Real*Real -> Real Real*Imaginary -> !Real Imaginary*Imaginary -> Real

    static Pow(expr, assumptions)
        Real**Integer -> Real Positive**Real -> Real Real**(Integer/Even) -> Real if base is nonnegative
        Real**(Integer/Odd) -> Real Imaginary**(Integer/Even) -> Real Imaginary**(Integer/Odd) ->
        not Real Imaginary**Real -> ? since Real could be 0 (giving real) or 1 (giving imaginary)
        b**Imaginary -> Real if log(b) is imaginary and b != 0 and exponent != integer multiple of
        I*pi/log(b) Real**Real -> ? e.g. sqrt(-1) is imaginary and sqrt(2) is not
```

Queries are used to ask information about expressions. Main method for this is ask():

```
sympy.assumptions.ask.ask(proposition, assumptions=True, context=AssumptionsContext([]))
    Method for inferring properties about objects.
```

### Syntax

- ask(proposition)
- ask(proposition, assumptions)  
where proposition is any boolean expression

### Examples

```
>>> from sympy import ask, Q, pi
>>> from sympy.abc import x, y
>>> ask(Q.rational(pi))
False
>>> ask(Q.even(x*y), Q.even(x) & Q.integer(y))
True
```

```
>>> ask(Q.prime(x*y), Q.integer(x) & Q.integer(y))
False
```

**Remarks** Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), Q.is_true(x > 0))
```

It is however a work in progress.

### 3.17.2 Querying

ask's optional second argument should be a boolean expression involving assumptions about objects in expr. Valid values include:

- Q.integer(x)
- Q.positive(x)
- Q.integer(x) & Q.positive(x)
- etc.

Q is a class in sympy.assumptions holding known predicates.

See documentation for the logic module for a complete list of valid boolean expressions.

You can also define a context so you don't have to pass that argument each time to function ask(). This is done by using the assuming context manager from module sympy.assumptions.

```
>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> facts = Q.positive(x), Q.positive(y)
>>> with assuming(*facts):
...     print(ask(Q.positive(2*x + y)))
True
```

### 3.17.3 Supported predicates

#### finite

Test that a function is bounded with respect to its variables. For example,  $\sin(x)$  is a bounded functions, but  $\exp(x)$  is not.

Examples:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> ask(Q.finite(exp(x)), ~Q.finite(x))
False
>>> ask(Q.finite(exp(x)), Q.finite(x))
True
>>> ask(Q.finite(sin(x)), ~Q.finite(x))
True
```

## commutative

Test that objects are commutative. By default, symbols in SymPy are considered commutative except otherwise stated.

Examples:

```
>>> from sympy import *
>>> x, y = symbols('x,y')
>>> ask(Q.commutative(x))
True
>>> ask(Q.commutative(x), ~Q.commutative(x))
False
>>> ask(Q.commutative(x*y), ~Q.commutative(x))
False
```

## complex

Test that expression belongs to the field of complex numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.complex(2))
True
>>> ask(Q.complex(I))
True
>>> x, y = symbols('x,y')
>>> ask(Q.complex(x+I*y), Q.real(x) & Q.real(y))
True
```

## even

Test that expression represents an even number, that is, a number that can be written in the form  $2^*n$ , n integer.

Examples:

```
>>> from sympy import *
>>> ask(Q.even(2))
True
>>> n = Symbol('n')
>>> ask(Q.even(2*n), Q.integer(n))
True
```

## extended\_real

Test that an expression belongs to the field of extended real numbers, that is, real numbers union { $\infty$ ,  $-\infty$ }.

Examples:

```
>>> from sympy import *
>>> ask(Q.extended_real(oo))
True
>>> ask(Q.extended_real(2))
True
```

```
>>> ask(Q.extended_real(x), Q.real(x))
True
```

## imaginary

Test that an expression belongs to the set of imaginary numbers, that is, it can be written as  $x*I$ , where  $x$  is real and  $I$  is the imaginary unit.

Examples:

```
>>> from sympy import *
>>> ask(Q.imaginary(2*I))
True
>>> x = Symbol('x')
>>> ask(Q.imaginary(x*I), Q.real(x))
True
```

## infinitesimal

Test that an expression is equivalent to an infinitesimal number.

Examples:

```
>>> from sympy import *
>>> ask(Q.infinitesimal(1/oo))
True
>>> x, y = symbols('x,y')
>>> ask(Q.infinitesimal(2*x), Q.infinitesimal(x))
True
>>> ask(Q.infinitesimal(x*y), Q.infinitesimal(x) & Q.finite(y))
True
```

## integer

Test that an expression belongs to the set of integer numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.integer(2))
True
>>> ask(Q.integer(sqrt(2)))
False
>>> x = Symbol('x')
>>> ask(Q.integer(x/2), Q.even(x))
True
```

## irrational

Test that an expression represents an irrational number.

Examples:

```
>>> from sympy import *
>>> ask(Q.irrational(pi))
True
>>> ask(Q.irrational(sqrt(2)))
True
>>> ask(Q.irrational(x*sqrt(2)), Q.rational(x))
True
```

## rational

Test that an expression represents a rational number.

Examples:

```
>>> from sympy import *
>>> ask(Q.rational(Rational(3, 4)))
True
>>> x, y = symbols('x,y')
>>> ask(Q.rational(x/2), Q.integer(x))
True
>>> ask(Q.rational(x/y), Q.integer(x) & Q.integer(y))
True
```

## negative

Test that an expression is less (strict) than zero.

Examples:

```
>>> from sympy import *
>>> ask(Q.negative(0.3))
False
>>> x = Symbol('x')
>>> ask(Q.negative(-x), Q.positive(x))
True
```

## Remarks

negative numbers are defined as real numbers that are not zero nor positive, so complex numbers (with nontrivial imaginary coefficients) will return False for this predicate. The same applies to Q.positive.

## positive

Test that a given expression is greater (strict) than zero.

Examples:

```
>>> from sympy import *
>>> ask(Q.positive(0.3))
True
>>> x = Symbol('x')
>>> ask(Q.positive(-x), Q.negative(x))
True
```

## Remarks

see Remarks for negative

## prime

Test that an expression represents a prime number.

Examples:

```
>>> from sympy import *
>>> ask(Q.prime(13))
True
```

Remarks: Use sympy.ntheory.isprime to test numeric values efficiently.

## real

Test that an expression belongs to the field of real numbers.

Examples:

```
>>> from sympy import *
>>> ask(Q.real(sqrt(2)))
True
>>> x, y = symbols('x,y')
>>> ask(Q.real(x*y), Q.real(x) & Q.real(y))
True
```

## odd

Test that an expression represents an odd number.

Examples:

```
>>> from sympy import *
>>> ask(Q.odd(3))
True
>>> n = Symbol('n')
>>> ask(Q.odd(2*n + 1), Q.integer(n))
True
```

## nonzero

Test that an expression is not zero.

Examples:

```
>>> from sympy import *
>>> x = Symbol('x')
>>> ask(Q.nonzero(x), Q.positive(x) | Q.negative(x))
True
```

### 3.17.4 Design

Each time ask is called, the appropriate Handler for the current key is called. This is always a subclass of sympy.assumptions.AskHandler. Its classmethods have the name's of the classes it supports. For example, a (simplified) AskHandler for the ask ‘positive’ would look like this:

```
class AskPositiveHandler(CommonHandler):

    def Mul(self):
        # return True if all argument's in self.expr.args are positive
        ...

    def Add(self):
        for arg in self.expr.args:
            if not ask(arg, positive, self.assumptions):
                break
        else:
            # if all argument's are positive
            return True
        ...

    ...
```

The .Mul() method is called when self.expr is an instance of Mul, the Add method would be called when self.expr is an instance of Add and so on.

### 3.17.5 Extensibility

You can define new queries or support new types by subclassing sympy.assumptions.AskHandler and registering that handler for a particular key by calling register\_handler:

```
sympy.assumptions.ask.register_handler(key, handler)
```

Register a handler in the ask system. key must be a string and handler a class inheriting from AskHandler:

```
>>> from sympy.assumptions import register_handler, ask, Q
>>> from sympy.assumptions.handlers import AskHandler
>>> class MersenneHandler(AskHandler):
...     # Mersenne numbers are in the form 2**n + 1, n integer
...     @staticmethod
...     def Integer(expr, assumptions):
...         import math
...         return ask(Q.integer(math.log(expr + 1, 2)))
>>> register_handler('mersenne', MersenneHandler)
>>> ask(Q.mersenne(7))
True
```

You can undo this operation by calling remove\_handler.

```
sympy.assumptions.ask.remove_handler(key, handler)
```

Removes a handler from the ask system. Same syntax as register\_handler

You can support new types<sup>6</sup> by adding a handler to an existing key. In the following example, we will create a new type MyType and extend the key ‘prime’ to accept this type (and return True)

```
>>> from sympy.core import Basic
>>> from sympy.assumptions import register_handler
>>> from sympy.assumptions.handlers import AskHandler
>>> class MyType(Basic):
```

<sup>6</sup> New type must inherit from Basic, otherwise an exception will be raised. This is a bug and should be fixed.

```

...
    pass
>>> class MyAskHandler(AskHandler):
...
    @staticmethod
...
    def MyType(expr, assumptions):
...
        return True
>>> a = MyType()
>>> register_handler('prime', MyAskHandler)
>>> ask(Q.prime(a))
True

```

### 3.17.6 Performance improvements

On queries that involve symbolic coefficients, logical inference is used. Work on improving satisfiable function (`sympy.logic.inference.satisfiable`) should result in notable speed improvements.

Logic inference used in one ask could be used to speed up further queries, and current system does not take advantage of this. For example, a truth maintenance system ([http://en.wikipedia.org/wiki/Truth\\_maintenance\\_system](http://en.wikipedia.org/wiki/Truth_maintenance_system)) could be implemented.

### 3.17.7 Misc

You can find more examples in the in the form of test under directory `sympy/assumptions/tests/`

## 3.18 Term rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently SymPy has several functions and basic built-in methods for performing various types of rewriting.

### 3.18.1 Expanding

The simplest rewrite rule is expanding expressions into a `_sparse_` form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

#### Expanding of arithmetic expressions involving products and powers:

```

>>> from sympy import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)*(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2

```

Arithmetic expand is done by default in `expand()` so the keyword `basic` can be omitted. However you can set `basic=False` to avoid this type of expand if you use rules described below. This give complete control on what is done with the expression.

Another type of expand rule is expanding complex valued expressions and putting them into a normal form. For this `complex` keyword is used. Note that it will always perform arithmetic expand to obtain the desired normal form:

```
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)

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

Note also that the same behavior can be obtained by using `as_real_imag()` method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using `collect` function:

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

>>> collect((x + I*y).expand(complex=True), I, evaluate=False)
{1: re(x) - im(y), I: re(y) + im(x)}
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use `rewrite()` to do specific type of rewrite:

```
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```

### 3.18.2 Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the `cse` function. Examples:

```
>>> from sympy import cse, sqrt, sin, pprint
>>> from sympy.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
[] , \ sin(x)

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
[(x, sin(x))] , \ x + 4 \ x + 5

>>> pprint(cse(sqrt(sin(x+1) + 5 + cos(y))*sqrt(sin(x+1) + 4 + cos(y))),
...     use_unicode=True)
[(x, sin(x + 1) + cos(y))] , \ x + 4 \ x + 5

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
[(x, -y), (x1, (x + x)(x + z))] , \ x1 + x1
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the “`optimizations`“ optional argument. A set of predefined basic optimizations can be applied by passing `optimizations='basic'`:

```
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y)), optimizations='basic'),
...     use_unicode=True)
[(x, -(x - y)(y - z))] , \ x + x
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

## 3.19 Series Expansions

The series module implements series expansions as a function and many related functions.

`class sympy.series.limits.Limit`

Represents a directional limit of `expr` at the point `z0`.

**Parameters** `expr` : `Expr`

algebraic expression

`z` : `Symbol`

variable of the `expr`

`z0` : `Expr`

limit point,  $z_0$

`dir` : {“+”, “-”, “real”}, optional

For `dir="+"` (default) it calculates the limit from the right ( $z \rightarrow z_0 + 0$ ) and for `dir="-"` the limit from the left ( $z \rightarrow z_0 - 0$ ). If `dir="real"`, the limit is the bidirectional real limit. For infinite `z0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

### Examples

```
>>> from sympy import Limit, sin
>>> from sympy.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir="-")
Limit(1/x, x, 0, dir='-')
```

`doit(**hints)`

Evaluates limit.

### Notes

First we handle some trivial cases (i.e. constant), then try Gruntz algorithm (see the `gruntz` (page 900) module).

`sympy.series.limits.limit(expr, z, z0, dir='+')`

Compute the directional limit of `expr` at the point `z0`.

**See also:**

`Limit` (page 897)

## Examples

```
>>> from sympy import limit, sin, oo
>>> from sympy.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0, dir="+")
oo
>>> limit(1/x, x, 0, dir="-")
-oo
>>> limit(1/x, x, oo)
0
```

`sympy.series.series.series(expr, x=None, x0=0, n=6, dir='+')  
Series expansion of expr in x around point x0.`

### See also:

[sympy.core.expr.Expr.series](#) (page 94)

`sympy.series.order.O`  
alias of [Order](#) (page 898)

`class sympy.series.order.Order`

Represents the limiting behavior of some function

The order of a function characterizes the function based on the limiting behavior of the function as it goes to some limit. Only taking the limit point to be a number is currently supported. This is expressed in big O notation [R346] (page 1246).

The formal definition for the order of a function  $g(x)$  about a point  $a$  is such that  $g(x) = O(f(x))$  as  $x \rightarrow a$  if and only if for any  $\delta > 0$  there exists a  $M > 0$  such that  $|g(x)| \leq M|f(x)|$  for  $|x - a| < \delta$ . This is equivalent to  $\lim_{x \rightarrow a} \sup |g(x)/f(x)| < \infty$ .

Let's illustrate it on the following example by taking the expansion of  $\sin(x)$  about 0:

$$\sin(x) = x - x^3/3! + O(x^5)$$

where in this case  $O(x^5) = x^5/5! - x^7/7! + \dots$ . By the definition of  $O$ , for any  $\delta > 0$  there is an  $M$  such that:

$$|x^5/5! - x^7/7! + \dots| \leq M|x^5| \text{ for } |x| < \delta$$

or by the alternate definition:

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| < \infty$$

which surely is true, because

$$\lim_{x \rightarrow 0} |(x^5/5! - x^7/7! + \dots)/x^5| = 1/5!$$

As it is usually used, the order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example,  $O(x^3)$  corresponds to any terms proportional to  $x^3, x^4, \dots$  and any higher power. For a polynomial, this leaves terms proportional to  $x^2, x$  and constants.

## Notes

In `O(f(x), x)` the expression `f(x)` is assumed to have a leading term. `O(f(x), x)` is automatically transformed to `O(f(x).as_leading_term(x), x)`.

```
O(expr*f(x), x) is O(f(x), x)
O(expr, x) is O(1)
O(0, x) is 0.
```

Multivariate O is also supported:

```
O(f(x, y), x, y) is transformed to O(f(x, y).as_leading_term(x,y).as_leading_term(y),
x, y)
```

In the multivariate case, it is assumed the limits w.r.t. the various symbols commute.

If no symbols are passed then all symbols in the expression are used and the limit point is assumed to be zero.

## References

[R346] (page 1246)

## Examples

```
>>> from sympy import O, oo, cos, pi
>>> from sympy.abc import x, y

>>> O(x + x**2)
O(x)
>>> O(x + x**2, (x, 0))
O(x)
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))

>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1, x, y)
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))

>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True

>>> O(x)*x
O(x**2)
>>> O(x) - O(x)
O(x)
>>> O(cos(x))
O(1)
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

```
contains(*args, **kwargs)
```

Return True if expr belongs to Order(self.expr, \*self.variables). Return False if self belongs to expr. Return None if the inclusion relation cannot be determined (e.g. when self and expr have different symbols).

```
sympy.series.residues.residue(expr, x, x0)
```

Finds the residue of `expr` at the point `x=x0`.

The residue is defined [R347] (page 1246) as the coefficient of  $1/(x - x_0)$  in the power series expansion around  $x = x_0$ .

This notion is essential for the Residue Theorem [R348] (page 1246)

## References

[R347] (page 1246), [R348] (page 1246)

## Examples

```
>>> from sympy import residue, sin
>>> from sympy.abc import x
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

### 3.19.1 The Gruntz Algorithm

This section explains the basics of the algorithm [R349] (page 1246) used for computing limits. Most of the time the `limit()` (page 897) function should just work. However it is still useful to keep in mind how it is implemented in case something does not work as expected.

First we define an ordering on functions of single variable  $x$  according to how rapidly varying they at infinity. Any two functions  $f(x)$  and  $g(x)$  can be compared using the properties of:

$$L = \lim_{x \rightarrow \infty} \frac{\log |f(x)|}{\log |g(x)|}$$

We shall say that  $f(x)$  dominates  $g(x)$ , written  $f(x) \succ g(x)$ , iff  $L = \pm\infty$ . We also say that  $f(x)$  and  $g(x)$  are of the same comparability class if neither  $f(x) \succ g(x)$  nor  $g(x) \succ f(x)$  and shall denote it as  $f(x) \asymp g(x)$ .

It is easy to show the following examples:

- $e^{e^x} \succ e^{x^2} \succ e^x \succ x \succ 42$
- $2 \asymp 3 \asymp -5$
- $x \asymp x^2 \asymp x^3 \asymp -x$
- $e^x \asymp e^{-x} \asymp e^{2x} \asymp e^{x+e^{-x}}$
- $f(x) \asymp 1/f(x)$

Using these definitions yields the following strategy for computing  $\lim_{x \rightarrow \infty} f(x)$ :

- Given the function  $f(x)$ , we find the set of *most rapidly varying subexpressions* (MRV set) of it. All items of this set belongs to the same comparability class. Let's say it is  $\{e^x, e^{2x}\}$ .
- Choose an expression  $\omega$  which is positive and tends to zero and which is in the same comparability class as any element of the MRV set. Such element always exists. Then we rewrite the MRV set using  $\omega$ , in our case  $\{\omega^{-1}, \omega^{-2}\}$ , and substitute it into  $f(x)$ .
- Let  $f(\omega)$  be the function which is obtained from  $f(x)$  after the rewrite step above. Consider all expressions independent of  $\omega$  as constants and compute the leading term of the power series of  $f(\omega)$  around  $\omega = 0^+$ :

$$f(\omega) = c_0\omega^{e_0} + c_1\omega^{e_1} + \dots$$

where  $e_0 < e_1 < e_2 \dots$

- If the leading exponent  $e_0 > 0$  then the limit is 0. If  $e_0 < 0$ , then the answer is  $\pm\infty$  (depends on sign of  $c_0$ ). Finally, if  $e_0 = 0$ , the limit is the limit of the leading coefficient  $c_0$ .

## Notes

This exposition glossed over several details. For example, limits could be computed recursively (steps 1 and 4). Please address to the Gruntz thesis [R349] (page 1246) for proof of the termination (pp. 52-60).

## References

`class sympy.series.gruntz.SubsSet`

Stores (expr, dummy) pairs, and how to rewrite expr-s.

The gruntz algorithm needs to rewrite certain expressions in term of a new variable w. We cannot use subs, because it is just too smart for us. For example:

```
> Omega=[exp(exp(_p - exp(-_p))/(1 - 1/_p)), exp(exp(_p))]
> O2=[exp(-exp(_p) + exp(-exp(-_p))*exp(_p)/(1 - 1/_p))/_w, 1/_w]
> e = exp(exp(_p - exp(-_p))/(1 - 1/_p)) - exp(exp(_p))
> e.subs(Omega[0],O2[0]).subs(Omega[1],O2[1])
-1/w + exp(exp(p)*exp(-exp(-p))/(1 - 1/p))
```

is really not what we want!

So we do it the hard way and keep track of all the things we potentially want to substitute by dummy variables. Consider the expression:

```
exp(x - exp(-x)) + exp(x) + x.
```

The mrv set is  $\{\exp(x), \exp(-x), \exp(x - \exp(-x))\}$ . We introduce corresponding dummy variables d1, d2, d3 and rewrite:

```
d3 + d1 + x.
```

This class first of all keeps track of the mapping expr->variable, i.e. will at this stage be a dictionary:

```
{exp(x): d1, exp(-x): d2, exp(x - exp(-x)): d3}.
```

[It turns out to be more convenient this way round.] But sometimes expressions in the mrv set have other expressions from the mrv set as subexpressions, and we need to keep track of that as well. In this case, d3 is really  $\exp(x - d2)$ , so rewrites at this stage is:

```
{d3: exp(x-d2)}.
```

The function rewrite uses all this information to correctly rewrite our expression in terms of w. In this case w can be chosen to be  $\exp(-x)$ , i.e. d2. The correct rewriting then is:

```
exp(-w)/w + 1/w + x.
```

```
meets(s2)
```

Tell whether or not self and s2 have non-empty intersection

```
union(s2, exps=None)
```

Compute the union of self and s2, adjusting exps

```
sympy.series.gruntz.build_expression_tree(Omega, rewrites)
```

Helper function for rewrite.

We need to sort Omega (mrv set) so that we replace an expression before we replace any expression in terms of which it has to be rewritten:

```
e1 ---> e2 ---> e3  
      \  
      -> e4
```

Here we can do e1, e2, e3, e4 or e1, e2, e4, e3. To do this we assemble the nodes into a tree, and sort them by height.

This function builds the tree, rewrites then sorts the nodes.

```
sympy.series.gruntz.calculate_series(e, x, logx=None)
```

Calculates at least one term of the series of “e” in “x”.

This is a place that fails most often, so it is in its own function.

```
sympy.series.gruntz.compare(a, b, x)
```

Returns “<” if  $a < b$ , “=” for  $a == b$ , “>” for  $a > b$

```
sympy.series.gruntz.gruntz(e, z, z0, dir='+')
```

Compute the limit of  $e(z)$  at the point  $z_0$  using the Gruntz algorithm.

$z_0$  can be any expression, including  $\infty$  and  $-\infty$ .

For  $dir=“+”$  (default) it calculates the limit from the right ( $z \rightarrow z_0 +$ ) and for  $dir=“-”$  the limit from the left ( $z \rightarrow z_0 -$ ). For infinite  $z_0$  ( $\infty$  or  $-\infty$ ), the dir argument doesn’t matter.

This algorithm is fully described in the module docstring in the gruntz.py file. It relies heavily on the series expansion. Most frequently, `gruntz()` is only used if the faster `limit()` function (which uses heuristics) fails.

```
sympy.series.gruntz.limitinf(*args, **kwargs)
```

Limit  $e(x)$  for  $x \rightarrow \infty$

```
sympy.series.gruntz.mrv(e, x)
```

Returns a SubsSet of most rapidly varying (mrv) subexpressions of ‘e’, and e rewritten in terms of these

```
sympy.series.gruntz.mrv_leadterm(*args, **kwargs)
```

Returns  $(c_0, e_0)$  for e.

```
sympy.series.gruntz.mrv_max1(f, g, exps, x)
```

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. `mrv_max1()` compares (two elements of) f and g and returns the set, which is in the higher comparability class of the union of both, if they have the same order of variation. Also returns exps, with the appropriate substitutions made.

```
sympy.series.gruntz.mrv_max3(f, expsf, g, expsg, union, expsboth, x)
```

Computes the maximum of two sets of expressions f and g, which are in the same comparability class, i.e. max() compares (two elements of) f and g and returns either (f, expsf) [if f is larger], (g, expsg) [if g is larger] or (union, expsboth) [if f, g are of the same class].

```
sympy.series.gruntz.rewrite(e, Omega, x, wsym)
```

e(x) ... the function Omega ... the mrv set wsym ... the symbol which is going to be used for w

Returns the rewritten e in terms of w and log(w). See test\_rewrite1() for examples and correct results.

```
sympy.series.gruntz.sign(*args, **kwargs)
```

Returns a sign of an expression e(x) for x->oo.

```
e > 0 for x sufficiently large ... 1
```

```
e == 0 for x sufficiently large ... 0
```

```
e < 0 for x sufficiently large ... -1
```

The result of this function is currently undefined if e changes sign arbitrarily often for arbitrarily large x (e.g. sin(x)).

Note that this returns zero only if e is *constantly* zero for x sufficiently large. [If e is constant, of course, this is just the same thing as the sign of e.]

## 3.20 Sets

### 3.20.1 Set

```
class sympy.sets.sets.Set
```

The base class for any kind of set.

This is not meant to be used directly as a container of items. It does not behave like the builtin `set`; see [FiniteSet](#) (page 910) for that.

Real intervals are represented by the [Interval](#) (page 908) class and unions of sets by the [Union](#) (page 910) class. The empty set is represented by the [EmptySet](#) (page 913) class and available as a singleton as `S.EmptySet`.

**boundary**

The boundary or frontier of a set

A point x is on the boundary of a set S if

1.x is in the closure of S. I.e. Every neighborhood of x contains a point in S.

2.x is not in the interior of S. I.e. There does not exist an open set centered on x contained entirely within S.

There are the points on the outer rim of S. If S is open then these points need not actually be contained within S.

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

#### Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

**complement(universe)**  
The complement of ‘self’ w.r.t the given the universe.

### Examples

```
>>> from sympy import Interval, S
>>> Interval(0, 1).complement(S.Reals)
(-oo, 0) U (1, oo)

>>> Interval(0, 1).complement(S.UniversalSet)
UniversalSet() \ [0, 1]
```

**contains(other)**  
Returns True if ‘other’ is contained in ‘self’ as an element.  
As a shortcut it is possible to use the ‘in’ operator:

### Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).contains(0.5)
True
>>> 0.5 in Interval(0, 1)
True
```

**inf**  
The infimum of ‘self’

### Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

**intersect(other)**  
Returns the intersection of ‘self’ and ‘other’.

```
>>> from sympy import Interval

>>> Interval(1, 3).intersect(Interval(1, 2))
[1, 2]
```

**intersection(other)**  
Alias for `intersect()` (page 904)

**is\_disjoint(other)**  
Returns True if ‘self’ and ‘other’ are disjoint

## References

[R350] (page 1246)

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

`is_proper_subset(other)`

Returns True if ‘self’ is a proper subset of ‘other’.

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

`is_proper_superset(other)`

Returns True if ‘self’ is a proper superset of ‘other’.

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

`is_subset(other)`

Returns True if ‘self’ is a subset of ‘other’.

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

`is_superset(other)`

Returns True if ‘self’ is a superset of ‘other’.

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True

isdisjoint(other)
    Alias for is\_disjoint\(\) (page 904)

issubset(other)
    Alias for is\_subset\(\) (page 905)

issuperset(other)
    Alias for is\_superset\(\) (page 905)

measure
    The (Lebesgue) measure of ‘self’
```

## Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2

powerset()
    Find the Power set of ‘self’.
```

## References

[R351] (page 1246)

## Examples

```
>>> from sympy import FiniteSet, EmptySet
>>> A = EmptySet()
>>> A.powerset()
{EmptySet()}
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet())
True
```

**sup**  
The supremum of ‘self’

## Examples

```
>>> from sympy import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

### union(*other*)

Returns the union of ‘self’ and ‘other’.

### Examples

As a shortcut it is possible to use the ‘+’ operator:

```
>>> from sympy import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
[0, 1] U [2, 3]
>>> Interval(0, 1) + Interval(2, 3)
[0, 1] U [2, 3]
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
(1, 2] U {3}
```

Similarly it is possible to use the ‘-‘ operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
(1, 2]
>>> Interval(1, 3) - FiniteSet(2)
[1, 2) U (2, 3]
```

### sympy.sets.sets.imageset(\*args)

Image of set under transformation *f*.

If this function can’t compute the image, it returns an unevaluated ImageSet object.

$$f(x) | x \in \text{self}$$

See also:

[sympy.sets.fancysets.ImageSet](#) (page 915)

### Examples

```
>>> from sympy import Interval, Symbol, imageset, sin, Lambda
>>> x = Symbol('x')

>>> imageset(x, 2*x, Interval(0, 2))
[0, 4]

>>> imageset(lambda x: 2*x, Interval(0, 2))
[0, 4]

>>> imageset(Lambda(x, sin(x)), Interval(-2, 1))
ImageSet(Lambda(x, sin(x)), [-2, 1])
```

## Elementary Sets

### 3.20.2 Interval

```
class sympy.sets.sets.Interval  
    Represents a real interval as a Set.
```

**Usage:** Returns an interval with end points “start” and “end”.

For left\_open=True (default left\_open is False) the interval will be open on the left. Similarly, for right\_open=True the interval will be open on the right.

#### Notes

- Only real end points are supported
- Interval(a, b) with a > b will return the empty set
- Use the evalf() method to turn an Interval into an mpmath ‘mpi’ interval instance

#### References

[R352] (page 1246)

#### Examples

```
>>> from sympy import Symbol, Interval  
>>> Interval(0, 1)  
[0, 1]  
>>> Interval(0, 1, False, True)  
[0, 1)  
>>> Interval.Ropen(0, 1)  
[0, 1)  
>>> Interval.Lopen(0, 1)  
(0, 1]  
>>> Interval.open(0, 1)  
(0, 1)  
  
>>> a = Symbol('a', extended_real=True)  
>>> Interval(0, a)  
[0, a]  
  
classmethod Lopen(a, b)  
    Return an interval not including the left boundary.  
  
classmethod Ropen(a, b)  
    Return an interval not including the right boundary.  
  
as_relational(x)  
    Rewrite an interval in terms of inequalities and logic operators.  
  
end  
    The right end point of ‘self’.  
  
    This property takes the same value as the ‘sup’ property.
```

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).end
1

is_left_unbounded
Return True if the left endpoint is negative infinity.

is_right_unbounded
Return True if the right endpoint is positive infinity.

left
The left end point of ‘self’.
This property takes the same value as the ‘inf’ property.
```

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0

left_open
True if ‘self’ is left-open.
```

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, left_open=True).left_open
True
>>> Interval(0, 1, left_open=False).left_open
False

classmethod open(a, b)
Return an interval including neither boundary.

right
The right end point of ‘self’.
This property takes the same value as the ‘sup’ property.
```

## Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).end
1

right_open
True if ‘self’ is right-open.
```

### Examples

```
>>> from sympy import Interval
>>> Interval(0, 1, right_open=True).right_open
True
>>> Interval(0, 1, right_open=False).right_open
False

start
The left end point of ‘self’.

This property takes the same value as the ‘inf’ property.
```

### Examples

```
>>> from sympy import Interval
>>> Interval(0, 1).start
0
```

## 3.20.3 FiniteSet

```
class sympy.sets.sets.FiniteSet
Represents a finite set of discrete numbers
```

### References

[R353] (page 1246)

### Examples

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
>>> 3 in FiniteSet(1, 2, 3, 4)
True

as_relational(symbol)
Rewrite a FiniteSet in terms of equalities and logic operators.
```

## Compound Sets

## 3.20.4 Union

```
class sympy.sets.sets.Union
Represents a union of sets as a Set (page 903).
```

See also:

[Intersection](#) (page 911)

## References

[R354] (page 1246)

## Examples

```
>>> from sympy import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
[1, 2] ∪ [3, 4]
```

The Union constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
[1, 3]
```

`as_relational(symbol)`

Rewrite a Union in terms of equalities and logic operators.

`static reduce(args)`

Simplify a `Union` (page 910) using known rules

We first start with global rules like ‘Merge all FiniteSets’

Then we iterate through all pairs and ask the constituent sets if they can simplify themselves with any other constituent

## 3.20.5 Intersection

`class sympy.sets.sets.Intersection`

Represents an intersection of sets as a `Set` (page 903).

**See also:**

`Union` (page 910)

## References

[R355] (page 1246)

## Examples

```
>>> from sympy import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
[2, 3]
```

We often use the `.intersect` method

```
>>> Interval(1,3).intersect(Interval(2,4))
[2, 3]
```

`as_relational(symbol)`

Rewrite an Intersection in terms of equalities and logic operators

```
static reduce(args)
    Simplify an intersection using known rules
    We first start with global rules like ‘if any empty sets return empty set’ and ‘distribute any unions’
    Then we iterate through all pairs and ask the constituent sets if they can simplify themselves with
    any other constituent
```

### 3.20.6 ProductSet

```
class sympy.sets.sets.ProductSet
    Represents a Cartesian Product of Sets.
    Returns a Cartesian product given several sets as either an iterable or individual arguments.
    Can use '*' operator on any sets for convenient shorthand.
```

#### Notes

- Passes most operations down to the argument sets
- Flattens Products of ProductSets

#### References

[R356] (page 1246)

#### Examples

```
>>> from sympy import Interval, FiniteSet, ProductSet
>>> I = Interval(0, 5); S = FiniteSet(1, 2, 3)
>>> ProductSet(I, S)
[0, 5] x {1, 2, 3}

>>> (2, 2) in ProductSet(I, S)
True

>>> Interval(0, 1) * Interval(0, 1) # The unit square
[0, 1] x [0, 1]

>>> coin = FiniteSet('H', 'T')
>>> set(coin**2)
set([(H, H), (H, T), (T, H), (T, T)])
```

### 3.20.7 Complement

```
class sympy.sets.sets.Complement
    Represents the set difference or relative complement of a set with another set.
     $A - B = \{x \in A | x \notin B\}$ 
See also:
```

[Intersection](#) (page 911), [Union](#) (page 910)

## References

<http://mathworld.wolfram.com/SetComplement.html>

## Examples

```
>>> from sympy import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}

static reduce(A, B)
Simplify a Complement (page 912).
```

## Singleton Sets

### 3.20.8 EmptySet

`class sympy.sets.sets.EmptySet`

Represents the empty set. The empty set is available as a singleton as S.EmptySet.

See also:

[UniversalSet](#) (page 913)

## References

[R357] (page 1246)

## Examples

```
>>> from sympy import S, Interval
>>> S.EmptySet
EmptySet()

>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet()
```

### 3.20.9 UniversalSet

`class sympy.sets.sets.UniversalSet`

Represents the set of all things. The universal set is available as a singleton as S.UniversalSet

See also:

[EmptySet](#) (page 913)

## References

[R358] (page 1246)

## Examples

```
>>> from sympy import S, Interval
>>> S.UniversalSet
UniversalSet()

>>> Interval(1, 2).intersect(S.UniversalSet)
[1, 2]
```

## Special Sets

### 3.20.10 Naturals

`class sympy.sets.fancysets.Naturals`

Represents the natural numbers (or counting numbers) which are all positive integers starting from 1. This set is also available as the Singleton, `S.Naturals`.

See also:

[Naturals0 \(page 914\)](#) non-negative integers (i.e. includes 0, too)

[Integers \(page 915\)](#) also includes negative integers

## Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```

### 3.20.11 Naturals0

`class sympy.sets.fancysets.Naturals0`

Represents the whole numbers which are all the non-negative integers, inclusive of zero.

See also:

[Naturals \(page 914\)](#) positive integers; does not include 0

[Integers \(page 915\)](#) also includes the negative integers

### 3.20.12 Integers

```
class sympy.sets.fancysets.Integers
```

Represents all integers: positive, negative and zero. This set is also available as the Singleton, S.Integers.

See also:

[Naturals0](#) (page 914) non-negative integers

[Integers](#) (page 915) positive and negative integers and zero

#### Examples

```
>>> from sympy import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
>>> next(iterable)
-1
>>> next(iterable)
2

>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

### 3.20.13 ImageSet

```
class sympy.sets.fancysets.ImageSet
```

Image of a set under a mathematical function

#### Examples

```
>>> from sympy import Symbol, S, ImageSet, FiniteSet, Lambda

>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>>> 4 in squares
True
>>> 5 in squares
False

>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
{1, 4, 9}

>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
```

4  
9  
16

## 3.21 Simplify

### 3.21.1 simplify

```
sympy.simplify.simplify(expr, ratio=1.7, measure=<function count_ops at 0x7fb9aef6e60>, fu=False)
```

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.

Nonetheless, especially for interactive use, or when you don’t know anything about the structure of the expression, `simplify()` tries to apply intelligent heuristics to make the input expression “simpler”. For example:

```
>>> from sympy import simplify, cos, sin
>>> from sympy.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> a
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
x + 1
```

Note that we could have obtained the same result by using specific simplification functions:

```
>>> from sympy import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying `simplify()` (page 916) may actually result in some more complicated expression. The default `ratio=1.7` prevents more extreme cases: if  $(\text{result length})/(\text{input length}) > \text{ratio}$ , then input is returned unmodified. The `measure` parameter lets you specify the function used to determine how complex an expression is. The function should take a single argument as an expression and return a number such that if expression `a` is more complex than expression `b`, then `measure(a) > measure(b)`. The default measure function is `count_ops()` (page 145), which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output can’t be longer than input.

```
>>> from sympy import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since `simplify(root)` would result in a slightly longer expression, `root` is returned unchanged instead:

```
>>> simplify(root, ratio=1) == root
True
```

If `ratio=oo`, `simplify` will be applied anyway:

```
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not necessarily the simplest, so setting `ratio` to 1 may not be a good idea. Heuristically, the default value `ratio=1.7` seems like a reasonable choice.

You can easily define your own measure function based on what you feel should represent the “size” or “complexity” of the input expression. Note that some choices, such as `lambda expr: len(str(expr))` may appear to be good metrics, but have other problems (in this case, the measure function may slow down simplify too much for very large expressions). If you don’t know what a good metric would be, the default, `count_ops`, is a good one.

For example:

```
>>> from sympy import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(-log(a) + 1))
>>> count_ops(g)
8
>>> count_ops(h)
5
```

So you can see that `h` is simpler than `g` using the `count_ops` metric. However, we may not like how `simplify` (in this case, using `logcombine`) has created the `b**(log(1/a) + 1)` term. A simple way to reduce this would be to give more weight to powers as operations in `count_ops`. We can do this by using the `visual=True` option:

```
>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB

>>> from sympy import Symbol, S
>>> def my_measure(expr):
...     POW = Symbol('POW')
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(POW, 10)
...     # Every other operation gets a weight of 1 (the default)
...     count = count.replace(Symbol, type(S.One))
...     return count
>>> my_measure(g)
8
>>> my_measure(h)
14
>>> 15./8 > 1.7 # 1.7 is the default ratio
True
>>> simplify(g, measure=my_measure)
-log(a)*log(b) + log(a) + log(b)
```

Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

### 3.21.2 collect

```
sympy.simplify.collect(expr, syms, func=None, evaluate=None, exact=False, distribute_order_term=True)
```

Collect additive terms of an expression.

This function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents. By the term symbol here are meant arbitrary expressions, which can contain powers, products, sums etc. In other words symbol is a pattern which will be searched for in the expression's terms.

The input expression is not expanded by `collect()` (page 918), so user is expected to provide an expression in an appropriate form. This makes `collect()` (page 918) more predictable as there is no magic happening behind the scenes. However, it is important to note, that powers of products are converted to products of powers using the `expand_power_base()` (page 148) function.

There are two possible types of output. First, if `evaluate` flag is set, this function will return an expression with collected terms or else it will return a dictionary with expressions up to rational powers as keys and collected coefficients as values.

**See also:**

`collect_const` (page 929), `collect_sqrt` (page 929), `rcollect` (page 920)

#### Examples

```
>>> from sympy import S, collect, expand, factor, Wild
>>> from sympy.abc import a, b, c, x, y, z
```

This function can collect symbolic coefficients in polynomials or rational expressions. It will manage to find all integer or rational powers of collection variable:

```
>>> collect(a*x**2 + b*x**2 + a*x - b*x + c, x)
c + x**2*(a + b) + x*(a - b)
```

The same result can be achieved in dictionary form:

```
>>> d = collect(a*x**2 + b*x**2 + a*x - b*x + c, x, evaluate=False)
>>> d[x**2]
a + b
>>> d[x]
a - b
>>> d[S.One]
c
```

You can also work with multivariate polynomials. However, remember that this function is greedy so it will care only about a single symbol at time, in specification order:

```
>>> collect(x**2 + y*x**2 + x*y + y + a*y, [x, y])
x**2*(y + 1) + x*y + y*(a + 1)
```

Also more complicated expressions can be used as patterns:

```
>>> from sympy import sin, log
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)

>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```
>>> w = Wild('w1')
>>> collect(a*x**y - b*x**y, w**y)
x**y*(a - b)
```

It is also possible to work with symbolic powers, although it has more complicated behavior, because in this case power's base and symbolic part of the exponent are treated as a single symbol:

```
>>> collect(a*x**c + b*x**c, x)
a*x**c + b*x**c
>>> collect(a*x**c + b*x**c, x**c)
x**c*(a + b)
```

However if you incorporate rationals to the exponents, then you will get well known behavior:

```
>>> collect(a*x**(2*c) + b*x**(2*c), x**c)
x**(2*c)*(a + b)
```

Note also that all previously stated facts about `collect()` (page 918) function apply to the exponential function, so you can get:

```
>>> from sympy import exp
>>> collect(a*exp(2*x) + b*exp(2*x), exp(x))
(a + b)*exp(2*x)
```

If you are interested only in collecting specific powers of some symbols then set `exact` flag in arguments:

```
>>> collect(a*x**7 + b*x**7, x, exact=True)
a*x**7 + b*x**7
>>> collect(a*x**7 + b*x**7, x**7, exact=True)
x**7*(a + b)
```

You can also apply this function to differential equations, where derivatives of arbitrary order can be collected. Note that if you collect with respect to a function or a derivative of a function, all derivatives of that function will also be collected. Use `exact=True` to prevent this from happening:

```
>>> from sympy import Derivative as D, collect, Function
>>> f = Function('f')(x)

>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), f)
(a + b)*Derivative(f(x), x, x)

>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), x, x) + b*Derivative(f(x), x, x)

>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
(a + b)*f(x) + (a + b)*Derivative(f(x), x)
```

Or you can even match both derivative order and exponent at the same time:

```
>>> collect(a*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), x, x)**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:

```
>>> f = expand((x + a + 1)**3)
>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

---

**Note:** Arguments are expected to be in expanded form, so you might have to call [expand\(\)](#) (page 142) prior to calling this function.

---

`sympy.simplify.simplify.rcollect(expr, *vars)`

Recursively collect sums in an expression.

See also:

[collect](#) (page 918), [collect\\_const](#) (page 929), [collect\\_sqrt](#) (page 929)

#### Examples

```
>>> from sympy.simplify import rcollect
>>> from sympy.abc import x, y

>>> expr = (x**2*y + x*y + x + y)/(x + y)

>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

### 3.21.3 separatevars

`sympy.simplify.simplify.separatevars(expr, symbols=[], dict=False, force=False)`

Separates variables in an expression, if possible. By default, it separates with respect to all symbols in an expression and collects constant coefficients that are independent of symbols.

If `dict=True` then the separated terms will be returned in a dictionary keyed to their corresponding symbols. By default, all symbols in the expression will appear as keys; if `symbols` are provided, then all those symbols will be used as keys, and any terms in the expression containing other symbols or non-symbols will be returned keyed to the string ‘coeff’. (Passing `None` for `symbols` will return the expression in a dictionary keyed to ‘coeff’.)

If `force=True`, then bases of powers will be separated regardless of assumptions on the symbols involved.

#### Notes

The order of the factors is determined by `Mul`, so that the separated expressions may not necessarily be grouped together.

Although factoring is necessary to separate variables in some expressions, it is not necessary in all cases, so one should not count on the returned factors being factored.

#### Examples

```
>>> from sympy.abc import x, y, z, alpha
>>> from sympy import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y

>>> e = 2*x**2*z*sin(y)+2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)
>>> separatevars(e, symbols=(x, y), dict=True)
{'coeff': 2*z, x: x**2, y: sin(y) + 1}
>>> separatevars(e, [x, y, alpha], dict=True)
{'coeff': 2*z, alpha: 1, x: x**2, y: sin(y) + 1}
```

If the expression is not really separable, or is only partially separable, separatevars will do the best it can to separate it by using factoring.

```
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then expr is returned unchanged or (if dict=True) then None is returned.

```
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) == None
True
```

### 3.21.4 nthroot

`sympy.simplify.simplify.nthroot(expr, n, max_len=4, prec=15)`  
compute a real nth-root of a sum of surds

**Parameters** `expr` : sum of surds

`n` : integer

`max_len` : maximum number of surds passed as constants to `nsimplify`

#### Examples

```
>>> from sympy.simplify.simplify import nthroot
>>> from sympy import Rational, sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

### 3.21.5 rad\_rationalize

`sympy.simplify.simplify.rad_rationalize(num, den)`

Rationalize num/den by removing square roots in the denominator; num and den are sum of terms whose squares are rationals

## Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.simplify import rad_rationalize
>>> rad_rationalize(sqrt(3), 1 + sqrt(2)/3)
(-sqrt(3) + sqrt(6)/3, -7/9)
```

## 3.21.6 radsimp

`sympy.simplify.simplify.radsimp(expr, symbolic=True, max_terms=4)`

Rationalize the denominator by removing square roots.

Note: the expression returned from `radsimp` must be used with caution since if the denominator contains symbols, it will be possible to make substitutions that violate the assumptions of the simplification process: that for a denominator matching  $a + b\sqrt{c}$ ,  $a \neq -b\sqrt{c}$ . (If there are no symbols, this assumption is made valid by collecting terms of  $\sqrt{c}$  so the match variable `a` does not contain  $\sqrt{c}$ .) If you do not want the simplification to occur for symbolic denominators, set `symbolic` to `False`.

If there are more than `max_terms` radical terms then the expression is returned unchanged.

## Examples

```
>>> from sympy import radsimp, sqrt, Symbol, denom, pprint, I
>>> from sympy import factor_terms, fraction, signsimp
>>> from sympy.simplify.simplify import collect_sqrt
>>> from sympy.abc import a, b, c

>>> radsimp(1/(I + 1))
(1 - I)/2
>>> radsimp(1/(2 + sqrt(2)))
(-sqrt(2) + 2)/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5)); pprint(ans)

      ---      ---      ---      ---
      \ / 5 *a + \ / 5 *b - \ / 2 *x - \ / 2 *y
-----
      2      2      2      2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y

>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True))

      ---      ---
      \ / 5 *(a + b) - \ / 2 *(x + y)
-----
```

```
2      2      2      2
5*a  + 10*a*b + 5*b  - 2*x  - 4*x*y - 2*y
```

If radicals in the denominator cannot be removed or there is no denominator, the original expression will be returned.

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

Results with symbols will not always be valid for all substitutions:

```
>>> eq = 1/(a + b*sqrt(c))
>>> eq.subs(a, b*sqrt(c))
1/(2*b*sqrt(c))
>>> radsimp(eq).subs(a, b*sqrt(c))
nan
```

If symbolic=False, symbolic denominators will not be transformed (but numeric denominators will still be processed):

```
>>> radsimp(eq, symbolic=False)
1/(a + b*sqrt(c))
```

### 3.21.7 ratsimp

`sympy.simplify.simplify.ratsimp(expr)`

Put an expression over a common denominator, cancel and reduce.

#### Examples

```
>>> from sympy import ratsimp
>>> from sympy.abc import x, y
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

### 3.21.8 fraction

`sympy.simplify.simplify.fraction(expr, exact=False)`

Returns a pair with expression's numerator and denominator. If the given expression is not a fraction then this function will return the tuple (expr, 1).

This function will not make any attempt to simplify nested fractions or to do any term rewriting at all.

If only one of the numerator/denominator pair is needed then use numer(expr) or denom(expr) functions respectively.

```
>>> from sympy import fraction, Rational, Symbol
>>> from sympy.abc import x, y

>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)
```

```
>>> fraction(1/y**2)
(1, y**2)

>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and ‘exact’ flag is unset, then structure this exponent’s structure will be analyzed and pretty fraction will be returned:

```
>>> from sympy import exp
>>> fraction(2*x**(-y))
(2, x**y)

>>> fraction(exp(-x))
(1, exp(x))

>>> fraction(exp(-x), exact=True)
(exp(-x), 1)
```

### 3.21.9 trigsimp

`sympy.simplify.simplify.trigsimp(expr, **opts)`  
reduces expression by using known trig identities

#### Notes

method: - Determine the method to use. Valid choices are ‘matching’ (default), ‘groebner’, ‘combined’, and ‘fu’. If ‘matching’, simplify the expression recursively by targeting common patterns. If ‘groebner’, apply an experimental groebner basis algorithm. In this case further options are forwarded to `trigsimp_groebner`, please refer to its docstring. If ‘combined’, first run the groebner basis algorithm with small default parameters, then run the ‘matching’ algorithm. ‘fu’ runs the collection of trigonometric transformations described by Fu, et al. (see the *fu* docstring).

#### Examples

```
>>> from sympy import trigsimp, sin, cos, log
>>> from sympy.abc import x, y
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```

Simplification occurs wherever trigonometric functions are located.

```
>>> trigsimp(log(e))
log(2)
```

Using *method = "groebner"* (or *"combined"*) might lead to greater simplification.

The old trigsimp routine can be accessed as with method ‘old’.

```
>>> from sympy import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

### 3.21.10 besselsimp

`sympy.simplify.simplify.besselsimp(expr)`  
Simplify bessel-type functions.

This routine tries to simplify bessel-type functions. Currently it only works on the Bessel J and I functions, however. It works by looking at all such functions in turn, and eliminating factors of “I” and “-1” (actually their polar equivalents) in front of the argument. Then, functions of half-integer order are rewritten using trigonometric functions and functions of integer order ( $> 1$ ) are rewritten using functions of low order. Finally, if the expression was changed, compute factorization of the result with factor().

```
>>> from sympy import besselj, besseli, besselsimp, polar_lift, I, S
>>> from sympy.abc import z, nu
>>> besselsimp(besselj(nu, z*polar_lift(-1)))
exp(I*pi*nu)*besselj(nu, z)
>>> besselsimp(besseli(nu, z*polar_lift(-I)))
exp(-I*pi*nu/2)*besselj(nu, z)
>>> besselsimp(besseli(S(-1)/2, z))
sqrt(2)*cosh(z)/(sqrt(pi)*sqrt(z))
>>> besselsimp(z*besseli(0, z) + z*(besseli(2, z))/2 + besseli(1, z))
3*z*besseli(0, z)/2
```

### 3.21.11 powsimp

`sympy.simplify.simplify.powsimp(expr, deep=False, combine='all', force=False, measure=<function count_ops at 0x7fb9aef6e60>)`  
reduces expression by combining powers with similar bases and exponents.

#### Notes

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.  $\sqrt{x}*\sqrt{y} \rightarrow \sqrt{x*y}$  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, Mul
>>> 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))
```

### 3.21.12 combsimp

```
sympy.simplify.simplify.combsimp(expr)
    Simplify combinatorial expressions.
```

This function takes as input an expression containing factorials, binomials, Pochhammer symbol and other “combinatorial” functions, and tries to minimize the number of those functions and reduce the size of their arguments. The result is given in terms of binomials and factorials.

The algorithm works by rewriting all combinatorial functions as expressions involving rising factorials (Pochhammer symbols) and applies recurrence relations and other transformations applicable to rising factorials, to reduce their arguments, possibly letting the resulting rising factorial to cancel. Rising factorials with the second argument being an integer are expanded into polynomial forms and finally all other rising factorial are rewritten in terms more familiar functions. If the initial expression contained any combinatorial functions, the result is expressed using binomial coefficients and gamma functions. If the initial expression consisted of gamma functions alone, the result is expressed in terms of gamma functions.

If the result is expressed using gamma functions, the following three additional steps are performed:

1. Reduce the number of gammas by applying the reflection theorem  $\text{gamma}(x)*\text{gamma}(1-x) == \pi/\sin(\pi*x)$ .
2. Reduce the number of gammas by applying the multiplication theorem  $\text{gamma}(x)*\text{gamma}(x+1/n)*...*\text{gamma}(x+(n-1)/n) == C*\text{gamma}(n*x)$ .
3. Reduce the number of prefactors by absorbing them into gammas, where possible.

All transformation rules can be found (or was derived from) here:

1. <http://functions.wolfram.com/GammaBetaErf/Pochhammer/17/01/02/>
2. <http://functions.wolfram.com/GammaBetaErf/Pochhammer/27/01/0005/>

#### Examples

```
>>> from sympy.simplify import combsimp
>>> from sympy import factorial, binomial
>>> from sympy.abc import n, k

>>> combsimp(factorial(n)/factorial(n - 3))
n*(n - 2)*(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
(n + 1)/(k + 1)
```

### 3.21.13 hypersimp

```
sympy.simplify.simplify.hypersimp(f, k)
```

Given combinatorial term  $f(k)$  simplify its consecutive term ratio i.e.  $f(k+1)/f(k)$ . The input term can be composed of functions and integer sequences which have equivalent representation in terms of gamma special function.

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.

3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If  $f(k)$  is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

1. W. Koepf, Algorithms for m-fold Hypergeometric Summation, Journal of Symbolic Computation (1995) 20, 399-417

### 3.21.14 hypersimilar

`sympy.simplify.simplify.hypersimilar(f, g, k)`

Returns True if 'f' and 'g' are hyper-similar.

Similarity in hypergeometric sense means that a quotient of  $f(k)$  and  $g(k)$  is a rational function in  $k$ . This procedure is useful in solving recurrence relations.

For more information see hypersimp().

### 3.21.15 nsimplify

`sympy.simplify.simplify.nsimplify(expr, constants=[], tolerance=None, full=False, rational=None)`

Find a simple representation for a number or, if there are free symbols or if rational=True, then replace Floats with their Rational equivalents. If no change is made and rational is not False then Floats will at least be converted to Rationals.

For numerical expressions, a simple formula that numerically matches the given numerical expression is sought (and the input should be possible to evalf to a precision of at least 30 digits).

Optionally, a list of (rationally independent) constants to include in the formula may be given.

A lower tolerance may be set to find less exact matches. If no tolerance is given then the least precise value will set the tolerance (e.g. Floats default to 15 digits of precision, so would be tolerance=10\*\*-15).

With full=True, a more extensive search is performed (this is useful to find simpler numbers when the tolerance is set low).

See also:

[sympy.core.function.nfloat](#) (page 149)

#### Examples

```
>>> from sympy import nsimplify, sqrt, GoldenRatio, exp, I, exp, pi
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nsimplify(I**I, [pi])
exp(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

### 3.21.16 collect\_sqrt

`sympy.simplify.simplify.collect_sqrt(expr, evaluate=None)`

Return `expr` with terms having common square roots collected together. If `evaluate` is `False` a count indicating the number of `sqrt`-containing terms will be returned and, if non-zero, the terms of the `Add` will be returned, else the expression itself will be returned as a single term. If `evaluate` is `True`, the expression with any collected terms will be returned.

Note: since  $I = \sqrt{-1}$ , it is collected, too.

See also:

`collect` (page 918), `collect_const` (page 929), `rcollect` (page 920)

#### Examples

```
>>> from sympy import sqrt
>>> from sympy.simplify.simplify import collect_sqrt
>>> from sympy.abc import a, b

>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]
>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)
```

If `evaluate` is `False` then the arguments will be sorted and returned as a list and a count of the number of `sqrt`-containing terms will be returned:

```
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)
>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)
>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

### 3.21.17 collect\_const

`sympy.simplify.simplify.collect_const(expr, *vars, **kwargs)`

A non-greedy collection of terms with similar number coefficients in an `Add` `expr`. If `vars` is given then only those constants will be targeted. Although any `Number` can also be targeted, if this is not desired set `Numbers=False` and no `Float` or `Rational` will be collected.

See also:

`collect` (page 918), `collect_sqrt` (page 929), `rcollect` (page 920)

#### Examples

```
>>> from sympy import sqrt
>>> from sympy.abc import a, s, x, y, z
>>> from sympy.simplify.simplify import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
```

```
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```

### 3.21.18 posify

`sympy.simplify.simplify.posify(eq)`

Return `eq` (with generic symbols made positive) and a restore dictionary.

Any symbol that has `positive=None` will be replaced with a positive dummy symbol having the same name. This replacement will allow more symbolic processing of expressions, especially those involving powers and logarithms.

A dictionary that can be sent to `subs` to restore `eq` to its original symbols is also returned.

```
>>> from sympy import posify, Symbol, log
>>> from sympy.abc import x
>>> posify(x + Symbol('p', positive=True) + Symbol('n', negative=True))
(_x + n + p, {_x: x})

>> log(1/x).expand() # should be log(1/x) but it comes back as -log(x) log(1/x)

>>> log(posify(1/x)[0]).expand() # take [0] and ignore replacements
-log(_x)
>>> eq, rep = posify(1/x)
>>> log(eq).expand().subs(rep)
-log(x)
>>> posify([x, 1 + x])
(_x, _x + 1, {_x: x})
```

### 3.21.19 powdenest

`sympy.simplify.simplify.powdenest(eq, force=False, polar=False)`

Collect exponents on powers as assumptions allow.

Given `(bb**be)**e`, this can be simplified as follows:

- if `bb` is positive, or
- `e` is an integer, or
- $|be| < 1$  then this simplifies to `bb**(be*e)`

Given a product of powers raised to a power,  $(bb1^{be1} * bb2^{be2} \dots)^{be}$ , simplification can be done as follows:

- if  $e$  is positive, the gcd of all  $bei$  can be joined with  $e$ ;
- all non-negative  $bb$  can be separated from those that are negative and their gcd can be joined with  $e$ ; autosimplification already handles this separation.
- integer factors from powers that have integers in the denominator of the exponent can be removed from any term and the gcd of such integers can be joined with  $e$

Setting `force` to True will make symbols that are not explicitly negative behave as though they are positive, resulting in more denesting.

Setting `polar` to True will do simplifications on the Riemann surface of the logarithm, also resulting in more denestings.

When there are sums of logs in `exp()` then a product of powers may be obtained e.g. `exp(3*(log(a) + 2*log(b))) -> a**3*b**6.`

## Examples

```
>>> from sympy.abc import a, b, x, y, z
>>> from sympy import Symbol, exp, log, sqrt, symbols, powdenest

>>> powdenest((x**(2*a/3))**(3*x))
(x**((2*a)/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**((3*x)
```

Assumptions may prevent expansion:

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

>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i,j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**((x*(i + j))
```

But `exp()` will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the `exp` to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**((3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest((x**(2*i))**(3*y))**x
```

```
((x**2*i)**3*y))**x
>>> powdenest((x**2*i)**3*y))**x, force=True)
x**6*i*x*y

>>> powdenest(((x**2*a/3)**3*y/i))**x
((x**2*a/3)**3*y/i)**x
>>> powdenest((x**2*i)*y**4*i)**z, force=True)
(x*y**2)**(2*i*z)

>>> n = Symbol('n', negative=True)

>>> powdenest((x**i)**y, force=True)
x**i*y
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

### 3.21.20 logcombine

`sympy.simplify.simplify.logcombine(expr, force=False)`

Takes logarithms and combines them using the following rules:

- $\log(x) + \log(y) == \log(x*y)$  if both are not negative
- $a*\log(x) == \log(x^a)$  if  $x$  is positive and  $a$  is real

If `force` is True then the assumptions above will be assumed to hold if there is no assumption already in place on a quantity. For example, if `a` is imaginary or the argument negative, force will not perform a combination but if `a` is a symbol with no assumptions the change will take place.

See also:

`posify` ([page 930](#)) replace all symbols with symbols having positive assumptions

#### Examples

```
>>> from sympy import Symbol, symbols, log, logcombine, I
>>> from sympy.abc import a, x, y, z
>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)
>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)
>>> x,y,z = symbols('x,y,z', positive=True)
>>> a = Symbol('a', extended_real=True)
>>> logcombine(a*log(x) + log(y) - log(z))
log(x**a*y/z)
```

The transformation is limited to factors and/or terms that contain logs, so the result depends on the initial state of expansion:

```
>>> eq = (2 + 3*I)*log(x)
>>> logcombine(eq, force=True) == eq
True
>>> logcombine(eq.expand(), force=True)
log(x**2) + I*log(x**3)
```

### 3.21.21 Square Root Denest

#### sqrtdenest

`sympy.simplify.sqrtdenest.sqrtdenest(expr, max_iter=3)`

Denests sqrt in an expression that contain other square roots if possible, otherwise returns the expr unchanged. This is based on the algorithms of [1].

See also:

`sympy.solvers.solvers.unrad` (page 1055)

#### References

[1] <http://researcher.watson.ibm.com/researcher/files/us-fagin/symb85.pdf>

[2] D. J. Jeffrey and A. D. Rich, ‘Simplifying Square Roots of Square Roots by Denesting’ (available at <http://www.cybertester.com/data/denest.pdf>)

#### Examples

```
>>> from sympy.simplify.sqrtdenest import sqrtdenest
>>> from sympy import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

### 3.21.22 Common Subexpression Elimination

#### cse

`sympy.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical')`

Perform common subexpression elimination on an expression.

**Parameters** `exprs` : list of sympy expressions, or a single sympy expression

The expressions to reduce.

`symbols` : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out. The `numbered_symbols` generator is useful. The default is a stream of symbols of the form “x0”, “x1”, etc. This must be an infinite iterator.

`optimizations` : list of (callable, callable) pairs

The (preprocessor, postprocessor) pairs of external optimization functions. Optionally ‘basic’ can be passed for a set of predefined basic optimizations. Such ‘basic’ optimizations were used by default in old implementation, however they can be really slow on larger expressions. Now, no pre or post optimizations are made by default.

`postprocess` : a function which accepts the two return values of cse and

returns the desired form of output from cse, e.g. if you want the replacements reversed the function might be the following lambda: `lambda r, e: return reversed(r), e`

**order** : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. If set to ‘canonical’, arguments will be canonically ordered. If set to ‘none’, ordering will be faster but dependent on expressions hashes, thus machine dependent and variable. For large expressions where speed is a concern, use the setting `order='none'`.

**Returns replacements** : list of (Symbol, expression) pairs

All of the common subexpressions that were replaced. Subexpressions earlier in this list might show up in subexpressions later in this list.

**reduced\_exprs** : list of sympy expressions

The reduced expressions with all of the replacements above.

### Examples

```
>>> from sympy import cse, SparseMatrix
>>> from sympy.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently,  $y + z$  will not get substituted if  $-y - z$  is used.

```
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

### opt\_cse

`sympy.simplify.cse_main.opt_cse(exprs, order='canonical')`

Find optimization opportunities in Adds, Muls, Pows and negative coefficient Muls

**Parameters exprs** : list of sympy expressions

The expressions to optimize.

**order** : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

**Returns opt\_subs** : dictionary of expression substitutions

The expression substitutions which can be useful to optimize CSE.

## Examples

```
>>> from sympy.simplify.cse_main import opt_cse
>>> from sympy.abc import x
>>> opt_subs = opt_cse([x**-2])
>>> print(opt_subs)
{x**(-2): 1/(x**2)}
```

### tree\_cse

`sympy.simplify.cse_main.tree_cse(exprs, symbols, opt_subs=None, order='canonical')`

Perform raw CSE on expression tree, taking opt\_subs into account.

**Parameters** `exprs` : list of sympy expressions

The expressions to reduce.

`symbols` : infinite iterator yielding unique Symbols

The symbols used to label the common subexpressions which are pulled out.

`opt_subs` : dictionary of expression substitutions

The expressions to be substituted before any CSE action is performed.

`order` : string, ‘none’ or ‘canonical’

The order by which Mul and Add arguments are processed. For large expressions where speed is a concern, use the setting `order='none'`.

## 3.21.23 Hypergeometric Function Expansion

### hyperexpand

`sympy.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default')`

Expand hypergeometric functions. If `allow_hyper` is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

## Examples

```
>>> from sympy.simplify.hyperexpand import hyperexpand
>>> from sympy.functions import hyper
>>> from sympy.abc import z
>>> hyperexpand(hyper([], [], z))
exp(z)
```

Non-hyperegeometric parts of the expression and hypergeometric expressions that are not recognised are left unchanged:

```
>>> hyperexpand(1 + hyper([1, 1, 1], [], z))
hyper((1, 1, 1), (), z) + 1
```

### 3.21.24 Traversal Tools

**use**

```
sympy.simplify.traversaltools.use(expr, func, level=0, args=(), kwargs={})  
    Use func to transform expr at the given level.
```

**Examples**

```
>>> from sympy import use, expand  
>>> from sympy.abc import x, y  
  
>>> f = (x + y)**2*x + 1  
  
>>> use(f, expand, level=2)  
x*(x**2 + 2*x*y + y**2) + 1  
>>> expand(f)  
x**3 + 2*x**2*y + x*y**2 + 1
```

### 3.21.25 EPath Tools

**EPath class**

```
class sympy.simplify.epathtools.EPath  
    Manipulate expressions using paths.
```

EPath grammar in EBNF notation:

```
literal   ::= /[A-Za-z_][A-Za-z_0-9]*/  
number   ::= /-?\d+/  
type     ::= literal  
attribute ::= literal "?"  
all      ::= "*"  
slice    ::= "[" number? ":" number? ":" number?")? "]"  
range   ::= all | slice  
query    ::= (type | attribute) ("|" (type | attribute))*  
selector ::= range | query range?  
path     ::= "/" selector ("/" selector)*
```

See the docstring of the epath() function.

```
apply(expr, func, args=None, kwargs=None)  
    Modify parts of an expression selected by a path.
```

**Examples**

```
>>> from sympy.simplify.epathtools import EPath  
>>> from sympy import sin, cos, E  
>>> from sympy.abc import x, y, z, t  
  
>>> path = EPath("/*[0]/Symbol")  
>>> expr = [(x, 1), 2, ((3, y), z)]
```

```
>>> path.apply(expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = EPath("/*/*/*Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

### select(expr)

Retrieve parts of an expression selected by a path.

#### Examples

```
>>> from sympy.simplify.epathtools import EPath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [(x, 1), (3, y), z]
```

```
>>> path.select(expr)
[x, y]
```

```
>>> path = EPath("/*/*/*Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.select(expr)
[x, x, y]
```

## epath

`sympy.simplify.epathtools.epath(path, expr=None, func=None, args=None, kwargs=None)`

Manipulate parts of an expression selected by a path.

This function allows to manipulate large nested expressions in single line of code, utilizing techniques to those applied in XML processing standards (e.g. XPath).

If `func` is `None`, `epath()` (page 937) retrieves elements selected by the `path`. Otherwise it applies `func` to each matching element.

Note that it is more efficient to create an `EPath` object and use the `select` and `apply` methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- select all:** /\* Equivalent of for arg in args::
- select slice:** /[0] or /[1:5] or /[1:5:2] Supports standard Python's slice syntax.
- select by type:** /list or /list|tuple Emulates `isinstance`.
- select by attribute:** /\_\_iter\_\_? Emulates `hasattr`.

**Parameters** `path` : str | `EPath`

A path as a string or a compiled EPath.

**expr** : Basic | iterable

An expression or a container of expressions.

**func** : callable (optional)

A callable that will be applied to matching parts.

**args** : tuple (optional)

Additional positional arguments to **func**.

**kwargs** : dict (optional)

Additional keyword arguments to **func**.

## Examples

```
>>> from sympy.simplify.epathtools import epath
>>> from sympy import sin, cos, E
>>> from sympy.abc import x, y, z, t

>>> path = "/*/[0]/Symbol"
>>> expr = [((x, 1), 2), ((3, y), z)]

>>> epath(path, expr)
[x, y]
>>> epath(path, expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]

>>> path = "/*/*/Symbol"
>>> expr = t + sin(x + 1) + cos(x + y + E)

>>> epath(path, expr)
[x, x, y]
>>> epath(path, expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

## 3.22 Details on the Hypergeometric Function Expansion Module

This page describes how the function `hyperexpand()` (page 935) and related code work. For usage, see the documentation of the `sympify` module.

### 3.22.1 Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [\[Roach1996\]](#) (page 1246) and [\[Roach1997\]](#) (page 1247).

Recall that the hypergeometric function is (initially) defined as

$${}_pF_q \left( \begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!}.$$

It turns out that there are certain differential operators that can change the  $a_p$  and  $p_q$  parameters by integers. If a sequence of such operators is known that converts the set of indices  $a_r^0$  and  $b_s^0$  into  $a_p$  and  $b_q$ , then we shall say the pair  $a_p, b_q$  is reachable from  $a_r^0, b_s^0$ . Our general strategy is thus as follows: given a set  $a_p, b_q$  of parameters, try to look up an origin  $a_r^0, b_s^0$  for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

## Notation

In the following, the symbol  $a$  will always denote a numerator parameter and the symbol  $b$  will always denote a denominator parameter. The subscripts  $p, q, r, s$  denote vectors of that length, so e.g.  $a_p$  denotes a vector of  $p$  numerator parameters. The subscripts  $i$  and  $j$  denote “running indices”, so they should usually be used in conjunction with a “for all  $i$ ”. E.g.  $a_i < 4$  for all  $i$ . Uppercase subscripts  $I$  and  $J$  denote a chosen, fixed index. So for example  $a_I > 0$  is true if the inequality holds for the one index  $I$  we are currently interested in.

## Incrementing and decrementing indices

Suppose  $a_i \neq 0$ . Set  $A(a_i) = \frac{z}{a_i} \frac{d}{dz} + 1$ . It is then easy to show that  $A(a_i)_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_p F_q \left( \begin{smallmatrix} a_p + e_i \\ b_q \end{smallmatrix} \middle| z \right)$ , where  $e_i$  is the  $i$ -th unit vector. Similarly for  $b_j \neq 1$  we set  $B(b_j) = \frac{z}{b_j - 1} \frac{d}{dz} + 1$  and find  $B(b_j)_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q - e_i \end{smallmatrix} \middle| z \right)$ . Thus we can increment upper and decrement lower indices at will, as long as we don’t go through zero. The  $A(a_i)$  and  $B(b_j)$  are called shift operators.

It is also easy to show that  $\frac{d}{dz} {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \frac{a_1 \dots a_p}{b_1 \dots b_q} {}_p F_q \left( \begin{smallmatrix} a_p + 1 \\ b_q + 1 \end{smallmatrix} \middle| z \right)$ , where  $a_p + 1$  is the vector  $a_1 + 1, a_2 + 1, \dots$  and similarly for  $b_q + 1$ . Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation:  $\left[ \frac{d}{dz} \prod_{j=1}^q B(b_j) - \frac{a_1 \dots a_p}{(b_1 - 1) \dots (b_q - 1)} \prod_{i=1}^p A(a_i) \right] {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$ . This holds if all shift operators are defined, i.e. if no  $a_i = 0$  and no  $b_j = 1$ . Clearing denominators and multiplying through by  $z$  we arrive at the following equation:  $\left[ z \frac{d}{dz} \prod_{j=1}^q (z \frac{d}{dz} + b_j - 1) - z \prod_{i=1}^p (z \frac{d}{dz} + a_i) \right] {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$ . Even though our derivation does not show it, it can be checked that this equation holds whenever the  ${}_p F_q$  is defined.

Notice that, under suitable conditions on  $a_I, b_J$ , each of the operators  $A(a_I)$ ,  $B(b_J)$  and  $z \frac{d}{dz}$  can be expressed in terms of  $A(a_I)$  or  $B(b_J)$ . Our next aim is to write the Hypergeometric differential equation as follows:  $[X A(a_I) - r] {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = 0$ , for some operator  $X$  and some constant  $r$  to be determined. If  $r \neq 0$ , then we can write this as  $\frac{-1}{r} X {}_p F_q \left( \begin{smallmatrix} a_p + e_I \\ b_q \end{smallmatrix} \middle| z \right) = {}_p F_q \left( \begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right)$ , and so  $\frac{-1}{r} X$  undoes the shifting of  $A(a_I)$ , whence it will be called an inverse-shift operator.

Now  $A(a_I)$  exists if  $a_I \neq 0$ , and then  $z \frac{d}{dz} = a_I A(a_I) - a_I$ . Observe also that all the operators  $A(a_i)$ ,  $B(b_j)$  and  $z \frac{d}{dz}$  commute. We have  $\prod_{i=1}^p (z \frac{d}{dz} + a_i) = (\prod_{i=1, i \neq I}^p (z \frac{d}{dz} + a_i)) a_I A(a_I)$ , so this gives us the first half of  $X$ . The other half does not have such a nice expression. We find  $z \frac{d}{dz} \prod_{j=1}^q (z \frac{d}{dz} + b_j - 1) = (a_I A(a_I) - a_I) \prod_{j=1}^q (a_I A(a_I) - a_I + b_j - 1)$ . Since the first half had no constant term, we infer  $r = -a_I \prod_{j=1}^q (b_j - 1 - a_I)$ .

This tells us under which conditions we can “un-shift”  $A(a_I)$ , namely when  $a_I \neq 0$  and  $r \neq 0$ . Substituting  $a_I - 1$  for  $a_I$  then tells us under what conditions we can decrement the index  $a_I$ . Doing a similar analysis for  $B(b_J)$ , we arrive at the following rules:

- An index  $a_I$  can be decremented if  $a_I \neq 1$  and  $a_I \neq b_j$  for all  $b_j$ .
- An index  $b_J$  can be incremented if  $b_J \neq -1$  and  $b_J \neq a_i$  for all  $a_i$ .

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

### Reduction of Order

Notice that, quite trivially, if  $a_I = b_J$ , we have  ${}_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z\right) = {}_{p-1}F_{q-1}\left(\begin{matrix} a_p^* \\ b_q^* \end{matrix} \middle| z\right)$ , where  $a_p^*$  means  $a_p$  with  $a_I$  omitted, and similarly for  $b_q^*$ . We call this reduction of order.

In fact, we can do even better. If  $a_I - b_J \in \mathbb{Z}_{>0}$ , then it is easy to see that  $\frac{(a_I)_n}{(b_J)_n}$  is actually a polynomial in  $n$ . It is also easy to see that  $(z \frac{d}{dz})^k z^n = n^k z^n$ . Combining these two remarks we find:

If  $a_I - b_J \in \mathbb{Z}_{>0}$ , then there exists a polynomial  $p(n) = p_0 + p_1 n + \dots$  (of degree  $a_I - b_J$ ) such that  $\frac{(a_I)_n}{(b_J)_n} = p(n)$  and  ${}_pF_q\left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z\right) = \left(p_0 + p_1 z \frac{d}{dz} + p_2 (z \frac{d}{dz})^2 + \dots\right) {}_{p-1}F_{q-1}\left(\begin{matrix} a_p^* \\ b_q^* \end{matrix} \middle| z\right)$ .

Thus any set of parameters  $a_p, b_q$  is reachable from a set of parameters  $c_r, d_s$  where  $c_i - d_j \in \mathbb{Z}$  implies  $c_i < d_j$ . Such a set of parameters  $c_r, d_s$  is called suitable. Our database of known formulae should only contain suitable origins. The reasons are twofold: firstly, working from suitable origins is easier, and secondly, a formula for a non-suitable origin can be deduced from a lower order formula, and we should put this one into the database instead.

### Moving Around in the Parameter Space

It remains to investigate the following question: suppose  $a_p, b_q$  and  $a_p^0, b_q^0$  are both suitable, and also  $a_i - a_i^0 \in \mathbb{Z}, b_j - b_j^0 \in \mathbb{Z}$ . When is  $a_p, b_q$  reachable from  $a_p^0, b_q^0$ ? It is clear that we can treat all parameters independently that are incongruent mod 1. So assume that  $a_i$  and  $b_j$  are congruent to  $r$  mod 1, for all  $i$  and  $j$ . The same then follows for  $a_i^0$  and  $b_j^0$ .

If  $r \neq 0$ , then any such  $a_p, b_q$  is reachable from any  $a_p^0, b_q^0$ . To see this notice that there exist constants  $c, c^0$ , congruent mod 1, such that  $a_i < c < b_j$  for all  $i$  and  $j$ , and similarly  $a_i^0 < c^0 < b_j^0$ . If  $n = c - c^0 > 0$  then we first inverse-shift all the  $b_j^0$   $n$  times up, and then similarly shift up all the  $a_i^0$   $n$  times. If  $n < 0$  then we first inverse-shift down the  $a_i^0$  and then shift down the  $b_j^0$ . This reduces to the case  $c = c^0$ . But evidently we can now shift or inverse-shift around the  $a_i^0$  arbitrarily so long as we keep them less than  $c$ , and similarly for the  $b_j^0$  so long as we keep them bigger than  $c$ . Thus  $a_p, b_q$  is reachable from  $a_p^0, b_q^0$ .

If  $r = 0$  then the problem is slightly more involved. WLOG no parameter is zero. We now have one additional complication: no parameter can ever move through zero. Hence  $a_p, b_q$  is reachable from  $a_p^0, b_q^0$  if and only if the number of  $a_i < 0$  equals the number of  $a_i^0 < 0$ , and similarly for the  $b_i$  and  $b_i^0$ . But in a suitable set of parameters, all  $b_j > 0$ ! This is because the Hypergeometric function is undefined if one of the  $b_j$  is a non-positive integer and all  $a_i$  are smaller than the  $b_j$ . Hence the number of  $b_j \leq 0$  is always zero.

We can thus associate to every suitable set of parameters  $a_p, b_q$ , where no  $a_i = 0$ , the following invariants:

- For every  $r \in [0, 1)$  the number  $\alpha_r$  of parameters  $a_i \equiv r \pmod{1}$ , and similarly the number  $\beta_r$  of parameters  $b_i \equiv r \pmod{1}$ .
- The number  $\gamma$  of integers  $a_i$  with  $a_i < 0$ .

The above reasoning shows that  $a_p, b_q$  is reachable from  $a_p^0, b_q^0$  if and only if the invariants  $\alpha_r, \beta_r, \gamma$  all agree. Thus in particular “being reachable from” is a symmetric relation on suitable parameters without zeros.

### Applying the Operators

If all goes well then for a given set of parameters we find an origin in our database for which we have a nice formula. We now have to apply (potentially) many differential operators to it. If we do this blindly then

the result will be very messy. This is because with Hypergeometric type functions, the derivative is usually expressed as a sum of two contiguous functions. Hence if we compute  $N$  derivatives, then the answer will involve  $2N$  contiguous functions! This is clearly undesirable. In fact we know from the Hypergeometric differential equation that we need at most  $\max(p, q + 1)$  contiguous functions to express all derivatives.

Hence instead of differentiating blindly, we will work with a  $\mathbb{C}(z)$ -module basis: for an origin  $a_r^0, b_s^0$  we either store (for particularly pretty answers) or compute a set of  $N$  functions (typically  $N = \max(r, s + 1)$ ) with the property that the derivative of any of them is a  $\mathbb{C}(z)$ -linear combination of them. In formulae, we store a vector  $B$  of  $N$  functions, a matrix  $M$  and a vector  $C$  (the latter two with entries in  $\mathbb{C}(z)$ ), with the following properties:

- ${}_rF_s \left( \begin{matrix} a_r^0 \\ b_s^0 \end{matrix} \middle| z \right) = CB$

- $z \frac{d}{dz} B = MB.$

Then we can compute as many derivatives as we want and we will always end up with  $\mathbb{C}(z)$ -linear combination of at most  $N$  special functions.

As hinted above,  $B$ ,  $M$  and  $C$  can either all be stored (for particularly pretty answers) or computed from a single  ${}_pF_q$  formula.

## Loose Ends

This describes the bulk of the hypergeometric function algorithm. There are a few further tricks, described in the `hyperexpand.py` source file. The extension to Meijer G-functions is also described there.

### 3.22.2 Meijer G-Functions of Finite Confluence

Slater's theorem essentially evaluates a  $G$ -function as a sum of residues. If all poles are simple, the resulting series can be recognised as hypergeometric series. Thus a  $G$ -function can be evaluated as a sum of Hypergeometric functions.

If the poles are not simple, the resulting series are not hypergeometric. This is known as the “confluent” or “logarithmic” case (the latter because the resulting series tend to contain logarithms). The answer depends in a complicated way on the multiplicities of various poles, and there is no accepted notation for representing it (as far as I know). However if there are only finitely many multiple poles, we can evaluate the  $G$  function as a sum of hypergeometric functions, plus finitely many extra terms. I could not find any good reference for this, which is why I work it out here.

Recall the general setup. We define

$$G(z) = \frac{1}{2\pi i} \int_L \frac{\prod_{j=1}^m \Gamma(b_j - s) \prod_{j=1}^n \Gamma(1 - a_j + s)}{\prod_{j=m+1}^q \Gamma(1 - b_j + s) \prod_{j=n+1}^p \Gamma(a_j - s)} z^s ds,$$

where  $L$  is a contour starting and ending at  $+\infty$ , enclosing all of the poles of  $\Gamma(b_j - s)$  for  $j = 1, \dots, n$  once in the negative direction, and no other poles. Also the integral is assumed absolutely convergent.

In what follows, for any complex numbers  $a, b$ , we write  $a \equiv b \pmod{1}$  if and only if there exists an integer  $k$  such that  $a - b = k$ . Thus there are double poles iff  $a_i \equiv a_j \pmod{1}$  for some  $i \neq j \leq n$ .

We now assume that whenever  $b_j \equiv a_i \pmod{1}$  for  $i \leq m, j > n$  then  $b_j < a_i$ . This means that no quotient of the relevant gamma functions is a polynomial, and can always be achieved by “reduction of order”. Fix a complex number  $c$  such that  $\{b_i | b_i \equiv c \pmod{1}, i \leq m\}$  is not empty. Enumerate this set as  $b, b + k_1, \dots, b + k_u$ , with  $k_i$  non-negative integers. Enumerate similarly  $\{a_j | a_j \equiv c \pmod{1}, j > n\}$  as  $b + l_1, \dots, b + l_v$ . Then  $l_i > k_j$  for all  $i, j$ . For finite confluence, we need to assume  $v \geq u$  for all such  $c$ .

Let  $c_1, \dots, c_w$  be distinct  $\pmod{1}$  and exhaust the congruence classes of the  $b_i$ . I claim

$$G(z) = -\sum_{j=1}^w (F_j(z) + R_j(z)),$$

where  $F_j(z)$  is a hypergeometric function and  $R_j(z)$  is a finite sum, both to be specified later. Indeed corresponding to every  $c_j$  there is a sequence of poles, at mostly finitely many of them multiple poles. This is where the  $j$ -th term comes from.

Hence fix again  $c$ , enumerate the relevant  $b_i$  as  $b, b+k_1, \dots, b+k_u$ . We will look at the  $a_j$  corresponding to  $a+l_1, \dots, a+l_u$ . The other  $a_i$  are not treated specially. The corresponding gamma functions have poles at (potentially)  $s = b+r$  for  $r = 0, 1, \dots$ . For  $r \geq l_u$ , pole of the integrand is simple. We thus set

$$R(z) = \sum_{r=0}^{l_u-1} res_{s=r+b}.$$

We finally need to investigate the other poles. Set  $r = l_u + t$ ,  $t \geq 0$ . A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where  $\delta_i = l_i - k_i$ .

Also

$$\begin{aligned} \Gamma(b_j - l_u - b - t) &= \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t}, \\ \Gamma(1 - a_j + l_u + b + t) &= \Gamma(1 - a_j + l_u + b) (1 - a_j + l_u + b)_t \end{aligned}$$

and

$$res_{s=b+l_u+t} \Gamma(b - s) = -\frac{(-1)^{l_u+t}}{(l_u + t)!} = -\frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$\begin{aligned} res_{s=b+l_u+t} &= -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^u \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^n \Gamma(1 - a_j + l_u + b) \prod_{j=1}^m \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^p \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^q \Gamma(1 - b_j + l_u + b)} \\ &\quad \times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^u \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^n (1 - a_j + l_u + b)_t \prod_{j=n+1}^p (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^m (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^q (1 - b_j + l_u + b)_t}, \end{aligned}$$

where the  $*$  means to omit the terms we treated specially.

We thus arrive at

$$F(z) = C \times {}_{p+1}F_q \left( \begin{matrix} 1, (1 + l_u - l_i), (1 + l_u + b - a_i)^* \\ 1 + l_u, (1 + l_u - k_i), (1 + l_u + b - b_i)^* \end{matrix} \middle| (-1)^{p-m-n} z \right),$$

where  $C$  designates the factor in the residue independent of  $t$ . (This result can also be written in slightly simpler form by converting all the  $l_u$  etc back to  $a_* - b_*$ , but doing so is going to require more notation still and is not helpful for computation.)

### 3.22.3 Extending The Hypergeometric Tables

Adding new formulae to the tables is straightforward. At the top of the file `sympy/simplify/hyperexpand.py`, there is a function called `add_formulae()` (page 947). Nested in

it are defined two helpers, `add(ap, bq, res)` and `addb(ap, bq, B, C, M)`, as well as dummies `a`, `b`, `c`, and `z`.

The first step in adding a new formula is by using `add(ap, bq, res)`. This declares `hyper(ap, bq, z) == res`. Here `ap` and `bq` may use the dummies `a`, `b`, and `c` as free symbols. For example the well-known formula  $\sum_0^{\infty} \frac{(-a)_n z^n}{n!} = (1-z)^a$  is declared by the following line: `add((-a, ), (), (1-z)**a)`.

From the information provided, the matrices  $B$ ,  $C$  and  $M$  will be computed, and the formula is now available when expanding hypergeometric functions. Next the test file `sympy/simplify/tests/test_hyperexpand.py` should be run, in particular the test `test_formulae`. This will test the newly added formula numerically. If it fails, there is (presumably) a typo in what was entered.

Since all newly-added formulae are probably relatively complicated, chances are that the automatically computed basis is rather suboptimal (there is no good way of testing this, other than observing very messy output). In this case the matrices  $B$ ,  $C$  and  $M$  should be computed by hand. Then the helper `addb` can be used to declare a hypergeometric formula with hand-computed basis.

## An example

Because this explanation so far might be very theoretical and difficult to understand, we walk through an explicit example now. We take the Fresnel function  $C(z)$  which obeys the following hypergeometric representation:

$$C(z) = z \cdot {}_1F_2 \left( \begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix} \middle| -\frac{\pi^2 z^4}{16} \right).$$

First we try to add this formula to the lookup table by using the (simpler) function `add(ap, bq, res)`. The first two arguments are simply the lists containing the parameter sets of  ${}_1F_2$ . The `res` argument is a little bit more complicated. We only know  $C(z)$  in terms of  ${}_1F_2(\dots | f(z))$  with  $f$  a function of  $z$ , in our case

$$f(z) = -\frac{\pi^2 z^4}{16}.$$

What we need is a formula where the hypergeometric function has only  $z$  as argument  ${}_1F_2(\dots | z)$ . We introduce the new complex symbol  $w$  and search for a function  $g(w)$  such that

$$f(g(w)) = w$$

holds. Then we can replace every  $z$  in  $C(z)$  by  $g(w)$ . In the case of our example the function  $g$  could look like

$$g(w) = \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}.$$

We get these functions mainly by guessing and testing the result. Hence we proceed by computing  $f(g(w))$  (and simplifying naively)

$$\begin{aligned} f(g(w)) &= -\frac{\pi^2 g(w)^4}{16} \\ &= -\frac{\pi^2 g \left( \frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}} \right)^4}{16} \\ &= -\frac{\pi^2 \frac{2^4}{\sqrt{\pi^4}} \exp\left(\frac{i\pi}{4}\right)^4 w^{\frac{1}{4} \cdot 4}}{16} \\ &= -\exp(i\pi) w \\ &= w \end{aligned}$$

and indeed get back  $w$ . (In case of branched functions we have to be aware of branch cuts. In that case we take  $w$  to be a positive real number and check the formula. If what we have found works for positive  $w$ , then just replace `exp()` (page 325) inside any branched function by `exp_polar()` (page 325) and what we get is right for *all*  $w$ .) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_1F_2 \left( \begin{matrix} \frac{1}{4}, \frac{5}{4} \\ \frac{1}{2} \end{matrix} \middle| w \right).$$

and trivially

$${}_1F_2 \left( \begin{matrix} \frac{1}{4}, \frac{5}{4} \\ \frac{1}{2} \end{matrix} \middle| w \right) = \frac{C(g(w))}{g(w)} = \frac{C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:

```
add([S(1)/4,
    [S(1)/2, S(5)/4],
    fresnelc(exp(pi*I/4)*root(z,4)*2/sqrt(pi)) / (exp(pi*I/4)*root(z,4)*2/sqrt(pi))
    )
```

Using this rule we will find that it works but the results are not really nice in terms of simplicity and number of special function instances included. We can obtain much better results by adding the formula to the lookup table in another way. For this we use the (more complicated) function `addb(ap, bq, B, C, M)`. The first two arguments are again the lists containing the parameter sets of  ${}_1F_2$ . The remaining three are the matrices mentioned earlier on this page.

We know that the  $n = \max(p, q + 1)$ -th derivative can be expressed as a linear combination of lower order derivatives. The matrix  $B$  contains the basis  $\{B_0, B_1, \dots\}$  and is of shape  $n \times 1$ . The best way to get  $B_i$  is to take the first  $n = \max(p, q + 1)$  derivatives of the expression for  ${}_pF_q$  and take out useful pieces. In our case we find that  $n = \max(1, 2 + 1) = 3$ . For computing the derivatives, we have to use the operator  $z \frac{d}{dz}$ . The first basis element  $B_0$  is set to the expression for  ${}_1F_2$  from above:

$$B_0 = \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}}$$

Next we compute  $z \frac{d}{dz} B_0$ . For this we can directly use SymPy!

```
>>> from sympy import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/
...     (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/(4*z) - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**1/4*exp(I*pi/4)/sqrt(pi))/(8*z**5/4))
>>> expand(_)
cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**1/4*exp(I*pi/4)/sqrt(pi))/(8*z**1/4)
```

Formatting this result nicely we obtain

$$B'_1 = -\frac{1}{4} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4} \cosh(2\sqrt{z})$$

Computing the second derivative we find

```
>>> from sympy import (Symbol, cosh, sqrt, pi, exp, I, fresnelc, root,
...                     diff, expand)
>>> z = Symbol("z")
```

```
>>> B1prime = cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*\
...      fresnelc(2*sqrt(z,4)*exp(I*pi/4)/sqrt(pi))/(8*sqrt(z,4))
>>> z * diff(B1prime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**1/4)*exp(I*pi/4)/sqrt(pi))
>>> expand(_)
sqrt(z)*sinh(2*sqrt(z))/4 - cosh(2*sqrt(z))/16 + sqrt(pi)*exp(-I*pi/4)*fresnelc(2*z**1/4)*exp(I*pi/4)/sqrt(pi))/(32*z)
```

which can be printed as

$$B'_2 = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh(2\sqrt{z}) + \frac{1}{4} \sinh(2\sqrt{z})\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose  $B_1$  and  $B_2$  as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right) \\ 2z^{\frac{1}{4}} \\ \cosh(2\sqrt{z}) \\ \sinh(2\sqrt{z})\sqrt{z} \end{pmatrix}$$

(This is in contrast to the basis  $B = (B_0, B'_1, B'_2)$  that would have been computed automatically if we used just `add(ap, bq, res)`.)

Because it must hold that  ${}_pF_q(\dots|z) = CB$  the entries of  $C$  are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the  $3 \times 3$  matrix  $M$  such that  $z \frac{d}{dz} B = MB$  holds. This is easy. We already computed the first part  $z \frac{d}{dz} B_0$  above. This gives us the first row of  $M$ . For the second row we have:

```
>>> from sympy import Symbol, cosh, sqrt, diff
>>> z = Symbol("z")
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> from sympy import Symbol, sinh, sqrt, expand, diff
>>> z = Symbol("z")
>>> B2 = sinh(2*sqrt(z))*sqrt(z)
>>> expand(z * diff(B2, z))
sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))
```

Now we have computed the entries of this matrix to be

$$M = \begin{pmatrix} -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}$$

Note that the entries of  $C$  and  $M$  should typically be rational functions in  $z$ , with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for `hyperexpand`.

### 3.22.4 Implemented Hypergeometric Formulae

A vital part of the algorithm is a relatively large table of hypergeometric function representations. The following automatically generated list contains all the representations implemented in SymPy (of course

many more are derived from them). These formulae are mostly taken from [Luke1969] (page 1247) and [Prudnikov1990] (page 1247). They are all tested numerically.

$${}_0F_0(|z|) = e^z$$

$${}_1F_0(a|z) = (-z+1)^{-a}$$

$${}_2F_1\left(\begin{matrix} a, a - \frac{1}{2} \\ 2a \end{matrix} \middle| z\right) = 2^{2a-1} (\sqrt{-z+1} + 1)^{-2a+1}$$

$${}_2F_1\left(\begin{matrix} 1, 1 \\ 2 \end{matrix} \middle| z\right) = -\frac{1}{z} \log(-z+1)$$

$${}_2F_1\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ \frac{3}{2} \end{matrix} \middle| z\right) = \frac{1}{\sqrt{z}} \operatorname{atanh}(\sqrt{z})$$

$${}_2F_1\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ \frac{3}{2} \end{matrix} \middle| z\right) = \frac{1}{\sqrt{z}} \operatorname{asin}(\sqrt{z})$$

$${}_2F_1\left(\begin{matrix} a, a + \frac{1}{2} \\ \frac{1}{2} \end{matrix} \middle| z\right) = \frac{1}{2} (\sqrt{z} + 1)^{-2a} + \frac{1}{2} (-\sqrt{z} + 1)^{-2a}$$

$${}_2F_1\left(\begin{matrix} a, -a \\ \frac{1}{2} \end{matrix} \middle| z\right) = \cos(2a \operatorname{asin}(\sqrt{z}))$$

$${}_2F_1\left(\begin{matrix} 1, 1 \\ \frac{3}{2} \end{matrix} \middle| z\right) = \frac{\operatorname{asin}(\sqrt{z})}{\sqrt{z}\sqrt{-z+1}}$$

$${}_2F_1\left(\begin{matrix} \frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix} \middle| z\right) = \frac{2K(z)}{\pi}$$

$${}_2F_1\left(\begin{matrix} -\frac{1}{2}, \frac{1}{2} \\ 1 \end{matrix} \middle| z\right) = \frac{2E(z)}{\pi}$$

$${}_3F_2\left(\begin{matrix} -\frac{1}{2}, 1, 1 \\ \frac{1}{2}, 2 \end{matrix} \middle| z\right) = -\frac{2\sqrt{z}}{3} \operatorname{atanh}(\sqrt{z}) + \frac{2}{3} - \frac{1}{3z} \log(-z+1)$$

$${}_3F_2\left(\begin{matrix} -\frac{1}{2}, 1, 1 \\ 2, 2 \end{matrix} \middle| z\right) = \left(\frac{4}{9} - \frac{16}{9z}\right) \sqrt{-z+1} + \frac{4}{3z} \log\left(\frac{1}{2}\sqrt{-z+1} + \frac{1}{2}\right) + \frac{16}{9z}$$

$${}_1F_1\left(\begin{matrix} 1 \\ b \end{matrix} \middle| z\right) = z^{-b+1} (b-1) e^z \gamma(b-1, z)$$

$${}_1F_1\left(\begin{matrix} a \\ 2a \end{matrix} \middle| z\right) = 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} e^{\frac{z}{2}} I_{a-\frac{1}{2}}\left(\frac{z}{2}\right) \Gamma\left(a + \frac{1}{2}\right)$$

$${}_1F_1\left(\begin{matrix} a \\ a+1 \end{matrix} \middle| z\right) = a(z e^{i\pi})^{-a} \gamma(a, z e^{i\pi})$$

$${}_1F_1\left(\begin{matrix} -\frac{1}{2} \\ \frac{1}{2} \end{matrix} \middle| z\right) = \sqrt{z} i \sqrt{\pi} \operatorname{erf}(\sqrt{z}i) + e^z$$

$${}_1F_2\left(\begin{matrix} 1 \\ \frac{3}{4}, \frac{5}{4} \end{matrix} \middle| z\right) = \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}}}{2\sqrt[4]{z}} \left( i \sinh(2\sqrt{z}) S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) + \cosh(2\sqrt{z}) C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) \right)$$

$${}_2F_2\left(\begin{matrix} \frac{1}{2}, a \\ \frac{3}{2}, a+1 \end{matrix} \middle| z\right) = -\frac{ai\sqrt{\pi}\sqrt{\frac{1}{z}}}{2a-1} \operatorname{erf}(\sqrt{z}i) - \frac{a(z e^{i\pi})^{-a}}{2a-1} \gamma(a, z e^{i\pi})$$

$${}_2F_2\left(\begin{matrix} 1, 1 \\ 2, 2 \end{matrix} \middle| z\right) = \frac{1}{z} (-\log(z) + \operatorname{Ei}(z)) - \frac{\gamma}{z}$$

$${}_0F_1\left(\begin{matrix} \\ \frac{1}{2} \end{matrix} \middle| z\right) = \cosh(2\sqrt{z})$$

$${}_0F_1\left(\begin{matrix} \\ b \end{matrix} \middle| z\right) = z^{-\frac{b}{2} + \frac{1}{2}} I_{b-1}(2\sqrt{z}) \Gamma(b)$$

$${}_0F_3\left(\begin{matrix} \\ \frac{1}{2}, a, a+\frac{1}{2} \end{matrix} \middle| z\right) = 2^{-2a} z^{-\frac{a}{2} + \frac{1}{4}} (I_{2a-1}(4\sqrt[4]{z}) + J_{2a-1}(4\sqrt[4]{z})) \Gamma(2a)$$

$${}_0F_3\left(\begin{matrix} \\ a, a+\frac{1}{2}, 2a \end{matrix} \middle| z\right) = \left(2\sqrt{z} e^{\frac{i\pi}{2}}\right)^{-2a+1} I_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}}\right) J_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} e^{\frac{i\pi}{4}}\right) \Gamma^2(2a)$$

$${}_1F_2\left(\begin{matrix} a \\ a-\frac{1}{2}, 2a \end{matrix} \middle| z\right) = 2 \cdot 4^{a-1} z^{-a+1} I_{a-\frac{3}{2}}(\sqrt{z}) I_{a-\frac{1}{2}}(\sqrt{z}) \Gamma\left(a-\frac{1}{2}\right) \Gamma\left(a+\frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}^2(\sqrt{z}) \Gamma^2\left(a+\frac{1}{2}\right)$$

$${}_1F_2\left(\begin{matrix} \frac{1}{2} \\ b, -b+2 \end{matrix} \middle| z\right) = \frac{\pi I_{-b+1}(\sqrt{z}) I_{b-1}(\sqrt{z})}{\sin(b\pi)} (-b+1)$$

$${}_1F_2\left(\begin{matrix} \frac{1}{2} \\ \frac{3}{2}, \frac{3}{2} \end{matrix} \middle| z\right) = \frac{1}{2\sqrt{z}} \operatorname{Shi}(2\sqrt{z})$$

$${}_1F_2\left(\begin{matrix} \frac{3}{4} \\ \frac{3}{2}, \frac{7}{4} \end{matrix} \middle| z\right) = \frac{3\sqrt{\pi}}{4z^{\frac{3}{4}}} e^{-\frac{3\pi}{4}i} S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right)$$

$${}_1F_2\left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix} \middle| z\right) = \frac{\sqrt{\pi} e^{-\frac{i\pi}{4}}}{2\sqrt[4]{z}} C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right)$$

$${}_2F_3\left(\begin{matrix} a, a+\frac{1}{2} \\ 2a, b, 2a-b+1 \end{matrix} \middle| z\right) = \left(\frac{\sqrt{z}}{2}\right)^{-2a+1} I_{2a-b}(\sqrt{z}) I_{b-1}(\sqrt{z}) \Gamma(b) \Gamma(2a-b+1)$$

$${}_2F_3\left(\begin{matrix} 1, 1 \\ 2, 2, \frac{3}{2} \end{matrix} \middle| z\right) = \frac{1}{z} (-\log(2\sqrt{z}) + \operatorname{Chi}(2\sqrt{z})) - \frac{\gamma}{z}$$

$${}_3F_3\left(\begin{matrix} 1, 1, a \\ 2, 2, a+1 \end{matrix} \middle| z\right) = \frac{a(-z)^{-a}}{(a-1)^2} (\Gamma(a) - \Gamma(a, -z)) + \frac{a}{z(a^2-2a+1)} (-a+1)(\log(-z) + \operatorname{Ei}(-z) + \gamma) - \frac{ae^z}{z(a^2-2a+1)} + \dots$$

```
sympy.simplify.hyperexpand.add_formulae(formulae)
```

Create our knowledge base.

### 3.22.5 References

## 3.23 Stats

SymPy statistics module

Introduces a random variable type into the SymPy language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc... or built with functions like FiniteRV.

Queries on random expressions can be made using the functions

Expression	Meaning
P(condition)	Probability
E(expression)	Expected value
variance(expression)	Variance
density(expression)	Probability Density Function
sample(expression)	Produce a realization
where(condition)	Where the condition is true

### 3.23.1 Examples

```
>>> from sympy.stats import P, E, variance, Die, Normal
>>> from sympy import Eq, simplify
>>> X, Y = Die('X', 6), Die('Y', 6) # Define two six sided dice
>>> Z = Normal('Z', 0, 1) # Declare a Normal random variable with mean 0, std 1
>>> P(X>3) # Probability X is greater than 3
1/2
>>> E(X+Y) # Expectation of the sum of two dice
7
>>> variance(X+Y) # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1)) # Probability of Z being greater than 1
-erf(sqrt(2)/2)/2 + 1/2
```

### 3.23.2 Random Variable Types

#### Finite Types

`sympy.stats.DiscreteUniform(name, items)`

Create a Finite Random Variable representing a uniform distribution over the input set.

Returns a RandomSymbol.

#### Examples

```
>>> from sympy.stats import DiscreteUniform, density
>>> from sympy import symbols

>>> X = DiscreteUniform('X', symbols('a b c')) # equally likely over a, b, c
>>> density(X).dict
{a: 1/3, b: 1/3, c: 1/3}
```

```
>>> Y = DiscreteUniform('Y', list(range(5))) # distribution over a range
>>> density(Y).dict
{0: 1/5, 1: 1/5, 2: 1/5, 3: 1/5, 4: 1/5}

sympy.stats.Die(name, sides=6)
Create a Finite Random Variable representing a fair die.

Returns a RandomSymbol.

>>> from sympy.stats import Die, density

>>> D6 = Die('D6', 6) # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}

>>> D4 = Die('D4', 4) # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}

sympy.stats.Bernoulli(name, p, succ=1, fail=0)
Create a Finite Random Variable representing a Bernoulli process.

Returns a RandomSymbol

>>> from sympy.stats import Bernoulli, density
>>> from sympy import S

>>> X = Bernoulli('X', S(3)/4) # 1-0 Bernoulli variable, probability = 3/4
>>> density(X).dict
{0: 1/4, 1: 3/4}

>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X).dict
{Heads: 1/2, Tails: 1/2}

sympy.stats.Coin(name, p=1/2)
Create a Finite Random Variable representing a Coin toss.

Probability p is the chance of getting "Heads." Half by default

Returns a RandomSymbol.

>>> from sympy.stats import Coin, density
>>> from sympy import Rational

>>> C = Coin('C') # A fair coin toss
>>> density(C).dict
{H: 1/2, T: 1/2}

>>> C2 = Coin('C2', Rational(3, 5)) # An unfair coin
>>> density(C2).dict
{H: 3/5, T: 2/5}

sympy.stats.Binomial(name, n, p, succ=1, fail=0)
Create a Finite Random Variable representing a binomial distribution.

Returns a RandomSymbol.
```

### Examples

```
>>> from sympy.stats import Binomial, density
>>> from sympy import S

>>> X = Binomial('X', 4, S.Half) # Four "coin flips"
>>> density(X).dict
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

`sympy.stats.Hypergeometric(name, N, m, n)`

Create a Finite Random Variable representing a hypergeometric distribution.

Returns a RandomSymbol.

### Examples

```
>>> from sympy.stats import Hypergeometric, density
>>> from sympy import S

>>> X = Hypergeometric('X', 10, 5, 3) # 10 marbles, 5 white (success), 3 draws
>>> density(X).dict
{0: 1/12, 1: 5/12, 2: 5/12, 3: 1/12}
```

`sympy.stats.FiniteRV(name, density)`

Create a Finite Random Variable given a dict representing the density.

Returns a RandomSymbol.

```
>>> from sympy.stats import FiniteRV, P, E

>>> density = {0: .1, 1: .2, 2: .3, 3: .4}
>>> X = FiniteRV('X', density)

>>> E(X)
2.00000000000000
>>> P(X>=2)
0.700000000000000
```

## Discrete Types

`sympy.stats.Geometric(name, p)`

Create a discrete random variable with a Geometric distribution.

The density of the Geometric distribution is given by

$$f(k) := p(1 - p)^{k-1}$$

**Parameters p:** A probability between 0 and 1

**Returns** A RandomSymbol.

## References

[1] [http://en.wikipedia.org/wiki/Geometric\\_distribution](http://en.wikipedia.org/wiki/Geometric_distribution) [2] <http://mathworld.wolfram.com/GeometricDistribution.html>

## Examples

```
>>> from sympy.stats import Geometric, density, E, variance
>>> from sympy import Symbol, S

>>> p = S.One / 5
>>> z = Symbol("z")

>>> X = Geometric("x", p)

>>> density(X)(z)
(4/5)**(z - 1)/5

>>> E(X)
5

>>> variance(X)
20
```

`sympy.stats.Poisson(name, lamda)`

Create a discrete random variable with a Poisson distribution.

The density of the Poisson distribution is given by

$$f(k) := \frac{\lambda^k e^{-\lambda}}{k!}$$

**Parameters** `lamda`: Positive number, a rate

**Returns** A RandomSymbol.

## References

[1] [http://en.wikipedia.org/wiki/Poisson\\_distribution](http://en.wikipedia.org/wiki/Poisson_distribution) [2] <http://mathworld.wolfram.com/PoissonDistribution.html>

## Examples

```
>>> from sympy.stats import Poisson, density, E, variance
>>> from sympy import Symbol, simplify

>>> rate = Symbol("lambda", positive=True)
>>> z = Symbol("z")

>>> X = Poisson("x", rate)

>>> density(X)(z)
lambda**z*exp(-lambda)/factorial(z)

>>> E(X)
lambda

>>> simplify(variance(X))
lambda
```

## Continuous Types

```
sympy.stats.Arcsin(name, a=0, b=1)
```

Create a Continuous Random Variable with an arcsin distribution.

The density of the arcsin distribution is given by

$$f(x) := \frac{1}{\pi \sqrt{(x-a)(b-x)}}$$

with  $x \in [a, b]$ . It must hold that  $-\infty < a < b < \infty$ .

**Parameters** **a** : Real number, the left interval boundary

**b** : Real number, the right interval boundary

**Returns** A RandomSymbol.

### References

[R383] (page 1247)

### Examples

```
>>> from sympy.stats import Arcsin, density
>>> from sympy import Symbol, simplify
```

```
>>> a = Symbol("a", extended_real=True)
>>> b = Symbol("b", extended_real=True)
>>> z = Symbol("z")
```

```
>>> X = Arcsin("x", a, b)
```

```
>>> density(X)(z)
1/(pi*sqrt((-a + z)*(b - z)))
```

```
sympy.stats.Benini(name, alpha, beta, sigma)
```

Create a Continuous Random Variable with a Benini distribution.

The density of the Benini distribution is given by

$$f(x) := e^{-\alpha \log \frac{x}{\sigma} - \beta \log^2 \left[ \frac{x}{\sigma} \right]} \left( \frac{\alpha}{x} + \frac{2\beta \log \frac{x}{\sigma}}{x} \right)$$

This is a heavy-tailed distribution and is also known as the log-Rayleigh distribution.

**Parameters** **alpha** : Real number,  $\alpha > 0$ , a shape

**beta** : Real number,  $\beta > 0$ , a shape

**sigma** : Real number,  $\sigma > 0$ , a scale

**Returns** A RandomSymbol.

### References

[R384] (page 1247), [R385] (page 1247)

## Examples

```
>>> from sympy.stats import Benini, density
>>> from sympy import Symbol, simplify, pprint

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = Benini("x", alpha, beta, sigma)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/   / z \ \ / z \ \ 2/ z \
| 2*beta*log|-----|| - alpha*log|-----| - beta*log|-----|
|alpha      \sigma/|      \sigma/      \sigma/
|----- + ----- *e
\ z          z      /
```

`sympy.stats.Beta(name, alpha, beta)`

Create a Continuous Random Variable with a Beta distribution.

The density of the Beta distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

with  $x \in [0, 1]$ .

**Parameters** `alpha` : Real number,  $\alpha > 0$ , a shape

`beta` : Real number,  $\beta > 0$ , a shape

**Returns** A RandomSymbol.

## References

[R386] (page 1247), [R387] (page 1247)

## Examples

```
>>> from sympy.stats import Beta, density, E, variance
>>> from sympy import Symbol, simplify, pprint, expand_func

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")

>>> X = Beta("x", alpha, beta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      beta - 1
z      *(-z + 1)
-----
beta(alpha, beta)
```

```
>>> expand_func(simplify(E(X, meijerg=True)))
alpha/(alpha + beta)

>>> simplify(variance(X, meijerg=True))
alpha*beta/((alpha + beta)**2*(alpha + beta + 1))
```

`sympy.stats.BetaPrime(name, alpha, beta)`

Create a continuous random variable with a Beta prime distribution.

The density of the Beta prime distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1+x)^{-\alpha-\beta}}{B(\alpha, \beta)}$$

with  $x > 0$ .

**Parameters** `alpha` : Real number,  $\alpha > 0$ , a shape

`beta` : Real number,  $\beta > 0$ , a shape

**Returns** A RandomSymbol.

## References

[R388] (page 1247), [R389] (page 1247)

## Examples

```
>>> from sympy.stats import BetaPrime, density
>>> from sympy import Symbol, pprint

>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")

>>> X = BetaPrime("x", alpha, beta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      -alpha - beta
z      *(z + 1)
-----
beta(alpha, beta)
```

`sympy.stats.Cauchy(name, x0, gamma)`

Create a continuous random variable with a Cauchy distribution.

The density of the Cauchy distribution is given by

$$f(x) := \frac{1}{\pi} \arctan\left(\frac{x - x_0}{\gamma}\right) + \frac{1}{2}$$

**Parameters** `x0` : Real number, the location

`gamma` : Real number,  $\gamma > 0$ , the scale

**Returns** A RandomSymbol.

## References

[R390] (page 1247), [R391] (page 1247)

## Examples

```
>>> from sympy.stats import Cauchy, density
>>> from sympy import Symbol

>>> x0 = Symbol("x0")
>>> gamma = Symbol("gamma", positive=True)
>>> z = Symbol("z")

>>> X = Cauchy("x", x0, gamma)

>>> density(X)(z)
1/(pi*gamma*(1 + (-x0 + z)**2/gamma**2))
```

`sympy.stats.Chi(name, k)`

Create a continuous random variable with a Chi distribution.

The density of the Chi distribution is given by

$$f(x) := \frac{2^{1-k/2} x^{k-1} e^{-x^2/2}}{\Gamma(k/2)}$$

with  $x \geq 0$ .

**Parameters** `k` : A positive Integer,  $k > 0$ , the number of degrees of freedom

**Returns** A RandomSymbol.

## References

[R392] (page 1247), [R393] (page 1247)

## Examples

```
>>> from sympy.stats import Chi, density, E, std
>>> from sympy import Symbol, simplify

>>> k = Symbol("k", integer=True)
>>> z = Symbol("z")

>>> X = Chi("x", k)

>>> density(X)(z)
2**(-k/2 + 1)*z**(k - 1)*exp(-z**2/2)/gamma(k/2)
```

`sympy.stats.ChiNoncentral(name, k, l)`

Create a continuous random variable with a non-central Chi distribution.

The density of the non-central Chi distribution is given by

$$f(x) := \frac{e^{-(x^2+\lambda^2)/2} x^k \lambda}{(\lambda x)^{k/2}} I_{k/2-1}(\lambda x)$$

with  $x \geq 0$ . Here,  $I_\nu(x)$  is the *modified Bessel function of the first kind* (page 391).

**Parameters** `k` : A positive Integer,  $k > 0$ , the number of degrees of freedom

`l` : Shift parameter

**Returns** A RandomSymbol.

## References

[R394] (page 1247)

## Examples

```
>>> from sympy.stats import ChiNoncentral, density, E, std
>>> from sympy import Symbol, simplify

>>> k = Symbol("k", integer=True)
>>> l = Symbol("l")
>>> z = Symbol("z")

>>> X = ChiNoncentral("x", k, l)

>>> density(X)(z)
1*z**k*(1*z)**(-k/2)*exp(-l**2/2 - z**2/2)*besseli(k/2 - 1, l*z)
```

`sympy.stats.ChiSquared(name, k)`

Create a continuous random variable with a Chi-squared distribution.

The density of the Chi-squared distribution is given by

$$f(x) := \frac{1}{2^{\frac{k}{2}} \Gamma\left(\frac{k}{2}\right)} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$$

with  $x \geq 0$ .

**Parameters** `k` : A positive Integer,  $k > 0$ , the number of degrees of freedom

**Returns** A RandomSymbol.

## References

[R395] (page 1247), [R396] (page 1247)

## Examples

```
>>> from sympy.stats import ChiSquared, density, E, variance
>>> from sympy import Symbol, simplify, combsimp, expand_func
```

```

>>> k = Symbol("k", integer=True, positive=True)
>>> z = Symbol("z")

>>> X = ChiSquared("x", k)

>>> density(X)(z)
2**(-k/2)*z**(k/2 - 1)*exp(-z/2)/gamma(k/2)

>>> combsimp(E(X))
k

>>> simplify(expand_func(variance(X)))
2*k

```

`sympy.stats.Dagum(name, p, a, b)`

Create a continuous random variable with a Dagum distribution.

The density of the Dagum distribution is given by

$$f(x) := \frac{ap}{x} \left( \frac{\left(\frac{x}{b}\right)^{ap}}{\left(\left(\frac{x}{b}\right)^a + 1\right)^{p+1}} \right)$$

with  $x > 0$ .

**Parameters** `p` : Real number,  $p > 0$ , a shape

`a` : Real number,  $a > 0$ , a shape

`b` : Real number,  $b > 0$ , a scale

**Returns** A RandomSymbol.

## References

[R397] (page 1247)

## Examples

```

>>> from sympy.stats import Dagum, density
>>> from sympy import Symbol, simplify

>>> p = Symbol("p", positive=True)
>>> b = Symbol("b", positive=True)
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")

>>> X = Dagum("x", p, a, b)

>>> density(X)(z)
a*p*(z/b)**(a*p)*((z/b)**a + 1)**(-p - 1)/z

```

`sympy.stats.Erlang(name, k, l)`

Create a continuous random variable with an Erlang distribution.

The density of the Erlang distribution is given by

$$f(x) := \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

with  $x \in [0, \infty]$ .

**Parameters** `k` : Integer

`l` : Real number,  $\lambda > 0$ , the rate

**Returns** A RandomSymbol.

## References

[R398] (page 1247), [R399] (page 1247)

## Examples

```
>>> from sympy.stats import Erlang, density, cdf, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> k = Symbol("k", integer=True, positive=True)
>>> l = Symbol("l", positive=True)
>>> z = Symbol("z")

>>> X = Erlang("x", k, l)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
k  k - 1  -l*z
l *z      *e
-----
gamma(k)

>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/  k*lowergamma(k, 0)  k*lowergamma(k, l*z)
|- ----- + ----- for z >= 0
<    gamma(k + 1)      gamma(k + 1)
|
\                           0
                                otherwise

>>> simplify(E(X))
k/l

>>> simplify(variance(X))
k/l**2
```

`sympy.stats.Exponential(name, rate)`

Create a continuous random variable with an Exponential distribution.

The density of the exponential distribution is given by

$$f(x) := \lambda \exp(-\lambda x)$$

with  $x > 0$ . Note that the expected value is  $1/\lambda$ .

**Parameters** `rate` : A positive Real number,  $\lambda > 0$ , the rate (or inverse scale/inverse mean)  
**Returns** A RandomSymbol.

## References

[R400] (page 1247), [R401] (page 1247)

## Examples

```
>>> from sympy.stats import Exponential, density, cdf, E
>>> from sympy.stats import variance, std, skewness
>>> from sympy import Symbol

>>> l = Symbol("lambda", positive=True)
>>> z = Symbol("z")

>>> X = Exponential("x", l)

>>> density(X)(z)
lambda*exp(-lambda*z)

>>> cdf(X)(z)
Piecewise((1 - exp(-lambda*z), z >= 0), (0, True))

>>> E(X)
1/lambda

>>> variance(X)
lambda**(-2)

>>> skewness(X)
2

>>> X = Exponential('x', 10)

>>> density(X)(z)
10*exp(-10*z)

>>> E(X)
1/10

>>> std(X)
1/10
```

`sympy.stats.FDistribution(name, d1, d2)`  
Create a continuous random variable with a F distribution.

The density of the F distribution is given by

$$f(x) := \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1 + d_2}}}}{x B\left(\frac{d_1}{2}, \frac{d_2}{2}\right)}$$

with  $x > 0$ .

**Parameters** **d1** :  $d_1 > 0$  a parameter  
**d2** :  $d_2 > 0$  a parameter  
**Returns** A RandomSymbol.

## References

[R402] (page 1247), [R403] (page 1247)

## Examples

```
>>> from sympy.stats import FDistribution, density
>>> from sympy import Symbol, simplify, pprint

>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")

>>> X = FDistribution("x", d1, d2)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
d2
--_
  2   /      d1      -d1 - d2
d2 * \ / (d1*z) *(d1*z + d2)
-----
           /d1  d2\
           z*beta|--, --|
           \2    2 /
```

`sympy.stats.FisherZ(name, d1, d2)`

Create a Continuous Random Variable with an Fisher's Z distribution.

The density of the Fisher's Z distribution is given by

$$f(x) := \frac{2d_1^{d_1/2} d_2^{d_2/2}}{B(d_1/2, d_2/2)} \frac{e^{d_1 z}}{(d_1 e^{2z} + d_2)^{(d_1+d_2)/2}}$$

**Parameters** **d1** :  $d_1 > 0$ , degree of freedom

**d2** :  $d_2 > 0$ , degree of freedom

**Returns** A RandomSymbol.

## References

[R404] (page 1247), [R405] (page 1247)

## Examples

```
>>> from sympy.stats import FisherZ, density
>>> from sympy import Symbol, simplify, pprint
```

```

>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")

>>> X = FisherZ("x", d1, d2)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      d1   d2
d1   d2   - - - -
--   --
      2   2
  2   2 / 2*z   \
2*d1 *d2 *d1*e + d2/   d1*z
----- *e
----- /d1   d2\
beta|--, --|
\2   2 /

```

`sympy.stats.Frechet(name, a, s=1, m=0)`

Create a continuous random variable with a Frechet distribution.

The density of the Frechet distribution is given by

$$f(x) := \frac{\alpha}{s} \left( \frac{x-m}{s} \right)^{-1-\alpha} e^{-(\frac{x-m}{s})^{-\alpha}}$$

with  $x \geq m$ .

**Parameters** `a` : Real number,  $a \in (0, \infty)$  the shape

`s` : Real number,  $s \in (0, \infty)$  the scale

`m` : Real number,  $m \in (-\infty, \infty)$  the minimum

**Returns** A RandomSymbol.

## References

[R406] (page 1247)

## Examples

```

>>> from sympy.stats import Frechet, density, E, std
>>> from sympy import Symbol, simplify

>>> a = Symbol("a", positive=True)
>>> s = Symbol("s", positive=True)
>>> m = Symbol("m", extended_real=True)
>>> z = Symbol("z")

>>> X = Frechet("x", a, s, m)

>>> density(X)(z)
a*((-m + z)/s)**(-a - 1)*exp(-((-m + z)/s)**(-a))/s

```

```
sympy.stats.Gamma(name, k, theta)
```

Create a continuous random variable with a Gamma distribution.

The density of the Gamma distribution is given by

$$f(x) := \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}$$

with  $x \in [0, 1]$ .

**Parameters** `k` : Real number,  $k > 0$ , a shape

`theta` : Real number,  $\theta > 0$ , a scale

**Returns** A RandomSymbol.

## References

[R407] (page 1247), [R408] (page 1247)

## Examples

```
>>> from sympy.stats import Gamma, density, cdf, E, variance
>>> from sympy import Symbol, pprint, simplify

>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> z = Symbol("z")

>>> X = Gamma("x", k, theta)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      -z
      -----
      -k  k - 1  theta
theta *z      *e
-----
gamma(k)

>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
      /      z \
      |      k*lowergamma|k, -----
      |      \        theta/
      k*lowergamma(k, 0)
<- ----- + ----- for z >= 0
      gamma(k + 1)      gamma(k + 1)
      |
      \                           otherwise

>>> E(X)
theta*gamma(k + 1)/gamma(k)

>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
      2
      k*theta
```

---

```
sympy.stats.GammaInverse(name, a, b)
```

Create a continuous random variable with an inverse Gamma distribution.

The density of the inverse Gamma distribution is given by

$$f(x) := \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(\frac{-\beta}{x}\right)$$

with  $x > 0$ .

**Parameters** **a** : Real number,  $a > 0$  a shape

**b** : Real number,  $b > 0$  a scale

**Returns** A RandomSymbol.

## References

[R409] (page 1247)

## Examples

```
>>> from sympy.stats import GammaInverse, density, cdf, E, variance
>>> from sympy import Symbol, pprint

>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = GammaInverse("x", a, b)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      -b
      ---
      a  -a - 1   z
      b *z        *e
      -----
                           gamma(a)
```

```
sympy.stats.Kumaraswamy(name, a, b)
```

Create a Continuous Random Variable with a Kumaraswamy distribution.

The density of the Kumaraswamy distribution is given by

$$f(x) := abx^{a-1}(1-x^a)^{b-1}$$

with  $x \in [0, 1]$ .

**Parameters** **a** : Real number,  $a > 0$  a shape

**b** : Real number,  $b > 0$  a scale

**Returns** A RandomSymbol.

## References

[R410] (page 1247)

## Examples

```
>>> from sympy.stats import Kumaraswamy, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Kumaraswamy("x", a, b)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      b - 1
      a - 1 /   a    \
a*b*z     * \ - z + 1/
```

`sympy.stats.Laplace(name, mu, b)`

Create a continuous random variable with a Laplace distribution.

The density of the Laplace distribution is given by

$$f(x) := \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

**Parameters** `mu` : Real number, the location (mean)

`b` : Real number,  $b > 0$ , a scale

**Returns** A RandomSymbol.

## References

[R411] (page 1247), [R412] (page 1247)

## Examples

```
>>> from sympy.stats import Laplace, density
>>> from sympy import Symbol

>>> mu = Symbol("mu")
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Laplace("x", mu, b)

>>> density(X)(z)
exp(-Abs(mu - z)/b)/(2*b)
```

`sympy.stats.Logistic(name, mu, s)`

Create a continuous random variable with a logistic distribution.

The density of the logistic distribution is given by

$$f(x) := \frac{e^{-(x-\mu)/s}}{s \left(1 + e^{-(x-\mu)/s}\right)^2}$$

**Parameters** `mu` : Real number, the location (mean)  
`s` : Real number,  $s > 0$  a scale  
**Returns** A RandomSymbol.

## References

[R413] (page 1247), [R414] (page 1247)

## Examples

```
>>> from sympy.stats import Logistic, density
>>> from sympy import Symbol

>>> mu = Symbol("mu", extended_real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")

>>> X = Logistic("x", mu, s)

>>> density(X)(z)
exp((mu - z)/s)/(s*(exp((mu - z)/s) + 1)**2)
```

`sympy.stats.LogNormal(name, mean, std)`

Create a continuous random variable with a log-normal distribution.

The density of the log-normal distribution is given by

$$f(x) := \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

with  $x \geq 0$ .

**Parameters** `mu` : Real number, the log-scale  
`sigma` : Real number,  $\sigma^2 > 0$  a shape  
**Returns** A RandomSymbol.

## References

[R415] (page 1248), [R416] (page 1248)

## Examples

```
>>> from sympy.stats import LogNormal, density
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu", extended_real=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = LogNormal("x", mu, sigma)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)

$$\frac{e^{-\frac{(-\mu + \ln(z))^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma z}$$


>>> X = LogNormal('x', 0, 1) # Mean 0, standard deviation 1

>>> density(X)(z)

$$\frac{\sqrt{2} e^{-\frac{z^2}{2}}}{\sqrt{\pi} z}$$

```

`sympy.stats.Maxwell(name, a)`

Create a continuous random variable with a Maxwell distribution.

The density of the Maxwell distribution is given by

$$f(x) := \sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3}$$

with  $x \geq 0$ .

**Parameters** `a` : Real number,  $a > 0$

**Returns** A RandomSymbol.

## References

[R417] (page 1248), [R418] (page 1248)

## Examples

```
>>> from sympy.stats import Maxwell, density, E, variance
>>> from sympy import Symbol, simplify

>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")

>>> X = Maxwell("x", a)

>>> density(X)(z)

$$\frac{\sqrt{2} z^2 e^{-z^2/(2a^2)}}{\sqrt{\pi} a^3}$$


>>> E(X)

$$\frac{2\sqrt{2} a}{\sqrt{\pi}}$$


>>> simplify(variance(X))

$$\frac{a^2 (8 - 3\pi)}{2}$$

```

---

```
sympy.stats.Nakagami(name, mu, omega)
```

Create a continuous random variable with a Nakagami distribution.

The density of the Nakagami distribution is given by

$$f(x) := \frac{2\mu^\mu}{\Gamma(\mu)\omega^\mu} x^{2\mu-1} \exp\left(-\frac{\mu}{\omega}x^2\right)$$

with  $x > 0$ .

**Parameters** `mu` : Real number,  $\mu \geq \frac{1}{2}$  a shape

`omega` : Real number,  $\omega > 0$ , the spread

**Returns** A RandomSymbol.

## References

[R419] (page 1248)

## Examples

```
>>> from sympy.stats import Nakagami, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu", positive=True)
>>> omega = Symbol("omega", positive=True)
>>> z = Symbol("z")

>>> X = Nakagami("x", mu, omega)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
              2
      -mu*z
      -----
      mu      -mu   2*mu - 1   omega
  2*mu *omega   *z           *e
  -----
                               gamma(mu)

>>> simplify(E(X, meijerg=True))
sqrt(mu)*sqrt(omega)*gamma(mu + 1/2)/gamma(mu + 1)

>>> V = simplify(variance(X, meijerg=True))
>>> pprint(V, use_unicode=False)
      2
      omega*gamma (mu + 1/2)
omega - -----
                           gamma(mu)*gamma(mu + 1)
```

```
sympy.stats.Normal(name, mean, std)
```

Create a continuous random variable with a Normal distribution.

The density of the Normal distribution is given by

$$f(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

**Parameters** `mu` : Real number, the mean  
`sigma` : Real number,  $\sigma^2 > 0$  the variance  
**Returns** A RandomSymbol.

## References

[R420] (page 1248), [R421] (page 1248)

## Examples

```
>>> from sympy.stats import Normal, density, E, std, cdf, skewness
>>> from sympy import Symbol, simplify, pprint, factor, together, factor_terms

>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = Normal("x", mu, sigma)

>>> density(X)(z)
sqrt(2)*exp(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)

>>> C = simplify(cdf(X))(z) # it needs a little more help...
>>> pprint(C, use_unicode=False)
      / ___ \
      | \ 2 *(-mu + z)|
erf|-----|
      \ 2*sigma   /  1
----- + -
      2             2

>>> simplify(skewness(X))
0

>>> X = Normal("x", 0, 1) # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*exp(-z**2/2)/(2*sqrt(pi))

>>> E(2*X + 1)
1

>>> simplify(std(2*X + 1))
2
```

`sympy.stats.Pareto(name, xm, alpha)`

Create a continuous random variable with the Pareto distribution.

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha x_m^\alpha}{x^{\alpha+1}}$$

with  $x \in [x_m, \infty]$ .

**Parameters** `xm` : Real number,  $x_m > 0$ , a scale  
`alpha` : Real number,  $\alpha > 0$ , a shape  
**Returns** A RandomSymbol.

#### References

[R422] (page 1248), [R423] (page 1248)

#### Examples

```
>>> from sympy.stats import Pareto, density
>>> from sympy import Symbol

>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")

>>> X = Pareto("x", xm, beta)

>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

`sympy.stats.QuadraticU(name, a, b)`

Create a Continuous Random Variable with a U-quadratic distribution.

The density of the U-quadratic distribution is given by

$$f(x) := \alpha(x - \beta)^2$$

with  $x \in [a, b]$ .

**Parameters** `a` : Real number

`b` : Real number,  $a < b$

**Returns** A RandomSymbol.

#### References

[R424] (page 1248)

#### Examples

```
>>> from sympy.stats import QuadraticU, density, E, variance
>>> from sympy import Symbol, simplify, factor, pprint

>>> a = Symbol("a", extended_real=True)
>>> b = Symbol("b", extended_real=True)
>>> z = Symbol("z")

>>> X = QuadraticU("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
|   /   a   b   \
| 12*(- - - + z)
|   \   2   2   /
<----- for And(a <= z, z <= b)
|
|           3
|   (-a + b)
|
\       0           otherwise
```

`sympy.stats.RaisedCosine(name, mu, s)`

Create a Continuous Random Variable with a raised cosine distribution.

The density of the raised cosine distribution is given by

$$f(x) := \frac{1}{2s} \left( 1 + \cos \left( \frac{x - \mu}{s} \pi \right) \right)$$

with  $x \in [\mu - s, \mu + s]$ .

**Parameters** `mu` : Real number

`s` : Real number,  $s > 0$

**Returns** A RandomSymbol.

## References

[R425] (page 1248)

## Examples

```
>>> from sympy.stats import RaisedCosine, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu", extended_real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")

>>> X = RaisedCosine("x", mu, s)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
|   /pi*(-mu + z)\_
|cos|-----| + 1
|   \   s   /
<----- for And(z <= mu + s, mu - s <= z)
|
|           2*s
|
\       0           otherwise
```

`sympy.stats.Rayleigh(name, sigma)`

Create a continuous random variable with a Rayleigh distribution.

The density of the Rayleigh distribution is given by

$$f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}$$

with  $x > 0$ .

**Parameters** `sigma` : Real number,  $\sigma > 0$

**Returns** A RandomSymbol.

## References

[R426] (page 1248), [R427] (page 1248)

## Examples

```
>>> from sympy.stats import Rayleigh, density, E, variance
>>> from sympy import Symbol, simplify

>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

>>> X = Rayleigh("x", sigma)

>>> density(X)(z)
z*exp(-z**2/(2*sigma**2))/sigma**2

>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2

>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2

sympy.stats.StudentT(name, nu)
```

Create a continuous random variable with a student's t distribution.

The density of the student's t distribution is given by

$$f(x) := \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

**Parameters** `nu` : Real number,  $\nu > 0$ , the degrees of freedom

**Returns** A RandomSymbol.

## References

[R428] (page 1248), [R429] (page 1248)

## Examples

```
>>> from sympy.stats import StudentT, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")

>>> X = StudentT("x", nu)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      nu   1
      - - - -
      2   2
      /   2 \
      |   z |
| 1 + --|
      \   nu/
-----
      /   nu \
  --- *beta|1/2, --|
      \   2 /
```

`sympy.stats.Triangular(name, a, b, c)`

Create a continuous random variable with a triangular distribution.

The density of the triangular distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

**Parameters** `a` : Real number,  $a \in (-\infty, \infty)$

`b` : Real number,  $a < b$

`c` : Real number,  $a \leq c \leq b$

**Returns** A RandomSymbol.

## References

[R430] (page 1248), [R431] (page 1248)

## Examples

```
>>> from sympy.stats import Triangular, density, E
>>> from sympy import Symbol, pprint

>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")
```

```
>>> X = Triangular("x", a,b,c)

>>> pprint(density(X)(z), use_unicode=False)
/
| -2*a + 2*z
|----- for And(a <= z, z < c)
|(-a + b)*(-a + c)
|
|      2
|----- for z = c
< -a + b
|
|      2*b - 2*z
|----- for And(z <= b, c < z)
|(-a + b)*(b - c)
|
\      0          otherwise
```

`sympy.stats.Uniform(name, left, right)`

Create a continuous random variable with a uniform distribution.

The density of the uniform distribution is given by

$$f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

with  $x \in [a, b]$ .

**Parameters** **a** : Real number,  $-\infty < a$  the left boundary

**b** : Real number,  $a < b < \infty$  the right boundary

**Returns** A RandomSymbol.

## References

[R432] (page 1248), [R433] (page 1248)

## Examples

```
>>> from sympy.stats import Uniform, density, cdf, E, variance, skewness
>>> from sympy import Symbol, simplify

>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")

>>> X = Uniform("x", a, b)

>>> density(X)(z)
Piecewise((1/(-a + b), And(a <= z, z <= b)), (0, True))

>>> cdf(X)(z)
-a/(-a + b) + z/(-a + b)
```

```
>>> simplify(E(X))
a/2 + b/2

>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

`sympy.stats.UniformSum(name, n)`

Create a continuous random variable with an Irwin-Hall distribution.

The probability distribution function depends on a single parameter  $n$  which is an integer.

The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!} \sum_{k=0}^{\lfloor x \rfloor} (-1)^k \binom{n}{k} (x-k)^{n-1}$$

**Parameters** `n` : A positive Integer,  $n > 0$

**Returns** A RandomSymbol.

## References

[R434] (page 1248), [R435] (page 1248)

## Examples

```
>>> from sympy.stats import UniformSum, density
>>> from sympy import Symbol, pprint

>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")

>>> X = UniformSum("x", n)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)

  _____
  \      k      n - 1 /n\
  )    (-1) *(-k + z)   *| |
  /
  /__,
k = 0
-----
(n - 1)!
```

`sympy.stats.VonMises(name, mu, k)`

Create a Continuous Random Variable with a von Mises distribution.

The density of the von Mises distribution is given by

$$f(x) := \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

with  $x \in [0, 2\pi]$ .

**Parameters** `mu` : Real number, measure of location  
`k` : Real number, measure of concentration  
**Returns** A RandomSymbol.

**References**

[R436] (page 1248), [R437] (page 1248)

**Examples**

```
>>> from sympy.stats import VonMises, density, E, variance
>>> from sympy import Symbol, simplify, pprint

>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")

>>> X = VonMises("x", mu, k)

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      k*cos(mu - z)
      e
-----
2*pi*besseli(0, k)
```

`sympy.stats.Weibull(name, alpha, beta)`

Create a continuous random variable with a Weibull distribution.

The density of the Weibull distribution is given by

$$f(x) := \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

**Parameters** `lambda` : Real number,  $\lambda > 0$  a scale

`k` : Real number,  $k > 0$  a shape

**Returns** A RandomSymbol.

**References**

[R438] (page 1248), [R439] (page 1248)

**Examples**

```
>>> from sympy.stats import Weibull, density, E, variance
>>> from sympy import Symbol, simplify

>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Weibull("x", 1, k)

>>> density(X)(z)
k*(z/lambda)**(k - 1)*exp(-(z/lambda)**k)/lambda

>>> simplify(E(X))
lambda*gamma(1 + 1/k)

>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

`sympy.stats.WignerSemicircle(name, R)`

Create a continuous random variable with a Wigner semicircle distribution.

The density of the Wigner semicircle distribution is given by

$$f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}$$

with  $x \in [-R, R]$ .

**Parameters** `R` : Real number,  $R > 0$ , the radius

**Returns** A *RandomSymbol*.

## References

[R440] (page 1248), [R441] (page 1248)

## Examples

```
>>> from sympy.stats import WignerSemicircle, density, E
>>> from sympy import Symbol, simplify
```

```
>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")
```

```
>>> X = WignerSemicircle("x", R)
```

```
>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)
```

```
>>> E(X)
0
```

`sympy.stats.ContinuousRV(symbol, density, set=(-oo, oo))`

Create a Continuous Random Variable given the following:

– a symbol – a probability density function – set on which the pdf is valid (defaults to entire real line)

Returns a *RandomSymbol*.

Many common continuous random variable types are already implemented. This function should be necessary only very rarely.

## Examples

```
>>> from sympy import Symbol, sqrt, exp, pi
>>> from sympy.stats import ContinuousRV, P, E

>>> x = Symbol("x")

>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)) # Normal distribution
>>> X = ContinuousRV(x, pdf)

>>> E(X)
0
>>> P(X>0)
1/2
```

### 3.23.3 Interface

`sympy.stats.P(condition, given_condition=None, numsamples=None, evaluate=True, **kwargs)`

Probability that a condition is true, optionally given a second condition

**Parameters** `expr` : Relational containing RandomSymbols

The condition of which you want to compute the probability

`given_condition` : Relational containing RandomSymbols

A conditional expression.  $P(X>1, X>0)$  is expectation of  $X>1$  given  $X>0$

`numsamples` : int

Enables sampling and approximates the probability with this many samples

`evalf` : Bool (defaults to True)

If sampling return a number rather than a complex expression

`evaluate` : Bool (defaults to True)

In case of continuous systems return unevaluated integral

## Examples

```
>>> from sympy.stats import P, Die
>>> from sympy import Eq
>>> X, Y = Die('X', 6), Die('Y', 6)
>>> P(X>3)
1/2
>>> P(Eq(X, 5), X>2) # Probability that X == 5 given that X > 2
1/4
>>> P(X>Y)
5/12
```

`sympy.stats.E(expr, condition=None, numsamples=None, evaluate=True, **kwargs)`

Returns the expected value of a random expression

**Parameters** `expr` : Expr containing RandomSymbols

The expression of which you want to compute the expectation value

**given** : Expr containing RandomSymbols  
A conditional expression.  $E(X, X>0)$  is expectation of  $X$  given  $X > 0$

**numsamples** : int  
Enables sampling and approximates the expectation with this many samples

**evalf** : Bool (defaults to True)  
If sampling return a number rather than a complex expression

**evaluate** : Bool (defaults to True)  
In case of continuous systems return unevaluated integral

### Examples

```
>>> from sympy.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8

>>> E(X, X>3) # Expectation of X given that it is above 3
5
```

`sympy.stats.density(expr, condition=None, evaluate=True, numsamples=None, **kwargs)`  
Probability density of a random expression, optionally given a second condition.

This density will take on different forms for different types of probability spaces. Discrete variables produce Dicts. Continuous variables produce Lambdas.

**Parameters** **expr** : Expr containing RandomSymbols  
The expression of which you want to compute the density value

**condition** : Relational containing RandomSymbols  
A conditional expression.  $\text{density}(X>1, X>0)$  is density of  $X>1$  given  $X>0$

**numsamples** : int  
Enables sampling and approximates the density with this many samples

### Examples

```
>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol

>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)

>>> density(D).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D).dict
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)(x)
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

---

```
sympy.stats.given(expr, condition=None, **kwargs)
```

Conditional Random Expression From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.

### Examples

```
>>> from sympy.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X>3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```
>>> from sympy.stats import Normal
>>> from sympy import pprint
>>> from sympy.abc import z

>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)

$$\frac{-(\bar{Y} + z)}{\sqrt{2} \cdot \sqrt{\pi}}$$

```

```
sympy.stats.where(condition, given_condition=None, **kwargs)
```

Returns the domain where a condition is True.

### Examples

```
>>> from sympy.stats import where, Die, Normal
>>> from sympy import symbols, And

>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)

>>> where(X**2<1)
Domain: And(-1 < x, x < 1)

>>> where(X**2<1).set
(-1, 1)

>>> where(And(D1<=D2 , D2<3))
Domain: Or(And(Eq(a, 1), Eq(b, 1)), And(Eq(a, 1), Eq(b, 2)), And(Eq(a, 2), Eq(b, 2)))
```

```
sympy.stats.variance(X, condition=None, **kwargs)
```

Variance of a random expression

Expectation of  $(X - E(X))^2$

## Examples

```
>>> from sympy.stats import Die, E, Bernoulli, variance
>>> from sympy import simplify, Symbol

>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)

>>> variance(2*X)
35/3

>>> simplify(variance(B))
p*(-p + 1)

sympy.stats.std(X, condition=None, **kwargs)
Standard Deviation of a random expression

Square root of the Expectation of (X-E(X))**2
```

## Examples

```
>>> from sympy.stats import Bernoulli, std
>>> from sympy import Symbol, simplify

>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)

>>> simplify(std(B))
sqrt(p*(-p + 1))

sympy.stats.sample(expr, condition=None, **kwargs)
A realization of the random expression
```

## Examples

```
>>> from sympy.stats import Die, sample
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6)

>>> die_roll = sample(X + Y + Z) # A random realization of three dice

sympy.stats.sample_iter(expr, condition=None, numsamples=oo, **kwargs)
Returns an iterator of realizations from the expression given a condition

expr: Random expression to be realized condition: A conditional expression (optional) numsamples:
Length of the iterator (defaults to infinity)

See also:
sympy.stats.sample      (page      980),      sympy.stats.rv.sampling_P      (page      982),
sympy.stats.rv.sampling_E (page  982),  sympy.stats.rv.sample_iter_lambdify (page  983),
sympy.stats.rv.sample_iter_subs (page 983)
```

## Examples

```
>>> from sympy.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X*X + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

### 3.23.4 Mechanics

SymPy Stats employs a relatively complex class hierarchy.

`RandomDomains` are a mapping of variables to possible values. For example we might say that the symbol `Symbol('x')` can take on the values  $\{1, 2, 3, 4, 5, 6\}$ .

```
class sympy.stats.rv.RandomDomain
```

A `PSpace`, or Probability Space, combines a `RandomDomain` with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density  $\{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6\}$  to fully define the roll of a fair die named `x`.

```
class sympy.stats.rv.PSpace
```

A `RandomSymbol` represents the `PSpace`'s symbol 'x' inside of SymPy expressions.

```
class sympy.stats.rv.RandomSymbol
```

The `RandomDomain` and `PSpace` classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

`RandomDomains` and `PSpaces` must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build `SingleDomains` and `SinglePSpaces` to represent single, univariate events and processes such as a single die or a single normal variable.

```
class sympy.stats.rv.SinglePSpace
```

```
class sympy.stats.rv.SingleDomain
```

Another common case is to collect together a set of such univariate random variables. A collection of independent `SinglePSpaces` or `SingleDomains` can be brought together to form a `ProductDomain` or `ProductPSpace`. These objects would be useful in representing three dice rolled together for example.

```
class sympy.stats.rv.ProductDomain
```

```
class sympy.stats.rv.ProductPSpace
```

The `Conditional` adjective is added whenever we add a global condition to a `RandomDomain` or `PSpace`. A common example would be three independent dice where we know their sum to be greater than 12.

```
class sympy.stats.rv.ConditionalDomain
```

We specialize further into `Finite` and `Continuous` versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

```
class sympy.stats.frv.FiniteDomain
```

```
class sympy.stats.frv.FinitePSpace
```

```
class sympy.stats.crv.ContinuousDomain
```

```
class sympy.stats.crv.ContinuousPSpace
```

Additionally there are a few specialized classes that implement certain common random variable types. There is for example a DiePSpace that implements SingleFinitePSpace and a NormalPSpace that implements SingleContinuousPSpace.

```
class sympy.stats.frv_types.DiePSpace  
class sympy.stats.crv_types.NormalPSpace
```

RandomVariables can be extracted from these objects using the PSpace.values method.

As previously mentioned SymPy Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow SymPy to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending SymPy Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions Die, Coin, FiniteRV, Normal, Exponential, etc.... These build the appropriate SinglePSpaces and return the corresponding RandomVariable. Conditional and Product spaces are formed in the natural construction of SymPy expressions and the use of interface functions E, Given, Density, etc....

```
sympy.stats.Die()
```

```
sympy.stats.Normal()
```

There are some additional functions that may be useful. They are largely used internally.

```
sympy.stats.rv.random_symbols(expr)
```

Returns all RandomSymbols within a SymPy Expression.

```
sympy.stats.rv.pspace(expr)
```

Returns the underlying Probability Space of a random expression.

For internal use.

## Examples

```
>>> from sympy.stats import pspace, Normal  
>>> from sympy.stats.rv import ProductPSpace  
>>> X = Normal('X', 0, 1)  
>>> pspace(2*X + 1) == X.pspace  
True
```

```
sympy.stats.rv.rs_swap(a, b)
```

Build a dictionary to swap RandomSymbols based on their underlying symbol.

i.e. if  $X = ('x', pspace1)$  and  $Y = ('x', pspace2)$  then  $X$  and  $Y$  match and the key, value pair  $\{X:Y\}$  will appear in the result

Inputs: collections a and b of random variables which share common symbols Output: dict mapping RVs in a to RVs in b

```
sympy.stats.rv.sampling_P(condition, given_condition=None, numsamples=1, evalf=True,  
                           **kwargs)
```

Sampling version of P

See also:

[sympy.stats.P](#) (page 977), [sympy.stats.rv.sampling\\_E](#) (page 982),  
[sympy.stats.rv.sampling\\_density](#) (page 983)

`sympy.stats.rv.sampling_E(expr, given_condition=None, numsamples=1, evalf=True, **kwargs)`  
Sampling version of E

See also:

`sympy.stats.P` (page 977), `sympy.stats.rv.sampling_P` (page 982),  
`sympy.stats.rv.sampling_density` (page 983)

`sympy.stats.rv.sample_iter_lambdify(expr, condition=None, numsamples=oo, **kwargs)`  
See `sample_iter`

Uses lambdify for computation. This is fast but does not always work.

`sympy.stats.rv.sample_iter_subs(expr, condition=None, numsamples=oo, **kwargs)`  
See `sample_iter`

Uses subs for computation. This is slow but almost always works.

`sympy.stats.rv.sampling_density(expr, given_condition=None, numsamples=1, **kwargs)`  
Sampling version of density

See also:

`sympy.stats.density` (page 978), `sympy.stats.rv.sampling_P` (page 982),  
`sympy.stats.rv.sampling_E` (page 982)

`sympy.stats.rv.independent(a, b)`  
Independence of two random expressions

Two expressions are independent if knowledge of one does not change computations on the other.

See also:

`sympy.stats.rv.dependent` (page 983)

## Examples

```
>>> from sympy.stats import Normal, independent, given
>>> from sympy import Tuple, Eq
```

```
>>> X, Y = Normal('X', 0, 1), Normal('Y', 0, 1)
>>> independent(X, Y)
True
>>> independent(2*X + Y, -Y)
False
>>> X, Y = given(Tuple(X, Y), Eq(X + Y, 3))
>>> independent(X, Y)
False
```

`sympy.stats.rv.dependent(a, b)`  
Dependence of two random expressions

Two expressions are independent if knowledge of one does not change computations on the other.

See also:

`sympy.stats.rv.independent` (page 983)

## Examples

```
>>> from sympy.stats import Normal, dependent, given
>>> from sympy import Tuple, Eq

>>> X, Y = Normal('X', 0, 1), Normal('Y', 0, 1)
>>> dependent(X, Y)
False
>>> dependent(2*X + Y, -Y)
True
>>> X, Y = given(Tuple(X, Y), Eq(X + Y, 3))
>>> dependent(X, Y)
True
```

## 3.24 ODE

### 3.24.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike Hint Functions (page 989), below) are intended for use by ordinary users of SymPy.

#### dsolve

```
sympy.solvers.ode.dsolve(eq, func=None, hint='default', simplify=True, ics=None, xi=None,
                         eta=None, x0=0, n=6, **kwargs)
```

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

## Examples

```
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos, symbols
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))

>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='1st_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> dsolve(eq, hint='almost_linear')
[Eq(f(x), -acos(C1/sqrt(-cos(x)**2)) + 2*pi), Eq(f(x), acos(C1/sqrt(-cos(x)**2)))]
>>> t = symbols('t')
>>> x, y = symbols('x, y', function=True)
>>> eq = (Eq(Derivative(x(t),t), 12*t*x(t) + 8*y(t)), Eq(Derivative(y(t),t), 21*x(t) + 7*t*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0 + C2*x0*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0**2, t)),
 Eq(y(t), C1*y0 + C2*y0*Integral(8*exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0**2, t) +
 exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0))]
>>> eq = (Eq(Derivative(x(t),t),x(t)*y(t)*sin(t)), Eq(Derivative(y(t),t),y(t)**2*sin(t)))
>>> dsolve(eq)
set([Eq(x(t), -exp(C1)/(C2*exp(C1) - cos(t))), Eq(y(t), -1/(C1 - cos(t)))])
```

## classify\_ode

```
sympy.solvers.ode.classify_ode(eq, func=None, dict=False, ics=None, **kwargs)
```

Returns a tuple of possible [dsolve\(\)](#) (page 984) classifications for an ODE.

The tuple is ordered so that first item is the classification that [dsolve\(\)](#) (page 984) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make [dsolve\(\)](#) (page 984) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the [dsolve\(\)](#) (page 984) docstring for different meta-hints you can use.

If `dict` is true, [classify\\_ode\(\)](#) (page 985) will return a dictionary of `hint:match` expression terms. This is intended for internal use by [dsolve\(\)](#) (page 984). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hintname` is the name of the hint without `_Integral`.

See [allhints](#) (page 989) or the [ode](#) (page 1033) docstring for a list of all supported hints that can be returned from [classify\\_ode\(\)](#) (page 985).

## Notes

These are remarks on hint names.

### \_Integral

If a classification has `_Integral` at the end, it will return the expression with an unevaluated [Integral](#) (page 551) class in it. Note that a hint may do this anyway if [integrate\(\)](#) (page 543) cannot do the integral, though just using an `_Integral` will do so much faster. Indeed, an `_Integral` hint will always be faster than its corresponding hint without `_Integral` because [integrate\(\)](#) (page 543) is an expensive routine. If [dsolve\(\)](#) (page 984) hangs, it is probably because [integrate\(\)](#) (page 87) is hanging on a tough or impossible integral. Try using an `_Integral` hint or `all_Integral` to get it return something.

Note that some hints do not have `_Integral` counterparts. This is because [integrate\(\)](#) (page 543) is not used in solving the ODE for those method. For example, `nth` order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no `nth_linear_homogeneous_constant_coeff_Integrate` hint. You can easily evaluate any unevaluated [Integral](#) (page 551)s in an expression by doing `expr.doit()`.

### Ordinals

Some hints contain an ordinal such as `1st_linear`. This is to help differentiate them from other hints, as well as from other methods that may not be implemented yet. If a hint has `nth` in it, such as the `nth_linear` hints, this means that the method used to applies to ODEs of any order.

### indep and dep

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of  $f(x)$ , then `indep` will refer to  $x$  and `dep` will refer to  $f$ .

### subs

If a hints has the word `subs` in it, it means the the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters

allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.

### `coeff`

The word `coeff` in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (`help(ode)`). This is contrast to `coefficients`, as in `undetermined_coefficients`, which refers to the common name of a method.

### `_best`

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a `_best` meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal `best` meta-hint.

## Examples

```
>>> from sympy import Function, classify_ode, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('separable', '1st_linear', '1st_homogeneous_coeff_best',
 '1st_homogeneous_coeff_subs_indep_div_dep',
 '1st_homogeneous_coeff_subs_dep_div_indep',
 '1st_power_series', 'lie_group',
 'nth_linear_constant_coeff_homogeneous',
 'separable_Integral', '1st_linear_Integral',
 '1st_homogeneous_coeff_subs_indep_div_dep_Integral',
 '1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('nth_linear_constant_coeff_undetermined_coefficients',
 'nth_linear_constant_coeff_variation_of_parameters',
 'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

## `checkodesol`

`sympy.solvers.ode.checkodesol(ode, sol, func=None, order='auto', solve_for_func=True)`

Substitutes `sol` into `ode` and checks that the result is 0.

This only works when `func` is one function, like  $f(x)$ . `sol` can be a single solution or a list of solutions. Each solution may be an `Equality` (page 120) that the solution satisfies, e.g. `Eq(f(x), C1)`, `Eq(f(x) + C1, 0)`; or simply an `Expr` (page 74), e.g. `f(x) - C1`. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the `func` argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:

1. Substitute the solution for  $f$  in the original equation. This only works if `ode` is solved for  $f$ . It will attempt to solve it first unless `solve_for_func == False`.
2. Take  $n$  derivatives of the solution, where  $n$  is the order of `ode`, and check to see if that is equal to the solution. This only works on exact ODEs.

3. Take the 1st, 2nd, ...,  $n$ th derivatives of the solution, each time solving for the derivative of  $f$  of that order (this will always be possible because  $f$  is a linear operator). Then back substitute each derivative into `ode` in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in 0, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be 0 if the first item is `True`. Note that sometimes this function will `False`, but with an expression that is identically equal to 0, instead of returning `True`. This is because `simplify()` (page 916) cannot reduce the expression to 0. If an expression returned by this function vanishes identically, then `sol` really is a solution to `ode`.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

### Examples

```
>>> from sympy import Eq, Function, checkodesol, symbols
>>> x, C1 = symbols('x,C1')
>>> f = Function('f')
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

### homogeneous\_order

`sympy.solvers.ode.homogeneous_order(eq, *symbols)`

Returns the order  $n$  if  $g$  is homogeneous and `None` if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function  $f(x, y, \dots)$  is homogeneous of order  $n$  if  $f(tx, ty, \dots) = t^n f(x, y, \dots)$ .

If the function is of two variables,  $F(x, y)$ , then  $f$  being homogeneous of any order is equivalent to being able to rewrite  $F(x, y)$  as  $G(x/y)$  or  $H(y/x)$ . This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 995) and `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 996)).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the list of symbols. If a declared function appears with different arguments than given in the list of symbols, `None` is returned.

### Examples

```
>>> from sympy import Function, homogeneous_order, sqrt
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
>>> homogeneous_order(f(x), f(x), x)
```

```
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

## infinitesimals

```
sympy.solvers.ode.infinitesimals(eq, func=None, order=None, hint='default', match=None)
```

The infinitesimal functions of an ordinary differential equation,  $\xi(x, y)$  and  $\eta(x, y)$ , are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE  $y' = f(x, y)$  would admit a Lie group  $x^* = X(x, y; \varepsilon) = x + \varepsilon\xi(x, y)$ ,  $y^* = Y(x, y; \varepsilon) = y + \varepsilon\eta(x, y)$  such that  $(y^*)' = f(x^*, y^*)$ . A change of coordinates, to  $r(x, y)$  and  $s(x, y)$ , can be performed so this Lie group becomes the translation group,  $r^* = r$  and  $s^* = s + \varepsilon$ . They are tangents to the coordinate curves of the new system.

Consider the transformation  $(x, y) \rightarrow (X, Y)$  such that the differential equation remains invariant.  $\xi$  and  $\eta$  are the tangents to the transformed coordinates  $X$  and  $Y$ , at  $\varepsilon = 0$ .

$$\left( \frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon} \right) |_{\varepsilon=0} = \xi, \left( \frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon} \right) |_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> from sympy import Function, diff, Eq, pprint
>>> from sympy.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y) # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h
... - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
>>> pprint(genform)
/d           d           \
|---(eta(x, y)) - ---((xi(x, y))*h(x, y) - eta(x, y)*--(h(x, y)) - h(x, y)*--(x
\dy          dx          /           d           2           d
i(x, y)) - xi(x, y)*--(h(x, y)) + --(eta(x, y)) = 0
dx           dx
```

Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for  $\xi$  and  $\eta$  (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, `hint` should be flagged as `all`, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to `hint`.

## References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

## Examples

```
>>> from sympy import Function, diff
>>> from sympy.solvers.ode import infinitesimals
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[{\eta(x, f(x)): exp(x**3/3), \xi(x, f(x)): 0}]
```

## checkinfsol

`sympy.solvers.ode.checkinfsol(eq, infinitesimals, func=None, order=None)`

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left( \frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where  $\eta$ , and  $\xi$  are the infinitesimals and  $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts `[{xi(x, y): inf, eta(x, y): inf}]`, corresponding to the output of the function `infinitesimals`. It returns a list of values of the form `[(True/False, sol)]` where `sol` is the value obtained after substituting the infinitesimals in the PDE. If it is `True`, then `sol` would be 0.

## 3.24.2 Hint Functions

These functions are intended for internal use by `dsolve()` (page 984) and others. Unlike User Functions (page 984), above, these are not intended for every-day use by ordinary SymPy users. Instead, functions such as `dsolve()` (page 984) should be used. Nonetheless, these functions contain useful information in their docstrings on the various ODE solving methods. For this reason, they are documented here.

### allhints

`sympy.solvers.ode.allhints = ('separable', '1st_exact', '1st_linear', 'Bernoulli', 'Riccati_special_minus2', '1st')`

This is a list of hints in the order that they should be preferred by `classify_ode()` (page 985). In general, hints earlier in the list should produce simpler solutions than those later in the list (for ODEs that fit both). For now, the order of this list is based on empirical observations by the developers of SymPy.

The hint used by `dsolve()` (page 984) for a specific ODE can be overridden (see the docstring).

In general, `_Integral` hints are grouped at the end of the list, unless there is a method that returns an unevaluable integral most of the time (which go near the end of the list anyway). `default`, `all`, `best`, and `all_Integral` meta-hints should not be included in this list, but `_best` and `_Integral` hints should be included.

### odesimp

`sympy.solvers.ode.odesimp(*args, **kwargs)`

Simplifies ODEs, including trying to solve for `func` and running `constantsimp()` (page 991).

It may use knowledge of the type of solution that the hint returns to apply additional simplifications.

It also attempts to integrate any [Integral](#) (page 551)s in the expression, if the hint is not an [\\_Integral](#) hint.

This function should have no effect on expressions returned by [dsolve\(\)](#) (page 984), as [dsolve\(\)](#) (page 984) already calls [odesimp\(\)](#) (page 989), but the individual hint functions do not call [odesimp\(\)](#) (page 989) (because the [dsolve\(\)](#) (page 984) wrapper does). Therefore, this function is designed for mainly internal use.

## Examples

```
>>> from sympy import sin, symbols, dsolve, pprint, Function
>>> from sympy.solvers.ode import odesimp
>>> x, u2, C1= symbols('x,u2,C1')
>>> f = Function('f')

>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral',
... simplify=False)
>>> pprint(eq, wrap_line=False)
      x
      -----
      f(x)
      /
      |
      | /      1 \ \
      | -|u2 + -----|
      | |      /1 \ \
      | |      sin|---|
      | |      \u2//_
      log(f(x)) = log(C1) + -----
                           2
                           u2
      |
      /
```

```
>>> pprint(odesimp(eq, f(x), 1, set([C1]),
... hint='1st_homogeneous_coeff_subs_indep_div_dep'
... ))
      x
      -----
      /f(x)\_
      tan|---|
      \2*x /
```

## constant\_renumber

`sympy.solvers.ode.constant_renumber(expr, symbolname, startnumber, endnumber)`

Renumber arbitrary constants in `expr` to have numbers 1 through  $N$  where  $N$  is `endnumber - startnumber + 1` at most. In the process, this reorders expression terms in a standard way.

This is a simple function that goes through and renames any [Symbol](#) (page 95) with a name in the form `symbolname + num` where `num` is in the range from `startnumber` to `endnumber`.

Symbols are renumbered based on `.sort_key()`, so they should be numbered roughly in the order that they appear in the final, printed expression. Note that this ordering is based in part on hashes, so it can produce different results on different machines.

The structure of this function is very similar to that of `constantsimp()` (page 991).

### Examples

```
>>> from sympy import symbols, Eq, pprint
>>> from sympy.solvers.ode import constant_renumber
>>> x, C0, C1, C2, C3, C4 = symbols('x,C:5')
```

Only constants in the given range (inclusive) are renumbered; the renumbering always starts from 1:

```
>>> constant_renumber(C1 + C3 + C4, 'C', 1, 3)
C1 + C2 + C4
>>> constant_renumber(C0 + C1 + C3 + C4, 'C', 2, 4)
C0 + 2*C1 + C2
>>> constant_renumber(C0 + 2*C1 + C2, 'C', 0, 1)
C1 + 3*C2
>>> pprint(C2 + C1*x + C3*x**2)
      2
C1*x + C2 + C3*x
>>> pprint(constant_renumber(C2 + C1*x + C3*x**2, 'C', 1, 3))
      2
C1 + C2*x + C3*x
```

### `constantsimp`

`sympy.solvers.ode.constantsimp(*args, **kwargs)`

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with `dsolve()` (page 984), and is not intended for general use.

Simplification is done by “absorbing” the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.

The symbols must all have the same name with numbers after it, for example, `C1`, `C2`, `C3`. The `symbolname` here would be ‘`C`’, the `startnumber` would be 1, and the `endnumber` would be 3. If the arbitrary constants are independent of the variable `x`, then the independent symbol would be `x`. There is no need to specify the dependent function, such as `f(x)`, because it already has the independent symbol, `x`, in it.

Because terms are “absorbed” into arbitrary constants and because constants are renumbered after simplifying, the arbitrary constants in `expr` are not necessarily equal to the ones of the same name in the returned result.

If two or more arbitrary constants are added, multiplied, or raised to the power of each other, they are first absorbed together into a single arbitrary constant. Then the new constant is combined into other terms if necessary.

Absorption of constants is done with limited assistance:

1. terms of `Add` (page 115)s are collected to try join constants so  $e^x(C_1 \cos(x) + C_2 \cos(x))$  will simplify to  $e^x C_1 \cos(x)$ ;

2. powers with exponents that are `Add` (page 115)s are expanded so  $e^{C_1+x}$  will be simplified to  $C_1 e^x$ .

Use `constant_renumber()` (page 990) to renumber constants after simplification or else arbitrary numbers on constants may appear, e.g.  $C_1 + C_3 x$ .

In rare cases, a single constant can be “simplified” into two constants. Every differential equation solution should have as many arbitrary constants as the order of the differential equation. The result here will be technically correct, but it may, for example, have  $C_1$  and  $C_2$  in an expression, when  $C_1$  is actually equal to  $C_2$ . Use your discretion in such situations, and also take advantage of the ability to use hints in `dsolve()` (page 984).

## Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, set([C1, C2, C3]))
C1*x
>>> constantsimp(C1 + 2 + x, set([C1, C2, C3]))
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, set([C1, C2, C3]))
C1 + C3*x
```

## sol\_simplicity

`sympy.solvers.ode.ode_sol_simplicity(sol, func, trysolving=True)`

Returns an extended integer representing how simple a solution to an ODE is.

The following things are considered, in order from most simple to least:

- `sol` is solved for `func`.
- `sol` is not solved for `func`, but can be if passed to `solve` (e.g., a solution returned by `dsolve(ode, func, simplify=False)`).
- If `sol` is not solved for `func`, then base the result on the length of `sol`, as computed by `len(str(sol))`.
- If `sol` has any unevaluated `Integral` (page 551)s, this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then `ode_sol_simplicity(sola, func) < ode_sol_simplicity(solb, func)`.

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

Simplicity	Return
<code>sol</code> solved for <code>func</code>	-2
<code>sol</code> not solved for <code>func</code> but can be	-1
<code>sol</code> is not solved nor solvable for <code>func</code>	<code>len(str(sol))</code>
<code>sol</code> contains an <code>Integral</code> (page 551)	<code>oo</code>

`oo` here means the SymPy infinity, which should compare greater than any integer.

If you already know `solve()` (page 1045) cannot solve `sol`, you can use `trysolving=False` to skip that step, which is the only potentially slow step. For example, `dsolve()` (page 984) with the `simplify=False` flag should do this.

If `sol` is a list of solutions, if the worst solution in the list returns `oo` it returns that, otherwise it returns `len(str(sol))`, that is, the length of the string representation of the whole list.

## Examples

This function is designed to be passed to `min` as the key argument, such as `min(listofsolutions, key=lambda i: ode_sol_simplicity(i, f(x)))`.

```
>>> from sympy import symbols, Function, Eq, tan, cos, sqrt, Integral
>>> from sympy.solvers.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')

>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x)) + f(x), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

## 1st\_exact

`sympy.solvers.ode.ode_1st_exact(eq, func, order, match)`

Solves 1st order exact ordinary differential equations.

A 1st order differential equation is called exact if it is the total differential of a function. That is, the differential equation

$$P(x, y) \partial x + Q(x, y) \partial y = 0$$

is exact if there is some function  $F(x, y)$  such that  $P(x, y) = \partial F / \partial x$  and  $Q(x, y) = \partial F / \partial y$ . It can be shown that a necessary and sufficient condition for a first order ODE to be exact is that  $\partial P / \partial y = \partial Q / \partial x$ . Then, the solution will be as given below:

```
>>> from sympy import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1 = symbols('x,y,t,x0,y0,C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(x0, t), (t, y0, y))), C1))
      x                               y
      /                               /
      |   P(t, y) dt + |   Q(x0, t) dt = C1
      |                   |
      /                   /
      x0                  y0
```

Where the first partials of  $P$  and  $Q$  exist and are continuous in a simply connected region.

A note: SymPy currently has no way to represent inert substitution on an expression, so the hint `1st_exact_Integral` will return an integral with  $dy$ . This is supposed to represent the function that you are solving for.

## References

- [http://en.wikipedia.org/wiki/Exact\\_differential\\_equation](http://en.wikipedia.org/wiki/Exact_differential_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 73

```
# indirect doctest
```

## Examples

```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

## 1st\_homogeneous\_coeff\_best

```
sympy.solvers.ode.ode_1st_homogeneous_coeff_best(eq, func, order, match)
```

Returns the best solution to an ODE from the two hints `1st_homogeneous_coeff_subs_dep_div_indep` and `1st_homogeneous_coeff_subs_indep_div_dep`.

This is as determined by `ode_sol_simplicity()` (page 992).

See the `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 996) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 995) docstrings for more information on these hints. Note that there is no `ode_1st_homogeneous_coeff_best_Integral` hint.

## References

- [http://en.wikipedia.org/wiki/Homogeneous\\_differential\\_equation](http://en.wikipedia.org/wiki/Homogeneous_differential_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

```
# indirect doctest
```

## Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_best', simplify=False))
      / 2      \
      | 3*x      |
log|----- + 1|
      | 2      |
      \f (x)    /
log(f(x)) = log(C1) - -----
                  3
```

## 1st\_homogeneous\_coeff\_subs\_dep\_div\_indep

`sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep(eq, func, order, match)`  
 Solves a 1st order differential equation with homogeneous coefficients using the substitution  $u_1 = \frac{\text{dependent variable}_i}{\text{independent variable}_i}$ .

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that  $P$  and  $Q$  are homogeneous and of the same order. A function  $F(x, y)$  is homogeneous of order  $n$  if  $F(xt, yt) = t^n F(x, y)$ . Equivalently,  $F(x, y)$  can be rewritten as  $G(y/x)$  or  $H(x/y)$ . See also the docstring of `homogeneous_order()` (page 987).

If the coefficients  $P$  and  $Q$  in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution  $y = u_1 x$  (i.e.  $u_1 = y/x$ ) will turn the differential equation into an equation separable in the variables  $x$  and  $u$ . If  $h(u_1)$  is the function that results from making the substitution  $u_1 = f(x)/x$  on  $P(x, f(x))$  and  $g(u_2)$  is the function that results from the substitution on  $Q(x, f(x))$  in the differential equation  $P(x, f(x)) + Q(x, f(x))f'(x) = 0$ , then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
 /f(x)\ /f(x)\ d
g|----| + h|----|---(f(x))
 \ x / \ x / dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
      f(x)
      -----
      x
      /
      |
      |      -h(u1)
      |----- d(u1)
      | u1*h(u1) + g(u1)
      |
      /
log(x) = C1 +
```

Where  $u_1 h(u_1) + g(u_1) \neq 0$  and  $x \neq 0$ .

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 994) and `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 996).

## References

- [http://en.wikipedia.org/wiki/Homogeneous\\_differential\\_equation](http://en.wikipedia.org/wiki/Homogeneous_differential_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

# indirect doctest

## Examples

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_indep', simplify=False))
      /      3 \
      |3*f(x)   f (x)|
log|----- + -----|
      | x      3 |
      \         x /
log(x) = log(C1) - -----
                  3
```

### 1st\_homogeneous\_coeff\_subs\_indep\_div\_dep

`sympy.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep(eq, func, order, match)`  
Solves a 1st order differential equation with homogeneous coefficients using the substitution  $u_2 = \frac{\text{independent variable}_i}{\text{dependent variable}_i}$ .

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that  $P$  and  $Q$  are homogeneous and of the same order. A function  $F(x, y)$  is homogeneous of order  $n$  if  $F(tx, yt) = t^n F(x, y)$ . Equivalently,  $F(x, y)$  can be rewritten as  $G(y/x)$  or  $H(x/y)$ . See also the docstring of `homogeneous_order()` (page 987).

If the coefficients  $P$  and  $Q$  in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution  $x = u_2y$  (i.e.  $u_2 = x/y$ ) will turn the differential equation into an equation separable in the variables  $y$  and  $u_2$ . If  $h(u_2)$  is the function that results from making the substitution  $u_2 = x/f(x)$  on  $P(x, f(x))$  and  $g(u_2)$  is the function that results from the substitution on  $Q(x, f(x))$  in the differential equation  $P(x, f(x)) + Q(x, f(x))f'(x) = 0$ , then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
      / x \ / x \ d
g|----| + h|----|---(f(x))
      \f(x)/ \f(x)/ dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
      x
      -----
      f(x)
      /
      |
      |      -g(u2)
      |      ----- d(u2)
      |      u2*g(u2) + h(u2)
      |
      /
f(x) = C1*e
```

Where  $u_2g(u_2) + h(u_2) \neq 0$  and  $f(x) \neq 0$ .

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 994) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 995).

## References

- [http://en.wikipedia.org/wiki/Homogeneous\\_differential\\_equation](http://en.wikipedia.org/wiki/Homogeneous_differential_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

# indirect doctest

## Examples

```
>>> from sympy import Function, pprint, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep',
... simplify=False))
      / 2 \
      | 3*x |
log|----- + 1|
      | 2   |
      \f (x) /
log(f(x)) = log(C1) - -----
                  3
```

## 1st\_linear

`sympy.solvers.ode.ode_1st_linear(eq, func, order, match)`

Solves 1st order linear differential equations.

These are differential equations of the form

$$\frac{dy}{dx} + P(x)y = Q(x).$$

These kinds of differential equations can be solved in a general way. The integrating factor  $e^{\int P(x) dx}$  will turn the equation into a separable equation. The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff, sin
>>> from sympy.abc import x
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x))
>>> pprint(genform)
      d
P(x)*f(x) + --(f(x)) = Q(x)
      dx
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))
      /   /           \
      |   |           |
      |   |           /   |   /
      |   |           |   |   /
      |   |           |   |   /
```

$$f(x) = \frac{C_1 + Q(x)e^{\int P(x)dx}}{e^{\int P(x)dx}}$$

## References

- [http://en.wikipedia.org/wiki/Linear\\_differential\\_equation#First\\_order\\_equation](http://en.wikipedia.org/wiki/Linear_differential_equation#First_order_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 92

# indirect doctest

## Examples

```
>>> f = Function('f')
>>> pprint(dsolve(Eq(x*diff(f(x), x) - f(x), x**2*sin(x)),
... f(x), '1st_linear'))
f(x) = x*(C1 - cos(x))
```

## Bernoulli

`sympy.solvers.ode.ode_Bernoulli(eq, func, order, match)`

Solves Bernoulli differential equations.

These are equations of the form

$$\frac{dy}{dx} + P(x)y = Q(x)y^n, n \neq 1.$$

The substitution  $w = 1/y^{1-n}$  will transform an equation of this form into one that is linear (see the docstring of `ode_1st_linear()` (page 997)). The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform)
      d           n
P(x)*f(x) + --(f(x)) = Q(x)*f (x)
      dx
```

```
>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'))
```

$$f(x) = \frac{C_1 + (-1 + n)*\int \frac{P(x)}{(1 - n)}dx - \int \frac{Q(x)}{(-1 + n)}dx}{e^{\int \frac{P(x)}{1 - n}dx}}$$

Note that the equation is separable when  $n = 1$  (see the docstring of `ode_separable()` (page 1004)).

```
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
... hint='separable_Integral'))
f(x)
/
| 1      / 
| - dy = C1 + | (-P(x) + Q(x)) dx
| y      /
|/
|
```

## References

- [http://en.wikipedia.org/wiki/Bernoulli\\_differential\\_equation](http://en.wikipedia.org/wiki/Bernoulli_differential_equation)
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 95

# indirect doctest

## Examples

```
>>> from sympy import Function, dsolve, Eq, pprint, log
>>> from sympy.abc import x
>>> f = Function('f')

>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2),
... f(x), hint='Bernoulli'))
1
f(x) = -----
/      log(x)  1 \
x*C1 + ----- + -|
 \      x       x/
```

## Liouville

`sympy.solvers.ode.ode_Liouville(eq, func, order, match)`

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

$$\frac{d^2y}{dx^2} + g(y) \left( \frac{dy}{dx} \right)^2 + h(x) \frac{dy}{dx}.$$

The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = Eq(diff(f(x), x, x) + g(f(x))*diff(f(x), x)**2 +
... h(x)*diff(f(x), x), 0)
>>> pprint(genform)
          2
        /d      \      2
                  d      d
```

```

g(f(x))*|-(f(x))| + h(x)*|-(f(x))| + |-(f(x))| = 0
          \dx           / dx           2
                                         dx

>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))
               f(x)
      /   /   /
      |   |   | | |
      |   |   |
      | - | h(x) dx |   | g(y) dy |
      |   |   |   |
      |   |   |   |
      |   |   |   |
C1 + C2* e   dx + e   dy = 0
      |   |
      |   |
      |   |
      |   |

```

## References

- Goldstein and Braun, “Advanced Methods for the Solution of Differential Equations”, pp. 98
  - <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville>

```
# indirect doctest
```

## Examples

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(diff(f(x)), x, x) + diff(f(x), x)**2/f(x) +
... diff(f(x), x)/x, f(x), hint='Liouville'))
-----  -----
[f(x) = -\sqrt{C1 + C2*log(x)}, f(x) = \sqrt{C1 + C2*log(x)} ]
```

## Riccati\_special\_minus2

```
sympy.solvers.ode.ode_Riccati_special_minus2(eq, func, order, match)
```

The general Riccati equation has the form

$$dy/dx = f(x)y^2 + g(x)y + h(x).$$

While it does not have a general solution [1], the “special” form,  $dy/dx = ay^2 - bx^c$ , does have solutions in many cases [2]. This routine returns a solution for  $a(dy/dx) = by^2 + cy/x + d/x^2$  that is obtained by using a suitable change of variables to reduce it to the special form and is valid when neither  $a$  nor  $b$  are zero and either  $c$  or  $d$  is zero.

```
>>> from sympy.abc import x, y, a, b, c, d
>>> from sympy.solvers.ode import dsolve, checkodesol
>>> from sympy import pprint, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = a*y.diff(x) - (b*y**2 + c*y/x + d/x**2)
>>> sol = dsolve(genform, y)
>>> pprint(sol, wrap_line=False)
```

```

f(x) = -----
          /   2
        -| a + c - \sqrt(4*b*d - (a + c) *tan[C1 + -----
          \           |   2
                  \sqrt(4*b*d - (a + c) *log(x))]|]
          2*a //|

```

>>> checkodesol(genform, sol, order=1)[0]

True

## References

- 1.<http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Riccati>
  - 2.<http://eqworld.ipmnet.ru/en/solutions/ode/ode0106.pdf> - <http://eqworld.ipmnet.ru/en/solutions/ode/ode0123.pdf>

### **nth\_linear\_constant\_coeff\_homogeneous**

```
sympy.solvers.ode.ode_nth_linear_constant_coeff_homogeneous(eq, func, order, match, re-
turns='sol')
```

Solves an  $n$ th order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation  $a_n m^n + a_{n-1} m^{n-1} + \dots + a_1 m + a_0 = 0$ . The solution will then be the sum of  $C_n x^i e^{rx}$  terms, for each where  $C_n$  is an arbitrary constant,  $r$  is a root of the characteristic equation and  $i$  is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms  $C_1 e^{3x} + C_2 x e^{3x}$ ). The exponential is usually expanded for complex roots using Euler's equation  $e^{Ix} = \cos(x) + I \sin(x)$ . Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as  $e^{ax} (C_1 \cos(bx) + C_2 \sin(bx))$ .

If SymPy cannot find exact roots to the characteristic equation, a `RootOf` (page 722) instance will be returned instead.

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(f(x).diff(x, 5) + 10*f(x).diff(x) - 2*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous')
...
Eq(f(x), C1*exp(x*RootOf(_x**5 + 10*_x - 2, 0)) +
C2*exp(x*RootOf(_x**5 + 10*_x - 2, 1)) +
C3*exp(x*RootOf(_x**5 + 10*_x - 2, 2)) +
C4*exp(x*RootOf(_x**5 + 10*_x - 2, 3)) +
C5*exp(x*RootOf(_x**5 + 10*_x - 2, 4)))
```

Note that because this method does not involve integration, there is no `nth_linear_constant_coeff_homogeneous_Integral` hint.

The following is for internal use:

- `returns = 'sol'` returns the solution to the ODE.

- `returns = 'list'` returns a list of linearly independent solutions, for use with non homogeneous solution methods like variation of parameters and undetermined coefficients. Note that, though the solutions should be linearly independent, this function does not explicitly check that. You can do `assert simplify(wronskian(sollist)) != 0` to check for linear independence. Also, `assert len(sollist) == order` will need to pass.
- `returns = 'both'`, return a dictionary `{'sol': <solution to ODE>, 'list': <list of linearly independent solutions>}`.

## References

- [http://en.wikipedia.org/wiki/Linear\\_differential\\_equation](http://en.wikipedia.org/wiki/Linear_differential_equation) section: Nonhomogeneous\_equation\_with\_constant\_coefficients
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 211

# indirect doctest

## Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) -
... 2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous'))
      x          -2*x
f(x) = (C1 + C2*x)*e  + (C3*sin(x) + C4*cos(x))*e
```

## `nth_linear_constant_coeff_undetermined_coefficients`

```
sympy.solvers.ode.ode_nth_linear_constant_coeff_undetermined_coefficients(eq, func, order,
                           match)
```

Solves an  $n$ th order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x),$$

where  $P(x)$  is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form  $ax^i e^{bx} \sin(cx + d)$  or  $ax^i e^{bx} \cos(cx + d)$ , where  $i$  is a non-negative integer and  $a, b, c$ , and  $d$  are constants. For example any polynomial in  $x$ , functions like  $x^2 e^{2x}$ ,  $x \sin(x)$ , and  $e^x \cos(x)$  can all be used. Products of  $\sin$ 's and  $\cos$ 's have a finite number of derivatives, because they can be expanded into  $\sin(ax)$  and  $\cos(bx)$  terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert  $\sin^2(x)$  into  $(1 + \cos(2x))/2$  to properly apply the method of undetermined coefficients on it.

This method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for each term will be a system of linear equations, which are solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient  $x$  to make them linearly independent.

## References

- [http://en.wikipedia.org/wiki/Method\\_of\\_undetermined\\_coefficients](http://en.wikipedia.org/wiki/Method_of_undetermined_coefficients)
  - M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 221
- # indirect doctest

## Examples

```
>>> from sympy import Function, dsolve, pprint, exp, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) -
... 4*exp(-x)*x**2 + cos(2*x), f(x),
... hint='nth_linear_constant_coeff_undetermined_coefficients'))
      4\
      |      -x    4*sin(2*x)   3*cos(2*x)
f(x) = |C1 + C2*x + --|*e   - ----- + -----
      \            3 /           25           25
```

## `nth_linear_constant_coeff_variation_of_parameters`

```
sympy.solvers.ode.ode_nth_linear_constant_coeff_variation_of_parameters(eq, func, order,
match)
```

Solves an  $n$ th order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1f'(x) + a_0f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{x=1}^n c_i(x)y_i(x),$$

where  $y_i$  is the  $i$ th solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by

$$\sum_{x=1}^n \left( \int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where  $W(x)$  is the Wronskian of the fundamental system (the system of  $n$  linearly independent solutions to the homogeneous equation), and  $W_i(x)$  is the Wronskian of the fundamental system with the  $i$ th column replaced with  $[0, 0, \dots, 0, P(x)]$ .

This method is general enough to solve any  $n$ th order inhomogeneous linear differential equation with constant coefficients, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it doesn't use integration, making it faster and more reliable.

Warning, using `simplify=False` with ‘`nth_linear_constant_coeff_variation_of_parameters`’ in `dsolve()` (page 984) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use `simplify=False` with ‘`nth_linear_constant_coeff_variation_of_parameters_Integral`’ for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

## References

- [http://en.wikipedia.org/wiki/Variation\\_of\\_parameters](http://en.wikipedia.org/wiki/Variation_of_parameters)
- <http://planetmath.org/VariationOfParameters>
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 233

```
# indirect doctest
```

## Examples

```
>>> from sympy import Function, dsolve, pprint, exp, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) +
... 3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
... hint='nth_linear_constant_coeff_variation_of_parameters'))
      3
      |           2   x *(6*log(x) - 11)| x
f(x) = |C1 + C2*x + C3*x + -----|*e
      \           36          /
```

## separable

```
sympy.solvers.ode.ode_separable(eq, func, order, match)
```

Solves separable 1st order differential equations.

This is any differential equation that can be written as  $P(y)\frac{dy}{dx} = Q(x)$ . The solution can then just be found by rearranging terms and integrating:  $\int P(y) dy = \int Q(x) dx$ . This hint uses `sympy.simplify.separatevars()` (page 920) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. `separatevars()` (page 920) is smart enough to do most expansion and factoring necessary to convert a separable equation  $F(x, y)$  into the proper form  $P(x) \cdot Q(y)$ . The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
      d
a(x)*b(f(x))**--(f(x)) = c(x)*d(f(x))
      dx
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
      f(x)
      /
      |           |
      | b(y)       | c(x)
      | ---- dy = C1 + | ---- dx
```

$$\frac{d(y)}{a(x)}$$

## References

- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 52

```
# indirect doctest
```

## Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
... hint='separable', simplify=False))
      / 2           \
      \             2
log\3*f (x) - 1/      x
----- = C1 + --
      6                  2
```

## almost\_linear

```
sympy.solvers.ode.ode_almost_linear(eq, func, order, match)
```

Solves an almost-linear differential equation.

The general form of an almost linear differential equation is

$$f(x)g(y)y + k(x)l(y) + m(x) = 0 \text{ where } l'(y) = g(y).$$

This can be solved by substituting  $l(y) = u(y)$ . Making the given substitution reduces it to a linear differential equation of the form  $u' + P(x)u + Q(x) = 0$ .

The general solution is

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, y, n
>>> f, g, k, l = map(Function, ['f', 'g', 'k', 'l'])
>>> genform = Eq(f(x)*(l(y).diff(y)) + k(x)*l(y) + g(x))
>>> pprint(genform)
      d
f(x)*--(l(y)) + g(x) + k(x)*l(y) = 0
      dy
>>> pprint(dsolve(genform, hint = 'almost_linear'))
      /   //  -y*g(x)          \\
      |   ||  -----    for k(x) = 0 ||
      |   ||  f(x)          ||  -y*k(x)
      |   ||  y*k(x)        ||  -----
      |   ||  f(x)          ||  ||*e
      |   || -g(x)*e       ||  ||
      |   ||----- otherwise ||
```

|      ||      k(x)      ||  
\\      //

See also:

[sympy.solvers.ode.ode\\_1st\\_linear\(\)](#) (page 997)

## References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

## Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), (C1 - Ei(x))*exp(-x))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
           -x
f(x) = (C1 - Ei(x))*e
```

## linear\_coefficients

[sympy.solvers.ode.ode\\_linear\\_coefficients\(\)](#) (eq, func, order, match)  
Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

$$y' + F \left( \frac{a_1 x + b_1 y + c_1}{a_2 x + b_2 y + c_2} \right) = 0,$$

where  $a_1, b_1, c_1, a_2, b_2, c_2$  are constants and  $a_1 b_2 - a_2 b_1 \neq 0$ .

This can be solved by substituting:

$$\begin{aligned} x &= x' + \frac{b_2 c_1 - b_1 c_2}{a_2 b_1 - a_1 b_2} \\ y &= y' + \frac{a_1 c_2 - a_2 c_1}{a_2 b_1 - a_1 b_2}. \end{aligned}$$

This substitution reduces the equation to a homogeneous differential equation.

See also:

[sympy.solvers.ode.ode\\_1st\\_homogeneous\\_coeff\\_best\(\)](#) (page 994),  
[sympy.solvers.ode.ode\\_1st\\_homogeneous\\_coeff\\_subs\\_indep\\_div\\_dep\(\)](#) (page 996),  
[sympy.solvers.ode.ode\\_1st\\_homogeneous\\_coeff\\_subs\\_dep\\_div\\_indep\(\)](#) (page 995)

## References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

## Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) - 1)]
>>> pprint(dsolve(eq, hint='linear_coefficients'))
      / 2      / 2
      -----  ----- 
[f(x) = -x - \sqrt{C1 + 7*x } - 1, f(x) = -x + \sqrt{C1 + 7*x } - 1]
```

### separable\_reduced

`sympy.solvers.ode.ode_separable_reduced(eq, func, order, match)`  
Solves a differential equation that can be reduced to the separable form.

The general form of this equation is

$$y' + (y/x)H(x^n y) = 0.$$

This can be solved by substituting  $u(y) = x^n y$ . The equation then reduces to the separable form  $\frac{u'}{u(\text{power}-H(u))} - \frac{1}{x} = 0$ .

The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)
      / n      \
d      f(x)*g\x *f(x)/
--(f(x)) + -----
dx          x
>>> pprint(dsolve(genform, hint='separable_reduced'))
      n
x *f(x)
  /
  |      1
  |  ----- dy = C1 + log(x)
  |  y*(n - g(y))
  |
  /
```

See also:

`sympy.solvers.ode.ode_separable()` (page 1004)

## References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

## Examples

```
>>> from sympy import Function, Derivative, pprint
>>> from sympy.solvers.ode import dsolve, classify_ode
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (-sqrt(C1*x**2 + 1) + 1)/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/x)]
>>> pprint(dsolve(eq, hint='separable_reduced'))
-----      -----
/ 2          / 2
- \sqrt{C1*x  + 1} + 1      \sqrt{C1*x  + 1} + 1
[f(x) = -----, f(x) = -----]
```

## lie\_group

`sympy.solvers.ode.ode_lie_group(eq, func, order, match)`

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate given system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE is quadrature and can be solved easily. It makes use of the `sympy.solvers.ode.infinitesimals()` (page 988) function which returns the infinitesimals of the transformation.

The coordinates  $r$  and  $s$  can be found by solving the following Partial Differential Equations.

$$\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} = 0$$

$$\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y) \frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y) \frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting  $r$  and  $s$  in terms of  $x$  and  $y$  again.

## References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

## Examples

```
>>> from sympy import Function, dsolve, Eq, exp, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
... hint='lie_group'))
      / 2\ 2
      | x | -x
f(x) = |C1 + --|*e
      \ 2 /
```

## 1st\_power\_series

`sympy.solvers.ode.ode_1st_power_series(eq, func, order, match)`

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation  $\frac{dy}{dx} = h(x, y)$ , a power series solution exists at a point  $x = x_0$  if  $h(x, y)$  is analytic at  $x_0$ . The solution is given by

$$y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},$$

where  $y(x_0) = b$  is the value of  $y$  at the initial value of  $x_0$ . To compute the values of the  $F_n(x_0, b)$  the following algorithm is followed, until the required number of terms are generated.

$$\begin{aligned} 1.F_1 &= h(x_0, b) \\ 2.F_{n+1} &= \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1 \end{aligned}$$

## References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

## Examples

```
>>> from sympy import Function, Derivative, pprint, exp
>>> from sympy.solvers.ode import dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
      3      4      5
      C1*x      C1*x      C1*x      / 6\
f(x) = C1 + C1*x - ----- + ----- + ----- + 0\x /
```

## 2nd\_power\_series\_ordinary

`sympy.solvers.ode.ode_2nd_power_series_ordinary(eq, func, order, match)`

Gives a power series solution to a second order homogeneous differential equation with polynomial

coefficients at an ordinary point. A homogenous differential equation is of the form

$$P(x) \frac{d^2y}{dx^2} + Q(x) \frac{dy}{dx} + R(x) = 0$$

For simplicity it is assumed that  $P(x)$ ,  $Q(x)$  and  $R(x)$  are polynomials, it is sufficient that  $\frac{Q(x)}{P(x)}$  and  $\frac{R(x)}{P(x)}$  exists at  $x_0$ . A recurrence relation is obtained by substituting  $y$  as  $\sum_{n=0}^{\infty} a_n x^n$ , in the differential equation, and equating the nth term. Using this relation various terms can be generated.

## References

- <http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx>
- George E. Simmons, “Differential Equations with Applications and Historical Notes”, p.p 176 - 184

## Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x, y
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
      / 4      2      \      / 2      \
      |x      x      |      | x      |      / 6 \
f(x) = C2*|--- - --- + 1| + C1*x*|--- + 1| + 0\x / \
      \24      2      /      \ 6      /
```

## 2nd\_power\_series\_regular

```
sympy.solvers.ode.ode_2nd_power_series_regular(eq, func, order, match)
```

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at a regular point. A second order homogenous differential equation is of the form

$$P(x) \frac{d^2y}{dx^2} + Q(x) \frac{dy}{dx} + R(x) = 0$$

A point is said to regular singular at  $x_0$  if  $x - x_0 \frac{Q(x)}{P(x)}$  and  $(x - x_0)^2 \frac{R(x)}{P(x)}$  are analytic at  $x_0$ . For simplicity  $P(x)$ ,  $Q(x)$  and  $R(x)$  are assumed to be polynomials. The algorithm for finding the power series solutions is:

- 1.Try expressing  $(x - x_0)P(x)$  and  $((x - x_0)^2)Q(x)$  as power series solutions about  $x_0$ . Find  $p_0$  and  $q_0$  which are the constants of the power series expansions.
- 2.Solve the indicial equation  $f(m) = m(m - 1) + m * p_0 + q_0$ , to obtain the roots  $m_1$  and  $m_2$  of the indicial equation.
- 3.If  $m_1 - m_2$  is a non integer there exists two series solutions. If  $m_1 = m_2$ , there exists only one solution. If  $m_1 - m_2$  is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form  $x^m \sum_{n=0}^{\infty} a_n x^n$ . The coefficients are determined by the following recurrence relation.  $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$ . For the case in which  $m_1 - m_2$  is an integer, it can be seen from the recurrence relation that for the lower root  $m$ , when  $n$  equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

**References**

- George E. Simmons, “Differential Equations with Applications and Historical Notes”, p.p 176 - 184

**Examples**

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x, y
>>> f = Function("f")
>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq))
      /   6   4   2   \
      | x   x   x   |
      |           \ 720  24  2   /   / 6\
C1*|--- + --- + --- + 1| + -----
      |120   6   /           x
f(x) = C2*|--- - --- + 1| + ----- + 0\x /
```

### 3.24.3 Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

**abaco1\_simple**

```
sympy.solvers.ode.lie_heuristic_abaco1_simple(match, comp=False)
```

The first heuristic uses the following four sets of assumptions on  $\xi$  and  $\eta$

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption  $\xi = 0$  and  $\eta$  to be a function of  $x$ , the PDE

$$\frac{\partial \eta}{\partial x} + \left( \frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to  $f'(x) - f \frac{\partial h}{\partial y} = 0$ . If  $\frac{\partial h}{\partial y}$  is a function of  $x$ , then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

**References**

- E.S Cheb-Terrab, L.G.S Duarte and L.A,C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

## abaco1\_product

```
sympy.solvers.ode.lie_heuristic_abaco1_product(match, comp=False)
```

The second heuristic uses the following two assumptions on  $\xi$  and  $\eta$

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if  $\frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h)$  is separable in  $x$  and  $y$ , then the separated factors containing  $x$  is  $f(x)$ , and  $g(y)$  is obtained by

$$e^{\int f \frac{\partial}{\partial x} \left( \frac{1}{f*h} \right) dy}$$

provided  $f \frac{\partial}{\partial x} \left( \frac{1}{f*h} \right)$  is a function of  $y$  only.

The second assumption holds good if  $\frac{dy}{dx} = h(x, y)$  is rewritten as  $\frac{dy}{dx} = \frac{1}{h(y,x)}$  and the same properties of the first assumption satisfies. After obtaining  $f(x)$  and  $g(y)$ , the coordinates are again interchanged, to get  $\eta$  as  $f(x) * g(y)$

### References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

## bivariate

```
sympy.solvers.ode.lie_heuristic_bivariate(match, comp=False)
```

The third heuristic assumes the infinitesimals  $\xi$  and  $\eta$  to be bi-variate polynomials in  $x$  and  $y$ . The assumption made here for the logic below is that  $h$  is a rational function in  $x$  and  $y$  though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

### References

- Lie Groups and Differential Equations pp. 327 - pp. 329

## chi

```
sympy.solvers.ode.lie_heuristic_chi(match, comp=False)
```

The aim of the fourth heuristic is to find the function  $\chi(x, y)$  that satisfies the PDE  $\frac{d\chi}{dx} + h \frac{d\chi}{dx} - \frac{\partial h}{\partial y} \chi = 0$ .

This assumes  $\chi$  to be a bivariate polynomial in  $x$  and  $y$ . By intuition,  $h$  should be a rational function in  $x$  and  $y$ . The method used here is to substitute a general binomial for  $\chi$  up to a certain maximum degree is reached. The coefficients of the polynomials, are calculated by collecting terms of the same order in  $x$  and  $y$ .

After finding  $\chi$ , the next step is to use  $\eta = \xi * h + \chi$ , to determine  $\xi$  and  $\eta$ . This can be done by dividing  $\chi$  by  $h$  which would give  $-\xi$  as the quotient and  $\eta$  as the remainder.

## References

- E.S Cheb-Terrab, L.G.S Duarte and L.A,C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

### abaco2\_similar

`sympy.solvers.ode.lie_heuristic_abaco2_similar(match, comp=False)`

This heuristic uses the following two assumptions on  $\xi$  and  $\eta$

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1. First  $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial y^2}}$  is calculated. Let us say this value is A

2. If this is constant, then  $h$  is matched to the form  $A(x) + B(x)e^{\frac{y}{C}}$  then,  $\frac{e^{\int \frac{A(x)}{C} dx}}{B(x)}$  gives  $f(x)$  and  $A(x) * f(x)$  gives  $g(x)$

3. Otherwise  $\frac{\frac{\partial A}{\partial X}}{\frac{\partial A}{\partial Y}} = \gamma$  is calculated. If

a]  $\gamma$  is a function of  $x$  alone

b]  $\frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) - \frac{\partial h}{\partial x}}{h + \gamma} = G$  is a function of  $x$  alone. then,  $e^{\int G dx}$  gives  $f(x)$  and  $-\gamma * f(x)$  gives  $g(x)$

The second assumption holds good if  $\frac{dy}{dx} = h(x, y)$  is rewritten as  $\frac{dy}{dx} = \frac{1}{h(y,x)}$  and the same properties of the first assumption satisfies. After obtaining  $f(x)$  and  $g(x)$ , the coordinates are again interchanged, to get  $\xi$  as  $f(x^*)$  and  $\eta$  as  $g(y^*)$

## References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

### function\_sum

`sympy.solvers.ode.lie_heuristic_function_sum(match, comp=False)`

This heuristic uses the following two assumptions on  $\xi$  and  $\eta$

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y} \left[ \left( h \frac{\partial^2}{\partial x^2} (h^{-1}) \right)^{-1} \right]$$

is separable in  $x$  and  $y$ ,

1. The separated factors containing  $y$  is  $\frac{\partial g}{\partial y}$ . From this  $g(y)$  can be determined.

2.The separated factors containing  $x$  is  $f''(x)$ .

3. $h \frac{\partial^2}{\partial x^2}(h^{-1})$  equals  $\frac{f''(x)}{f(x)+g(y)}$ . From this  $f(x)$  can be determined.

The second assumption holds good if  $\frac{dy}{dx} = h(x, y)$  is rewritten as  $\frac{dy}{dx} = \frac{1}{h(y,x)}$  and the same properties of the first assumption satisfies. After obtaining  $f(x)$  and  $g(y)$ , the coordinates are again interchanged, to get  $\eta$  as  $f(x) + g(y)$ .

For both assumptions, the constant factors are separated among  $g(y)$  and  $f''(x)$ , such that  $f''(x)$  obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

## References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

### abaco2\_unique\_unknown

```
sympy.solvers.ode.lie_heuristic_abaco2_unique_unknown(match, comp=False)
```

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1.A list of all functions and non-integer powers containing  $x$  and  $y$

2.Loop over each element  $f$  in the list, find  $\frac{\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial y}} = R$

If it is separable in  $x$  and  $y$ , let  $X$  be the factors containing  $x$ . Then

a] Check if  $\xi = X$  and  $\eta = -\frac{X}{R}$  satisfy the PDE. If yes, then return  $\xi$  and  $\eta$

b] Check if  $\xi = \frac{-R}{X}$  and  $\eta = -\frac{1}{X}$  satisfy the PDE. If yes, then return  $\xi$  and  $\eta$

If not, then check if

a]  $\xi = -R, \eta = 1$

b]  $\xi = 1, \eta = -\frac{1}{R}$

are solutions.

## References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

### abaco2\_unique\_general

```
sympy.solvers.ode.lie_heuristic_abaco2_unique_general(match, comp=False)
```

This heuristic finds if infinitesimals of the form  $\eta = f(x), \xi = g(y)$  without making any assumptions on  $h$ .

The complete sequence of steps is given in the paper mentioned below.

## References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

**linear**

```
sympy.solvers.ode.lie_heuristic_linear(match, comp=False)
```

This heuristic assumes

$$1. \xi = ax + by + c \text{ and}$$

$$2. \eta = fx + gy + h$$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that  $h$  need not be a rational function in this case.

**References**

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

**3.24.4 System of ODEs**

These functions are intended for internal use by `dsolve()` (page 984) for system of differential equations.

**system\_of\_odes\_linear\_2eq\_order1\_type1**

```
sympy.solvers.ode.linear_2eq_order1_type1(x, y, t, r)
```

It is classified under system of two linear homogeneous first-order constant-coefficient ordinary differential equations.

The equations which come under this type are

$$x' = ax + by,$$

$$y' = cx + dy$$

The characteristics equation is written as

$$\lambda^2 + (a + d)\lambda + ad - bc = 0$$

and its discriminant is  $D = (a - d)^2 + 4bc$ . There are several cases

1. Case when  $ad - bc \neq 0$ . The origin of coordinates,  $x = y = 0$ , is the only stationary point; it is - a node if  $D = 0$  - a node if  $D > 0$  and  $ad - bc > 0$  - a saddle if  $D > 0$  and  $ad - bc < 0$  - a focus if  $D < 0$  and  $a + d \neq 0$  - a centre if  $D < 0$  and  $a + d \neq 0$ .

1.1. If  $D > 0$ . The characteristic equation has two distinct real roots  $\lambda_1$  and  $\lambda_2$ . The general solution of the system in question is expressed as

$$x = C_1 be^{\lambda_1 t} + C_2 be^{\lambda_2 t}$$

$$y = c_1(\lambda_1 - a)e^{\lambda_1 t} + c_2(\lambda_2 - a)e^{\lambda_2 t}$$

where  $C_1$  and  $C_2$  being arbitrary constants

1.2. If  $D < 0$ . The characteristics equation has two conjugate roots,  $\lambda_1 = \sigma + i\beta$  and  $\lambda_2 = \sigma - i\beta$ . The general solution of the system is given by

$$x = be^{\sigma t}(C_1 \sin(\beta t) + C_2 \cos(\beta t))$$

$$y = e^{\sigma t}([( \sigma - a)C_1 - \beta C_2] \sin(\beta t) + [\beta C_1 + (\sigma - a)C_2 \cos(\beta t)])$$

1.3. If  $D = 0$  and  $a \neq d$ . The characteristic equation has two equal roots,  $\lambda_1 = \lambda_2$ . The general solution of the system is written as

$$x = 2b(C_1 + \frac{C_2}{a-d} + C_2 t)e^{\frac{a+d}{2}t}$$

$$y = [(d-a)C_1 + C_2 + (d-a)C_2 t]e^{\frac{a+d}{2}t}$$

1.4. If  $D = 0$  and  $a = d \neq 0$  and  $b = 0$

$$x = C_1 e^{at}, y = (cC_1 t + C_2) e^{at}$$

1.5. If  $D = 0$  and  $a = d \neq 0$  and  $c = 0$

$$x = (bC_1 t + C_2) e^{at}, y = C_1 e^{at}$$

2. Case when  $ad - bc = 0$  and  $a^2 + b^2 > 0$ . The whole straight line  $ax + by = 0$  consists of singular points. The orginal system of differential equaitons can be rewritten as

$$x' = ax + by, y' = k(ax + by)$$

2.1 If  $a + bk \neq 0$ , solution will be

$$x = bC_1 + C_2 e^{(a+bk)t}, y = -aC_1 + kC_2 e^{(a+bk)t}$$

2.2 If  $a + bk = 0$ , solution will be

$$x = C_1(bkt - 1) + bC_2 t, y = k^2 bC_1 t + (bk^2 t + 1)C_2$$

## system\_of\_odes\_linear\_2eq\_order1\_type2

`sympy.solvers.ode.linear_2eq_order1_type2(x, y, t, r)`

The equations of this type are

$$x' = ax + by + k1, y' = cx + dy + k2$$

The general solution og this system is given by sum of its particular solution and the general solution of the corresponding homogeneous system is obtained from type1.

1. When  $ad - bc \neq 0$ . The particular solution will be  $x = x_0$  and  $y = y_0$  where  $x_0$  and  $y_0$  are determined by solving linear system of equations

$$ax_0 + by_0 + k1 = 0, cx_0 + dy_0 + k2 = 0$$

2. When  $ad - bc = 0$  and  $a^2 + b^2 > 0$ . In this case, the system of equation becomes

$$x' = ax + by + k_1, y' = k(ax + by) + k_2$$

2.1 If  $\sigma = a + bk \neq 0$ , particular solution is given by

$$x = b\sigma^{-1}(c_1k - c_2)t - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + (c_2 - c_1k)t$$

2.2 If  $\sigma = a + bk = 0$ , particular solution is given by

$$x = \frac{1}{2}b(c_2 - c_1k)t^2 + c_1t$$

$$y = kx + (c_2 - c_1k)t$$

### system\_of\_odes\_linear\_2eq\_order1\_type3

`sympy.solvers.ode.linear_2eq_order1_type3(x, y, t, r)`

The equations of this type of ode are

$$x' = f(t)x + g(t)y$$

$$y' = g(t)x + f(t)y$$

The solution of such equations is given by

$$x = e^F(C_1e^G + C_2e^{-G}), y = e^F(C_1e^G - C_2e^{-G})$$

where  $C_1$  and  $C_2$  are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

### system\_of\_odes\_linear\_2eq\_order1\_type4

`sympy.solvers.ode.linear_2eq_order1_type4(x, y, t, r)`

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = -g(t)x + f(t)y$$

The solution is given by

$$x = F(C_1 \cos(G) + C_2 \sin(G)), y = F(-C_1 \sin(G) + C_2 \cos(G))$$

where  $C_1$  and  $C_2$  are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

**system\_of\_odes.linear\_2eq\_order1\_type5**

```
sympy.solvers.ode.linear_2eq_order1_type5(x, y, t, r)
```

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = ag(t)x + [f(t) + bg(t)]y$$

The transformation of

$$x = e^{\int f(t) dt} u, y = e^{\int f(t) dt} v, T = \int g(t) dt$$

leads to a system of constant coefficient linear differential equations

$$u'(T) = v, v'(T) = au + bv$$

**system\_of\_odes.linear\_2eq\_order1\_type6**

```
sympy.solvers.ode.linear_2eq_order1_type6(x, y, t, r)
```

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y$$

This is solved by first multiplying the first equation by  $-a$  and adding it to the second equation to obtain

$$y' - ax' = -ah(t)(y - ax)$$

Setting  $U = y - ax$  and integrating the equation we arrive at

$$y - ax = C_1 e^{-a \int h(t) dt}$$

and on substituting the value of  $y$  in first equation give rise to first order ODEs. After solving for  $x$ , we can obtain  $y$  by substituting the value of  $x$  in second equation.

**system\_of\_odes.linear\_2eq\_order1\_type7**

```
sympy.solvers.ode.linear_2eq_order1_type7(x, y, t, r)
```

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = h(t)x + p(t)y$$

Differentiating the first equation and substituting the value of  $y$  from second equation will give a second-order linear equation

$$gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0$$

This above equation can be easily integrated if following conditions are satisfied.

1.  $f'gp - g^2h + fg' - f'g = 0$
2.  $f'gp - g^2h + fg' - f'g = ag, fg + gp + g' = bg$

If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as  $x = x_0(t)$  and  $y = y_0(t)$ . Then the general solution is expressed as

$$x = C_1x_0(t) + C_2x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt$$

$$y = C_1y_0(t) + C_2\left[\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt\right]$$

where  $C_1$  and  $C_2$  are arbitrary constants and

$$F(t) = e^{\int f(t) dt}, P(t) = e^{\int p(t) dt}$$

### system\_of\_odes.linear\_2eq\_order2\_type1

`sympy.solvers.ode.linear_2eq_order2_type1(x, y, t, r)`

System of two constant-coefficient second-order linear homogeneous differential equations

$$x'' = ax + by$$

$$y'' = cx + dy$$

The characteristic equation for above equations

$$\lambda^4 - (a + d)\lambda^2 + ad - bc = 0$$

whose discriminant is  $D = (a - d)^2 + 4bc \neq 0$

1. When  $ad - bc \neq 0$

1.1. If  $D \neq 0$ . The characteristic equation has four distinct roots,  $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ . The general solution of the system is

$$x = C_1be^{\lambda_1 t} + C_2be^{\lambda_2 t} + C_3be^{\lambda_3 t} + C_4be^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 - a)e^{\lambda_1 t} + C_2(\lambda_2^2 - a)e^{\lambda_2 t} + C_3(\lambda_3^2 - a)e^{\lambda_3 t} + C_4(\lambda_4^2 - a)e^{\lambda_4 t}$$

where  $C_1, \dots, C_4$  are arbitrary constants.

1.2. If  $D = 0$  and  $a \neq d$ :

$$x = 2C_1(bt + \frac{2bk}{a-d})e^{\frac{kt}{2}} + 2C_2(bt + \frac{2bk}{a-d})e^{-\frac{kt}{2}} + 2bC_3te^{\frac{kt}{2}} + 2bC_4te^{-\frac{kt}{2}}$$

$$y = C_1(d - a)te^{\frac{kt}{2}} + C_2(d - a)te^{-\frac{kt}{2}} + C_3[(d - a)t + 2k]e^{\frac{kt}{2}} + C_4[(d - a)t - 2k]e^{-\frac{kt}{2}}$$

where  $C_1, \dots, C_4$  are arbitrary constants and  $k = \sqrt{2(a + d)}$

1.3. If  $D = 0$  and  $a = d \neq 0$  and  $b = 0$ :

$$x = 2\sqrt{a}C_1e^{\sqrt{a}t} + 2\sqrt{a}C_2e^{-\sqrt{a}t}$$

$$y = cC_1te^{\sqrt{at}} - cC_2te^{-\sqrt{at}} + C_3e^{\sqrt{at}} + C_4e^{-\sqrt{at}}$$

1.4. If  $D = 0$  and  $a = d \neq 0$  and  $c = 0$ :

$$x = bC_1te^{\sqrt{at}} - bC_2te^{-\sqrt{at}} + C_3e^{\sqrt{at}} + C_4e^{-\sqrt{at}}$$

$$y = 2\sqrt{a}C_1e^{\sqrt{at}} + 2\sqrt{a}C_2e^{-\sqrt{at}}$$

2. When  $ad - bc = 0$  and  $a^2 + b^2 > 0$ . Then the original system becomes

$$x'' = ax + by$$

$$y'' = k(ax + by)$$

2.1. If  $a + bk \neq 0$ :

$$x = C_1e^{t\sqrt{a+bk}} + C_2e^{-t\sqrt{a+bk}} + C_3bt + C_4b$$

$$y = C_1ke^{t\sqrt{a+bk}} + C_2ke^{-t\sqrt{a+bk}} - C_3at - C_4a$$

2.2. If  $a + bk = 0$ :

$$x = C_1bt^3 + C_2bt^2 + C_3t + C_4$$

$$y = kx + 6C_1t + 2C_2$$

## system\_of\_odes.linear\_2eq\_order2\_type2

`sympy.solvers.ode.linear_2eq_order2_type2(x, y, t, r)`

The equations in this type are

$$x'' = a_1x + b_1y + c_1$$

$$y'' = a_2x + b_2y + c_2$$

The general solution of this system is given by the sum of its particular solution and the general solution of the homogeneous system. The general solution is given by the linear system of 2 equation of order 2 and type 1

1. If  $a_1b_2 - a_2b_1 \neq 0$ . A particular solution will be  $x = x_0$  and  $y = y_0$  where the constants  $x_0$  and  $y_0$  are determined by solving the linear algebraic system

$$a_1x_0 + b_1y_0 + c_1 = 0, a_2x_0 + b_2y_0 + c_2 = 0$$

2. If  $a_1b_2 - a_2b_1 = 0$  and  $a_1^2 + b_1^2 > 0$ . In this case, the system in question becomes

$$x'' = ax + by + c_1, y'' = k(ax + by) + c_2$$

2.1. If  $\sigma = a + bk \neq 0$ , the particular solution will be

$$x = \frac{1}{2}b\sigma^{-1}(c_1k - c_2)t^2 - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

2.2. If  $\sigma = a + bk = 0$ , the particular solution will be

$$x = \frac{1}{24}b(c_2 - c_1k)t^4 + \frac{1}{2}c_1t^2$$

$$y = kx + \frac{1}{2}(c_2 - c_1k)t^2$$

**system\_of\_odes.linear\_2eq\_order2\_type3**

```
sympy.solvers.ode.linear_2eq_order2_type3(x, y, t, r)
```

These type of equation is used for describing the horizontal motion of a pendulum taking into account the Earth rotation. The solution is given with  $a^2 + 4b > 0$ :

$$x = C_1 \cos(\alpha t) + C_2 \sin(\alpha t) + C_3 \cos(\beta t) + C_4 \sin(\beta t)$$

$$y = -C_1 \sin(\alpha t) + C_2 \cos(\alpha t) - C_3 \sin(\beta t) + C_4 \cos(\beta t)$$

where  $C_1, \dots, C_4$  and

$$\alpha = \frac{1}{2}a + \frac{1}{2}\sqrt{a^2 + 4b}, \beta = \frac{1}{2}a - \frac{1}{2}\sqrt{a^2 + 4b}$$

**system\_of\_odes.linear\_2eq\_order2\_type4**

```
sympy.solvers.ode.linear_2eq_order2_type4(x, y, t, r)
```

These equations are found in the theory of oscillations

$$x'' + a_1x' + b_1y' + c_1x + d_1y = k_1e^{i\omega t}$$

$$y'' + a_2x' + b_2y' + c_2x + d_2y = k_2e^{i\omega t}$$

The general solution of this linear nonhomogeneous system of constant-coefficient differential equations is given by the sum of its particular solution and the general solution of the corresponding homogeneous system (with  $k_1 = k_2 = 0$ )

1. A particular solution is obtained by the method of undetermined coefficients:

$$x = A_*e^{i\omega t}, y = B_*e^{i\omega t}$$

On substituting these expressions into the original system of differential equations, one arrive at a linear nonhomogeneous system of algebraic equations for the coefficients  $A$  and  $B$ .

2. The general solution of the homogeneous system of differential equations is determined by a linear combination of linearly independent particular solutions determined by the method of undetermined coefficients in the form of exponentials:

$$x = Ae^{\lambda t}, y = Be^{\lambda t}$$

On substituting these expressions into the original system and colleting the coefficients of the unknown  $A$  and  $B$ , one obtains

$$(\lambda^2 + a_1\lambda + c_1)A + (b_1\lambda + d_1)B = 0$$

$$(a_2\lambda + c_2)A + (\lambda^2 + b_2\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions  $A, B$  to exist. This requirement results in the following characteristic equation for  $\lambda$

$$(\lambda^2 + a_1\lambda + c_1)(\lambda^2 + b_2\lambda + d_2) - (b_1\lambda + d_1)(a_2\lambda + c_2) = 0$$

If all roots  $k_1, \dots, k_4$  of this equation are distict, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1\lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1\lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1\lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1\lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + a_1\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + a_1\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + a_1\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + a_1\lambda_4 + c_1)e^{\lambda_4 t}$$

**system\_of\_odes.linear\_2eq\_order2\_type5**

```
sympy.solvers.ode.linear_2eq_order2_type5(x, y, t, r)
```

The equation which come under this catagory are

$$x'' = a(ty' - y)$$

$$y'' = b(tx' - x)$$

The transformation

$$u = tx' - x, b = ty' - y$$

leads to the first-order system

$$u' = atv, v' = btu$$

The general solution of this system is given by

If  $ab > 0$ :

$$u = C_1ae^{\frac{1}{2}\sqrt{ab}t^2} + C_2ae^{-\frac{1}{2}\sqrt{ab}t^2}$$

$$v = C_1\sqrt{ab}e^{\frac{1}{2}\sqrt{ab}t^2} - C_2\sqrt{ab}e^{-\frac{1}{2}\sqrt{ab}t^2}$$

If  $ab < 0$ :

$$u = C_1a \cos(\frac{1}{2}\sqrt{|ab|}t^2) + C_2a \sin(-\frac{1}{2}\sqrt{|ab|}t^2)$$

$$v = C_1\sqrt{|ab|} \sin(\frac{1}{2}\sqrt{|ab|}t^2) + C_2\sqrt{|ab|} \cos(-\frac{1}{2}\sqrt{|ab|}t^2)$$

where  $C_1$  and  $C_2$  are arbitrary constants. On substituting the value of  $u$  and  $v$  in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3t + t \int \frac{u}{t^2} dt, y = C_4t + t \int \frac{u}{t^2} dt$$

where  $C_3$  and  $C_4$  are arbitrary constants.

**system\_of\_odes.linear\_2eq\_order2\_type6**

```
sympy.solvers.ode.linear_2eq_order2_type6(x, y, t, r)
```

The equations are

$$x'' = f(t)(a_1x + b_1y)$$

$$y'' = f(t)(a_2x + b_2y)$$

If  $k_1$  and  $k_2$  are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1b_2 - a_2b_1 = 0$$

Then by multiplying appropriate constants and adding together original equations we obtain two independent equations:

$$z_1'' = k_1f(t)z_1, z_1 = a_2x + (k_1 - a_1)y$$

$$z_2'' = k_2f(t)z_2, z_2 = a_2x + (k_2 - a_1)y$$

Solving the equations will give the values of  $x$  and  $y$  after obtaining the value of  $z_1$  and  $z_2$  by solving the differential equation and substuting the result.

**system\_of\_odes.linear\_2eq\_order2\_type7**

```
sympy.solvers.ode.linear_2eq_order2_type7(x, y, t, r)
The equations are given as
```

$$x'' = f(t)(a_1x' + b_1y')$$

$$y'' = f(t)(a_2x' + b_2y')$$

If  $k_1$  and ' $k_2$ ' are roots of the quadratic equation

$$k^2 - (a_1 + b_2)k + a_1b_2 - a_2b_1 = 0$$

Then the system can be reduced by adding together the two equations multiplied by appropriate constants give following two independent equations:

$$z_1'' = k_1f(t)z_1', z_1 = a_2x + (k_1 - a_1)y$$

$$z_2'' = k_2f(t)z_2', z_2 = a_2x + (k_2 - a_1)y$$

Integrating these and returning to the original variables, one arrives at a linear algebraic system for the unknowns  $x$  and  $y$ :

$$a_2x + (k_1 - a_1)y = C_1 \int e^{k_1 F(t)} dt + C_2$$

$$a_2x + (k_2 - a_1)y = C_3 \int e^{k_2 F(t)} dt + C_4$$

where  $C_1, \dots, C_4$  are arbitrary constants and  $F(t) = \int f(t) dt$

**system\_of\_odes.linear\_2eq\_order2\_type8**

```
sympy.solvers.ode.linear_2eq_order2_type8(x, y, t, r)
```

The equation of this category are

$$x'' = af(t)(ty' - y)$$

$$y'' = bf(t)(tx' - x)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the system of first-order equations

$$u' = atf(t)v, v' = bt f(t)u$$

The general solution of this system has the form

If  $ab > 0$ :

$$u = C_1ae^{\sqrt{ab} \int tf(t) dt} + C_2ae^{-\sqrt{ab} \int tf(t) dt}$$

$$v = C_1 \sqrt{ab} e^{\sqrt{ab} \int t f(t) dt} - C_2 \sqrt{ab} e^{-\sqrt{ab} \int t f(t) dt}$$

If  $ab < 0$ :

$$u = C_1 a \cos(\sqrt{|ab|} \int t f(t) dt) + C_2 a \sin(-\sqrt{|ab|} \int t f(t) dt)$$

$$v = C_1 \sqrt{|ab|} \sin(\sqrt{|ab|} \int t f(t) dt) + C_2 \sqrt{|ab|} \cos(-\sqrt{|ab|} \int t f(t) dt)$$

where  $C_1$  and  $C_2$  are arbitrary constants. On substituting the value of  $u$  and  $v$  in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{u}{t^2} dt$$

where  $C_3$  and  $C_4$  are arbitrary constants.

### system\_of\_odes\_linear\_2eq\_order2\_type9

```
sympy.solvers.ode.linear_2eq_order2_type9(x, y, t, r)
```

$$t^2 x'' + a_1 t x' + b_1 t y' + c_1 x + d_1 y = 0$$

$$t^2 y'' + a_2 t x' + b_2 t y' + c_2 x + d_2 y = 0$$

These system of equations are euler type.

The substitution of  $t = \sigma e^\tau (\sigma \neq 0)$  leads to the system of constant coefficient linear differential equations

$$x'' + (a_1 - 1)x' + b_1 y' + c_1 x + d_1 y = 0$$

$$y'' + a_2 x' + (b_2 - 1)y' + c_2 x + d_2 y = 0$$

The general solution of the homogeneous system of differential equations is determined by a linear combination of linearly independent particular solutions determined by the method of undetermined coefficients in the form of exponentials

$$x = A e^{\lambda t}, y = B e^{\lambda t}$$

On substituting these expressions into the original system and collecting the coefficients of the unknown  $A$  and  $B$ , one obtains

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)A + (b_1 \lambda + d_1)B = 0$$

$$(a_2 \lambda + c_2)A + (\lambda^2 + (b_2 - 1)\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions  $A, B$  to exist. This requirement results in the following characteristic equation for  $\lambda$

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)(\lambda^2 + (b_2 - 1)\lambda + d_2) - (b_1 \lambda + d_1)(a_2 \lambda + c_2) = 0$$

If all roots  $k_1, \dots, k_4$  of this equation are distinct, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1 \lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1 \lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1 \lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1 \lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + (a_1 - 1)\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + (a_1 - 1)\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + (a_1 - 1)\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + (a_1 - 1)\lambda_4 + c_2)e^{\lambda_4 t}$$

**system\_of\_odes.linear\_2eq\_order2\_type10**

```
sympy.solvers.ode.linear_2eq_order2_type10(x, y, t, r)
The equation of this category are
```

$$(\alpha t^2 + \beta t + \gamma)^2 x'' = ax + by$$

$$(\alpha t^2 + \beta t + \gamma)^2 y'' = cx + dy$$

The transformation

$$\tau = \int \frac{1}{\alpha t^2 + \beta t + \gamma} dt, u = \frac{x}{\sqrt{|\alpha t^2 + \beta t + \gamma|}}, v = \frac{y}{\sqrt{|\alpha t^2 + \beta t + \gamma|}}$$

leads to a constant coefficient linear system of equations

$$u'' = (a - \alpha\gamma + \frac{1}{4}\beta^2)u + bv$$

$$v'' = cu + (d - \alpha\gamma + \frac{1}{4}\beta^2)v$$

These system of equations obtained can be solved by type1 of System of two constant-coefficient second-order linear homogeneous differential equations.

**system\_of\_odes.linear\_2eq\_order2\_type11**

```
sympy.solvers.ode.linear_2eq_order2_type11(x, y, t, r)
The equations which comes under this type are
```

$$x'' = f(t)(tx' - x) + g(t)(ty' - y)$$

$$y'' = h(t)(tx' - x) + p(t)(ty' - y)$$

The transformation

$$u = tx' - x, v = ty' - y$$

leads to the linear system of first-order equations

$$u' = tf(t)u + tg(t)v, v' = th(t)u + tp(t)v$$

On substituting the value of  $u$  and  $v$  in transformed equation gives value of  $x$  and  $y$  as

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt.$$

where  $C_3$  and  $C_4$  are arbitrary constants.

**system\_of\_odes.linear\_3eq\_order1\_type1**

```
sympy.solvers.ode.linear_3eq_order1_type1(x, y, z, t, r)
```

$$x' = ax$$

$$y' = bx + cy$$

$$z' = dx + ky + pz$$

Solution of such equations are forward substitution. Solving first equations gives the value of  $x$ , substituting it in second and third equation and solving second equation gives  $y$  and similarly substituting  $y$  in third equation give  $z$ .

$$x = C_1 e^{at}$$

$$y = \frac{bC_1}{a - c} e^{at} + C_2 e^{ct}$$

$$z = \frac{C_1}{a - p} \left( d + \frac{bk}{a - c} \right) e^{at} + \frac{kC_2}{c - p} e^{ct} + C_3 e^{pt}$$

where  $C_1, C_2$  and  $C_3$  are arbitrary constants.

### system\_of\_odes\_linear\_3eq\_order1\_type2

```
sympy.solvers.ode.linear_3eq_order1_type2(x, y, z, t, r)
```

The equations of this type are

$$x' = cy - bz$$

$$y' = az - cx$$

$$z' = bx - ay$$

1. First integral:

$$ax + by + cz = A \quad - (1)$$

$$x^2 + y^2 + z^2 = B^2 \quad - (2)$$

where  $A$  and  $B$  are arbitrary constants. It follows from these integrals that the integral lines are circles formed by the intersection of the planes (1) and sphere (2)

2. Solution:

$$x = aC_0 + kC_1 \cos(kt) + (cC_2 - bC_3) \sin(kt)$$

$$y = bC_0 + kC_2 \cos(kt) + (aC_2 - cC_3) \sin(kt)$$

$$z = cC_0 + kC_3 \cos(kt) + (bC_2 - aC_3) \sin(kt)$$

where  $k = \sqrt{a^2 + b^2 + c^2}$  and the four constants of integration,  $C_1, \dots, C_4$  are constrained by a single relation,

$$aC_1 + bC_2 + cC_3 = 0$$

**system\_of\_odes.linear\_3eq\_order1\_type3**

```
sympy.solvers.ode.linear_3eq_order1_type3(x, y, z, t, r)
    Equations of this system of ODEs
```

$$ax' = bc(y - z)$$

$$by' = ac(z - x)$$

$$cz' = ab(x - y)$$

1. First integral:

$$a^2x + b^2y + c^2z = A$$

where  $A$  is an arbitrary constant. It follows that the integral lines are plane curves.

2. Solution:

$$x = C_0 + kC_1 \cos(kt) + a^{-1}bc(C_2 - C_3) \sin(kt)$$

$$y = C_0 + kC_2 \cos(kt) + ab^{-1}c(C_3 - C_1) \sin(kt)$$

$$z = C_0 + kC_3 \cos(kt) + abc^{-1}(C_1 - C_2) \sin(kt)$$

where  $k = \sqrt{a^2 + b^2 + c^2}$  and the four constants of integration,  $C_1, \dots, C_4$  are constrained by a single relation

$$a^2C_1 + b^2C_2 + c^2C_3 = 0$$

**system\_of\_odes.linear\_3eq\_order1\_type4**

```
sympy.solvers.ode.linear_3eq_order1_type4(x, y, z, t, r)
    Equations:
```

$$x' = (a_1f(t) + g(t))x + a_2f(t)y + a_3f(t)z$$

$$y' = b_1f(t)x + (b_2f(t) + g(t))y + b_3f(t)z$$

$$z' = c_1f(t)x + c_2f(t)y + (c_3f(t) + g(t))z$$

The transformation

$$x = e^{\int g(t) dt} u, y = e^{\int g(t) dt} v, z = e^{\int g(t) dt} w, \tau = \int f(t) dt$$

leads to the system of constant coefficient linear differential equations

$$u' = a_1u + a_2v + a_3w$$

$$v' = b_1u + b_2v + b_3w$$

$$w' = c_1u + c_2v + c_3w$$

These system of equations are solved by homogeneous linear system of constant coefficients of  $n$  equations of first order. Then substituting the value of  $u, v$  and  $w$  in transformed equation gives value of  $x, y$  and  $z$ .

## system\_of\_odes.linear\_neq\_order1\_type1

```
sympy.solvers.ode.linear_neq_order1_type1(match_)
```

System of  $n$  first-order constant-coefficient linear nonhomogeneous differential equation

$$y'_k = a_{k1}y_1 + a_{k2}y_2 + \dots + a_{kn}y_n; k = 1, 2, \dots, n$$

or that can be written as  $\vec{y}' = A\vec{y}$  where  $\vec{y}$  is matrix of  $y_k$  for  $k = 1, 2, \dots, n$  and  $A$  is a  $n \times n$  matrix.

Since these equations are equivalent to a first order homogeneous linear differential equation. So the general solution will contain  $n$  linearly independent parts and solution will consist some type of exponential functions. Assuming  $y = \vec{v}e^{rt}$  is a solution of the system where  $\vec{v}$  is a vector of coefficients of  $y_1, \dots, y_n$ . Substituting  $y$  and  $y' = r\vec{v}e^{rt}$  into the equation  $\vec{y}' = A\vec{y}$ , we get

$$r\vec{v}e^{rt} = A\vec{v}e^{rt}$$

$$r\vec{v} = A\vec{v}$$

where  $r$  comes out to be eigenvalue of  $A$  and vector  $\vec{v}$  is the eigenvector of  $A$  corresponding to  $r$ . There are three possibilities of eigenvalues of  $A$

- $n$  distinct real eigenvalues
- complex conjugate eigenvalues
- eigenvalues with multiplicity  $k$

1. When all eigenvalues  $r_1, \dots, r_n$  are distinct with  $n$  different eigenvectors  $v_1, \dots, v_n$  then the solution is given by

$$\vec{y} = C_1 e^{r_1 t} \vec{v}_1 + C_2 e^{r_2 t} \vec{v}_2 + \dots + C_n e^{r_n t} \vec{v}_n$$

where  $C_1, C_2, \dots, C_n$  are arbitrary constants.

2. When some eigenvalues are complex then in order to make the solution real, we take a linear combination: if  $r = a + bi$  has an eigenvector  $\vec{v} = \vec{w}_1 + i\vec{w}_2$  then to obtain real-valued solutions to the system, replace the complex-valued solutions  $e^{rx}\vec{v}$  with real-valued solution  $e^{ax}(\vec{w}_1 \cos(bx) - \vec{w}_2 \sin(bx))$  and for  $r = a - bi$  replace the solution  $e^{-rx}\vec{v}$  with  $e^{ax}(\vec{w}_1 \sin(bx) + \vec{w}_2 \cos(bx))$

3. If some eigenvalues are repeated. Then we get fewer than  $n$  linearly independent eigenvectors, we miss some of the solutions and need to construct the missing ones. We do this via generalized eigenvectors, vectors which are not eigenvectors but are close enough that we can use to write down the remaining solutions. For a eigenvalue  $r$  with eigenvector  $\vec{w}$  we obtain  $\vec{w}_2, \dots, \vec{w}_k$  using

$$(A - rI).\vec{w}_2 = \vec{w}$$

$$(A - rI).\vec{w}_3 = \vec{w}_2$$

⋮

$$(A - rI).\vec{w}_k = \vec{w}_{k-1}$$

Then the solutions to the system for the eigenspace are  $e^{rt}[\vec{w}], e^{rt}[t\vec{w} + \vec{w}_2], e^{rt}[\frac{t^2}{2}\vec{w} + t\vec{w}_2 + \vec{w}_3], \dots, e^{rt}[\frac{t^{k-1}}{(k-1)!}\vec{w} + \frac{t^{k-2}}{(k-2)!}\vec{w}_2 + \dots + t\vec{w}_{k-1} + \vec{w}_k]$

So, If  $\vec{y}_1, \dots, \vec{y}_n$  are  $n$  solution of obtained from three categories of  $A$ , then general solution to the system  $\vec{y}' = A\vec{y}$

$$\vec{y} = C_1 \vec{y}_1 + C_2 \vec{y}_2 + \dots + C_n \vec{y}_n$$

**system\_of\_odes\_nonlinear\_2eq\_order1\_type1**

```
sympy.solvers.ode._nonlinear_2eq_order1_type1(x, y, t, eq)
    Equations:
```

$$x' = x^n F(x, y)$$

$$y' = g(y)F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if  $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} dy]^{\frac{1}{1-n}}$$

if  $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} dy}$$

where  $C_1$  and  $C_2$  are arbitrary constants.

**system\_of\_odes\_nonlinear\_2eq\_order1\_type2**

```
sympy.solvers.ode._nonlinear_2eq_order1_type2(x, y, t, eq)
    Equations:
```

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y)F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} dy = t + C_2$$

where

if  $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} \log(C_1 - \lambda \int \frac{1}{g(y)} dy)$$

if  $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)} dy$$

where  $C_1$  and  $C_2$  are arbitrary constants.

### system\_of\_odes\_nonlinear\_2eq\_order1\_type3

```
sympy.solvers.ode._nonlinear_2eq_order1_type3(x, y, t, eq)
Autonomous system of general form
```

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming  $y = y(x, C_1)$  where  $C_1$  is an arbitrary constant is the general solution of the first-order equation

$$F(x, y)y'_x = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))} dx = t + C_1$$

### system\_of\_odes\_nonlinear\_2eq\_order1\_type4

```
sympy.solvers.ode._nonlinear_2eq_order1_type4(x, y, t, eq)
Equation:
```

$$x' = f_1(x)g_1(y)\phi(x, y, t)$$

$$y' = f_2(x)g_2(y)\phi(x, y, t)$$

First integral:

$$\int \frac{f_2(x)}{f_1(x)} dx - \int \frac{g_1(y)}{g_2(y)} dy = C$$

where  $C$  is an arbitrary constant.

On solving the first integral for  $x$  (resp.,  $y$ ) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining  $y$  (resp.,  $x$ ).

### system\_of\_odes\_nonlinear\_2eq\_order1\_type5

```
sympy.solvers.ode._nonlinear_2eq_order1_type5(func, t, eq)
Clairaut system of ODEs
```

$$x = tx' + F(x', y')$$

$$y = ty' + G(x', y')$$

The following are solutions of the system

(i) straight lines:

$$x = C_1t + F(C_1, C_2), y = C_2t + G(C_1, C_2)$$

where  $C_1$  and  $C_2$  are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

**system\_of\_odes\_nonlinear\_3eq\_order1\_type1**

```
sympy.solvers.ode._nonlinear_3eq_order1_type1(x, y, z, t, eq)
    Equations:
```

$$ax' = (b - c)yz, \quad by' = (c - a)zx, \quad cz' = (a - b)xy$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where  $C_1$  and  $C_2$  are arbitrary constants. On solving the integrals for  $y$  and  $z$  and on substituting the resulting expressions into the first equation of the system, we arrives at a separable first-order equation on  $x$ . Similarly doing that for other two equations, we will arrive at first order equation on  $y$  and  $z$  too.

**References**

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0401.pdf>

**system\_of\_odes\_nonlinear\_3eq\_order1\_type2**

```
sympy.solvers.ode._nonlinear_3eq_order1_type2(x, y, z, t, eq)
    Equations:
```

$$ax' = (b - c)yzf(x, y, z, t)$$

$$by' = (c - a)zxf(x, y, z, t)$$

$$cz' = (a - b)xyf(x, y, z, t)$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where  $C_1$  and  $C_2$  are arbitrary constants. On solving the integrals for  $y$  and  $z$  and on substituting the resulting expressions into the first equation of the system, we arrives at a first-order differential equations on  $x$ . Similarly doing that for other two equations we will arrive at first order equation on  $y$  and  $z$ .

**References**

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0402.pdf>

### system\_of\_odes\_nonlinear\_3eq\_order1\_type3

```
sympy.solvers.ode._nonlinear_3eq_order1_type3(x, y, z, t, eq)
    Equations:
```

$$x' = cF_2 - bF_3, \quad y' = aF_3 - cF_1, \quad z' = bF_1 - aF_2$$

where  $F_n = F_n(x, y, z, t)$ .

1. First Integral:

$$ax + by + cz = C_1,$$

where  $C$  is an arbitrary constant.

2. If we assume function  $F_n$  to be independent of  $t$ , i.e.,  $F_n = F_n(x, y, z)$ . Then, on eliminating  $t$  and  $z$  from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{aF_3(x, y, z) - cF_1(x, y, z)}{cF_2(x, y, z) - bF_3(x, y, z)}$$

where  $z = \frac{1}{c}(C_1 - ax - by)$

#### References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0404.pdf>

### system\_of\_odes\_nonlinear\_3eq\_order1\_type4

```
sympy.solvers.ode._nonlinear_3eq_order1_type4(x, y, z, t, eq)
    Equations:
```

$$x' = czF_2 - byF_3, \quad y' = axF_3 - czF_1, \quad z' = byF_1 - axF_2$$

where  $F_n = F_n(x, y, z, t)$

1. First integral:

$$ax^2 + by^2 + cz^2 = C_1$$

where  $C$  is an arbitrary constant.

2. Assuming the function  $F_n$  is independent of  $t$ :  $F_n = F_n(x, y, z)$ . Then on eliminating  $t$  and  $z$  from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{axF_3(x, y, z) - czF_1(x, y, z)}{czF_2(x, y, z) - byF_3(x, y, z)}$$

where  $z = \pm\sqrt{\frac{1}{c}(C_1 - ax^2 - by^2)}$

#### References

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0405.pdf>

**system\_of\_odes\_nonlinear\_3eq\_order1\_type5**

```
sympy.solvers.ode._nonlinear_3eq_order1_type5(x, y, t, eq)
```

$$x' = x(cF_2 - bF_3), \quad y' = y(aF_3 - cF_1), \quad z' = z(bF_1 - aF_2)$$

where  $F_n = F_n(x, y, z, t)$  and are arbitrary functions.

First Integral:

$$|x|^a |y|^b |z|^c = C_1$$

where  $C$  is an arbitrary constant. If the function  $F_n$  is independent of  $t$ , then, by eliminating  $t$  and  $z$  from the first two equations of the system, one arrives at a first-order equation.

**References**

-<http://eqworld.ipmnet.ru/en/solutions/sysode/sode0406.pdf>

**3.24.5 Information on the `ode` module**

This module contains `dsolve()` (page 984) and different helper functions that it uses.

`dsolve()` (page 984) solves ordinary differential equations. See the docstring on the various functions for their uses. Note that partial differential equations support is in `pde.py`. Note that hint functions have docstrings describing their various methods, but they are intended for internal use. Use `dsolve(ode, func, hint=hint)` to solve an ODE using a specific hint. See also the docstring on `dsolve()` (page 984).

**Functions in this module**

These are the user functions in this module:

- [dsolve\(\)](#) (page 984) - Solves ODEs.
- [classify\\_ode\(\)](#) (page 985) - Classifies ODEs into possible hints for `dsolve()` (page 984).
- [checkodesol\(\)](#) (page 986) - Checks if an equation is the solution to an ODE.
- [homogeneous\\_order\(\)](#) (page 987) - Returns the homogeneous order of an expression.
- [infinitesimals\(\)](#) (page 988) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.

These are the non-solver helper functions that are for internal use. The user should use the various options to `dsolve()` (page 984) to obtain the functionality provided by these functions:

- [odesimp\(\)](#) (page 989) - Does all forms of ODE simplification.
- [ode\\_sol\\_simplicity\(\)](#) (page 992) - A key function for comparing solutions by simplicity.
- [constantsimp\(\)](#) (page 991) - Simplifies arbitrary constants.
- [constant\\_renumber\(\)](#) (page 990) - Renumber arbitrary constants.
- [\\_handle\\_Integral\(\)](#) (page 1036) - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

**Currently implemented solver methods**

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run `help(ode)`):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or  $dx$  and  $dy$  are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
- 1st order Bernoulli differential equations.
- Power series solutions for first order differential equations.
- Lie Group method of solving first order differential equations.
- 2nd order Liouville differential equations.
- Power series solutions for second order differential equations at ordinary and regular singular points.
- $n$ th order linear homogeneous differential equation with constant coefficients.
- $n$ th order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
- $n$ th order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

### Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a `classify_ode()` (page 985) function, which takes in an ODE and tells you what hints, if any, will solve the ODE. It does this without attempting to solve the ODE, so it is fast. Each solving method is a hint, and it has its own function, named `ode_<hint>`. That function takes in the ODE and any match expression gathered by `classify_ode()` (page 985) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 551) class. `dsolve()` (page 984), which is the user wrapper function around all of this, will then call `odesimp()` (page 989) on the result, which, among other things, will attempt to solve the equation for the dependent variable (the function we are solving for), simplify the arbitrary constants in the expression, and evaluate any integrals, if the hint allows it.

### How to add new solution methods

If you have an ODE that you want `dsolve()` (page 984) to be able to solve, try to avoid adding special case code here. Instead, try finding a general method that will solve your ODE, as well as others. This way, the `ode` (page 1033) module will become more robust, and unhindered by special case hacks. WolphramAlpha and Maple's DETools[odeadvisor] function are two resources you can use to classify a specific ODE. It is also better for a method to work with an  $n$ th order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using `min` with `ode_sol_simplicity` as the key argument. See `ode_1st_homogeneous_coeff_best()` (page 994), for example. The function that uses your method will be called `ode_<hint>()`, so the hint must only use characters that are allowed in a Python function name (alphanumeric characters and the underscore ‘\_’ character). Include a function for every hint, except for `Integral` hints (`dsolve()` (page 984) takes care of those automatically). Hint names should be all lowercase, unless a word is commonly capitalized (such as `Integral` or `Bernoulli`). If you have a hint that you do not want to run with `all_Integral` that doesn't have an `_Integral` counterpart (such as a `best` hint that would defeat the purpose of `all_Integral`), you will need to remove it manually in the `dsolve()` (page 984) code. See also the `classify_ode()` (page 985) docstring for guidelines on writing a hint name.

Determine *in general* how the solutions returned by your method compare with other methods that can potentially solve the same ODEs. Then, put your hints in the `allhints` (page 989) tuple in the order that they should be called. The ordering of this tuple determines which hints are default. Note that exceptions are ok, because it is easy for the user to choose individual hints with `dsolve()` (page 984). In general, `_Integral` variants should go at the end of the list, and `_best` variants should go before the various hints they apply to. For example, the `undetermined_coefficients` hint comes before the `variation_of_parameters` hint because, even though variation of parameters is more general than undetermined coefficients, undetermined coefficients generally returns cleaner results for the ODEs that it can solve than variation of parameters does, and it does not require integration, so it is much faster.

Next, you need to have a match expression or a function that matches the type of the ODE, which you should put in `classify_ode()` (page 985) (if the match function is more than just a few lines, like `_undetermined_coefficients_match()` (page 1036), it should go outside of `classify_ode()` (page 985)). It should match the ODE without solving for it as much as possible, so that `classify_ode()` (page 985) remains fast and is not hindered by bugs in solving code. Be sure to consider corner cases. For example, if your solution method involves dividing by something, make sure you exclude the case where that division will be 0.

In most cases, the matching of the ODE will also give you the various parts that you need to solve it. You should put that in a dictionary (`.match()` will do this for you), and add that as `matching_hints['hint'] = matchdict` in the relevant part of `classify_ode()` (page 985). `classify_ode()` (page 985) will then send this to `dsolve()` (page 984), which will send it to your function as the `match` argument. Your function should be named `ode_<hint>(eq, func, order, match)`. If you need to send more information, put it in the ‘‘match’’ dictionary. For example, if you had to substitute in a dummy variable in `classify_ode()` (page 985) to match the ODE, you will need to pass it to your function using the `match` dict to access it. You can access the independent variable using `func.args[0]`, and the dependent variable (the function you are trying to solve for) as `func.func`. If, while trying to solve the ODE, you find that you cannot, raise `NotImplementedError`. `dsolve()` (page 984) will catch this error with the `all` meta-hint, rather than causing the whole routine to fail.

Add a docstring to your function that describes the method employed. Like with anything else in SymPy, you will need to add a doctest to the docstring, in addition to real tests in `test_ode.py`. Try to maintain consistency with the other hint functions’ docstrings. Add your method to the list at the top of this docstring. Also, add your method to `ode.rst` in the `docs/src` directory, so that the Sphinx docs will pull its docstring into the main SymPy documentation. Be sure to make the Sphinx documentation by running `make html` from within the doc directory to verify that the docstring formats correctly.

If your solution method involves integrating, use `Integral()` (page 551) instead of `integrate()` (page 543). This allows the user to bypass hard/slow integration by using the `_Integral` variant of your hint. In most cases, calling `sympy.core.basic.Basic.doit()` (page 65) will integrate your solution. If this is not the case, you will need to write special code in `_handle_Integral()` (page 1036). Arbitrary constants should be symbols named `C1`, `C2`, and so on. All solution methods should return an equality instance. If you need an arbitrary number of arbitrary constants, you can use `constants = numbered_symbols(prefix='C', cls=Symbol, start=1)`. If it is possible to solve for the dependent function in a general way, do so. Otherwise, do as best as you can, but do not call `solve` in your `ode_<hint>()` function. `odesimp()` (page 989) will attempt to solve the solution for you, so you do not need to do that. Lastly, if your ODE has a common simplification that can be applied to your solutions, you can add a special case in `odesimp()` (page 989) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many `log()` (page 329) terms, so `odesimp()` (page 989) calls `logcombine()` (page 932) on them (it also helps to write the arbitrary constant as `log(C1)` instead of `C1` in this case). Also consider common ways that you can rearrange your solution to have `constantsimp()` (page 991) take better advantage of it. It is better to put simplification in `odesimp()` (page 989) than in your method, because it can then be turned off with the `simplify` flag in `dsolve()` (page 984). If you have any extraneous simplification in your function, be sure to only run it using `if match.get('simplify', True):`, especially if it can be slow or if it can reduce the domain of the solution.

Finally, as with every contribution to SymPy, your method will need to be tested. Add a test for each method in `test_ode.py`. Follow the conventions there, i.e., test the solver using `dsolve(eq, f(x), hint=your_hint)`, and also test the solution using `checkodesol()` (page 986) (you can put these in a separate tests and skip/XFAIL if it runs too slow/doesn't work). Be sure to call your hint specifically in `dsolve()` (page 984), that way the test won't be broken simply by the introduction of another matching hint. If your method works for higher order ( $>1$ ) ODEs, you will need to run `sol = constant_renumber(sol, 'C', 1, order)` for each solution, where `order` is the order of the ODE. This is because `constant_renumber` renames the arbitrary constants by printing order, which is platform dependent. Try to test every corner case of your solver, including a range of orders if it is a  $n$ th order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in `test_ode.py`, so if anything is broken, one of those tests will surely fail.

`sympy.solvers.ode._undetermined_coefficients_match(expr, x)`

Returns a trial function match if undetermined coefficients can be applied to `expr`, and `None` otherwise.

A trial expression can be found for an expression for use with the method of undetermined coefficients if the expression is an additive/multiplicative combination of constants, polynomials in  $x$  (the independent variable of `expr`),  $\sin(ax + b)$ ,  $\cos(ax + b)$ , and  $e^{ax}$  terms (in other words, it has a finite number of linearly independent derivatives).

Note that you may still need to multiply each term returned here by sufficient  $x$  to make it linearly independent with the solutions to the homogeneous equation.

This is intended for internal use by `undetermined_coefficients` hints.

SymPy currently has no way to convert  $\sin^n(x) \cos^m(y)$  into a sum of only  $\sin(ax)$  and  $\cos(bx)$  terms, so these are not implemented. So, for example, you will need to manually convert  $\sin^2(x)$  into  $[1 + \cos(2x)]/2$  to properly apply the method of undetermined coefficients on it.

## Examples

```
>>> from sympy import log, exp
>>> from sympy.solvers.ode import _undetermined_coefficients_match
>>> from sympy.abc import x
>>> _undetermined_coefficients_match(9*x*exp(x) + exp(-x), x)
{'test': True, 'trialset': set([x*exp(x), exp(-x), exp(x)])}
>>> _undetermined_coefficients_match(log(x), x)
{'test': False}
```

`sympy.solvers.ode._handle_Integral(expr, func, order, hint)`

Converts a solution with Integrals in it into an actual solution.

For most hints, this simply runs `expr.doit()`.

## 3.25 PDE

### 3.25.1 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. They are intended for user use.

## pde\_separate

```
sympy.solvers.pde.pde_separate(eq, fun, sep, strategy='mul')
```

Separate variables in partial differential equation either by additive or multiplicative separation approach. It tries to rewrite an equation so that one of the specified variables occurs on a different side of the equation than the others.

### Parameters

- **eq** – Partial differential equation
- **fun** – Original function  $F(x, y, z)$
- **sep** – List of separated functions  $[X(x), u(y, z)]$
- **strategy** – Separation strategy. You can choose between additive separation ('add') and multiplicative separation ('mul') which is default.

See also:

[sympy.solvers.pde.pde\\_separate\\_add](#) (page 1037), [sympy.solvers.pde.pde\\_separate\\_mul](#) (page 1038)

### Examples

```
>>> from sympy import E, Eq, Function, pde_separate, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')

>>> eq = Eq(D(u(x, t), x), E**u(x, t)*D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]

>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), x, x)/X(x), Derivative(T(t), t, t)/T(t)]
```

## pde\_separate\_add

```
sympy.solvers.pde.pde_separate_add(eq, fun, sep)
```

Helper function for searching additive separable solutions.

Consider an equation of two independent variables  $x, y$  and a dependent variable  $w$ , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) + Y(y, z)$$

### Examples

```
>>> from sympy import E, Eq, Function, pde_separate_add, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')

>>> eq = Eq(D(u(x, t), x), E**u(x, t)*D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

## pde\_separate\_mul

```
sympy.solvers.pde.pde_separate_mul(eq, fun, sep)
```

Helper function for searching multiplicative separable solutions.

Consider an equation of two independent variables  $x, y$  and a dependent variable  $w$ , we look for the product of two functions depending on different arguments:

$$w(x, y, z) = X(x) * u(y, z)$$

### Examples

```
>>> from sympy import Function, Eq, pde_separate_mul, Derivative as D
>>> from sympy.abc import x, y
>>> u, X, Y = map(Function, 'uXY')

>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), x, x)/X(x), Derivative(Y(y), y, y)/Y(y)]
```

## pdsolve

```
sympy.solvers.pde.pdsolve(eq, func=None, hint='default', dict=False, solvefun=None, **kwargs)
```

Solves any (supported) kind of partial differential equation.

### Usage

`pdsolve(eq, f(x,y), hint)` -> Solve partial differential equation `eq` for function `f(x,y)`, using method `hint`.

### Details

**eq can be any supported partial differential equation** (see the pde docstring for supported methods). This can either be an Equality, or an expression, which is assumed to be equal to 0.

**f(x,y) is a function of two variables whose derivatives in that variable make up the partial differential equation.** In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it couldn't be detected).

**hint is the solving method that you want pdsolve to use.** Use `classify_pde(eq, f(x,y))` to get all of the possible hints for a PDE. The default hint, 'default', will use whatever hint is returned first by `classify_pde()`. See Hints below for more options that you can use for `hint`.

**solvefun is the convention used for arbitrary functions returned by the PDE solver.** If not set by the user, it is set by default to be `F`.

### Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to `pdsolve()`:

**"default":** This uses whatever hint is returned first by `classify_pde()`. This is the default argument to `pdsolve()`.

**"all":** To make `pdsolve` apply all relevant classification hints, use `pdsolve(PDE, func, hint="all")`. This will return a dictionary of `hint:solution` terms. If a hint causes `pdsolve`

to raise the `NotImplementedError`, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- `order`: The order of the PDE. See also `ode_order()` in `deutils.py`
- `default`: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by `classify_pde()`.

**“all\_Integral”**: This is the same as “all”, except if a hint also has a corresponding “\_Integral” hint, it only returns the “\_Integral” hint. This is useful if “all” causes `pdsolve()` to hang because of a difficult or impossible integral. This meta-hint will also be much faster than “all”, because `integrate()` is an expensive routine.

See also the `classify_pde()` docstring for more info on hints, and the `pde` docstring for a list of all supported hints.

## Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x, y # x and y are the independent variables
>>> f = Function("f")(x, y) # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See `test_pde.py` for many tests, which serves also as a set of examples for how to use `pdsolve()`.
- `pdsolve` always returns an Equality class (except for the case when the hint is “all” or “all\_Integral”). Note that it is not possible to get an explicit solution for  $f(x, y)$  as in the case of ODE's
- Do `help(pde.pde_hintname)` to get help more information on a specific hint

## Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> pdsolve(eq)
Eq(f(x, y), F(3*x - 2*y)*exp(-2*x/13 - 3*y/13))
```

## classify\_pde

`sympy.solvers.pde.classify_pde(eq, func=None, dict=False, **kwargs)`

Returns a tuple of possible `pdsolve()` classifications for a PDE.

The tuple is ordered so that first item is the classification that `pdsolve()` uses to solve the PDE by default. In general, classifications near the beginning of the list will produce better solutions faster than those near the end, though there are always exceptions. To make `pdsolve` use a different classification,

use `pdsolve(PDE, func, hint=<classification>)`. See also the `pdsolve()` docstring for different meta-hints you can use.

If `dict` is true, `classify_pde()` will return a dictionary of `hint:match` expression terms. This is intended for internal use by `pdsolve()`. Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by doing `help(pde.pde_hintname)`, where `hintname` is the name of the hint without “`_Integral`”.

See `sympy.pde.allhints` or the `sympy.pde` docstring for a list of all supported hints that can be returned from `classify_pde`.

## Examples

```
>>> from sympy.solvers.pde import classify_pde
>>> from sympy import Function, diff, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```

## checkpdesol

`sympy.solvers.pde.checkpdesol(pde, sol, func=None, solve_for_func=True)`

Checks if the given solution satisfies the partial differential equation.

`pde` is the partial differential equation which can be given in the form of an equation or an expression. `sol` is the solution for which the `pde` is to be checked. This can also be given in an equation or an expression form. If the function is not provided, the helper function `_preprocess` from `deutils` is used to identify the function.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

The following methods are currently being implemented to check if the solution satisfies the PDE:

1. Directly substitute the solution in the PDE and check. If the solution hasn't been solved for `f`, then it will solve for `f` provided `solve_for_func` hasn't been set to `False`.

If the solution satisfies the PDE, then a tuple `(True, 0)` is returned. Otherwise a tuple `(False, expr)` where `expr` is the value obtained after substituting the solution in the PDE. However if a known solution returns `False`, it may be due to the inability of `doit()` to simplify it to zero.

## Examples

```
>>> from sympy import Function, symbols, diff
>>> from sympy.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
```

```
>>> assert checkpdesol(eq, sol)[0]
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, (x*F(4*x - 3*y) - 6*F(4*x - 3*y)/25 + 4*Subs(Derivative(F(_xi_1), _xi_1), (_xi_1,), (4*x - 3*y,)))*exp
```

### 3.25.2 Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

#### pde\_1st\_linear\_constant\_coeff\_homogeneous

```
sympy.solvers.pde.pde_1st_linear_constant_coeff_homogeneous(eq, func, order, match, solve-  
fun)
```

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{df(x, y)}{dx} + b \frac{df(x, y)}{dy} + cf(x, y) = 0$$

where  $a$ ,  $b$  and  $c$  are constants.

The general solution is of the form:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
      d          d
a>--(f(x, y)) + b--(f(x, y)) + c*f(x, y)
      dx          dy
>>> pprint(pdsolve(genform))
      -c*(a*x + b*y)
      -----
      2      2
      a  + b
f(x, y) = F(-a*y + b*x)*e
```

#### References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

#### Examples

```

>>> from sympy.solvers.pde import (
... pde_1st_linear_constant_coeff_homogeneous)
>>> from sympy import pdsolve
>>> from sympy import Function, diff, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y))
Eq(f(x, y), F(x - y)*exp(-x/2 - y/2))
>>> pprint(pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y)))
      x   y
      - - - -
      2   2
f(x, y) = F(x - y)*e

```

## pde\_1st\_linear\_constant\_coeff

```
sympy.solvers.pde.pde_1st_linear_constant_coeff(eq, func, order, match, solvefun)
```

Solves a first order linear partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a \frac{df(x,y)}{dx} + b \frac{df(x,y)}{dy} + cf(x,y) = G(x,y)$$

where  $a$ ,  $b$  and  $c$  are constants and  $G(x, y)$  can be an arbitrary function in  $x$  and  $y$ .

The general solution of the PDE is:

## References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

## Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), (F(4*x + 2*y) + exp(x/2 + 4*y)/15)*exp(x/2 - y))
```

## pde\_1st\_linear\_variable\_coeff

```
sympy.solvers.pde.pde_1st_linear_variable_coeff(eq, func, order, match, solvefun)
```

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

$$a(x,y) \frac{df(x,y)}{dx} + a(x,y) \frac{df(x,y)}{dy} + c(x,y)f(x,y) - G(x,y)$$

where  $a(x, y)$ ,  $b(x, y)$ ,  $c(x, y)$  and  $G(x, y)$  are arbitrary functions in  $x$  and  $y$ . This PDE is converted into an ODE by making the following transformation.

- 1]  $\xi$  as  $x$   
 2]  $\eta$  as the constant in the solution to the differential equation  $\frac{dy}{dx} = -\frac{b}{a}$

Making the following substitutions reduces it to the linear ODE

$$a(\xi, \eta) \frac{du}{d\xi} + c(\xi, \eta)u - d(\xi, \eta) = 0$$

which can be solved using `dsolve`.

The general form of this PDE is:

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy.abc import x, y
>>> from sympy import Function, pprint
>>> a, b, c, G, f= [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform)
      d          d
-G(x, y) + a(x, y)*f(x, y) + b(x, y)*--(f(x, y)) + c(x, y)*--(f(x, y))
      dx          dy
```

## References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

## Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, diff, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), F(x*y)*exp(y**2/2) + 1)
```

### 3.25.3 Information on the pde module

This module contains `pdsolve()` and different helper functions that it uses. It is heavily inspired by the `ode` module and hence the basic infrastructure remains the same.

#### Functions in this module

These are the user functions in this module:

- `pdsolve()` - Solves PDE's
- `classify_pde()` - Classifies PDEs into possible hints for `dsolve()`.
- **`pde_separate()` - Separate variables in partial differential equation either by additive or multiplicative separation approach.**

These are the helper functions in this module:

- `pde_separate_add()` - Helper function for searching additive separable solutions.
- **`pde_separate_mul()` - Helper function for searching multiplicative separable solutions.**

#### Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various `pde_hint()` functions for more information on each (run `help(pde)`):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.

- 1st order linear partial differential equations with variable coefficients.

## 3.26 Solvers

The *solvers* module in SymPy implements methods for solving equations.

### 3.26.1 Algebraic equations

Use `solve()` (page 1045) to solve algebraic equations. We suppose all equations are equaled to 0, so solving  $x^{**2} == 1$  translates into the following code:

```
>>> from sympy.solvers import solve
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> solve(x**2 - 1, x)
[-1, 1]
```

The first argument for `solve()` (page 1045) is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solvers.solve(f, *symbols, **flags)`  
Algebraically solves equations and systems of equations.

**Currently supported are:**

- polynomial,
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions.

Input is formed as:

**•f**

- a single Expr or Poly that must be zero,
- an Equality
- a Relational expression or boolean
- iterable of one or more of the above

**•symbols (object(s) to solve for) specified as**

- none given (other non-numeric objects will be used)
- single symbol
- denested list of symbols e.g. `solve(f, x, y)`
- ordered iterable of symbols e.g. `solve(f, [x, y])`

**•flags**

- ‘dict’=True (default is False) return list (perhaps empty) of solution mappings
- ‘set’=True (default is False) return list of symbols and set of tuple(s) of solution(s)

‘**exclude=[] (default)**’ don’t try to solve for any of the free symbols in exclude; if expressions are given, the free symbols in them will be extracted automatically.

‘**check=True (default)**’ If False, don’t do any testing of solutions. This can be useful if one wants to include solutions that make any denominator zero.

‘**numerical=True (default)**’ do a fast numerical check if  $f$  has only one symbol.

‘**minimal=True (default is False)**’ a very fast, minimal testing.

‘**warn=True (default is False)**’ show a warning if checksol() could not conclude.

‘**simplify=True (default)**’ simplify all but polynomials of order 3 or greater before returning them and (if check is not False) use the general simplify function on the solutions and the expression obtained when they are substituted into the function which should be zero

‘**force=True (default is False)**’ make positive all symbols without assumptions regarding sign.

‘**rational=True (default)**’ recast Floats as Rational; if this option is not used, the system containing floats may fail to solve because of issues with polys. If rational=None, Floats will be recast as rationals but the answer will be recast as Floats. If the flag is False then nothing will be done to the Floats.

‘**manual=True (default is False)**’ do not use the polys/matrix method to solve a system of equations, solve them one at a time as you might “manually”

‘**implicit=True (default is False)**’ allows solve to return a solution for a pattern in terms of other functions that contain that pattern; this is only needed if the pattern is inside of some invertible function like cos, exp, ....

‘**particular=True (default is False)**’ instructs solve to try to find a particular solution to a linear system with as many zeros as possible; this is very expensive

‘**quick=True (default is False)**’ when using particular=True, use a fast heuristic instead to find a solution with many zeros (instead of using the very slow method guaranteed to find the largest number of zeros possible)

‘**cubics=True (default)**’ return explicit solutions when cubic expressions are encountered

‘**quartics=True (default)**’ return explicit solutions when quartic expressions are encountered

‘**quintics=True (default)**’ return explicit solutions (if possible) when quintic expressions are encountered

## Notes

assumptions aren’t checked when `solve()` input involves relational or bools.

When the solutions are checked, those that make any denominator zero are automatically excluded. If you do not want to exclude such solutions then use the `check=False` option:

```
>>> from sympy import sin, limit
>>> solve(sin(x)/x) # 0 is excluded
[pi]
```

If `check=False` then a solution to the numerator being zero is found:  $x = 0$ . In this case, this is a spurious solution since  $\sin(x)/x$  has the well known limit (without discontinuity) of 1 at  $x = 0$ :

---

```
>>> solve(sin(x)/x, check=False)
[0, pi]
```

In the following case, however, the limit exists and is equal to the the value of  $x = 0$  that is excluded when check=True:

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve(eq, x)
[]
>>> solve(eq, x, check=False)
[0]
>>> limit(eq, x, 0, '-')
0
>>> limit(eq, x, 0, '+')
0
```

## Examples

The output varies according to the input and can be seen by example:

```
>>> from sympy import solve, Poly, Eq, Function, exp
>>> from sympy.abc import x, y, z, a, b
>>> f = Function('f')
```

- boolean or univariate Relational

```
>>> solve(x < 3)
And(-oo < x, x < 3)
```

- to always get a list of solution mappings, use flag dict=True

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> solve([x - 3, y - 1], dict=True)
[{x: 3, y: 1}]
```

- to get a list of symbols and set of solution(s) use flag set=True

```
>>> solve([x**2 - 3, y - 1], set=True)
([x, y], set([-sqrt(3), 1, (sqrt(3), 1)]))
```

- single expression and single symbol that is in the expression

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
>>> solve(x**2 - y**2, x, set=True)
([x], set([-y, y]))
>>> solve(x**4 - 1, x, set=True)
([x], set([-1, 1, -I, I]))
```

- single expression with no symbol that is in the expression

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

- single expression with no symbol given

In this case, all free symbols will be selected as potential symbols to solve for. If the equation is univariate then a list of solutions is returned; otherwise – as is the case when symbols are given as an iterable of length > 1 – a list of mappings will be returned.

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2)
[{x: -y}, {x: y}]
>>> solve(z**2*x**2 - z**2*y**2)
[{x: -y}, {x: y}, {z: 0}]
>>> solve(z**2*x - z**2*y**2)
[{x: y**2}, {z: 0}]
```

- when an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save one from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method.

```
>>> solve(f(x) - x, f(x))
[x]
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
>>> solve(f(x).diff(x) - f(x) - x, f(x))
[-x + Derivative(f(x), x)]
>>> solve(x + exp(x)**2, exp(x), set=True)
([exp(x)], set([-sqrt(-x), (sqrt(-x),)]))

>>> from sympy import Indexed, IndexedBase, Tuple, sqrt
>>> A = IndexedBase('A')
>>> eqs = Tuple(A[1] + A[2] - 3, A[1] - A[2] + 1)
>>> solve(eqs, eqs.atoms(Indexed))
{A[1]: 1, A[2]: 2}
```

–To solve for a *symbol* implicitly, use ‘implicit=True’:

```
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-exp(x)]
```

–It is possible to solve for anything that can be targeted with subs:

```
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2)
{y: -2 + sqrt(3), x + 2: -sqrt(3)}
```

–Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2)
{y: -sqrt(3)/(x + 3), x + 2: (-2*x - 6 + sqrt(3))/(x + 3)}
>>> solve(eqs, y*x, x)
{x: -y - 4, x*y: -3*y - sqrt(3)}
```

–if you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1) # /!\ -1 is targeted, too
[x/(y - 1)]
>>> [_.subs(z, -1) for _ in solve((x - y + 1).subs(-1, z), 1)]
[-x + y]
```

–To solve for a function within a derivative, use dsolve.

- single expression and more than 1 symbol

–when there is a linear solution

```
>>> solve(x - y**2, x, y)
[{x: y**2}]
>>> solve(x**2 - y, x, y)
[{y: x**2}]
```

–when undetermined coefficients are identified

\*that are linear

```
>>> solve((a + b)*x - b + 2, a, b)
{a: -2, b: 2}
```

\*that are nonlinear

```
>>> solve((a + b)*x - b**2 + 2, a, b, set=True)
([a, b], set([-sqrt(2), sqrt(2), (sqrt(2), -sqrt(2))]))
```

–if there is no linear solution then the first successful attempt for a nonlinear solution will be returned

```
>>> solve(x**2 - y**2, x, y)
[{x: -y}, {x: y}]
>>> solve(x**2 - y**2/exp(x), x, y)
[{x: 2*LambertW(y/2)}]
>>> solve(x**2 - y**2/exp(x), y, x)
[{y: -x*sqrt(exp(x))}, {y: x*sqrt(exp(x))}]
```

- iterable of one or more of the above

–involving relational or bools

```
>>> solve([x < 3, x - 2])
Eq(x, 2)
>>> solve([x > 3, x - 2])
False
```

–when the system is linear

\*with a solution

```
>>> solve([x - 3], x)
{x: 3}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y, z)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - z), z, x, y)
{x: -5*y + 2, z: 21*y - 6}
```

\*without a solution

```
>>> solve([x + 3, x - 3])
[]
```

–when the system is not linear

```
>>> solve([x**2 + y - 2, y**2 - 4], x, y, set=True)
([x, y], set([-2, -2), (0, 2), (2, -2)))
```

–if no symbols are given, all free symbols will be selected and a list of mappings returned

```
>>> solve([x - 2, x**2 + y])
[{x: 2, y: -4}]
>>> solve([x - 2, x**2 + f(x)], set([f(x), x]))
[{x: 2, f(x): -4}]
```

–if any equation doesn't depend on the symbol(s) given it will be eliminated from the equation set and an answer may be given implicitly in terms of variables that were not of interest

```
>>> solve([x - y, y - 3], x)
{x: y}
```

`sympy.solvers.solvers.solve_linear(lhs, rhs=0, symbols=[], exclude=[])`

Return a tuple derived from  $f = lhs - rhs$  that is either:

**(numerator, denominator) of  $f$**  If this comes back as (0, 1) it means that  $f$  is independent of the symbols in `symbols`, e.g:

```
y*cos(x)**2 + y*sin(x)**2 - y = y*(0) = 0
cos(x)**2 + sin(x)**2 = 1
```

If it comes back as (0, 0) there is no solution to the equation amongst the symbols given.

If the numerator is not zero then the function is guaranteed to be dependent on a symbol in `symbols`.

or

(symbol, solution) where symbol appears linearly in the numerator of  $f$ , is in `symbols` (if given) and is not in `exclude` (if given).

No simplification is done to  $f$  other than and `mul=True` expansion, so the solution will correspond strictly to a unique solution.

## Examples

```
>>> from sympy.solvers.solvers import solve_linear
>>> from sympy.abc import x, y, z
```

These are linear in x and 1/x:

```
>>> solve_linear(x + y**2)
(x, -y**2)
>>> solve_linear(1/x - y**2)
(x, y**(-2))
```

When not linear in x or y then the numerator and denominator are returned.

```
>>> solve_linear(x**2/y**2 - 3)
(x**2 - 3*y**2, y**2)
```

If the numerator is a symbol then (0, 0) is returned if the solution for that symbol would have set any denominator to 0:

```
>>> solve_linear(1/(1/x - 2))
(0, 0)
>>> 1/(1/x) # to SymPy, this looks like x ...
x
>>> solve_linear(1/(1/x)) # so a solution is given
(x, 0)
```

If x is allowed to cancel, then this appears linear, but this sort of cancellation is not done so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> solve_linear(x**2*(1/x - z**2/x))
(x**2*(-z**2 + 1), x)
```

You can give a list of what you prefer for x candidates:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

You can also indicate what variables you don't want to consider:

```
>>> solve_linear(x + y + z, exclude=[x, z])
(y, -x - z)
```

If only x was excluded then a solution for y or z might be obtained.

`sympy.solvers.solvers.solve_linear_system(system, *symbols, **flags)`

Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Respectively, this procedure will return None or a dictionary with solutions. In the case of underdetermined systems, all arbitrary parameters are skipped. This may cause a situation in which an empty dictionary is returned. In that case, all symbols can be assigned arbitrary values.

Input to this functions is a Nx(M+1) matrix, which means it has to be in augmented form. If you prefer to enter N equations and M unknowns then use `solve(Neqs, *Msymbols)` instead. Note: a local copy of the matrix is made by this routine so the matrix that is passed will not be modified.

The algorithm used here is fraction-free Gaussian elimination, which results, after elimination, in an upper-triangular matrix. Then solutions are found using back-substitution. This approach is more efficient and compact than the Gauss-Jordan method.

```
>>> from sympy import Matrix, solve_linear_system
>>> from sympy.abc import x, y
```

Solve the following system:

```
x + 4 y == 2
-2 x + y == 14
```

```
>>> system = Matrix(( (1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y)
{x: -6, y: 2}
```

A degenerate system returns an empty dictionary.

```
>>> system = Matrix(( (0,0,0), (0,0,0) ))
>>> solve_linear_system(system, x, y)
{}
```

`sympy.solvers.solvers.solve_linear_system.LU(matrix, syms)`

Solves the augmented matrix system using LUsolve and returns a dictionary in which solutions are keyed to the symbols of syms as ordered.

The matrix must be invertible.

See also:

`sympy.matrices.matrices.MatrixBase.LUsolve` (page 584)

## Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.solvers import solve_linear_system_LU

>>> solve_linear_system_LU(Matrix([
... [1, 2, 0, 1],
... [3, 2, 2, 1],
... [2, 0, 0, 1]]), [x, y, z])
{x: 1/2, y: 1/4, z: -1/2}
```

`sympy.solvers.solvers.solve_undetermined_coeffs(equ, coeffs, sym, **flags)`

Solve equation of a type  $p(x; a_1, \dots, a_k) == q(x)$  where both p, q are univariate polynomials and f depends on k parameters. The result of this functions is a dictionary with symbolic values of those parameters with respect to coefficients in q.

This functions accepts both Equations class instances and ordinary SymPy expressions. Specification of parameters and variable is obligatory for efficiency and simplicity reason.

```
>>> from sympy import Eq
>>> from sympy.abc import a, b, c, x
>>> from sympy.solvers import solve_undetermined_coeffs

>>> solve_undetermined_coeffs(Eq(2*a*x + a+b, x), [a, b], x)
{a: 1/2, b: -1/2}

>>> solve_undetermined_coeffs(Eq(a*c*x + a+b, x), [a, b], x)
{a: 1/c, b: -1/c}
```

```
sympy.solvers.solvers.nsolve(*args, **kwargs)
Solve a nonlinear equation system numerically:
```

```
nsolve(f, [args], x0, modules=['mpmath'], **kwargs)
```

f is a vector function of symbolic expressions representing the system. args are the variables. If there is only one variable, this argument can be omitted. x0 is a starting vector close to a solution.

Use the modules keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of lambdify.

Overdetermined systems are supported.

```
>>> from sympy import Symbol, nsolve
>>> import sympy
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
[-1.19287309935246]
[ 1.27844411169911]
```

For one-dimensional functions the syntax is simplified:

```
>>> from sympy import sin, nsolve
>>> from sympy.abc import x
>>> nsolve(sin(x), x, 2)
3.14159265358979
>>> nsolve(sin(x), 2)
3.14159265358979
```

mpmath.findroot is used, you can find there more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that this routine works only with the numerator of the function in the one-dimensional case, and for very steep functions near the root this may lead to a failure in the verification of the root. In this case you should use the flag *verify = False* and independently verify the solution.

```
>>> from sympy import cos, cosh
>>> from sympy.abc import i
>>> f = cos(x)*cosh(x) - 1
>>> nsolve(f, 3.14*100)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (1.39267e+230 > 2.1684e-19)
>>> ans = nsolve(f, 3.14*100, verify=False); ans
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:

```
>>> bounds = lambda i: (3.14*i, 3.14*(i + 1))
>>> nsolve(f, bounds(100), solver='biseect', verify=False)
315.730061685774
```

```
sympy.solvers.solvers.check_assumptions(expr, **assumptions)
Checks whether expression expr satisfies all assumptions.

assumptions is a dict of assumptions: {'assumption': True|False, ...}.
```

### Examples

```
>>> from sympy import Symbol, pi, I, exp
>>> from sympy.solvers.solvers import check_assumptions

>>> check_assumptions(-5, integer=True)
True
>>> check_assumptions(pi, extended_real=True, integer=False)
True
>>> check_assumptions(pi, extended_real=True, negative=True)
False
>>> check_assumptions(exp(I*pi/7), extended_real=False)
True

>>> x = Symbol('x', extended_real=True, positive=True)
>>> check_assumptions(2*x + 1, extended_real=True, positive=True)
True
>>> check_assumptions(-2*x - 5, extended_real=True, positive=True)
False
```

*None* is returned if `check_assumptions()` could not conclude.

```
>>> check_assumptions(2*x - 1, extended_real=True, positive=True)
>>> z = Symbol('z')
>>> check_assumptions(z, extended_real=True)
```

```
sympy.solvers.solvers.checksol(f, symbol, sol=None, **flags)
Checks whether sol is a solution of equation f == 0.
```

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag `simplify=True`, the values in the dictionary will be simplified. *f* can be a single equation or an iterable of equations. A solution must satisfy all equations in *f* to be considered valid; if a solution does not satisfy any equation, *False* is returned; if one or more checks are inconclusive (and none are *False*) then *None* is returned.

### Examples

```
>>> from sympy import symbols
>>> from sympy.solvers import checksol
>>> x, y = symbols('x,y')
>>> checksol(x**4 - 1, x, 1)
True
>>> checksol(x**4 - 1, x, 0)
False
>>> checksol(x**2 + y**2 - 5**2, {x: 3, y: 4})
True
```

To check if an expression is zero using `checksol`, pass it as *f* and send an empty dictionary for `symbol`:

```
>>> checksol(x**2 + x - x*(x + 1), {})
True
```

None is returned if `checksol()` could not conclude.

**flags:**

- ‘numerical=True (default)’ do a fast numerical check if `f` has only one symbol.
- ‘minimal=True (default is False)’ a very fast, minimal testing.
- ‘warn=True (default is False)’ show a warning if `checksol()` could not conclude.
- ‘simplify=True (default)’ simplify solution before substituting into function and simplify the function before trying specific simplifications
- ‘force=True (default is False)’ make positive all symbols without assumptions regarding sign.

`sympy.solvers.solvers.unrad(eq, *syms, **flags)`

Remove radicals with symbolic arguments and return (`eq, cov`), None or raise an error:

None is returned if there are no radicals to remove.

`NotImplementedError` is raised if there are radicals and they cannot be removed or if the relationship between the original symbols and the change of variable needed to rewrite the system as a polynomial cannot be solved.

Otherwise the tuple, (`eq, cov`), is returned where:

```
““eq“, ““cov“
    ““eq“ is an equation without radicals (in the symbol(s) of
    interest) whose solutions are a superset of the solutions to the
    original expression. ““eq“ might be re-written in terms of a new
    variable; the relationship to the original variables is given by
    ““cov“ which is a list containing ““v“ and ““v**p - b“ where
    ““p“ is the power needed to clear the radical and ““b“ is the
    radical now expressed as a polynomial in the symbols of interest.
    For example, for sqrt(2 - x) the tuple would be
    ““(c, c**2 - 2 + x)“. The solutions of ““eq“ will contain
    solutions to the original equation (if there are any).
```

`syms` an iterable of symbols which, if provided, will limit the focus of radical removal: only radicals with one or more of the symbols of interest will be cleared. All free symbols are used if `syms` is not set.

`flags` are used internally for communication during recursive calls. Two options are also recognized:

““take“, when defined, is interpreted as a single-argument function  
that returns True if a given Pow should be handled.

Radicals can be removed from an expression if:

- \* all bases of the radicals are the same; a change of variables is done in this case.
- \* if all radicals appear in one term of the expression
- \* there are only 4 terms with `sqrt()` factors or there are less than four terms having `sqrt()` factors
- \* there are only two terms with radicals

## Examples

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy.abc import x
>>> from sympy import sqrt, Rational, root, real_roots, solve

>>> unrad(sqrt(x)*x**Rational(1, 3) + 2)
(x**5 - 64, [])
>>> unrad(sqrt(x) + root(x + 1, 3))
(x**3 - x**2 - 2*x - 1, [])
>>> eq = sqrt(x) + root(x, 3) - 2
>>> unrad(eq)
(_p**3 + _p**2 - 2, [_p, _p**6 - x])
```

### 3.26.2 Ordinary Differential equations (ODEs)

See *ODE* (page 984).

### 3.26.3 Partial Differential Equations (PDEs)

See *PDE* (page 1036).

### 3.26.4 Deutils (Utilities for solving ODE's and PDE's)

`sympy.solvers.deutils.ode_order(expr, func)`

Returns the order of a given differential equation with respect to func.

This function is implemented recursively.

#### Examples

```
>>> from sympy import Function
>>> from sympy.solvers.deutils import ode_order
>>> from sympy.abc import x
>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 +
... f(x).diff(x), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```

### 3.26.5 Recurrence Equations

`sympy.solvers.recurr.rsolve(f, y, init=None)`

Solve univariate recurrence with rational coefficients.

Given  $k$ -th order linear recurrence  $L y = f$ , or equivalently:

$$a_k(n)y(n+k) + a_{k-1}(n)y(n+k-1) + \cdots + a_0(n)y(n) = f(n)$$

where  $a_i(n)$ , for  $i = 0, \dots, k$ , are polynomials or rational functions in  $n$ , and  $f$  is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in  $n$ , finds all solutions or returns `None`, if none were found.

Initial conditions can be given as a dictionary in two forms:

```
1.{ n_0 : v_0, n_1 : v_1, ..., n_m : v_m }
2.{ y(n_0) : v_0, y(n_1) : v_1, ..., y(n_m) : v_m }
```

or as a list  $L$  of values:

```
L = [ v_0, v_1, ..., v_m ]
```

where  $L[i] = v_i$ , for  $i = 0, \dots, m$ , maps to  $y(n_i)$ .

**See also:**

[rsolve\\_poly](#) (page 1057), [rsolve\\_ratio](#) (page 1058), [rsolve\\_hyper](#) (page 1058)

## Examples

Lets consider the following recurrence:

$$(n - 1)y(n + 2) - (n^2 + 3n - 2)y(n + 1) + 2n(n + 1)y(n) = 0$$

```
>>> from sympy import Function, rsolve
>>> from sympy.abc import n
>>> y = Function('y')

>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)

>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)

>>> rsolve(f, y(n), { y(0):0, y(1):3 })
3*2**n - 3*factorial(n)
```

`sympy.solvers.recurr.rsolve_poly(coeffs, f, n, **hints)`

Given linear recurrence operator  $L$  of order  $k$  with polynomial coefficients and inhomogeneous equation  $Ly = f$ , where  $f$  is a polynomial, we seek for all polynomial solutions over field  $K$  of characteristic zero.

The algorithm performs two basic steps:

1. Compute degree  $N$  of the general polynomial solution.
2. Find all polynomials of degree  $N$  or less of  $Ly = f$ .

There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it's smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with  $N + 1$  unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only  $r$  indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovsek.

It is possible to generalize the algorithm implemented here to the case of linear q-difference and differential equations.

Lets say that we would like to compute  $m$ -th Bernoulli polynomial up to a constant. For this we can use  $b(n+1) - b(n) = mn^{m-1}$  recurrence, which has solution  $b(n) = B_m + C$ . For example:

```
>>> from sympy import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)

>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

## References

[R377] (page 1248), [R378] (page 1248), [R379] (page 1248)

`sympy.solvers.recurr.rsolve_ratio(coeffs, f, n, **hints)`

Given linear recurrence operator  $L$  of order  $k$  with polynomial coefficients and inhomogeneous equation  $L y = f$ , where  $f$  is a polynomial, we seek for all rational solutions over field  $K$  of characteristic zero.

This procedure accepts only polynomials, however if you are interested in solving recurrence with rational coefficients then use `rsolve` which will pre-process the given equation and run this procedure with polynomial arguments.

The algorithm performs two basic steps:

1. Compute polynomial  $v(n)$  which can be used as universal denominator of any rational solution of equation  $L y = f$ .
2. Construct new linear difference equation by substitution  $y(n) = u(n)/v(n)$  and solve it for  $u(n)$  finding all its polynomial solutions. Return `None` if none were found.

Algorithm implemented here is a revised version of the original Abramov's algorithm, developed in 1989. The new approach is much simpler to implement and has better overall efficiency. This method can be easily adapted to q-difference equations case.

Besides finding rational solutions alone, this functions is an important part of Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.

**See also:**

`rsolve_hyper` (page 1058)

## References

[R380] (page 1248)

## Examples

```
>>> from sympy.abc import x
>>> from sympy.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x,
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

`sympy.solvers.recurr.rsolve_hyper(coeffs, f, n, **hints)`

Given linear recurrence operator  $L$  of order  $k$  with polynomial coefficients and inhomogeneous equation  $L y = f$  we seek for all hypergeometric solutions over field  $K$  of characteristic zero.

The inhomogeneous part can be either hypergeometric or a sum of a fixed number of pairwise dissimilar hypergeometric terms.

The algorithm performs three basic steps:

1. Group together similar hypergeometric terms in the inhomogeneous part of  $Ly = f$ , and find particular solution using Abramov's algorithm.
2. Compute generating set of  $L$  and find basis in it, so that all solutions are linearly independent.
3. Form final solution with the number of arbitrary constants equal to dimension of basis of  $L$ .

Term  $a(n)$  is hypergeometric if it is annihilated by first order linear difference equations with polynomial coefficients or, in simpler words, if consecutive term ratio is a rational function.

The output of this procedure is a linear combination of fixed number of hypergeometric terms. However the underlying method can generate larger class of solutions - D'Alembertian terms.

Note also that this method not only computes the kernel of the inhomogeneous equation, but also reduces it to a basis so that solutions generated by this procedure are linearly independent

## References

[R381] (page 1248), [R382] (page 1248)

## Examples

```
>>> from sympy.solvers import rsolve_hyper
>>> from sympy.abc import x

>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 + sqrt(5)/2)**x + C1*(-sqrt(5)/2 + 1/2)**x

>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2
```

## 3.26.6 Systems of Polynomial Equations

`sympy.solvers.polysys.solve_poly_system(seq, *gens, **args)`  
Solve a system of polynomial equations.

## Examples

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

>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]
```

`sympy.solvers.polysys.solve_triangulated(polys, *gens, **args)`  
Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

## References

1. Patrizia Gianni, Teo Mora, Algebraic Solution of System of Polynomial Equations using Groebner Bases, AAECC-5 on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS 356 247–257, 1989

## Examples

```
>>> from sympy.solvers.polysys import solve_triangulated
>>> from sympy.abc import x, y, z

>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]

>>> solve_triangulated(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

## 3.26.7 Diophantine Equations (DEs)

See *Diophantine* (page 1060)

## 3.26.8 Inequalities

See *Inequality Solvers* (page 1078)

# 3.27 Diophantine

## 3.27.1 Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form,  $f(x_1, x_2, \dots, x_n) = 0$  where  $n \geq 2$  and  $x_1, x_2, \dots, x_n$  are integer variables. If we can find  $n$  integers  $a_1, a_2, \dots, a_n$  such that  $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$  satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in <sup>7</sup> and <sup>8</sup>.

Currently, following five types of Diophantine equations can be solved using `diophantine()` (page 1064) and other helper functions of the Diophantine module.

- Linear Diophantine equations:  $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$ .
- General binary quadratic equation:  $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation:  $ax^2 + by^2 + cz^2 + dxy + eyz + fzr = 0$
- Extended Pythagorean equation:  $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$

<sup>7</sup> Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations

<sup>8</sup> Diophantine Equation, Wolfram Mathworld, [online]. Available: <http://mathworld.wolfram.com/DiophantineEquation.html>

- General sum of squares:  $x_1^2 + x_2^2 + \dots + x_n^2 = k$

### 3.27.2 Module structure

This module contains `diophantine()` (page 1064) and helper functions that are needed to solve certain Diophantine equations. It's structured in the following manner.

- `diophantine()` (page 1064)
  - `diop_solve()` (page 1065)
    - \* `classify_diop()` (page 1065)
    - \* `diop_linear()` (page 1066)
    - \* `diop_quadratic()` (page 1067)
    - \* `diop_ternary_quadratic()` (page 1070)
    - \* `diop_general_pythagorean()` (page 1072)
    - \* `diop_general_sum_of_squares()` (page 1072)
  - `merge_solution()` (page 1074)

When an equation is given to `diophantine()` (page 1064), it factors the equation(if possible) and solves the equation given by each factor by calling `diop_solve()` (page 1065) separately. Then all the results are combined using `merge_solution()` (page 1074).

`diop_solve()` (page 1065) internally uses `classify_diop()` (page 1065) to find the type of the equation(and some other details) given to it and then calls the appropriate solver function based on the type returned. For example, if `classify_diop()` (page 1065) returned "linear" as the type of the equation, then `diop_solve()` (page 1065) calls `diop_linear()` (page 1066) to solve the equation.

Each of the functions, `diop_linear()` (page 1066), `diop_quadratic()` (page 1067), `diop_ternary_quadratic()` (page 1070), `diop_general_pythagorean()` (page 1072) and `diop_general_sum_of_squares()` (page 1072) solves a specific type of equations and the type can be easily guessed by it's name.

Apart from these functions, there are a considerable number of other functions in the "Diophantine Module" and all of them are listed under User functions and Internal functions.

### 3.27.3 Tutorial

First, let's import the highest API of the Diophantine module.

```
>>> from sympy.solvers.diophantine import diophantine
```

Before we start solving the equations, we need to define the variables.

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

Let's start by solving the easiest type of Diophantine equations, i.e. linear Diophantine equations. Let's solve  $2x + 3y = 5$ . Note that although we write the equation in the above form, when we input the equation to any of the functions in Diophantine module, it needs to be in the form  $eq = 0$ .

```
>>> diophantine(2*x + 3*y - 5)
set([(3*t - 5, -2*t + 5)])
```

Note that stepping one more level below the highest API, we can solve the very same equation by calling `diop_solve()` (page 1065).

```
>>> from sympy.solvers.diophantine import diop_solve
>>> diop_solve(2*x + 3*y - 5)
(3*t - 5, -2*t + 5)
```

Note that it returns a tuple rather than a set. `diophantine()` (page 1064) always return a set of tuples. But `diop_solve()` (page 1065) may return a single tuple or a set of tuples depending on the type of the equation given.

We can also solve this equation by calling `diop_linear()` (page 1066), which is what `diop_solve()` (page 1065) calls internally.

```
>>> from sympy.solvers.diophantine import diop_linear
>>> diop_linear(2*x + 3*y - 5)
(3*t - 5, -2*t + 5)
```

If the given equation has no solutions then the outputs will look like below.

```
>>> diophantine(2*x + 4*y - 3)
set()
>>> diop_solve(2*x + 4*y - 3)
(None, None)
>>> diop_linear(2*x + 4*y - 3)
(None, None)
```

Note that except for the highest level API, in case of no solutions, a tuple of `None` are returned. Size of the tuple is the same as the number of variables. Also, one can specifically set the parameter to be used in the solutions by passing a customized parameter. Consider the following example.

```
>>> m = symbols("m", integer=True)
>>> diop_solve(2*x + 3*y - 5, m)
(3*m - 5, -2*m + 5)
```

Please note that for the moment, user can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases associated with the equation would help a lot. Please refer <sup>9</sup> and <sup>10</sup> for detailed analysis of different cases and the nature of the solutions. Let us define  $\Delta = b^2 - 4ac$  w.r.t. the binary quadratic  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ .

When  $\Delta < 0$ , there are either no solutions or only a finite number of solutions.

```
>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
set([(2, 1), (5, 1)])
```

In the above equation  $\Delta = (-4)^2 - 4 * 1 * 8 = -16$  and hence only a finite number of solutions exist.

When  $\Delta = 0$  we might have either no solutions or parameterized solutions.

```
>>> diophantine(3*x**2 - 6*x*y + 3*y**2 - 3*x + 7*y - 5)
set()
>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
set([(t, -t), (t, -t + 3)])
```

<sup>9</sup> Methods to solve  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ , [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>

<sup>10</sup> Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , [online], Available: <http://www.jpr2718.org/ax2p.pdf>

The most interesting case is when  $\Delta > 0$  and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where  $\Delta = 8$ .

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> from sympy import expand
>>> n = symbols("n", integer=True)
>>> s = diophantine(x**2 - 2*y**2 - 2*x - 4*y, n)
>>> x_n, y_n = s.pop()
>>> expand(x_n)
-(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 - sqrt(2)*(2*sqrt(2) + 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1
>>> expand(y_n)
-sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 + sqrt(2)*(2*sqrt(2) + 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1
```

Here  $n$  is an integer. Although  $x_n$  and  $y_n$  may not look like integers, substituting in specific values for  $n$  (and simplifying) shows that they are. For example consider the following example where we set  $n$  equal to 9.

```
>>> from sympy import simplify
>>> simplify(x_n.subs({n: 9}))
-9369318
```

Any binary quadratic of the form  $ax^2 + bxy + cy^2 + dx + ey + f = 0$  can be transformed to an equivalent form  $X^2 - DY^2 = N$ .

```
>>> from sympy.solvers.diophantine import find_DN, diop_DN, transformation_to_DN
>>> find_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
(5, 920)
```

So, the above equation is equivalent to the equation  $X^2 - 5Y^2 = 920$  after a linear transformation. If we want to find the linear transformation, we can use `transformation_to_DN()` (page 1069)

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
```

Here  $A$  is a 2 X 2 matrix and  $B$  is a 2 X 1 matrix such that the transformation

$$\begin{bmatrix} X \\ Y \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + B$$

gives the equation  $X^2 - 5Y^2 = 920$ . Values of  $A$  and  $B$  are as belows.

```
>>> A
Matrix([
[1/10, 3/10],
[0, 1/5]])
>>> B
Matrix([
[1/5],
[-11/5]])
```

We can solve an equation of the form  $X^2 - DY^2 = N$  by passing  $D$  and  $N$  to `diop_DN()` (page 1067)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation does not have solutions.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form  $ax^2 + by^2 + cz^2 + dxy + eyz + fz^2 = 0$ . These type of equations either have infinitely many solutions or no solutions (except the obvious solution  $(0, 0, 0)$ )

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
set()
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
set([-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2])
```

If you are only interested about a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use `diop_ternary_quadratic()` (page 1070).

```
>>> from sympy.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
(-4, 5, 1)
```

`diop_ternary_quadratic()` (page 1070) first converts the given equation to an equivalent equation of the form  $w^2 = AX^2 + BY^2$  and then it uses `descent()` (page 1071) to solve the latter equation. You can refer to the docs of `transformation_to_normal()` (page 1078) to find more on this. The equation  $w^2 = AX^2 + BY^2$  can be solved more easily by using the Aforementioned `descent()` (page 1071).

```
>>> from sympy.solvers.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*Y**2 + Z**2
(1, 0, 1)
```

Here the solution tuple is in the order (w, Y, Z)

The extended Pythagorean equation,  $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$  and the general sum of squares equation,  $x_1^2 + x_2^2 + \dots + x_n^2 = k$  can also be solved using the Diophantine module.

```
>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
set([(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5, 60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)])
```

function `diop_general_pythagorean()` (page 1072) can also be called directly to solve the same equation. This is true about the general sum of squares too. Either you can call `diop_general_pythagorean()` (page 1072) or use the high level API.

```
>>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
set([(8, 4, 4, 4, 0, 0)])
```

If you want to get a more thorough idea about the the Diophantine module please refer to the following blog.  
<http://thilinaatsympy.wordpress.com/>

## 3.27.4 References

## 3.27.5 User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions are intended for use by ordinary users of SymPy.

### diophantine

`sympy.solvers.diophantine.diophantine(eq, param=t)`

Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.

For example, when solving,  $x^2 - y^2 = 0$  this is treated as  $(x+y)(x-y) = 0$  and  $x+y = 0$  and  $x-y = 0$  are solved independently and combined. Each term is solved by calling `diop_solve()`.

Output of `diophantine()` is a set of tuples. Each tuple represents a solution of the input equation. In a tuple, solution for each variable is listed according to the alphabetic order of input variables. i.e. if we have an equation with two variables  $a$  and  $b$ , first element of the tuple will give the solution for  $a$  and the second element will give the solution for  $b$ .

**See also:**

`sympy.solvers.diophantine.diop_solve` (page 1065)

### Examples

```
>>> from sympy.solvers.diophantine import diophantine
>>> from sympy.abc import x, y, z
>>> diophantine(x**2 - y**2)
set([(-t, -t), (t, -t)])

#>>> diophantine(x*(2*x + 3*y - z)) #set([(0, n1, n2), (3*t - z, -2*t + z, z)]) #>>> diophantine(x**2 + 3*x*y + 4*x) #set([(0, n1), (3*t - 4, -t)])
```

### diop\_solve

`sympy.solvers.diophantine.diop_solve(eq, param=t)`

Solves the diophantine equation `eq`.

Similar to `diophantine()` but doesn't try to factor `eq` as latter does. Uses `classify_diop()` to determine the type of the equation and calls the appropriate solver function.

**See also:**

`sympy.solvers.diophantine.diophantine` (page 1064)

### Examples

```
>>> from sympy.solvers.diophantine import diop_solve
>>> from sympy.abc import x, y, z, w
>>> diop_solve(2*x + 3*y - 5)
(3*t - 5, -2*t + 5)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(3*t + 4*z - 5, -4*t - 4*z + 5, z)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t, -t - 3*y + 4*z + 6, y, z)
>>> diop_solve(x**2 + y**2 - 5)
set([(-2, -1), (-2, 1), (2, -1), (2, 1)])
```

### classify\_diop

`sympy.solvers.diophantine.classify_diop(eq)`

Helper routine used by `diop_solve()` to find the type of the `eq` etc.

Returns a tuple containing the type of the diophantine equation along with the variables(free symbols) and their coefficients. Variables are returned as a list and coefficients are returned as a dict with the key being the respective term and the constant term is keyed to `Integer(1)`. Type is an element in the set {“linear”, “binary\_quadratic”, “general\_pythagorean”, “homogeneous\_ternary\_quadratic”, “univariate”, “general\_sum\_of\_squares”}

## Examples

```
>>> from sympy.solvers.diophantine import classify_diop
>>> from sympy.abc import x, y, z, w, t
>>> classify_diop(4*x + 6*y - 4)
([x, y], {1: -4, x: 4, y: 6}, 'linear')
>>> classify_diop(x + 3*y - 4*z + 5)
([x, y, z], {1: 5, x: 1, y: 3, z: -4}, 'linear')
>>> classify_diop(x**2 + y**2 - x*y + x + 5)
([x, y], {1: 5, x: 1, x**2: 1, y: 0, y**2: 1, x*y: -1}, 'binary_quadratic')
```

## diop\_linear

`sympy.solvers.diophantine.diop_linear(eq, param=t)`

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form  $a_1x_1 + a_2x_2 + \dots + a_nx_n = 0$  where  $a_1, a_2, \dots, a_n$  are integer constants and  $x_1, x_2, \dots, x_n$  are integer variables.

See also:

`sympy.solvers.diophantine.diop_quadratic` (page 1067), `sympy.solvers.diophantine.diop_ternary_quadratic` (page 1070), `sympy.solvers.diophantine.diop_general_pythagorean` (page 1072),  
`sympy.solvers.diophantine.diop_general_sum_of_squares` (page 1072)

## Examples

```
>>> from sympy.solvers.diophantine import diop_linear
>>> from sympy.abc import x, y, z, t
>>> from sympy import Integer
>>> diop_linear(2*x - 3*y - 5) #solves equation 2*x - 3*y - 5 = 0
(-3*t - 5, -2*t - 5)
```

Here  $x = -3t - 5$  and  $y = -2t - 5$

```
>>> diop_linear(2*x - 3*y - 4*z - 3)
(-3*t - 4*z - 3, -2*t - 4*z - 3, z)
```

## base\_solution\_linear

`sympy.solvers.diophantine.base_solution_linear(c, a, b, t=None)`

Return the base solution for a linear diophantine equation with two variables.

Used by `diop_linear()` to find the base solution of a linear Diophantine equation. If `t` is given then the parametrized solution is returned.

## Examples

```
>>> from sympy.solvers.diophantine import base_solution_linear
>>> from sympy.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
```

```
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, -2*t + 5)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

## diop\_quadratic

`sympy.solvers.diophantine.diop_quadratic(eq, param=t)`

Solves quadratic diophantine equations.

i.e. equations of the form  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ . Returns a set containing the tuples  $(x, y)$  which contains the solutions. If there are no solutions then  $(None, None)$  is returned.

**See also:**

`sympy.solvers.diophantine.diop_linear` (page 1066), `sympy.solvers.diophantine.diop_ternary_quadratic` (page 1070), `sympy.solvers.diophantine.diop_general_sum_of_squares` (page 1072),  
`sympy.solvers.diophantine.diop_general_pythagorean` (page 1072)

## References

[R359] (page 1249), [R360] (page 1249)

## Examples

```
>>> from sympy.abc import x, y, t
>>> from sympy.solvers.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
set([-1, -1])
```

## diop\_DN

`sympy.solvers.diophantine.diop_DN(D, N, t=t)`

Solves the equation  $x^2 - Dy^2 = N$ .

Mainly concerned in the case  $D > 0, D$  is not a perfect square, which is the same as generalized Pell equation. To solve the generalized Pell equation this function Uses LMM algorithm. Refer [R361] (page 1249) for more details on the algorithm. Returns one solution for each class of the solutions. Other solutions of the class can be constructed according to the values of  $D$  and  $N$ . Returns a list containing the solution tuples  $(x, y)$ .

**See also:**

`sympy.solvers.diophantine.find_DN` (page 1070), `sympy.solvers.diophantine.diop_bf_DN` (page 1068)

## References

[R361] (page 1249)

## Examples

```
>>> from sympy.solvers.diophantine import diop_DN
>>> diop_DN(13, -4) # Solves equation  $x^{**2} - 13*y^{**2} = -4$ 
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation  $x^2 - 13y^2 = -4$  given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form (x, y), i.e. solution (3, 1) means that  $x = 3$  and  $y = 1$ .

```
>>> diop_DN(986, 1) # Solves equation  $x^{**2} - 986*y^{**2} = 1$ 
[(49299, 1570)]
```

## cornacchia

`sympy.solvers.diophantine.cornacchia(a, b, m)`  
Solves  $ax^2 + by^2 = m$  where  $\gcd(a, b) = 1 = \gcd(a, m)$  and  $a, b > 0$ .

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with  $\gcd(x, y) = 1$ . So this method can't be used to find the solutions of  $x^2 + y^2 = 20$  since the only solution to former is  $(x, y) = (4, 2)$  and it is not primitive. When ' $a = b = 1$ ', only the solutions with  $x \geq y$  are found. For more details, see the References.

## References

[R362] (page 1249), [R363] (page 1249)

## Examples

```
>>> from sympy.solvers.diophantine import cornacchia
>>> cornacchia(2, 3, 35) # equation  $2x^{**2} + 3y^{**2} = 35$ 
set([(2, 3), (4, 1)])
>>> cornacchia(1, 1, 25) # equation  $x^{**2} + y^{**2} = 25$ 
set([(4, 3)])
```

## diop\_bf\_DN

`sympy.solvers.diophantine.diop_bf_DN(D, N, t=t)`  
Uses brute force to solve the equation,  $x^2 - Dy^2 = N$ .

Mainly concerned with the generalized Pell equation which is the case when  $D > 0, D$  is not a perfect square. For more information on the case refer [R364] (page 1249). Let  $(t, u)$  be the minimal positive solution of the equation  $x^2 - Dy^2 = 1$ . Then this method requires  $\sqrt{\frac{|N|(t\pm 1)}{2D}}$  to be small.

## See also:

`sympy.solvers.diophantine.diop_DN` (page 1067)

## References

[R364] (page 1249)

## Examples

```
>>> from sympy.solvers.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

## `transformation_to_DN`

`sympy.solvers.diophantine.transformation_to_DN(eq)`

This function transforms general quadratic,  $ax^2 + bxy + cy^2 + dx + ey + f = 0$  to more easy to deal with  $X^2 - DY^2 = N$  form.

This is used to solve the general quadratic equation by transforming it to the latter form. Refer [R365] (page 1249) for more detailed information on the transformation. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

$$\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$$

See also:

`sympy.solvers.diophantine.find_DN` (page 1070)

## References

[R365] (page 1249)

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine import transformation_to_DN
>>> from sympy.solvers.diophantine import classify_diop
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([
[1/26, 3/26],
[0, 1/13]])
>>> B
Matrix([
[-6/13],
[-4/13]])
```

A, B returned are such that  $\text{Transpose}((x \ y)) = A * \text{Transpose}((X \ Y)) + B$ . Substituting these values for  $x$  and  $y$  and a bit of simplifying work will give an equation of the form  $x^2 - Dy^2 = N$ .

```
>>> from sympy.abc import X, Y
>>> from sympy import Matrix, simplify, Subs
>>> u = (A*Matrix([X, Y]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([X, Y]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for  $x$  and  $y$  and do `simplify()`.

```
>>> eq = simplify(Subs(x**2 - 3*x*y - y**2 - 2*y + 1, (x, y), (u, v)).doit())
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.

```
>>> eq * 676
X**2 - 13*Y**2 + 884
```

If only the final equation is needed, `find_DN()` can be used.

## find\_DN

```
sympy.solvers.diophantine.find_DN(eq)
```

This function returns a tuple,  $(D, N)$  of the simplified form,  $x^2 - Dy^2 = N$ , corresponding to the general quadratic,  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ .

Solving the general quadratic is then equivalent to solving the equation  $X^2 - DY^2 = N$  and transforming the solutions by using the transformation matrices returned by `transformation_to_DN()`.

**See also:**

`sympy.solvers.diophantine.transformation_to_DN` (page 1069)

## References

[R366] (page 1249)

## Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine import find_DN
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)
```

Interpretation of the output is that we get  $X^2 - 13Y^2 = -884$  after transforming  $x^2 - 3xy - y^2 - 2y + 1$  using the transformation returned by `transformation_to_DN()`.

## diop\_ternary\_quadratic

```
sympy.solvers.diophantine.diop_ternary_quadratic(eq)
```

Solves the general quadratic ternary form,  $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$ .

Returns a tuple  $(x, y, z)$  which is a base solution for the above equation. If there are no solutions,  $(None, None, None)$  is returned.

## Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
```

```
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y - 8*x*y)
(9, 1, 5)
```

## square\_factor

```
sympy.solvers.diophantine.square_factor(a)
```

Returns an integer  $c$  s.t.  $a = c^2k$ ,  $c, k \in \mathbb{Z}$ . Here  $k$  is square free.

### Examples

```
>>> from sympy.solvers.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(36)
6
>>> square_factor(1)
1
```

## descent

```
sympy.solvers.diophantine.descent(A, B)
```

Lagrange's descent() with lattice-reduction to find solutions to  $x^2 = Ay^2 + Bz^2$ .

Here  $A$  and  $B$  should be square free and pairwise prime. Always should be called with suitable  $A$  and  $B$  so that the above equation has solutions.

This is more faster than the normal Lagrange's descent algorithm because the gaussian reduction is used.

### References

[R367] (page 1249)

### Examples

```
>>> from sympy.solvers.diophantine import descent
>>> descent(3, 1) # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

$(x, y, z) = (1, 0, 1)$  is a solution to the above equation.

```
>>> descent(41, -113)
(-16, -3, 1)
```

## diop\_general\_pythagorean

```
sympy.solvers.diophantine.diop_general_pythagorean(eq, param=m)
    Solves the general pythagorean equation,  $a_1^2x_1^2 + a_2^2x_2^2 + \dots + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$ .
```

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

### Examples

```
>>> from sympy.solvers.diophantine import diop_general_pythagorean
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 + 15*m3**2 + 15*m4**2, 15*m1*m4, 12*m2*m3)
```

## diop\_general\_sum\_of\_squares

```
sympy.solvers.diophantine.diop_general_sum_of_squares(eq, limit=1)
    Solves the equation  $x_1^2 + x_2^2 + \dots + x_n^2 - k = 0$ .
```

Returns at most `limit` number of solutions. Currently there is no way to set `limit` using higher level API's like `diophantine()` or `diop_solve()` but that will be fixed soon.

### Examples

```
>>> from sympy.solvers.diophantine import diop_general_sum_of_squares
>>> from sympy.abc import a, b, c, d, e, f
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
set([(0, 48, 5, 4, 0)])
```

## partition

```
sympy.solvers.diophantine.partition(n, k=None, zeros=False)
```

Returns a generator that can be used to generate partitions of an integer  $n$ .

A partition of  $n$  is a set of positive integers which add upto  $n$ . For example, partitions of 3 are 3 , 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If `k` equals `None`, then all possible partitions are returned irrespective of their size, otherwise only the partitions of size `k` are returned. If there are no partitions of  $n$  with size `k` then an empty tuple is returned. If the `zero` parameter is set to `True` then a suitable number of zeros are added at the end of every partition of size less than `k`.

`zero` parameter is considered only if `k` is not `None`. When the partitions are over, the last `next()` call throws the `StopIteration` exception, so this function should always be used inside a `try - except` block.

### Examples

```
>>> from sympy.solvers.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(3, 1, 1)
>>> next(g)
(2, 2, 1)
```

### sum\_of\_three\_squares

`sympy.solvers.diophantine.sum_of_three_squares(n)`

Returns a 3-tuple  $(a, b, c)$  such that  $a^2 + b^2 + c^2 = n$  and  $a, b, c \geq 0$ .

Returns  $(\text{None}, \text{None}, \text{None})$  if  $n = 4^a(8m + 7)$  for some  $a, m \in \mathbb{Z}$ . See [R368] (page 1249) for more details.

#### References

[R368] (page 1249)

#### Examples

```
>>> from sympy.solvers.diophantine import sum_of_three_squares
>>> sum_of_three_squares(44542)
(207, 37, 18)
```

### sum\_of\_four\_squares

`sympy.solvers.diophantine.sum_of_four_squares(n)`

Returns a 4-tuple  $(a, b, c, d)$  such that  $a^2 + b^2 + c^2 + d^2 = n$ .

Here  $a, b, c, d \geq 0$ .

#### References

[R369] (page 1249)

#### Examples

```
>>> from sympy.solvers.diophantine import sum_of_four_squares
>>> sum_of_four_squares(3456)
(8, 48, 32, 8)
>>> sum_of_four_squares(1294585930293)
(0, 1137796, 2161, 1234)
```

### 3.27.6 Internal Functions

These functions are intended for the internal use in Diophantine module.

#### merge\_solution

```
sympy.solvers.diophantine.merge_solution(var, var_t, solution)
```

This is used to construct the full solution from the solutions of sub equations.

For example when solving the equation  $(x - y)(x^2 + y^2 - z^2) = 0$ , solutions for each of the equations  $x - y = 0$  and  $x^2 + y^2 - z^2$  are found independently. Solutions for  $x - y = 0$  are  $(x, y) = (t, t)$ . But we should introduce a value for  $z$  when we output the solution for the original equation. This function converts  $(t, t)$  into  $(t, t, n_1)$  where  $n_1$  is an integer parameter.

#### divisible

```
sympy.solvers.diophantine.divisible(a, b)
```

Returns *True* if  $a$  is divisible by  $b$  and *False* otherwise.

#### extended\_euclid

```
sympy.solvers.diophantine.extended_euclid(a, b)
```

For given  $a, b$  returns a tuple containing integers  $x, y$  and  $d$  such that  $ax + by = d$ . Here  $d = \gcd(a, b)$ .

#### Examples

```
>>> from sympy.solvers.diophantine import extended_euclid
>>> extended_euclid(4, 6)
(-1, 1, 2)
>>> extended_euclid(3, 5)
(2, -1, 1)
```

#### PQa

```
sympy.solvers.diophantine.PQa(P_0, Q_0, D)
```

Returns useful information needed to solve the Pell equation.

There are six sequences of integers defined related to the continued fraction representation of  $\frac{P+\sqrt{D}}{Q}$ , namely  $\{P_i\}, \{Q_i\}, \{a_i\}, \{A_i\}, \{B_i\}, \{G_i\}$ .  $\text{PQa}()$  Returns these values as a 6-tuple in the same order as mentioned above. Refer [R370] (page 1249) for more detailed information.

#### References

[R370] (page 1249)

#### Examples

```
>>> from sympy.solvers.diophantine import PQa
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```

## equivalent

```
sympy.solvers.diophantine.equivalent(u, v, r, s, D, N)
```

Returns True if two solutions  $(u, v)$  and  $(r, s)$  of  $x^2 - Dy^2 = N$  belongs to the same equivalence class and False otherwise.

Two solutions  $(u, v)$  and  $(r, s)$  to the above equation fall to the same equivalence class iff both  $(ur - Dvs)$  and  $(us - vr)$  are divisible by  $N$ . See reference [R371] (page 1249). No test is performed to test whether  $(u, v)$  and  $(r, s)$  are actually solutions to the equation. User should take care of this.

### References

[R371] (page 1249)

## Examples

```
>>> from sympy.solvers.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

## simplified

```
sympy.solvers.diophantine.simplified(x, y, z)
Simplify the solution  $(x, y, z)$ .
```

## parametrize\_ternary\_quadratic

```
sympy.solvers.diophantine.parametrize_ternary_quadratic(eq)
```

Returns the parametrized general solution for the ternary quadratic equation `eq` which has the form  $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$ .

### References

[R372] (page 1249)

## Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import parametrize_ternary_quadratic
>>> parametrize_ternary_quadratic(x**2 + y**2 - z**2)
(2*p*q, p**2 - q**2, p**2 + q**2)
```

Here  $p$  and  $q$  are two co-prime integers.

```
>>> parametrize_ternary_quadratic(3*x**2 + 2*y**2 - z**2 - 2*x*y + 5*y*z - 7*y*z)
(2*p**2 - 2*p*q - q**2, 2*p**2 + 2*p*q - q**2, 2*p**2 - 2*p*q + 3*q**2)
>>> parametrize_ternary_quadratic(124*x**2 - 30*y**2 - 7729*z**2)
(-1410*p**2 - 363263*q**2, 2700*p**2 + 30916*p*q - 695610*q**2, -60*p**2 + 5400*p*q + 15458*q**2)
```

## diop\_ternary\_quadratic\_normal

`sympy.solvers.diophantine.diop_ternary_quadratic_normal(eq)`

Solves the quadratic ternary diophantine equation,  $ax^2 + by^2 + cz^2 = 0$ .

Here the coefficients  $a$ ,  $b$ , and  $c$  should be non zero. Otherwise the equation will be a quadratic binary or univariate equation. If solvable, returns a tuple  $(x, y, z)$  that satisfies the given equation. If the equation does not have integer solutions,  $(None, None, None)$  is returned.

### Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine import diop_ternary_quadratic_normal
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

## ldescent

`sympy.solvers.diophantine.ldescent(A, B)`

Uses Lagrange's method to find a non trivial solution to  $w^2 = Ax^2 + By^2$ .

Here,  $A \neq 0$  and  $B \neq 0$  and  $A$  and  $B$  are square free. Output a tuple  $(w_0, x_0, y_0)$  which is a solution to the above equation.

### References

[R373] (page 1249), [R374] (page 1249)

### Examples

```
>>> from sympy.solvers.diophantine import ldescent
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that  $x = -1, y = 0$  and  $w = 2$  is a solution to the equation  $w^2 = 4x^2 - 7y^2$

```
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```

## gaussian\_reduce

`sympy.solvers.diophantine.gaussian_reduce(w, a, b)`

Returns a reduced solution  $(x, z)$  to the congruence  $X^2 - aZ^2 \equiv 0 \pmod{b}$  so that  $x^2 + |a|z^2$  is minimal.

## References

[R375] (page 1249), [R376] (page 1249)

## holzer

`sympy.solvers.diophantine.holzer(x_0, y_0, z_0, a, b, c)`

Simplify the solution  $(x_0, y_0, z_0)$  of the equation  $ax^2 + by^2 = cz^2$  with  $a, b, c > 0$  and  $z_0^2 \geq |ab|$  to a new reduced solution  $(x, y, z)$  such that  $z^2 \leq |ab|$ .

## prime\_as\_sum\_of\_two\_squares

`sympy.solvers.diophantine.prime_as_sum_of_two_squares(p)`

Represent a prime  $p$  which is congruent to 1 mod 4, as a sum of two squares.

## Examples

```
>>> from sympy.solvers.diophantine import prime_as_sum_of_two_squares
>>> prime_as_sum_of_two_squares(5)
(2, 1)
```

## pairwise\_prime

`sympy.solvers.diophantine.pairwise_prime(a, b, c)`

Transform  $ax^2 + by^2 + cz^2 = 0$  into an equivalent equation  $a'x^2 + b'y^2 + c'z^2 = 0$  where  $a', b', c'$  are pairwise relatively prime.

Returns a tuple containing  $a', b', c'$ .  $\gcd(a, b, c)$  should equal 1 for this to work. The solutions for  $ax^2 + by^2 + cz^2 = 0$  can be recovered from the solutions of  $a'x^2 + b'y^2 + c'z^2 = 0$ .

See also:

`sympy.solvers.diophantine.make_prime` (page 1078), `sympy.solvers.diophantine.reconstruct` (page 1078)

## Examples

```
>>> from sympy.solvers.diophantine import pairwise_prime
>>> pairwise_prime(6, 15, 10)
(5, 2, 3)
```

**make\_prime**

```
sympy.solvers.diophantine.make_prime(a, b, c)
```

Transform the equation  $ax^2 + by^2 + cz^2 = 0$  to an equivalent equation  $a'x^2 + b'y^2 + c'z^2 = 0$  with  $\gcd(a', b') = 1$ .

Returns a tuple  $(a', b', c')$  which satisfies above conditions. Note that in the returned tuple  $\gcd(a', c')$  and  $\gcd(b', c')$  can take any value.

See also:

```
sympy.solvers.diophantine.reconstruct (page 1078)
```

**Examples**

```
>>> from sympy.solvers.diophantine import make_prime
>>> make_prime(4, 2, 7)
(2, 1, 14)
```

**reconstruct**

```
sympy.solvers.diophantine.reconstruct(a, b, z)
```

Reconstruct the  $z$  value of an equivalent solution of  $ax^2 + by^2 + cz^2$  from the  $z$  value of a solution of a transformed version of the above equation.

**transformation\_to\_normal**

```
sympy.solvers.diophantine.transformation_to_normal(eq)
```

Returns the transformation Matrix from general ternary quadratic equation  $eq$  to normal form.

General form of the ternary quadratic equation is  $ax^2 + by^2 + cz^2 + dxy + eyz + fxz$ . This function returns a 3X3 transformation Matrix which transforms the former equation to the form  $ax^2 + by^2 + cz^2 = 0$ . This is not used in solving ternary quadratics. Only implemented for the sake of completeness.

## 3.28 Inequality Solvers

```
sympy.solvers.inequalities.solve_rational_inequalities(eqs)
```

Solve a system of rational inequalities with rational coefficients.

See also:

```
solve_poly_inequality (page 1079)
```

**Examples**

```
>>> from sympy.abc import x
>>> from sympy import Poly
>>> from sympy.solvers.inequalities import solve_rational_inequalities
```

```
>>> solve_rational_inequalities([[  
... ((Poly(-x + 1), Poly(1, x)), '>='),  
... ((Poly(-x + 1), Poly(1, x)), '<=')]]))  
{1}
```

```
>>> solve_rational_inequalities([[  
... ((Poly(x), Poly(1, x)), '!='),  
... ((Poly(-x + 1), Poly(1, x)), '>=')]]))  
(-oo, 0) U (0, 1]
```

`sympy.solvers.inequalities.solve_poly_inequality(poly, rel)`

Solve a polynomial inequality with rational coefficients.

See also:

[solve\\_poly\\_inequalities](#) (page 1079)

### Examples

```
>>> from sympy import Poly  
>>> from sympy.abc import x  
>>> from sympy.solvers.inequalities import solve_poly_inequality  
  
>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')  
[{0}]  
  
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')  
[(-oo, -1), (-1, 1), (1, oo)]  
  
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')  
[{-1}, {1}]
```

`sympy.solvers.inequalities.solve_poly_inequalities(polys)`

Solve polynomial inequalities with rational coefficients.

### Examples

```
>>> from sympy.solvers.inequalities import solve_poly_inequalities  
>>> from sympy.polys import Poly  
>>> from sympy.abc import x  
>>> solve_poly_inequalities(((  
... Poly(x**2 - 3, ">"), (  
... Poly(-x**2 + 1, ">"))))  
(-oo, -sqrt(3)) U (-1, 1) U (sqrt(3), oo)
```

`sympy.solvers.inequalities.reduce_rational_inequalities(exprs, gen, relational=True)`

Reduce a system of rational inequalities with rational coefficients.

### Examples

```
>>> from sympy import Poly, Symbol  
>>> from sympy.solvers.inequalities import reduce_rational_inequalities
```

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

>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)

>>> reduce_rational_inequalities([[x + 2 > 0]], x)
And(-2 < x, x < oo)
>>> reduce_rational_inequalities([[(x + 2, ">")]], x)
And(-2 < x, x < oo)
>>> reduce_rational_inequalities([[x + 2]], x)
Eq(x, -2)
```

`sympy.solvers.inequalities.reduce_abs_inequality(expr, rel, gen)`

Reduce an inequality with nested absolute values.

See also:

[reduce\\_abs\\_inequalities](#) (page 1080)

### Examples

```
>>> from sympy import Abs, Symbol
>>> from sympy.solvers.inequalities import reduce_abs_inequality
>>> x = Symbol('x', extended_real=True)

>>> reduce_abs_inequality(Abs(x - 5) - 3, '<', x)
And(2 < x, x < 8)

>>> reduce_abs_inequality(Abs(x + 2)*3 - 13, '<', x)
And(-19/3 < x, x < 7/3)
```

`sympy.solvers.inequalities.reduce_abs_inequalities(exprs, gen)`

Reduce a system of inequalities with nested absolute values.

See also:

[reduce\\_abs\\_inequality](#) (page 1080)

### Examples

```
>>> from sympy import Abs, Symbol
>>> from sympy.abc import x
>>> from sympy.solvers.inequalities import reduce_abs_inequalities
>>> x = Symbol('x', extended_real=True)

>>> reduce_abs_inequalities([(Abs(3*x - 5) - 7, '<'),
... (Abs(x + 25) - 13, '>')], x)
And(-2/3 < x, Or(And(-12 < x, x < oo), And(-oo < x, x < -38)), x < 4)

>>> reduce_abs_inequalities([(Abs(x - 4) + Abs(3*x - 5) - 7, '<')], x)
And(1/2 < x, x < 4)
```

`sympy.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`

Reduce a system of inequalities with rational coefficients.

**Examples**

```
>>> from sympy import sympify as S, Symbol
>>> from sympy.abc import x, y
>>> from sympy.solvers.inequalities import reduce_inequalities

>>> reduce_inequalities(S(0) <= x + 3, [])
And(-3 <= x, x < oo)

>>> reduce_inequalities(S(0) <= x + y*2 - 1, [x])
-2*y + 1 <= x
```

`sympy.solvers.inequalities.solve_univariate_inequality(expr, gen, relational=True)`  
Solves a real univariate inequality.

**Examples**

```
>>> from sympy.solvers.inequalities import solve_univariate_inequality
>>> from sympy.core.symbol import Symbol
>>> x = Symbol('x', extended_real=True)

>>> solve_univariate_inequality(x**2 >= 4, x)
Or(And(-oo < x, x <= -2), And(2 <= x, x < oo))

>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
(-oo, -2] U [2, oo)
```

## 3.29 Solveset

This module contains functions to solve a single equation for a single variable.

Use `solveset()` (page 1081) to solve equations or expressions (assumed to be equal to 0) for a single variable. Solving an equation like  $x^{**}2 == 1$  can be done as follows:

```
>>> from sympy.solvers.solveset import *
>>> from sympy import Symbol, Eq
>>> x = Symbol('x')
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
```

Or one may manually rewrite the equation as an expression equal to 0:

```
>>> solveset(x**2 - 1, x)
{-1, 1}
```

The first argument for `solveset()` (page 1081) is an expression (equal to zero) or an equation and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solveset.solveset(f, symbol=None)`  
Solves a given inequality or equation with set as output

**Parameters** `f` : Expr or a relational.

The target equation or inequality

`symbol` : Symbol

The variable for which the equation is solved

**Returns** Set

A set of values for *symbol* for which *f* is True or is equal to zero. An *EmptySet* is returned if no solution is found.

*solveset* claims to be complete in the solution set that it returns.

**Raises** `NotImplementedError`

The algorithms for to find the solution of the given equation are not yet implemented.

**ValueError**

The input is not valid.

**RuntimeError**

It is a bug, please report to the github issue tracker.

**'solveset' uses two underlying functions 'solveset\_real' and 'solveset\_complex' to solve equations. They are the solvers for real and complex domain respectively. The domain of the solver is decided by the assumption on the variable for which the equation is being solved.**

See also:

`solveset_real` (page 1083) solver for real domain

`solveset_complex` (page 1084) solver for complex domain

**Examples**

```
>>> from sympy import exp, Symbol, Eq, pprint
>>> from sympy.solvers.solveset import solveset
>>> from sympy.abc import x
```

• Symbols in Sympy are complex by default. A complex variable will lead to the solving of the equation in complex domain.

```
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers()}
```

• If you want to solve equation in real domain by the *solveset* interface, then specify the variable to real. Alternatively use *solveset\_real*.

```
>>> x = Symbol('x', extended_real=True)
>>> solveset(exp(x) - 1, x)
{0}
>>> solveset(Eq(exp(x), 1), x)
{0}
```

- Inequalities are always solved in the real domain irrespective of the assumption on the variable for which the inequality is solved.

```
>>> solveset(exp(x) > 1, x)
(0, oo)
```

`sympy.solvers.solveset.solveset_real(f, symbol)`  
Solves a real valued equation.

**Parameters** `f` : Expr

The target equation

`symbol` : Symbol

The variable for which the equation is solved

**Returns** Set

A set of values for `symbol` for which `f` is equal to zero. An `EmptySet` is returned if no solution is found.

`solveset_real` claims to be complete in the set of the solution it returns.

**Raises** `NotImplementedError`

The algorithms for to find the solution of the given equation are not yet implemented.

**ValueError**

The input is not valid.

**RuntimeError**

It is a bug, please report to the github issue tracker.

**See Also**

=====

`solveset_complex` : solver for complex domain

## Examples

```
>>> from sympy import Symbol, exp, sin, sqrt, I
>>> from sympy.solvers.solveset import solveset_real
>>> x = Symbol('x', extended_real=True)
>>> a = Symbol('a', extended_real=True, finite=True, positive=True)
>>> solveset_real(x**2 - 1, x)
{-1, 1}
>>> solveset_real(sqrt(5*x + 6) - 2 - x, x)
{-1, 2}
>>> solveset_real(x - I, x)
EmptySet()
>>> solveset_real(x - a, x)
{a}
>>> solveset_real(exp(x) - a, x)
{log(a)}
```

- In case the equation has infinitely many solutions an infinitely indexed *ImageSet* is returned.

```
>>> solveset_real(sin(x) - 1, x)
ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
```

- If the equation is true for any arbitrary value of the symbol a *S.Reals* set is returned.

```
>>> solveset_real(x - x, x)
(-oo, oo)
```

`sympy.solvers.solveset.solveset_complex(f, symbol)`  
Solve a complex valued equation.

**Parameters** `f` : Expr

The target equation

`symbol` : Symbol

The variable for which the equation is solved

**Returns** Set

A set of values for `symbol` for which `f` equal to zero. An *EmptySet* is returned if no solution is found.

`solveset_complex` claims to be complete in the solution set that it returns.

**Raises** `NotImplementedError`

The algorithms for to find the solution of the given equation are not yet implemented.

**ValueError**

The input is not valid.

**RuntimeError**

It is a bug, please report to the github issue tracker.

**See also:**

`solveset_real` ([page 1083](#)) solver for real domain

**Examples**

```
>>> from sympy import Symbol, exp
>>> from sympy.solvers.solveset import solveset_complex
>>> from sympy.abc import x, a, b, c
>>> solveset_complex(a*x**2 + b*x + c, x)
{-b/(2*a) - sqrt(-4*a*c + b**2)/(2*a), -b/(2*a) + sqrt(-4*a*c + b**2)/(2*a)}
```

- Due to the fact that complex extension of my real valued functions are multivariate even some simple equations can have infinitely many solution.

```
>>> solveset_complex(exp(x) - 1, x)
ImageSet(Lambda(_n, 2*_n*I*pi), Integers())
```

---

```
sympy.solvers.solveset.invert_real(f_x, y, x)
```

Inverts a real valued function

Reduces the real valued equation  $f(x) = y$  to a set of equations  $\{g_1(x) = h_1(y), g_2(x) = h_2(y), \dots, g_n(x) = h_n(y)\}$  where  $g_i(x)$  is a simpler function than  $f(x)$ . The return value is a tuple  $(g(x), \text{set\_h})$ , where  $g(x)$  is a function of  $x$  and  $\text{set\_h}$  is the set of functions  $\{h_1(y), h_2(y), \dots, h_n(y)\}$ . Here,  $y$  is not necessarily a symbol.

See also:

[invert\\_complex](#) (page 1085)

### Examples

```
>>> from sympy.solvers.solveset import invert_real
>>> from sympy import tan, Abs, exp
>>> from sympy.abc import x, y, n
>>> invert_real(exp(Abs(x)), y, x)
(x, {-log(y), log(y)})
>>> invert_real(exp(x), 1, x)
(x, {0})
>>> invert_real(Abs(x**31 + x), y, x)
(x**31 + x, {-y, y})
>>> invert_real(tan(x), y, x)
(x, ImageSet(Lambda(_n, _n*pi + atan(y)), Integers())))
```

```
sympy.solvers.solveset.invert_complex(f_x, y, x)
```

Inverts a complex valued function.

Reduces the complex valued equation  $f(x) = y$  to a set of equations  $\{g_1(x) = h_1(y), g_2(x) = h_2(y), \dots, g_n(x) = h_n(y)\}$  where  $g_i(x)$  is a simpler function than  $f(x)$ . The return value is a tuple  $(g(x), \text{set\_h})$ , where  $g(x)$  is a function of  $x$  and  $\text{set\_h}$  is the set of function  $\{h_1(y), h_2(y), \dots, h_n(y)\}$ . Here,  $y$  is not necessarily a symbol.

Note that *invert\_complex* and *invert\_real* don't always produce the same result even for a seemingly simple function like `exp(x)` because the complex extension of real valued `log` is multivariate in the complex system and has infinitely many branches. If you are working with real values only or you are not sure with function to use you should use *invert\_real*.

See also:

[invert\\_real](#) (page 1084)

### Examples

```
>>> from sympy.solvers.solveset import invert_complex
>>> from sympy.abc import x, y
>>> from sympy import exp, log
>>> invert_complex(log(x), y, x)
(x, {exp(y)})
>>> invert_complex(log(x), 0, x) # Second parameter is not a symbol
(x, {1})
>>> invert_complex(exp(x), y, x)
(x, ImageSet(Lambda(_n, I*(2*_n*pi + arg(y)) + log(Abs(y))), Integers())))
```

```
sympy.solvers.solveset.domain_check(f, symbol, p)
```

Returns False if point p is infinite or any subexpression of f is infinite or becomes so after replacing symbol with p. If none of these conditions is met then True will be returned.

### Examples

```
>>> from sympy import Mul, oo
>>> from sympy.abc import x
>>> from sympy.solvers.solveset import domain_check
>>> g = 1/(1 + (1/(x + 1))**2)
>>> domain_check(g, x, -1)
False
>>> domain_check(x**2, x, 0)
True
>>> domain_check(1/x, x, oo)
False
```

- The function relies on the assumption that the original form of the equation has not been changed by automatic simplification.

```
>>> domain_check(x/x, x, 0) # x/x is automatically simplified to 1
True
```

- To deal with automatic evaluations use evaluate=False:

```
>>> domain_check(Mul(x, 1/x, evaluate=False), x, 0)
False
```

## 3.29.1 Diophantine Equations (DEs)

See *Diophantine* (page 1060)

## 3.29.2 Inequalities

See *Inequality Solvers* (page 1078)

## 3.29.3 Ordinary Differential equations (ODEs)

See *ODE* (page 984).

## 3.29.4 Partial Differential Equations (PDEs)

See *PDE* (page 1036).

## 3.30 Tensor Module

A module to manipulate symbolic objects with indices including tensors

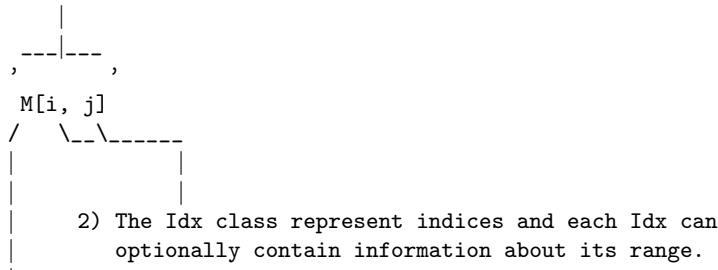
### 3.30.1 Contents

#### Indexed Objects

Module that defines indexed objects

The classes IndexedBase, Indexed and Idx would represent a matrix element  $M[i, j]$  as in the following graph:

- 1) The `Indexed` class represents the entire indexed object.



- 3) `IndexedBase` represents the ‘stem’ of an indexed object, here ‘`M`’. The stem used by itself is usually taken to represent the entire array.

There can be any number of indices on an Indexed object. No transformation properties are implemented in these Base objects, but implicit contraction of repeated indices is supported.

Note that the support for complicated (i.e. non-atomic) integer expressions as indices is limited. (This should be improved in future releases.)

#### Examples

To express the above matrix element example you would write:

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of Indexed objects:

```
>>> x = IndexedBase('x')
>>> M[i, j]*x[j]
x[j]*M[i, j]
```

If the indexed objects will be converted to component based arrays, e.g. with the code printers or the autowrap framework, you also need to provide (symbolic or numerical) dimensions. This can be done by passing an optional shape parameter to `IndexedBase` upon construction:

```
>>> dim1, dim2 = symbols('dim1 dim2', integer=True)
>>> A = IndexedBase('A', shape=(dim1, 2*dim1, dim2))
>>> A.shape
(dim1, 2*dim1, dim2)
>>> A[i, j, 3].shape
(dim1, 2*dim1, dim2)
```

If an IndexedBase object has no shape information, it is assumed that the array is as large as the ranges of its indices:

```
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
(m, n)
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods `get_indices()` and `get_contraction_structure()`:

```
>>> from sympy.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j])
({set([i])}, {})
>>> get_contraction_structure(A[i, j, j])
{(j,): set([A[i, j, j]])}
```

See the appropriate docstrings for a detailed explanation of the output.

### class `sympy.tensor.indexed.Idx`

Represents an integer index as an Integer or integer expression.

There are a number of ways to create an `Idx` object. The constructor takes two arguments:

`label` An integer or a symbol that labels the index.

`range` Optionally you can specify a range as either

- Symbol or integer: This is interpreted as a dimension. Lower and upper bounds are set to 0 and range - 1, respectively.
- tuple: The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: the `Idx` constructor is rather pedantic in that it only accepts integer arguments. The only exception is that you can use `oo` and `-oo` to specify an unbounded range. For all other cases, both label and bounds must be declared as integers, e.g. if `n` is given as an argument then `n.is_integer` must return `True`.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

### Examples

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols, oo
>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer Symbol is created and the bounds are both `None`:

```
>>> idx = Idx('qwerty'); idx
qwerty
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

The label can be a literal integer instead of a string/Symbol:

```
>>> idx = Idx(2, n); idx.lower, idx.upper
(0, n - 1)
>>> idx.label
2
```

#### label

Returns the label (Integer or integer expression) of the Idx object.

### Examples

```
>>> from sympy import Idx, Symbol
>>> Idx(2).label
2
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

#### lower

Returns the lower bound of the Index.

### Examples

```
>>> from sympy import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

**upper**

Returns the upper bound of the Index.

**Examples**

```
>>> from sympy import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

**class sympy.tensor.indexed.Indexed**

Represents a mathematical object with indices.

```
>>> from sympy.tensor import Indexed, IndexedBase, Idx
>>> from sympy import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects are created via IndexedBase:

```
>>> A = IndexedBase('A')
>>> Indexed('A', i, j) == A[i, j]
True
```

**base**

Returns the IndexedBase of the Indexed object.

**Examples**

```
>>> from sympy.tensor import Indexed, IndexedBase, Idx
>>> from sympy import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

**indices**

Returns the indices of the Indexed object.

**Examples**

```
>>> from sympy.tensor import Indexed, Idx
>>> from sympy import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```

**ranges**

Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains `None` instead of a tuple.

### Examples

```
>>> from sympy import Indexed, Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

#### rank

Returns the rank of the `Indexed` object.

### Examples

```
>>> from sympy.tensor import Indexed, Idx
>>> from sympy import symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

#### shape

Returns a list with dimensions of each index.

`Dimensions` is a property of the array, not of the indices. Still, if the `IndexedBase` does not define a `shape` attribute, it is assumed that the ranges of the indices correspond to the shape of the array.

```
>>> from sympy.tensor.indexed import IndexedBase, Idx
>>> from sympy import symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

## class `sympy.tensor.indexed.IndexedBase`

Represent the base or stem of an indexed object

The `IndexedBase` class represent an array that contains elements. The main purpose of this class is to allow the convenient creation of objects of the `Indexed` class. The `__getitem__` method of `IndexedBase` returns an instance of `Indexed`. Alone, without indices, the `IndexedBase` class can be used as a notation for e.g. matrix equations, resembling what you could do with the `Symbol` class. But, the `IndexedBase` class adds functionality that is not available for `Symbol` instances:

- An IndexedBase object can optionally store shape information. This can be used in to check array conformance and conditions for numpy broadcasting. (TODO)
- An IndexedBase object implements syntactic sugar that allows easy symbolic representation of array operations, using implicit summation of repeated indices.
- The IndexedBase object symbolizes a mathematical structure equivalent to arrays, and is recognized as such for code generation and automatic compilation and wrapping.

```
>>> from sympy.tensor import IndexedBase, Idx
>>> from sympy import symbols
>>> A = IndexedBase('A'); A
A
>>> type(A)
<class 'sympy.tensor.indexed.IndexedBase'>
```

When an IndexedBase object receives indices, it returns an array with named axes, represented by an Indexed object:

```
>>> i, j = symbols('i j', integer=True)
>>> A[i, j, 2]
A[i, j, 2]
>>> type(A[i, j, 2])
<class 'sympy.tensor.indexed.Indexed'>
```

The IndexedBase constructor takes an optional shape argument. If given, it overrides any shape information in the indices. (But not the index ranges!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

#### args

Returns the arguments used to create this IndexedBase object.

#### Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).args
(A, (x, y))
```

#### label

Returns the label of the IndexedBase object.

#### Examples

```
>>> from sympy import IndexedBase
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

**shape**

Returns the shape of the IndexedBase object.

**Examples**

```
>>> from sympy import IndexedBase, Idx, Symbol
>>> from sympy.abc import x, y
>>> IndexedBase('A', shape=(x, y)).shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

**Methods**

Module with functions operating on IndexedBase, Indexed and Idx objects

- Check shape conformance
- Determine indices in resulting expression

etc.

Methods in this module could be implemented by calling methods on Expr objects instead. When things stabilize this could be a useful refactoring.

`sympy.tensor.index_methods.get_contraction_structure(expr)`

Determine dummy indices of `expr` and describe its structure

By *dummy* we mean indices that are summation indices.

The structure of the expression is determined and described as follows:

1.A conforming summation of Indexed objects is described with a dict where the keys are summation indices and the corresponding values are sets containing all terms for which the summation applies. All Add objects in the SymPy expression tree are described like this.

2.For all nodes in the SymPy expression tree that are *not* of type Add, the following applies:

If a node discovers contractions in one of its arguments, the node itself will be stored as a key in the dict. For that key, the corresponding value is a list of dicts, each of which is the result of a recursive call to `get_contraction_structure()`. The list contains only dicts for the non-trivial deeper contractions, omitting dicts with None as the one and only key.

---

**Note:** The presence of expressions among the dictionary keys indicates multiple levels of index contractions. A nested dict displays nested contractions and may itself contain dicts from a deeper level. In practical calculations the summation in the deepest nested level must be calculated first so that the outer expression can access the resulting indexed object.

## Examples

```
>>> from sympy.tensor.index_methods import get_contraction_structure
>>> from sympy import symbols, default_sort_key
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j])
{(i,): set([x[i]*y[i]]), (j,): set([A[j, j]])}
>>> get_contraction_structure(x[i]*y[j])
{None: set([x[i]*y[j]])}
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None]
set([x[i, i]*y[j, j]])
```

Factors are contracted “first”:

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key)
[{(i,): set([x[i, i]]), (j,): set([y[j, j]])}]
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a Mul with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(x[j]*A[i, j] + y[i])*x[i], (i,)]
>>> d[(i,)]
set([(x[j]*A[i, j] + y[i])*x[i]])
>>> d[x[i]*(A[i, j]*x[j] + y[i])]
[{None: set([y[i]]), (j,): set([x[j]*A[i, j]])}]
```

Powers with contractions in either base or exponent will also be found as keys in the dictionary, mapping to a list of results from recursive calls:

```
>>> d = get_contraction_structure(A[j, j]**A[i, i])
>>> d[None]
set([A[j, j]**A[i, i]])
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0]
{(j,): set([A[j, j]])}
>>> nested_contractions[1]
{(i,): set([A[i, i]])}
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> from sympy import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
```

```

...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...     # treat outermost contractions here

```

`sympy.tensor.index_methods.get_indices(expr)`  
Determine the outer indices of expression `expr`

By *outer* we mean indices that are not summation indices. Returns a set and a dict. The set contains outer indices and the dict contains information about index symmetries.

### Examples

```

>>> from sympy.tensor.index_methods import get_indices
>>> from sympy import symbols
>>> from sympy.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)

```

The indices of the total expression is determined, Repeated indices imply a summation, for instance the trace of a matrix A:

```

>>> get_indices(A[i, i])
({set(), {}})

```

In the case of many terms, the terms are required to have identical outer indices. Else an `IndexConformanceException` is raised.

```

>>> get_indices(x[i] + A[i, j]*y[j])
({set([i]), {}})

```

### Exceptions

An `IndexConformanceException` means that the terms ar not compatible, e.g.

```

>>> get_indices(x[i] + y[j])
(...)
IndexConformanceException: Indices are not consistent: x(i) + y(j)

```

**Warning:** The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```

>>> get_indices((x[i] + A[i, j]*y[j])*x[j])
({set([i, j]), {}})

```

This is correct and may appear convenient, but you need to be careful with this as SymPy will happily `.expand()` the product, if requested. The resulting expression would mix the outer *j* with the dummies inside the parenthesis, which makes it a different expression. To be on the safe side, it is best to avoid such ambiguities by using unique indices for all contractions that should be held separate.

## Tensor

```
class sympy.tensor.tensor._TensorManager
    Class to manage tensor properties.
```

### Notes

Tensors belong to tensor commutation groups; each group has a label `comm`; there are predefined labels:

0 tensors commuting with any other tensor

1 tensors anticommuting among themselves

2 tensors not commuting, apart with those with `comm=0`

Other groups can be defined using `set_comm`; tensors in those groups commute with those with `comm=0`; by default they do not commute with any other group.

`clear()`

Clear the `TensorManager`.

`comm_i2symbol(i)`

Returns the symbol corresponding to the commutation group number.

`comm_symbols2i(i)`

get the commutation group number corresponding to `i`

`i` can be a symbol or a number or a string

If `i` is not already defined its commutation group number is set.

`get_comm(i, j)`

Return the commutation parameter for commutation group numbers `i, j`

see `_TensorManager.set_comm`

`set_comm(i, j, c)`

set the commutation parameter `c` for commutation groups `i, j`

**Parameters** `i, j` : symbols representing commutation groups

`c` : group commutation number

### Notes

`i, j` can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default `c=None`.

The group commutation number `c` is assigned in correspondence to the group commutation symbols; it can be

0 commuting

1 anticommuting

None no commutation property

## Examples

`G` and `GH` do not commute with themselves and commute with each other; `A` is commuting.

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead, TensorManager
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0,i1,i2,i3,i4 = tensor_indices('i0:5', Lorentz)
>>> A = tensorhead('A', [Lorentz], [[1]])
>>> G = tensorhead('G', [Lorentz], [[1]], 'Gcomm')
>>> GH = tensorhead('GH', [Lorentz], [[1]], 'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

### `set_comms(*args)`

set the commutation group numbers `c` for symbols `i`, `j`

**Parameters** `args` : sequence of (`i`, `j`, `c`)

## class `sympy.tensor.TensorIndexType`

A `TensorIndexType` is characterized by its name and its metric.

**Parameters** `name` : name of the tensor type

`metric` : metric symmetry or metric object or `None`

`dim` : dimension, it can be a symbol or an integer or `None`

`eps_dim` : dimension of the epsilon tensor

`dummy_fmt` : name of the head of dummy indices

## Notes

The `metric` parameter can be: `metric = False` symmetric metric (in Riemannian geometry)

`metric = True` antisymmetric metric (for spinor calculus)

`metric = None` there is no metric

`metric` can be an object having `name` and `antisym` attributes.

If there is a metric the metric is used to raise and lower indices.

In the case of antisymmetric metric, the following raising and lowering conventions will be adopted:

`psi(a) = g(a, b)*psi(-b); chi(-a) = chi(b)*g(-b, -a)`

`g(-a, b) = delta(-a, b); g(b, -a) = -delta(a, -b)`

where `delta(-a, b) = delta(b, -a)` is the Kronecker `delta` (see `TensorIndex` for the conventions on indices).

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of  $SU(N)$  is ‘covariant’ and the conjugate representation is ‘contravariant’; for  $N > 2$  they are linearly independent.

`eps_dim` is by default equal to `dim`, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if `eps_dim` is not an integer `epsilon` is `None`.

## Examples

```
>>> from sympy.tensor.tensor import TensorIndexType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> Lorentz.metric
metric(Lorentz, Lorentz)
```

Examples with metric components data added, this means it is working on a fixed basis:

```
>>> Lorentz.data = [1, -1, -1, -1]
>>> Lorentz
TensorIndexType(Lorentz, 0)
>>> Lorentz.data
[[1 0 0 0]
[0 -1 0 0]
[0 0 -1 0]
[0 0 0 -1]]
```

## Attributes

<code>name</code>	(it is ‘metric’ or metric.name)
<code>metric_name</code>	
<code>metric_antisym</code>	
<code>metric</code>	(the metric tensor)
<code>delta</code>	(Kronecker delta)
<code>epsilon</code>	(the Levi-Civita epsilon tensor)
<code>dim</code>	
<code>dim_eps</code>	
<code>dummy_fmt</code>	
<code>data</code>	(a property to add ndarray values, to work in a specified basis.)

class `sympy.tensor.TensorIndex`

Represents an abstract tensor index.

**Parameters** `name` : name of the index, or `True` if you want it to be automatically assigned  
`tensortype` : `TensorIndexType` of the index  
`is_up` : flag for contravariant index

## Notes

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a `-` to the index name.

Dummy indices have a name with head given by `tensortype._dummy_fmt`

## Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorIndex, TensorSymmetry, TensorType, get_symmetric_g
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i = TensorIndex('i', Lorentz); i
i
```

```
>>> sym1 = TensorSymmetry(*get_symmetric_group_sgs(1))
>>> S1 = TensorType([Lorentz], sym1)
>>> A, B = S1('A,B')
>>> A(i)*B(-i)
A(L_0)*B(-L_0)
```

If you want the index name to be automatically assigned, just put `True` in the `name` field, it will be generated using the reserved character `_` in front of its name, in order to avoid conflicts with possible existing indices:

```
>>> i0 = TensorIndex(True, Lorentz)
>>> i0
_i0
>>> i1 = TensorIndex(True, Lorentz)
>>> i1
_i1
>>> A(i0)*B(-i1)
A(_i0)*B(-_i1)
>>> A(i0)*B(-i0)
A(L_0)*B(-L_0)
```

## Attributes

<code>name</code>	
<code>tensortype</code>	
<code>is_up</code>	

`sympy.tensor.tensor.tensor_indices(s, typ)`

Returns list of tensor indices given their names and their types

**Parameters** `s` : string of comma separated names of indices

`typ` : list of `TensorIndexType` of the indices

## Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
```

`class sympy.tensor.tensor.TensorSymmetry`

Monoterm symmetry of a tensor

**Parameters** `bsgs` : tuple (base, sgs) BSGS of the symmetry of the tensor

**See also:**

`sympy.combinatorics.tensor_can.get_symmetric_group_sgs` (page 243)

## Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor, are not covered.

## Examples

Define a symmetric tensor

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, TensorSymmetry, TensorType, get_symmetric_group_sgs
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = TensorSymmetry(get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

## Attributes

base	(base of the BSGS)
generators	(generators of the BSGS)
rank	(rank of the tensor)

`sympy.tensor.tensor.tensorsymmetry(*args)`

Return a `TensorSymmetry` object.

One can represent a tensor with any monoterm slot symmetry group using a BSGS.

`args` can be a BSGS `args[0]` base `args[1]` sgs

Usually tensors are in (direct products of) representations of the symmetric group; `args` can be a list of lists representing the shapes of Young tableaux

## Notes

For instance: [[1]] vector [[1]\*n] symmetric tensor of rank n [[n]] antisymmetric tensor of rank n [[2, 2]] monoterm slot symmetry of the Riemann tensor [[1], [1]] vector\*vector [[2], [1], [1]] (antisymmetric tensor)\*vector\*vector

Notice that with the shape [2, 2] we associate only the monoterm symmetries of the Riemann tensor; this is an abuse of notation, since the shape [2, 2] corresponds usually to the irreducible representation characterized by the monoterm symmetries and by the cyclic symmetry.

## Examples

Symmetric tensor using a Young tableau

```
>>> from sympy.tensor.tensor import TensorIndexType, TensorType, tensorsymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Symmetric tensor using a BSGS (base, strong generator set)

```
>>> from sympy.tensor.tensor import TensorSymmetry, get_symmetric_group_sgs
>>> sym2 = tensorsymmetry(*get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

`class sympy.tensor.tensor.TensorType`

Class of tensor types.

**Parameters** `index_types` : list of `TensorIndexType` of the tensor indices  
`symmetry` : `TensorSymmetry` of the tensor

### Examples

Define a symmetric tensor

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorsymmetry, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

### Attributes

<code>index_types</code>	
<code>symmetry</code>	
<code>types</code>	(list of <code>TensorIndexType</code> without repetitions)

class `sympy.tensor.tensor.TensorHead`

Tensor head of the tensor

**Parameters** `name` : name of the tensor  
`typ` : list of `TensorIndexType`  
`comm` : commutation group number

### Notes

A `TensorHead` belongs to a commutation group, defined by a symbol on number `comm` (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default `comm` is 0, the group of the commuting tensors.

### Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorhead, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> A = tensorhead('A', [Lorentz, Lorentz], [[1],[1]])
```

Examples with ndarray values, the components data assigned to the `TensorHead` object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor, see the other examples.

```
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+2*i for j in range(4)] for i in range(4)]
```

in order to retrieve data, it is also necessary to specify abstract indices enclosed by round brackets, then numerical indices inside square brackets.

```
>>> A(i0, i1)[0, 0]
0
>>> A(i0, i1)[2, 3] == 3+2*2
True
```

Notice that square brackets create a valued tensor expression instance:

```
>>> A(i0, i1)
A(i0, i1)
```

To view the data, just type:

```
>>> A.data
[[0 1 2 3]
 [2 3 4 5]
 [4 5 6 7]
 [6 7 8 9]]
```

Turning to a tensor expression, covariant indices get the corresponding components data corrected by the metric:

```
>>> A(i0, -i1).data
[[0 -1 -2 -3]
 [2 -3 -4 -5]
 [4 -5 -6 -7]
 [6 -7 -8 -9]]

>>> A(-i0, -i1).data
[[0 -1 -2 -3]
 [-2 3 4 5]
 [-4 5 6 7]
 [-6 7 8 9]]
```

while if all indices are contravariant, the `ndarray` remains the same

```
>>> A(i0, i1).data
[[0 1 2 3]
 [2 3 4 5]
 [4 5 6 7]
 [6 7 8 9]]
```

When all indices are contracted and components data are added to the tensor, accessing the data will return a scalar, no numpy object. In fact, numpy ndarrays are dropped to scalars if they contain only one element.

```
>>> A(i0, -i0)
A(L_0, -L_0)
>>> A(i0, -i0).data
-18
```

It is also possible to assign components data to an indexed tensor, i.e. a tensor with specified covariant and contravariant components. In this example, the covariant components data of the Electromagnetic tensor are injected into  $A$ :

```
>>> from sympy import symbols
>>> Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
>>> c = symbols('c', positive=True)
```

Let's define  $F$ , an antisymmetric tensor, we have to assign an antisymmetric matrix to it, because  $[[2]]$  stands for the Young tableau representation of an antisymmetric set of two elements:

```
>>> F = tensorhead('A', [Lorentz, Lorentz], [[2]])
>>> F(-i0, -i1).data = [
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:

```
>>> F(i0, i1).data
[[0 -E_x/c -E_y/c -E_z/c]
 [E_x/c 0 -B_z B_y]
 [E_y/c B_z 0 -B_x]
 [E_z/c -B_y B_x 0]]
```

and the mixed contravariant-covariant form:

```
>>> F(i0, -i1).data
[[0 E_x/c E_y/c E_z/c]
 [E_x/c 0 B_z -B_y]
 [E_y/c -B_z 0 B_x]
 [E_z/c B_y -B_x 0]]
```

To convert the numpy's ndarray to a sympy matrix, just cast:

```
>>> from sympy import Matrix
>>> Matrix(F.data)
Matrix([
[ 0, -E_x/c, -E_y/c, -E_z/c],
[E_x/c, 0, -B_z, B_y],
[E_y/c, B_z, 0, -B_x],
[E_z/c, -B_y, B_x, 0]])
```

Still notice, in this last example, that accessing components data from a tensor without specifying the indices is equivalent to assume that all indices are contravariant.

It is also possible to store symbolic components data inside a tensor, for example, define a four-momentum-like tensor:

```
>>> from sympy import symbols
>>> P = tensorhead('P', [Lorentz], [[1]])
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> P.data = [E, px, py, pz]
```

The contravariant and covariant components are, respectively:

```
>>> P(i0).data
[E p_x p_y p_z]
>>> P(-i0).data
[E -p_x -p_y -p_z]
```

The contraction of a 1-index tensor by itself is usually indicated by a power by two:

```
>>> P(i0)**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

As the power by two is clearly identical to  $P_\mu P^\mu$ , it is possible to simply contract the `TensorHead` object, without specifying the indices

```
>>> P**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

### Attributes

name	
index_types	
rank	
types	( equal to typ.types)
symmetry	(equal to typ.symmetry)
comm	(commutation group)

`commutes_with(other)`

Returns 0 if `self` and `other` commute, 1 if they anticommute.

Returns `None` if `self` and `other` neither commute nor anticommute.

```
class sympy.tensor.tensor.TensExpr
Abstract base class for tensor expressions
```

### Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A `TensExpr` can be a `TensAdd` or a `TensMul`.

`TensAdd` objects are put in canonical form using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

`TensMul` objects are formed by products of component tensors, and include a coefficient, which is a SymPy expression.

In the internal representation contracted indices are represented by `(ipos1, ipos2, icomp1, icomp2)`, where `icomp1` is the position of the component tensor with contravariant index, `ipos1` is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

`get_matrix()`

Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

### Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorsymmetry, TensorType
>>> from sympy import ones
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1]*2)
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> A = S2('A')
```

The tensor `A` is symmetric in its indices, as can be deduced by the [1, 1] Young tableau when constructing `sym2`. One has to be careful to assign symmetric component data to `A`, as the symmetry properties of data are currently not checked to be compatible with the defined tensor symmetry.

```
>>> from sympy.tensor.tensor import tensor_indices, tensorhead
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+i for j in range(4)] for i in range(4)]
>>> A(i0, i1).get_matrix()
Matrix([
[0, 1, 2, 3],
[1, 2, 3, 4],
[2, 3, 4, 5],
[3, 4, 5, 6]])
```

It is possible to perform usual operation on matrices, such as the matrix multiplication:

```
>>> A(i0, i1).get_matrix()*ones(4, 1)
Matrix([
[ 6],
[10],
[14],
[18]])
```

class `sympy.tensor.tensor.TensAdd`

Sum of tensors

**Parameters** `free_args` : list of the free indices

#### Notes

Sum of more than one tensor are put automatically in canonical form.

#### Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensorhead, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(a) + q(a); t
p(a) + q(a)
>>> t(b)
p(b) + q(b)
```

Examples with components data added to the tensor expression:

```
>>> from sympy import eye
>>> Lorentz.data = [1, -1, -1, -1]
>>> a, b = tensor_indices('a, b', Lorentz)
>>> p.data = [2, 3, -2, 7]
>>> q.data = [2, 3, -2, 7]
>>> t = p(a) + q(a); t
p(a) + q(a)
>>> t(b)
p(b) + q(b)
```

The following are:  $2^{**2} - 3^{**2} - 2^{**2} - 7^{**2} ==> -58$

```
>>> (p(a)*p(-a)).data
-58
```

```
>>> p(a)**2
-58
```

### Attributes

args	(tuple of addends)
rank	(rank of the tensor)
free_args	(list of the free indices in sorted order)

### canon\_bp()

canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

### contract\_metric(*g*)

Raise or lower indices with the metric *g*

**Parameters** *g* : metric

**contract\_all** : if True, eliminate all *g* which are contracted

### Notes

see the `TensorIndexType` docstring for the contraction conventions

### static from\_TIDS\_list(*coeff*, *tids\_list*)

Given a list of coefficients and a list of TIDS objects, construct a `TensAdd` instance, equivalent to the one that would result from creating single instances of `TensMul` and then adding them.

### Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead, TensAdd
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j = tensor_indices('i,j', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]**2, [[1]**2])
>>> eA = 3*A(i, j)
>>> eB = 2*B(j, i)
>>> t1 = eA._tids
>>> t2 = eB._tids
>>> c1 = eA.coeff
>>> c2 = eB.coeff
>>> TensAdd.from_TIDS_list([c1, c2], [t1, t2])
2*B(i, j) + 3*A(i, j)
```

If the coefficient parameter is a scalar, then it will be applied as a coefficient on all TIDS objects.

```
>>> TensAdd.from_TIDS_list(4, [t1, t2])
4*A(i, j) + 4*B(i, j)
```

### fun\_eval(\**index\_tuples*)

Return a tensor with free indices substituted according to `index_tuples`

**Parameters** *index\_types* : list of tuples (`old_index`, `new_index`)

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j) + A(i, -j)
>>> t.fun_eval((i, k), (-j, 1))
A(k, L_0)*B(l, -L_0) + A(k, 1)

substitute_indices(*index_tuples)
Return a tensor with free indices substituted according to index_tuples

Parameters index_types : list of tuples (old_index, new_index)
```

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.substitute_indices((i,j), (j, k))
A(j, L_0)*B(-L_0, -k)

class sympy.tensor.tensor.TensMul
Product of tensors

Parameters coeff : SymPy coefficient of the tensor
args
```

**Notes**

args[0] list of TensorHead of the component tensors.

args[1] list of (ind, ipos, icomp) where ind is a free index, ipos is the slot position of ind in the icomp-th component tensor.

args[2] list of tuples representing dummy indices. (ipos1, ipos2, icomp1, icomp2) indicates that the contravariant dummy index is the ipos1-th slot position in the icomp1-th component tensor; the corresponding covariant index is in the ipos2 slot position in the icomp2-th component tensor.

**Attributes**

components	(list of TensorHead of the component tensors)
types	(list of nonrepeated TensorIndexType)
free	(list of (ind, ipos, icomp), see Notes)
dum	(list of (ipos1, ipos2, icomp1, icomp2), see Notes)
ext_rank	(rank of the tensor counting the dummy indices)
rank	(rank of the tensor)
coeff	(SymPy coefficient of the tensor)
free_args	(list of the free indices in sorted order)
is_canon_bp	(True if the tensor is in canonical form)

**canon\_bp()**

Canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = tensorhead('A', [Lorentz]*2, [[2]])
>>> t = A(m0,-m1)*A(m1,-m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
>>> t = A(m0,-m1)*A(m1,-m2)*A(m2,-m0)
>>> t.canon_bp()
0
```

**contract\_metric(*g*)**

Raise or lower indices with the metric *g*

**Parameters** *g* : metric

**Notes**

see the `TensorIndexType` docstring for the contraction conventions

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
>>> t.contract_metric(g).canon_bp()
p(L_0)*q(-L_0)
```

**fun\_eval(\**index\_tuples*)**

Return a tensor with free indices substituted according to `index_tuples`  
`index_types` list of tuples (`old_index`, `new_index`)

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.fun_eval((i, k), (-j, l))
A(k, L_0)*B(-L_0, l)
```

**get\_indices()**

Returns the list of indices of the tensor

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
```

**perm2tensor(*g*, *canon\_bp=False*)**

Returns the tensor corresponding to the permutation *g*

For further details, see the method in TIDS with the same name.

**sorted\_components()**

Returns a tensor with sorted components calling the corresponding method in a TIDS object.

**split()**

Returns a list of tensors, whose product is **self**

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

**Examples**

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

**sympy.tensor.tensor.canon\_bp(*p*)**

Butler-Portugal canonicalization

**sympy.tensor.tensor.tensor\_mul(\**a*)**

product of tensors

**sympy.tensor.tensor.riemann\_cyclic\_replace(*t\_r*)**

replace Riemann tensor with an equivalent expression

$R(m,n,p,q) \rightarrow 2/3*R(m,n,p,q) - 1/3*R(m,q,n,p) + 1/3*R(m,p,n,q)$

**sympy.tensor.tensor.riemann\_cyclic(*t2*)**

replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.

This trick is discussed in the reference guide to Cadabra.

## Examples

```
>>> from sympy.tensor.tensor import TensorIndexType, tensor_indices, tensorhead, riemann_cyclic
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = tensorhead('R', [Lorentz]*4, [[2, 2]])
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

## 3.31 Utilities

This module contains some general purpose utilities that are used across SymPy.

Contents:

### 3.31.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the [Tensor Module](#) (page 1086). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], x[j]*A[i, j])
```

Because the code printers treat Indexed objects with repeated indices as a summation, the above equality instance will be translated to low-level code for a matrix vector product. This is how you tell SymPy to generate the code, compile it and wrap it as a python function:

```
>>> matvec = autowrap(instruction)
```

That's it. Now let's test it with some numpy arrays. The default wrapper backend is f2py. The wrapper function it provides is set up to accept python lists, which it will silently convert to numpy arrays. So we can test the matrix vector product like this:

```
>>> M = [[0, 1],
...        [1, 0]]
>>> matvec(M, [2, 3])
[ 3.  2.]
```

### Implementation details

The autowrap module is implemented with a backend consisting of CodeWrapper objects. The base class `CodeWrapper` takes care of details about module name, filenames and options. It also contains the driver routine, which runs through all steps in the correct order, and also takes care of setting up and removing the temporary working directory.

The actual compilation and wrapping is done by external resources, such as the system installed f2py command. The Cython backend runs a distutils setup script in a subprocess. Subclasses of CodeWrapper takes care of these backend-dependent details.

## API Reference

Module for compiling codegen output, and wrap the binary for use in python.

---

**Note:** To use the autowrap module it must first be imported

```
>>> from sympy.utilities.autowrap import autowrap
```

---

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, i.e.

```
>>> from sympy.abc import x,y
>>> expr = ((x - y)**(25)).expand()
>>> binary_callable = autowrap(expr)
>>> binary_callable(1, 2)
-1.0
```

The callable returned from autowrap() is a binary python function, not a SymPy object. If it is desired to use the compiled function in symbolic expressions, it is better to use binary\_function() which returns a SymPy Function object. The binary callable is attached as the `_imp_` attribute and invoked when a numerical evaluation is requested with evalf(), or with lambdify().

```
>>> from sympy.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y
y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y:2})
0.e-110
```

The idea is that a SymPy user will primarily be interested in working with mathematical expressions, and should not have to learn details about wrapping tools in order to evaluate expressions numerically, even if they are computationally expensive.

When is this useful?

1. For computations on large arrays, Python iterations may be too slow, and depending on the mathematical expression, it may be difficult to exploit the advanced index operations provided by NumPy.
2. For *really* long expressions that will be called repeatedly, the compiled binary should be significantly faster than SymPy's .evalf()
3. If you are generating code with the codegen utility in order to use it in another project, the automatic python wrappers let you test the binaries immediately from within SymPy.
4. To create customized ufuncs for use with numpy arrays. See *ufuncify*.

When is this module NOT the best approach?

1. If you are really concerned about speed or memory optimizations, you will probably get better results by working directly with the wrapper tools and the low level code. However, the files generated by this utility may provide a useful starting point and reference code. Temporary files will be left intact if you supply the keyword tempdir="path/to/files/".
2. If the array computation can be handled easily by numpy, and you don't need the binaries for another project.

```
class sympy.utilities.autowrap.CodeWrapper(generator, filepath=None, flags=[], verbose=False)
    Base Class for code wrappers

class sympy.utilities.autowrap.CythonCodeWrapper(*args, **kwargs)
    Wrapper that uses Cython

    dump_pyx(routines, f, prefix)
        Write a Cython file with python wrappers

        This file contains all the definitions of the routines in c code and refers to the header file.

class sympy.utilities.autowrap.DummyWrapper(generator,      filepath=None,      flags=[],      verbose=False)
    Class used for testing independent of backends

class sympy.utilities.autowrap.F2PyCodeWrapper(generator,   filepath=None,   flags=[],   verbose=False)
    Wrapper that uses f2py

class sympy.utilities.autowrap.UfuncifyCodeWrapper(generator, filepath=None, flags=[], verbose=False)
    Wrapper for Ufuncify

    dump_c(routines, f, prefix)
        Write a C file with python wrappers

        This file contains all the definitions of the routines in c code.

sympy.utilities.autowrap.autowrap(*args, **kwargs)
    Generates python callable binaries based on the math expression.
```

### Parameters `expr`

The SymPy expression that should be wrapped as a binary routine.

### `language` : string, optional

If supplied, (options: ‘C’ or ‘F95’), specifies the language of the generated code.  
If `None` [default], the language is inferred based upon the specified backend.

### `backend` : string, optional

Backend used to wrap the generated code. Either ‘f2py’ [default], or ‘cython’.

### `tempdir` : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

### `args` : iterable, optional

An iterable of symbols. Specifies the argument sequence for the function.

### `flags` : iterable, optional

Additional option flags that will be passed to the backend.

### `verbose` : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

### `helpers` : iterable, optional

Used to define auxillary expressions needed for the main `expr`. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. Autowrap will then make sure that the compiled main expression

can link to the helper routine. Items should be tuples with (<function\_name>, <sympy\_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

`sympy.utilities.autowrap.binary_function(symfunc, expr, **kwargs)`

Returns a sympy function with expr as binary implementation

This is a convenience function that automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with implemented\_function().

```
>>> from sympy.abc import x, y
>>> from sympy.utilities.autowrap import binary_function
>>> expr = ((x - y)**(25)).expand()
>>> f = binary_function('f', expr)
>>> type(f)
<class 'sympy.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0
```

`sympy.utilities.autowrap.ufuncify(*args, **kwargs)`

Generates a binary function that supports broadcasting on numpy arrays.

**Parameters** `args` : iterable

Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.

**expr**

A SymPy expression that defines the element wise operation.

**language** : string, optional

If supplied, (options: ‘C’ or ‘F95’), specifies the language of the generated code.  
If `None` [default], the language is inferred based upon the specified backend.

**backend** : string, optional

Backend used to wrap the generated code. Either ‘numpy’ [default], ‘cython’, or ‘f2py’.

**tempdir** : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

**flags** : iterable, optional

Additional option flags that will be passed to the backend

**verbose** : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

#### helpers : iterable, optional

Used to define auxillary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in the `helpers` iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<function\_name>, <sympy\_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

#### References

[1] <http://docs.scipy.org/doc/numpy/reference/ufuncs.html>

#### Examples

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f)
numpy.ufunc
>>> f([1, 2, 3], 2)
array([ 3.,  6., 11.])
>>> f(np.arange(5), 3)
array([ 3.,  4.,  7., 12., 19.])
```

For the F2Py and Cython backends, inputs are required to be equal length 1-dimensional arrays. The F2Py backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='F2Py')
>>> f_fortran(1, 2)
3
>>> f_fortran(numpy.array([1, 2, 3]), numpy.array([1.0, 2.0, 3.0]))
array([2.,  6., 12.])
>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got int)
>>> f_cython(numpy.array([1.0]), numpy.array([2.0]))
array([ 3.])
```

### 3.31.2 Codegen

This module provides functionality to generate directly compilable code from SymPy expressions. The `codegen` function is the user interface to the code generation functionality in SymPy. Some details of the implementation is given below for advanced users that may want to use the framework directly.

---

**Note:** The `codegen` callable is not in the `sympy` namespace automatically, to use it you must first execute

```
>>> from sympy.utilities.codegen import codegen
```

---

## Implementation Details

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the `codegen()` function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

## Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

## API Reference

module for generating C, C++, Fortran77, Fortran90 and Octave/Matlab routines that evaluate sympy expressions. This module is work in progress. Only the milestones with a '+' character in the list below have been completed.

— How is `sympy.utilities.codegen` different from `sympy.printing.ccode`? —

We considered the idea to extend the printing routines for sympy functions in such a way that it prints complete compilable code, but this leads to a few unsurmountable issues that can only be tackled with dedicated code generator:

- For C, one needs both a code and a header file, while the printing routines generate just one string. This code generator can be extended to support .pyf files for f2py.
- SymPy functions are not concerned with programming-technical issues, such as input, output and input-output arguments. Other examples are contiguous or non-contiguous arrays, including headers of other libraries such as gsl or others.
- It is highly interesting to evaluate several sympy functions in one C routine, eventually sharing common intermediate results with the help of the cse routine. This is more than just printing.
- From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.

- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

- First working version with scalar input arguments, generating C code, tests
- Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
- Integer and Real numbers as input and output
- Output arguments
- InputOutput arguments
- Sort input/output arguments properly
- Contiguous array arguments (numpy matrices)
- Also generate .pyf code for f2py (in autowrap module)
- Isolate constants and evaluate them beforehand in double precision
- Fortran 90
- Octave/Matlab
- Common Subexpression Elimination
- User defined comments in the generated code
- Optional extra include lines for libraries/objects that can eval special functions
- Test other C compilers and libraries: gcc, tcc, libtcc, gcc+gsl, ...
- Contiguous array arguments (sympy matrices)
- Non-contiguous array arguments (sympy matrices)
- ccode must raise an error when it encounters something that can not be translated into c. ccode(integrate(sin(x)/x, x)) does not make sense.
- Complex numbers as input and output
- A default complex datatype
- Include extra information in the header: date, user, hostname, sha1 hash, ...
- Fortran 77
- C++
- Python
- ...

```
class sympy.utilities.codegen.Routine(name, arguments, results, local_vars)
```

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

```
result_variables
    Returns a list of OutputArgument, InOutArgument and Result.

    If return values are present, they are at the end of the list.

variables
    Returns a set of all variables possibly used in the routine.

    For routines with unnamed return values, the dummies that may or may not be used will be
    included in the set.

class sympy.utilities.codegen.DataType(cname, fname, pyname, octname)
    Holds strings for a certain datatype in different languages.

sympy.utilities.codegen.get_default_datatype(expr)
    Derives an appropriate datatype based on the expression.

class sympy.utilities.codegen.Argument(name,      datatype=None,      dimensions=None,      precision=None)
    An abstract Argument data structure: a name and a data type.

    This structure is refined in the descendants below.

class sympy.utilities.codegen.Result(expr, name=None, result_var=None, datatype=None, dimensions=None, precision=None)
    An expression for a return value.

    The name result is used to avoid conflicts with the reserved word "return" in the python language. It
    is also shorter than ReturnValue.

    These may or may not need a name in the destination (e.g., "return(x*y)" might return a value without
    ever naming it).

class sympy.utilities.codegen.CodeGen(project='project')
    Abstract class for the code generators.

    dump_code(routines, f, prefix, header=True, empty=True)
        Write the code by calling language specific methods.

        The generated file contains all the definitions of the routines in low-level code and refers to the
        header file if appropriate.

        Parameters routines : list
            A list of Routine instances.

        f : file-like
            Where to write the file.

        prefix : string
            The filename prefix, used to refer to the proper header file. Only the basename
            of the prefix is used.

        header : bool, optional
            When True, a header comment is included on top of each source file. [default :
            True]

        empty : bool, optional
            When True, empty lines are included to structure the source files. [default :
            True]
```

```
routine(name, expr, argument_sequence)
```

Creates an Routine object that is appropriate for this language.

This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If `argument_sequence` is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

```
write(routines, prefix, to_files=False, header=True, empty=True)
```

Writes all the source code files for the given routines.

The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

**Parameters** `routines` : list

A list of Routine instances to be written

`prefix` : string

The prefix for the output files

`to_files` : bool, optional

When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]

`header` : bool, optional

When True, a header comment is included on top of each source file. [default: True]

`empty` : bool, optional

When True, empty lines are included to structure the source files. [default: True]

```
class sympy.utilities.codegen.CCodeGen(project='project')
```

Generator for C code.

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.c and <prefix>.h respectively.

```
dump_c(routines, f, prefix, header=True, empty=True)
```

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters** `routines` : list

A list of Routine instances.

`f` : file-like

Where to write the file.

`prefix` : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

`header` : bool, optional

When True, a header comment is included on top of each source file. [default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files. [default : True]

`dump_h(routines, f, prefix, header=True, empty=True)`

Writes the C header file.

This file contains all the function declarations.

**Parameters** `routines` : list

A list of Routine instances.

`f` : file-like

Where to write the file.

`prefix` : string

The filename prefix, used to construct the include guards. Only the basename of the prefix is used.

`header` : bool, optional

When True, a header comment is included on top of each source file. [default : True]

`empty` : bool, optional

When True, empty lines are included to structure the source files. [default : True]

`get_prototype(routine)`

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: [http://en.wikipedia.org/wiki/Function\\_prototype](http://en.wikipedia.org/wiki/Function_prototype)

`class sympy.utilities.codegen.FCodeGen(project='project')`

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

`dump_f95(routines, f, prefix, header=True, empty=True)`

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters** `routines` : list

A list of Routine instances.

`f` : file-like

Where to write the file.

`prefix` : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file. [default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files. [default : True]

`dump_h(routines, f, prefix, header=True, empty=True)`

Writes the interface to a header file.

This file contains all the function declarations.

**Parameters** `routines` : list

A list of Routine instances.

`f` : file-like

Where to write the file.

`prefix` : string

The filename prefix.

**header** : bool, optional

When True, a header comment is included on top of each source file. [default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files. [default : True]

`get_interface(routine)`

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, a CodeGenError is raised.

See: [http://en.wikipedia.org/wiki/Function\\_prototype](http://en.wikipedia.org/wiki/Function_prototype)

`class sympy.utilitiescodegen.OctaveCodeGen(project='project')`

Generator for Octave code.

The `.write()` method inherited from CodeGen will output a code file <prefix>.m.

Octave .m files usually contain one function. That function name should match the filename (`prefix`). If you pass multiple `name_expr` pairs, the latter ones are presumed to be private functions accessed by the primary function.

You should only pass inputs to `argument_sequence`: outputs are ordered according to their order in `name_expr`.

`dump_m(routines, f, prefix, header=True, empty=True, inline=True)`

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

**Parameters** `routines` : list

A list of Routine instances.

**f** : file-like

Where to write the file.

**prefix** : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

**header** : bool, optional

When True, a header comment is included on top of each source file. [default : True]

**empty** : bool, optional

When True, empty lines are included to structure the source files. [default : True]

**routine**(*name, expr, argument\_sequence*)

Specialized Routine creation for Octave.

```
sympy.utilities.codegen_codegen(name_expr,      language,      prefix=None,      project='project',
                                 to_files=False,     header=True,      empty=True,      argu-
                                 ment_sequence=None)
```

Generate source code for expressions in a given language.

**Parameters** **name\_expr** : tuple, or list of tuples

A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class Equality) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.

**language** : string

A string that indicates the source code language. This is case insensitive. Currently, ‘C’, ‘F95’ and ‘Octave’ are supported. ‘Octave’ generates code compatible with both Octave and Matlab.

**prefix** : string, optional

A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first name\_expr tuple is used.

**project** : string, optional

A project name, used for making unique preprocessor instructions. [default: “project”]

**to\_files** : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

**header** : bool, optional

When True, a header is written on top of each source file. [default: True]

**empty** : bool, optional

When True, empty lines are used to structure the code. [default: True]

**argument\_sequence** : iterable, optional

Sequence of arguments for the routine in a preferred order. A CodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

## Examples

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     ("f", x+y*z), "C", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
    double f_result;
    f_result = x + y*z;
    return f_result;
}
>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double f(double x, double y, double z);
#endif
```

Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```
>>> from sympy.abc import f, g
>>> from sympy import Eq
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     [("myfcn", x + y), ("fcn2", [Eq(f, 2*x), Eq(g, y)])],
...     "C", header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
    double myfcn_result;
    myfcn_result = x + y;
    return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
    (*f) = 2*x;
    (*g) = y;
}
```

`sympy.utilities.codegen.make_routine(name, expr, argument_sequence=None, language='F95')`

A factory that makes an appropriate Routine from an expression.

**Parameters** `name` : string

The name of this routine in the generated code.

**expr** : expression or list/tuple of expressions

A SymPy expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.

**argument\_sequence** : list or tuple, optional

List arguments for the routine in a preferred order. If omitted, the results are language dependent, for example, alphabetical order or in the same order as the given expressions.

**language** : string, optional

Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: “F95”].

**A decision about whether to use output arguments or return values is made depending on both the language and the particular mathematical expressions.**

For an expression of type Equality, the left hand side is typically made into an OutputArgument (or perhaps an InOutArgument if appropriate). Otherwise, typically, the calculated expression is made a return values of the routine.

## Examples

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.abc import x, y, f, g
>>> from sympy import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> from sympy import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x, 2]])])
>>> [arg.result_var for arg in r.results]
[result_5397460570204848505]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                         InOutArgument)
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]

>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_8598435338387848786]
>>> [a.expr for a in r.arguments if isinstance(a, OutputArgument)]
[1, Matrix([[x, 2]])]

>>> [a.name for a in r.arguments if isinstance(a, InOutArgument)]
[g]
>>> [a.expr for a in r.arguments if isinstance(a, InOutArgument)]
[g + x]
```

### 3.31.3 Decorator

Useful utility decorators.

```
sympy.utilities.decorator.conserve_mpmath_dps(func)
```

After the function finishes, resets the value of mpmath.mp.dps to the value it had before the function was run.

```
sympy.utilities.decorator.doctest_depends_on(exe=None,           modules=None,           dis-
                                              able_viewers=None)
```

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

```
class sympy.utilities.decorator.no_attrs_in_subclass(cls, f)
```

Don't 'inherit' certain attributes from a base class

```
>>> from sympy.utilities.decorator import no_attrs_in_subclass
```

```
>>> class A(object):
...     x = 'test'
```

```
>>> A.x = no_attrs_in_subclass(A, A.x)
```

```
>>> class B(A):
...     pass
```

```
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False
```

```
sympy.utilities.decorator.public(obj)
```

Append obj's name to global \_\_all\_\_ variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling \_\_all\_\_ variables manually, you just don't have to repeat yourself (object's name). You also know if object is public at definition site, not at some random location (where \_\_all\_\_ was set).

Note that in multiple decorator setup (in almost all cases) @public decorator must be applied before any other decorators, because it relies on the pointer to object's global namespace. If you apply other decorators first, @public may end up modifying the wrong namespace.

## Examples

```
>>> from sympy.utilities.decorator import public

>>> __all__
Traceback (most recent call last):
...
NameError: name '__all__' is not defined

>>> @public
... def some_function():
...     pass

>>> __all__
['some_function']
```

`sympy.utilities.decorator.threaded(func)`

Apply `func` to sub-elements of an object, including [Add](#) (page 115).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 1125) decorator allows threading over elements of [Add](#) (page 115) class. If this behavior is not desirable use `xthreaded()` (page 1125) decorator.

Functions using this decorator must have the following signature:

```
@threaded
def function(expr, *args, **kwargs):
```

`sympy.utilities.decorator.threaded_factory(func, use_add)`

A factory for `threaded` decorators.

`sympy.utilities.decorator.xthreaded(func)`

Apply `func` to sub-elements of an object, excluding [Add](#) (page 115).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of `threaded()` (page 1125) decorator disallows threading over elements of [Add](#) (page 115) class. If this behavior is not desirable use `threaded()` (page 1125) decorator.

Functions using this decorator must have the following signature:

```
@xthreaded
def function(expr, *args, **kwargs):
```

### 3.31.4 Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

`sympy.utilities.enumerative.multiset_partitions_taocp(multiplicities)`

Enumerates partitions of a multiset.

#### Parameters multiplicities

list of integer multiplicities of the components of the multiset.

## Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[[['a', 'b', 'b']],
[['a', 'b'], ['b']],
[['a'], ['b', 'b']],
[['a'], ['b'], ['b']]]

sympy.utilities.enumerative.factoring_visitor(state, primes)
```

Use with `multiset_partitions_taocp` to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

## Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```
>>> from sympy.utilities.enumerative import factoring_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> from sympy import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]

sympy.utilities.enumerative.list_visitor(state, components)
```

Return a list of lists to represent the partition.

## Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc') # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3]) # for multiset '1 2 2 3
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

```
class sympy.utilities.enumerative.MultisetPartitionTraverser
    Has methods to enumerate and count the partitions of a multiset.
```

This implements a refactored and extended version of Knuth's algorithm 7.1.2.5M [AOCP] (page 1249)."

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of `multiset_partitions_taocp`.

**See also:**

`multiset_partitions_taocp` (page 1125)

## References

[AOCP] (page 1249), [Factorisatio] (page 1249), [Yorgey] (page 1249)

## Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

`count_partitions(multiplicities)`  
Returns the number of partitions of a multiset whose components have the multiplicities given in `multiplicities`.  
For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.

## Notes

If one looks at the workings of Knuth's algorithm M [AOCP] (page 1249), it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

1. This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)

2.Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no \_range version of count\_partitions. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

### Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9,8,2])
288716
>>> m.count_partitions([2,2])
9
>>> del m

enum_all(multiplicities)
Enumerate the partitions of a multiset.
```

### See also:

[multiset\\_partitions\\_taocp](#) ([page 1125](#)) which provides the same result as this method, but is about twice as fast. Hence, enum\_all is primarily useful for testing. Also see the function for a discussion of states and visitors.

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2,2])
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'a'], ['b'], ['b']],
[['a', 'a'], ['b'], ['b'], ['b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']],
[['a', 'b'], ['a'], ['b']],
[['a'], ['a'], ['b', 'b']],
[['a'], ['a'], ['b'], ['b']]]
```

```
enum_large(multiplicities, lb)
Enumerate the partitions of a multiset with lb < num(parts)

Equivalent to enum_range(multiplicities, lb, sum(multiplicities))
```

#### Parameters multiplicities

list of multiplicities of the components of the multiset.

#### lb

Number of parts in the partition must be greater than this lower bound.

**See also:**

[enum\\_all](#) (page 1128), [enum\\_small](#) (page 1129), [enum\\_range](#) (page 1129)

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a'], ['b'], ['b']],
[['a', 'b'], ['a'], ['b']],
[['a'], ['a'], ['b', 'b']],
[['a'], ['a'], ['b'], ['b']]]
```

**enum\_range(*multiplicities*, *lb*, *ub*)**

Enumerate the partitions of a multiset with  $lb < \text{num}(\text{parts}) \leq ub$ .

In particular, if partitions with exactly  $k$  parts are desired, call with  $(\text{multiplicities}, k - 1, k)$ . This method generalizes `enum_all`, `enum_small`, and `enum_large`.

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

**enum\_small(*multiplicities*, *ub*)**

Enumerate multiset partitions with no more than  $ub$  parts.

Equivalent to `enum_range(multiplicities, 0, ub)`

#### Parameters **multiplicities**

list of multiplicities of the components of the multiset.

#### ub

Maximum number of parts

**See also:**

[enum\\_all](#) (page 1128), [enum\\_large](#) (page 1128), [enum\\_range](#) (page 1129)

### Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
```

```
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth [AOCP] (page 1249).

### 3.31.5 Iterables

#### cartes

Returns the cartesian product of sequences as a generator.

Examples::

```
>>> from sympy.utilities.iterables import cartes
>>> list(cartes([1,2,3], 'ab'))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')]
```

#### variations

variations(seq, n) Returns all the variations of the list of size n.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to True, or the variations without repetition if set to False.

Examples::

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1,2,3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1,2,3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

#### partitions

Although the combinatorics module contains Partition and IntegerPartition classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: `partitions` and `multiset_partitions`. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine `kbins` that will give a variety of permutations of partitions.

partitions:

```
>>> from sympy.utilities.iterables import partitions
>>> [p.copy() for s, p in partitions(7, m=2, size=True) if s == 2]
[{:1: 1, :6: 1}, {:2: 1, :5: 1}, {:3: 1, :4: 1}]
```

multiset\_partitions:

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> [p for p in multiset_partitions([3, 2])]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> [p for p in multiset_partitions([1, 1, 1, 2], 2)]
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

kbins:

```
>>> from sympy.utilities.iterables import kbins
>>> def show(k):
...     rv = []
...     for p in k:
...         rv.append(''.join([''.join(j) for j in p]))
...     return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0)) # same as multiset_partitions
['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
 'B,AC', 'B,CA',
 'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
 'B,AC', 'BC,A',
 'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
 'B,AC', 'B,CA', 'BA,C', 'BC,A',
 'C,AB', 'C,BA', 'CA,B', 'CB,A']
```

## Docstring

`sympy.utilities.iterables.binary_partitions(n)`

Generates the binary partition of n.

A binary partition consists only of numbers that are powers of two. Each step reduces a  $2^{**k+1}$  to  $2^{**k}$  and  $2^{**k}$ . Thus 16 is converted to 8 and 8.

Reference: TAOCP 4, section 7.2.1.5, problem 64

## Examples

```
>>> from sympy.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
...     print(i)
...
[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]
```

```
sympy.utilities.iterables.bracelets(n, k)
```

Wrapper to necklaces to return a free (unrestricted) necklace.

```
sympy.utilities.iterables.capture(func)
```

Return the printed output of func().

*func* should be a function without arguments that produces output with print statements.

```
>>> from sympy.utilities.iterables import capture
>>> from sympy import pprint
>>> from sympy.abc import x
>>> def foo():
...     print('hello world!')
...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x))
'2\n-\n x\n'
```

```
sympy.utilities.iterables.common_prefix(*seqs)
```

Return the subsequence that is a common start of sequences in *seqs*.

```
>>> from sympy.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

```
sympy.utilities.iterables.common_suffix(*seqs)
```

Return the subsequence that is a common ending of sequences in *seqs*.

```
>>> from sympy.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

```
sympy.utilities.iterables.dict_merge(*dicts)
```

Merge dictionaries into a single dictionary.

```
sympy.utilities.iterables.filter_symbols(iterator, exclude)
```

Only yield elements from *iterator* that do not occur in *exclude*.

**Parameters** **iterator** : iterable

iterator to take elements from

**exclude** : iterable

elements to exclude

**Returns** **iterator** : iterator

filtered iterator

```
sympy.utilities.iterables.flatten(iterable, levels=None, cls=None)
```

Recursively denest iterable containers.

```
>>> from sympy.utilities.iterables import flatten

>>> flatten([1, 2, 3])
[1, 2, 3]
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set `levels` flag to the desired number of levels:

```
>>> ls = [[(-2, -1), (1, 2)], [(0, 0)]]

>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If `cls` argument is specified, it will only flatten instances of that class, for example:

```
>>> from sympy.core import Basic
>>> class MyOp(Basic):
...     pass
...
>>> flatten([MyOp(1, MyOp(2, 3))], cls=MyOp)
[1, 2, 3]
```

adapted from [http://kogs-www.informatik.uni-hamburg.de/~meine/python\\_tricks](http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks)

```
sympy.utilities.iterables.generate_bell(n)
```

Return permutations of  $[0, 1, \dots, n - 1]$  such that each permutation differs from the last by the exchange of a single pair of neighbors. The  $n!$  permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the `next_trotterjohnson` method of the Permutation class (which generates the same sequence in a different manner).

**See also:**

`sympy.combinatorics.permutations.Permutation.next_trotterjohnson` (page 176)

## References

- [http://en.wikipedia.org/wiki/Method\\_ringing](http://en.wikipedia.org/wiki/Method_ringing)
- <http://stackoverflow.com/questions/4856615/recursive-permutation/4857018>
- <http://programminggeeks.com/bell-algorithm-for-permutation/>
- [http://en.wikipedia.org/wiki/Steinhaus%20%93Johnson%20%93Trotter\\_algorithm](http://en.wikipedia.org/wiki/Steinhaus%20%93Johnson%20%93Trotter_algorithm)
- Generating involutions, derangements, and relatives by ECO Vincent Vajnovszki, DMTCS vol 1 issue 12, 2010

## Examples

```
>>> from itertools import permutations
>>> from sympy.utilities.iterables import generate_bell
>>> from sympy import zeros, Matrix
```

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by `permutations` and `generate_bell`:

```
>>> list(permutations(range(4)))[::5]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4))[::5]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each “bell” permutation always has only two elements out of place relative to the previous permutation (and so the signature (+/-1) of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix([j - 3 for j in list(p)]) # make largest zero
>>> m.print_nonzero('X')
[XXX XXXXXX XXXXXX XXX]
[XX XX XXXX XX XXXX XX XX]
[X XXXX XX XXXX XX XXXX X]
[ XXXXXX XXXXXX XXXXXX ]
```

`sympy.utilities.iterables.generate_derangements(perm)`

Routine to generate unique derangements.

TODO: This will be rewritten to use the ECO operator approach once the permutations branch is in master.

**See also:**

`sympy.functions.combinatorial.factorials.subfactorial` (page 348)

## Examples

```
>>> from sympy.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 3]))
[[1, 0, 3, 2], [1, 2, 3, 0], [1, 3, 0, 2], [2, 0, 3, 1], [2, 3, 0, 1], [2, 3, 1, 0], [3, 0, 1, 2], [3, 2,
```

```
>>> list(generate_derangements([0, 1, 1]))
[]
```

`sympy.utilities.iterables.generate_involutions(n)`

Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.

Reference: <http://mathworld.wolfram.com/PermutationInvolution.html>

### Examples

```
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10

sympy.utilities.iterables.generate_oriented_forest(n)
```

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.

Reference: [1] T. Beyer and S.M. Hedetniemi: constant time generation of rooted trees, SIAM J. Computing Vol. 9, No. 4, November 1980 [2] <http://stackoverflow.com/questions/1633833/oriented-forest-taocp-algorithm-in-python>

### Examples

```
>>> from sympy.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0], [0, 1, 1, 1], [0, 1, 1, 0], [0, 1, 0, 1], [0, 1,
```

```
sympy.utilities.iterables.group(seq, multiple=True)
Splits a sequence into a list of lists of equal, adjacent elements.
```

See also:

[multiset](#) (page 1138)

### Examples

```
>>> from sympy.utilities.iterables import group
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 3), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 2), (3, 1), (2, 2), (1, 1)]
```

```
sympy.utilities.iterables.has_dups(seq)
Return True if there are any duplicate elements in seq.
```

## Examples

```
>>> from sympy.utilities.iterables import has_dups
>>> from sympy import Dict, Set

>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

`sympy.utilities.iterables.has_variety(seq)`  
Return True if there are any different elements in `seq`.

## Examples

```
>>> from sympy.utilities.iterables import has_variety

>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

`sympy.utilities.iterables.ibin(n, bits=0, str=False)`  
Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If `bits` is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0, ..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for `bits`, e.g. ‘all’.

If the bit *string* is desired pass `str=True`.

## Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
>>> ibin(2, 4)[::-1]
[0, 1, 0, 0]
```

If all lists corresponding to 0 to  $2^{**n} - 1$ , pass a non-integer for `bits`:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use `str=True`:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1] # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

`sympy.utilities.iterables.interactive_traversal(expr)`

Traverse a tree asking a user which branch to choose.

`sympy.utilities.iterables.kbins(l, k, ordered=None)`

Return sequence l partitioned into k bins.

See also:

`partitions` (page 1141), `multiset_partitions` (page 1139)

## Examples

```
>>> from sympy.utilities.iterables import kbins
```

The default is to give the items in the same order, but grouped into k partitions without any reordering:

```
>>> from __future__ import print_function
>>> for p in kbins(list(range(5)), 2):
...     print(p)
...
[[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]
```

The `ordered` flag which is either `None` (to give the simple partition of the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

```
A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]
```

the following values for `ordered` have the shown meanings:

```
00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A
```

```
>>> for ordered in [None, 0, 1, 10, 11]:
...     print('ordered = %s' % ordered)
...     for p in kbins(list(range(3)), 2, ordered=ordered):
...         print('    %s' % p)
...
ordered = None
    [[0], [1, 2]]
    [[0, 1], [2]]
ordered = 0
    [[0, 1], [2]]
```

```
[[[0, 2], [1]],  
 [[0], [1, 2]]]  
ordered = 1  
[[[0], [1, 2]],  
 [[0], [2, 1]],  
 [[1], [0, 2]],  
 [[1], [2, 0]],  
 [[2], [0, 1]],  
 [[2], [1, 0]]]  
ordered = 10  
[[[0, 1], [2]],  
 [[2], [0, 1]],  
 [[0, 2], [1]],  
 [[1], [0, 2]],  
 [[0], [1, 2]],  
 [[1, 2], [0]]]  
ordered = 11  
[[[0], [1, 2]],  
 [[0, 1], [2]],  
 [[0], [2, 1]],  
 [[0, 2], [1]],  
 [[1], [0, 2]],  
 [[1, 0], [2]],  
 [[1], [2, 0]],  
 [[1, 2], [0]],  
 [[2], [0, 1]],  
 [[2, 0], [1]],  
 [[2], [1, 0]],  
 [[2, 1], [0]]]  
  
sympy.utilities.iterables.minlex(seq, directed=True, is_set=False, small=None)  
Return a tuple where the smallest element appears first; if directed is True (default) then the order  
is preserved, otherwise the sequence will be reversed if that gives a smaller ordering.  
If every element appears only once then is_set can be set to True for more efficient processing.  
If the smallest element is known at the time of calling, it can be passed and the calculation of the  
smallest element will be omitted.
```

## Examples

```
>>> from sympy.combinatorics.polyhedron import minlex  
>>> minlex((1, 2, 0))  
(0, 1, 2)  
>>> minlex((1, 0, 2))  
(0, 2, 1)  
>>> minlex((1, 0, 2), directed=False)  
(0, 1, 2)  
  
>>> minlex('11010011000', directed=True)  
'000011010011'  
>>> minlex('11010011000', directed=False)  
'000011001011'
```

```
sympy.utilities.iterables.multiset(seq)
```

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

See also:

[group](#) (page 1135)

### Examples

```
>>> from sympy.utilities.iterables import multiset
>>> multiset('mississippi')
{'i': 4, 'm': 1, 'p': 2, 's': 4}

sympy.utilities.iterables.multiset_combinations(m, n, g=None)
Return the unique combinations of size n from multiset m.
```

### Examples

```
>>> from sympy.utilities.iterables import multiset_combinations
>>> from itertools import combinations
>>> [''.join(i) for i in multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']

>>> def count(f, s): return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```
>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)
```

[sympy.utilities.iterables.multiset\\_partitions](#)(*multiset*, *m*=None)

Return unique partitions of the given multiset (in list form). If *m* is None, all multisets will be returned, otherwise only partitions with *m* parts will be returned.

If *multiset* is an integer, a range [0, 1, ..., *multiset* - 1] will be supplied.

See also:

[partitions](#) (page 1141), [sympy.combinatorics.partitions.Partition](#) (page 160), [sympy.combinatorics.partitions.IntegerPartition](#) (page 161), [sympy.functions.combinatorial.numbers.nT](#) (page 357)

### Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the `partitions` routine. If one is counting partitions then it is better to use the `nT` function.

### Examples

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
 [[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
 [[1], [2, 3, 4]]]
>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]
```

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> (list(multiset_partitions(a)) ==
...     list(multiset_partitions(sorted(a))))
True
```

If m is omitted then all partitions will be returned:

```
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```

`sympy.utilities.iterables.multiset_permutations(m, size=None, g=None)`

Return the unique permutations of multiset `m`.

## Examples

```
>>> from sympy.utilities.iterables import multiset_permutations
>>> from sympy import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

`sympy.utilities.iterables.necklaces(n, k, free=False)`

A routine to generate necklaces that may (`free=True`) or may not (`free=False`) be turned over to be viewed. The “necklaces” returned are comprised of `n` integers (beads) with `k` different values (colors). Only unique necklaces are returned.

## References

<http://mathworld.wolfram.com/Necklace.html>

## Examples

```
>>> from sympy.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
...     return ''.join(s[j] for j in i)
```

The “unrestricted necklace” is sometimes also referred to as a “bracelet” (an object that can be turned over, a sequence that can be reversed) and the term “necklace” is used to imply a sequence that cannot be reversed. So ACB == ABC for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B)
set(['ACB'])

>>> list(necklaces(4, 2))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 1),
 (0, 1, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1)]

>>> [show('.o', i) for i in bracelets(4, 2)]
['....', '...o', '..oo', '.o.o', '.ooo', 'oooo']

sympy.utilities.iterables.numbered_symbols(prefix='x', cls=None, start=0, exclude=[], *args,
                                         **assumptions)
```

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in *exclude*.

**Parameters** `prefix` : str, optional

The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.

`cls` : class, optional

The class to use. By default, it uses Symbol, but you can also use Wild or Dummy.

`start` : int, optional

The start number. By default, it is 0.

**Returns** `sym` : Symbol

The subscripted symbols.

```
sympy.utilities.iterables.partitions(n, m=None, k=None, size=False)
```

Generate all partitions of integer n ( $\geq 0$ ).

**Parameters** “`m`“ : integer (default gives partitions of all sizes)

limits number of parts in partition (mnemonic: m, maximum parts)

“`k`“ : integer (default gives partitions number from 1 through n)

limits the numbers that are kept in the partition (mnemonic: k, keys)

“`size`“ : bool (default False, only partition is returned)

when `True` then (M, P) is returned where M is the sum of the multiplicities and P is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is {4: 1}, “4: one of them”.

See also:

`sympy.combinatorics.partitions.Partition` (page 160), `sympy.combinatorics.partitions.IntegerPartition` (page 161)

### Examples

```
>>> from sympy.utilities.iterables import partitions
```

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```
>>> for p in partitions(6, k=2):
...     print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m:

```
>>> for p in partitions(6, m=2):
...     print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

Note that the `_same_` dictionary object is returned each time. This is for speed: generating each partition goes quickly, taking constant time, independent of n.

```
>>> [p for p in partitions(6, k=2)]
[{1: 6}, {1: 6}, {1: 6}, {1: 6}]
```

If you want to build a list of the returned dictionaries then make a copy of them:

```
>>> [p.copy() for p in partitions(6, k=2)]
[{2: 3}, {1: 2, 2: 2}, {1: 4, 2: 1}, {1: 6}]
>>> [(M, p.copy()) for M, p in partitions(6, k=2, size=True)]
[(3, {2: 3}), (4, {1: 2, 2: 2}), (5, {1: 4, 2: 1}), (6, {1: 6})]
```

**Reference:** modified from Tim Peter’s version to allow for k and m values:  
[code.activestate.com/recipes/218332-generator-for-integer-partitions/](http://code.activestate.com/recipes/218332-generator-for-integer-partitions/)

`sympy.utilities.iterables.postfixes(seq)`  
Generate all postfixes of a sequence.

## Examples

```
>>> from sympy.utilities.iterables import postfixes

>>> list(postfixes([1,2,3,4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

sympy.utilities.iterables.postorder\_traversal(*node*, *keys=None*)  
Do a postorder traversal of a tree.

This generator recursively yields nodes that it has visited in a postorder fashion. That is, it descends through the tree depth-first to yield all of a node's children's postorder traversal before yielding the node itself.

**Parameters** *node* : sympy expression

The expression to traverse.

**keys** : (default None) sort key(s)

The key(s) used to sort args of Basic objects. When None, args of Basic objects are processed in arbitrary order. If key is defined, it will be passed along to ordered() as the only key(s) to use to sort the arguments; if key is simply True then the default keys of ordered will be used (node count and default\_sort\_key).

## Examples

```
>>> from sympy.utilities.iterables import postorder_traversal
>>> from sympy.abc import w, x, y, z
```

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(postorder_traversal(w + (x + y)*z))
[z, y, x, x + y, z*(x + y), w, w + z*(x + y)]
>>> list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

sympy.utilities.iterables.prefixes(*seq*)

Generate all prefixes of a sequence.

## Examples

```
>>> from sympy.utilities.iterables import prefixes

>>> list(prefixes([1,2,3,4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

sympy.utilities.iterables.reshape(*seq*, *how*)

Reshape the sequence according to the template in *how*.

## Examples

```
>>> from sympy.utilities import reshape
>>> seq = list(range(1, 9))

>>> reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]

>>> reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]

>>> reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]

>>> reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]

>>> reshape(seq, ((2,), [2])) # etc...
[((1, 2), [3, 4]), ((5, 6), [7, 8])]

>>> reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]

>>> reshape(tuple(seq), ([[1], 1, (2,)],))
(([1], 2, (3, 4)), ([5], 6, (7, 8)))

>>> reshape(tuple(seq), ([1], 1, (2,)))
(([1], 2, (3, 4)), ([5], 6, (7, 8)))

>>> reshape(list(range(12)), [2, [3], set([2]), (1, (3,), 1)])
[[0, 1, [2, 3, 4], set([5, 6]), (7, (8, 9, 10), 11)]]
```

`sympy.utilities.iterables.rotate_left(x, y)`

Left rotates a list x by the number of steps specified in y.

### Examples

```
>>> from sympy.utilities.iterables import rotate_left
>>> a = [0, 1, 2]
>>> rotate_left(a, 1)
[1, 2, 0]
```

`sympy.utilities.iterables.rotate_right(x, y)`

Right rotates a list x by the number of steps specified in y.

### Examples

```
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
>>> rotate_right(a, 1)
[2, 0, 1]
```

`sympy.utilities.iterables.runs(seq, op=<built-in function gt>)`

Group the sequence into lists in which successive elements all compare the same with the comparison operator, op: op(seq[i + 1], seq[i]) is True from all elements in a run.

## Examples

```
>>> from sympy.utilities.iterables import runs
>>> from operator import ge
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

`sympy.utilities.iterables.sift(seq, keyfunc)`

Sift the sequence, `seq` into a dictionary according to `keyfunc`.

OUTPUT: each element in `expr` is stored in a list keyed to the value of `keyfunc` for the element.

See also:

`sympy.core.compatibility.ordered` (page 154)

## Examples

```
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp

>>> sift(range(5), lambda x: x % 2)
{0: [0, 2, 4], 1: [1, 3]}
```

`sift()` returns a `defaultdict()` object, so any key that has no matches will give `[]`.

```
>>> sift([x], lambda x: x.is_commutative)
{True: [x]}
>>> _[False]
[]
```

Sometimes you won't know how many keys you will get:

```
>>> sift([sqrt(x), exp(x), (y**x)**2],
...       lambda x: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

If you need to sort the sifted items it might be better to use `ordered` which can economically apply multiple sort keys to a sequence while sorting.

`sympy.utilities.iterables.subsets(seq, k=None, repetition=False)`

Generates all k-subsets (combinations) from an n-element set, `seq`.

A k-subset of an n-element set is any subset of length exactly k. The number of k-subsets of an n-element set is given by  $\text{binomial}(n, k)$ , whereas there are  $2^{**n}$  subsets all together. If k is None then all  $2^{**n}$  subsets will be returned from shortest to longest.

## Examples

```
>>> from sympy.utilities.iterables import subsets
```

`subsets(seq, k)` will return the  $n!/k!(n - k)!$  k-subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be "taken":

```
>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```

`subsets(seq, k, repetition=True)` will return the  $(n - 1 + k)!/k!/(n - 1)!$  combinations *with* repetition:

```
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

`sympy.utilities.iterables.take(iter, n)`

Return `n` items from `iter` iterator.

`sympy.utilities.iterables.topological_sort(graph, key=None)`

Topological sort of graph's vertices.

**Parameters “graph” : tuple[list, list[tuple[T, T]]]**

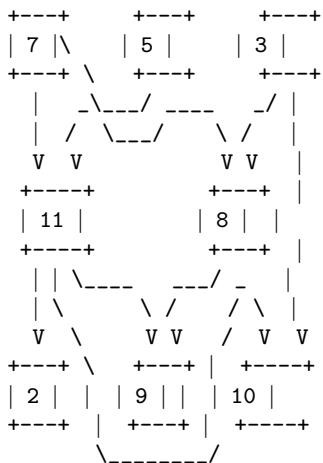
A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.

**“key” : callable[T] (optional)**

Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

## Examples

Consider a graph:



where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
...         (11, 2), (11, 9), (11, 10), (8, 9)]
```

To compute a topological sort for graph  $(V, E)$  issue:

```
>>> from sympy.utilities.iterables import topological_sort

>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use `key` parameter:

```
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
...
ValueError: cycle detected
```

**See also:**

[http://en.wikipedia.org/wiki/Topological\\_sorting](http://en.wikipedia.org/wiki/Topological_sorting)

`sympy.utilities.iterables.unflatten(iter, n=2)`

Group `iter` into tuples of length `n`. Raise an error if the length of `iter` is not a multiple of `n`.

`sympy.utilities.iterables.uniq(seq, result=None)`

Yield unique elements from `seq` as an iterator. The second parameter `result` is used internally; it is not necessary to pass anything for this.

## Examples

```
>>> from sympy.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False

>>> list(uniq(dat))
[1, 4, 5, 2]
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

`sympy.utilities.iterables.variations(seq, n, repetition=False)`

Returns a generator of the  $n$ -sized variations of `seq` (size  $N$ ). `repetition` controls whether items in `seq` can appear more than once;

## Examples

`variations(seq, n)` will return  $N! / (N - n)!$  permutations without repetition of `seq`'s elements:

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

`variations(seq, n, True)` will return the  $N^{**}n$  permutations obtained by allowing repetition of elements:

```
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

### 3.31.6 Lambdify

This module provides convenient functions to transform sympy expressions to lambda functions which can be used to calculate numerical values very fast.

`sympy.utilities.lambdify.implemented_function(symfunc, implementation)`

Add numerical `implementation` to function `symfunc`.

`symfunc` can be an `UndefinedFunction` instance, or a name string. In the latter case we create an `UndefinedFunction` instance with that name.

Be aware that this is a quick workaround, not a general method to create special symbolic functions. If you want to create a symbolic function to be used by all the machinery of SymPy you should subclass the `Function` class.

**Parameters** `symfunc` : str or `UndefinedFunction` instance

If `str`, then create new `UndefinedFunction` with this as name. If `symfunc` is a `sympy` function, attach implementation to it.

**implementation** : callable

numerical implementation to be called by `evalf()` or `lambdify`

**Returns** `afunc` : `sympy.FunctionClass` instance

function with attached implementation

#### Examples

```
>>> from sympy.abc import x
>>> from sympy.utilities.lambdify import lambdify, implemented_function
>>> from sympy import Function
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

`sympy.utilities.lambdify.lambdastr(args, expr, printer=None, dummify=False)`

Returns a string that can be evaluated to a lambda function.

## Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.lambdify import lambdastr
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x,y,z), [z,y,x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, lambdastr will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(*list(_flatten_args_([_0,_1])))'
```

`sympy.utilities.lambdify.lambdify(args, expr, modules=None, printer=None, use_imps=True, dummify=True)`

Returns a lambda function for fast calculation of numerical values.

If not specified differently by the user, SymPy functions are replaced as far as possible by either python-math, numpy (if available) or mpmath functions - exactly in this order. To change this behavior, the “modules” argument can be used. It accepts:

- the strings “math”, “mpmath”, “numpy”, “numexpr”, “sympy”
- any modules (e.g. math)
- dictionaries that map names of sympy functions to arbitrary functions
- lists that contain a mix of the arguments above, with higher priority given to entries appearing first.

The default behavior is to substitute all arguments in the provided expression with dummy symbols. This allows for applied functions (e.g.  $f(t)$ ) to be supplied as arguments. Call the function with `dummify=False` if dummy substitution is unwanted (and `args` is not a string). If you want to view the lambdified function or provide “sympy” as the module, you should probably set `dummify=False`.

For functions involving large array calculations, numexpr can provide a significant speedup over numpy. Please note that the available functions for numexpr are more limited than numpy but can be expanded with `implemented_function` and user defined subclasses of Function. If specified, numexpr may be the only option in modules. The official list of numexpr functions can be found at: <https://github.com/pydata/numexpr#supported-functions>

## Examples

```
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import w, x, y, z

>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1, 2, 3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
```

```
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> row(1, 2)
Matrix([[1, 3]])
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function.:

```
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

A more robust way of handling this is to always work with flattened arguments:

```
>>> from sympy.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in *expr* can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. Usually you attach this using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_imps` input parameter is False.

### 3.31.7 Memoization

```
sympy.utilities.memoization.assoc_recurrence_memo(base_seq)
```

Memo decorator for associated sequences defined by recurrence starting from base

`base_seq(n)` – callable to get base sequence elements

`XXX` works only for  $P_n = P_0$  cases `XXX` works only for  $m \leq n$  cases

```
sympy.utilities.memoization.recurrence_memo(initial)
```

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the `specfun/combinatorial` module

### 3.31.8 Miscellaneous

Miscellaneous stuff that doesn't really fit anywhere else.

```
sympy.utilities.misc.debug(*args)
```

Print `*args` if `SYMPY_DEBUG` is True, else do nothing.

```
sympy.utilities.misc.debug_decorator(func)
```

If `SYMPY_DEBUG` is True, it will print a nice execution tree with arguments and results of all decorated functions, else do nothing.

```
sympy.utilities.misc.find_executable(executable, path=None)
    Try to find 'executable' in the directories listed in 'path' (a string listing directories separated by 'os.pathsep'; defaults to os.environ['PATH']). Returns the complete filename or None if not found

sympy.utilities.misc.rawlines(s)
    Return a cut-and-pastable string that, when printed, is equivalent to the input. The string returned is formatted so it can be indented nicely within tests; in some cases it is wrapped in the dedent function which has to be imported from textwrap.
```

## Examples

Note: because there are characters in the examples below that need to be escaped because they are themselves within a triple quoted docstring, expressions below look more complicated than they would be if they were printed in an interpreter window.

```
>>> from sympy.utilities.misc import rawlines
>>> from sympy import TableForm
>>> s = str(TableForm([[1, 10]], headings=(None, ['a', 'bee'])))
>>> print(rawlines(s)) # the \ appears as \ when printed
(
    'a bee\n'
    '----\n'
    '1 10 '
)
>>> print(rawlines('\'\'\'this
... that\'\''))
dedent('\'\'\
    this
    that
\'\'')
>>> print(rawlines('\'\'\'this
... that
... \'\''))
dedent('\'\'\
    this
    that
\'\'')
>>> s = "'''this
... is a triple \
...
'''"
>>> print(rawlines(s))
dedent("'''\
    this
    is a triple \
'''")
>>> print(rawlines('\'\'\'this
... that
... \'\''))
(
    'this\n'
    'that\n'
    ', '
)
```

### 3.31.9 PKGDATA

pkgdata is a simple, extensible way for a package to acquire data file resources.

The getResource function is equivalent to the standard idioms, such as the following minimal implementation:

```
import sys, os

def getResource(identifier, pkgname=__name__):
    pkgpath = os.path.dirname(sys.modules[pkgname].__file__)
    path = os.path.join(pkgpath, identifier)
    return open(os.path.normpath(path), mode='rb')
```

When a `_loader_` is present on the module given by `_name_`, it will defer `getResource` to its `get_data` implementation and return it as a file-like object (such as `StringIO`).

```
sympy.utilities.pkgdata.getResource(identifier, pkgname='sympy.utilities.pkgdata')
```

Acquire a readable object for a given package name and identifier. An `IOError` will be raised if the resource can not be found.

For example:

```
mydata = getResource('mypkgdata.jpg').read()
```

Note that the package name must be fully qualified, if given, such that it would be found in `sys.modules`.

In some cases, `getResource` will return a real file object. In that case, it may be useful to use its `name` attribute to get the path rather than use it as a file-like object. For example, you may be handing data off to a C API.

### 3.31.10 pytest

py.test hacks to support XFAIL/XPASS

```
sympy.utilities.pytest.raises(expectedException, code=None)
```

Tests that `code` raises the exception `expectedException`.

`code` may be a callable, such as a lambda expression or function name.

If `code` is not given or `None`, `raises` will return a context manager for use in `with` statements; the code to execute then comes from the scope of the `with`.

`raises()` does nothing if the callable raises the expected exception, otherwise it raises an `AssertionError`.

#### Examples

```
>>> from sympy.utilities.pytest import raises

>>> raises(ZeroDivisionError, lambda: 1/0)
>>> raises(ZeroDivisionError, lambda: 1/2)
Traceback (most recent call last):
...
AssertionError: DID NOT RAISE

>>> with raises(ZeroDivisionError):
...     n = 1/0
>>> with raises(ZeroDivisionError):
```

```
...     n = 1/2
Traceback (most recent call last):
...
AssertionError: DID NOT RAISE
```

Note that you cannot test multiple statements via `with raises`:

```
>>> with raises(ZeroDivisionError):
...     n = 1/0      # will execute and raise, aborting the ``with``
...     n = 9999/0 # never executed
```

This is just what `with` is supposed to do: abort the contained statement sequence at the first exception and let the context manager deal with the exception.

To test multiple statements, you'll need a separate `with` for each:

```
>>> with raises(ZeroDivisionError):
...     n = 1/0      # will execute and raise
>>> with raises(ZeroDivisionError):
...     n = 9999/0 # will also execute and raise
```

### 3.31.11 Randomised Testing

Helpers for randomized testing

```
sympy.utilities.randtest.random_complex_number(a=2, b=-1, c=3, d=1, rational=False)
```

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee  $b \leq \text{Im } z \leq d$ ,  $a \leq \text{Re } z \leq c$

```
sympy.utilities.randtest.test_derivative_numerically(f, z, tol=1e-06, a=2, b=-1, c=3, d=1)
```

Test numerically that the symbolically computed derivative of  $f$  with respect to  $z$  is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

#### Examples

```
>>> from sympy import sin
>>> from sympy.abc import x
>>> from sympy.utilities.randtest import test_derivative_numerically as td
>>> td(sin(x), x)
True
```

```
sympy.utilities.randtest.verify_numerically(f, g, z=None, tol=1e-06, a=2, b=-1, c=3, d=1)
```

Test numerically that  $f$  and  $g$  agree when evaluated in the argument  $z$ .

If  $z$  is `None`, all symbols will be tested. This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

#### Examples

```
>>> from sympy import sin, cos
>>> from sympy.abc import x
>>> from sympy.utilities.randtest import verify_numerically as tn
```

```
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
True
```

### 3.31.12 Run Tests

This is our testing framework.

Goals:

- it should be compatible with py.test and operate very similarly (or identically)
- doesn't require any external dependencies
- preferably all the functionality should be in this file only
- no magic, just import the test file and execute the test functions, that's it
- portable

```
class sympy.utilities.runtests.PyTestReporter(verbose=False, tb='short', colors=True,
                                              force_colors=False, split=None)
```

Py.test like reporter. Should produce output identical to py.test.

```
write(text, color='', align='left', width=None, force_colors=False)
      Prints a text on the screen.
```

It uses sys.stdout.write(), so no readline library is necessary.

**Parameters** **color** : choose from the colors below, “” means default color

**align** : “left”/“right”, “left” is a normal print, “right” is aligned on  
the right-hand side of the screen, filled with spaces if necessary

**width** : the screen width

```
class sympy.utilities.runtests.Reporter
    Parent class for all reporters.
```

```
class sympy.utilities.runtests.SymPyDocTestFinder(verbose=False,
                                                    parser=<doctest.DocTestParser instance
                                                    at 0x7fb9b04e320>, recurse=True, exclude_empty=True)
```

A class used to extract the DocTests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types: modules, functions, classes, methods, staticmethods, classmethods, and properties.

Modified from doctest’s version by looking harder for code in the case that it looks like the the code comes from a different module. In the case of decorated functions (e.g. @vectorize) they appear to come from a different module (e.g. multidemensional) even though their code is not there.

```
class sympy.utilities.runtests.SymPyDocTestRunner(checker=None, verbose=None, option_flags=0)
```

A class used to run DocTest test cases, and accumulate statistics. The **run** method is used to process a single DocTest case. It returns a tuple (**f**, **t**), where **t** is the number of test cases tried, and **f** is the number of test cases that failed.

Modified from the doctest version to not reset the sys.displayhook (see issue 5140).

See the docstring of the original DocTestRunner for more information.

```
run(test, compileflags=None, out=None, clear_globs=True)
```

Run the examples in `test`, and display the results using the writer function `out`.

The examples are run in the namespace `test.globs`. If `clear_globs` is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use `clear_globs=False`.

`compileflags` gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to `globs`.

The output of each example is checked using `SympyDocTestRunner.check_output`, and the results are formatted by the `SympyDocTestRunner.report_*` methods.

```
class sympy.utilities.runtests.SympyOutputChecker
```

Compared to the `OutputChecker` from the `stdlib` our `OutputChecker` class supports numerical comparison of floats occurring in the output of the doctest examples

```
check_output(want, got, optionflags)
```

Return True iff the actual output from an example (`got`) matches the expected output (`want`). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See the documentation for `TestRunner` for more information about option flags.

```
sympy.utilities.runtests.convert_to_native_paths(lst)
```

Converts a list of ‘/’ separated paths into a list of native (os.sep separated) paths and converts to lowercase if the system is case insensitive.

```
sympy.utilities.runtests.doctest(*paths, **kwargs)
```

Runs doctests in all \*.py files in the `sympy` directory which match any of the given strings in `paths` or all tests if `paths=[]`.

Notes:

- Paths can be entered in native system format or in unix, forward-slash format.
- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

## Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.doctest()
```

Run one file:

```
>>> sympy.doctest("sympy/core/basic.py")
>>> sympy.doctest("polynomial.rst")
```

Run all tests in `sympy/functions/` and some particular file:

```
>>> sympy.doctest("/functions", "basic.py")
```

Run any file having polynomial in its name, `doc/modules/polynomial.rst`, `sympy/functions/special/polynomials.py`, and `sympy/polys/polynomial.py`:

```
>>> sympy.doctest("polynomial")
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form ‘a/b’, which will run part a of b. Note that the regular doctests and the Sphinx doctests are split independently. For instance, to run the first half of the test suite:

```
>>> sympy.doctest(split='1/2')
```

The `subprocess` and `verbose` options are the same as with the function `test()`. See the docstring of that function for more information.

`sympy.utilities.runtests.get_sympy_dir()`

Returns the root sympy directory and set the global value indicating whether the system is case sensitive or not.

```
sympy.utilities.runtests.run_all_tests(test_args=(),      test_kwargs={},      doctest_args=(),
                                         doctest_kwargs={},      examples_args=(),      exam-
                                         ples_kwargs={'quiet': True})
```

Run all tests.

Right now, this runs the regular tests (py.test), the doctests (bin/doctest), the examples (examples/all.py), and the sage tests (see `sympy/external/tests/test_sage.py`).

This is what `setup.py test` uses.

You can pass arguments and keyword arguments to the test functions that support them (for now, `test`, `doctest`, and the `examples`). See the docstrings of those functions for a description of the available options.

For example, to run the solvers tests with colors turned off:

```
>>> from sympy.utilities.runtests import run_all_tests
>>> run_all_tests(test_args=("solvers",),
...     test_kwargs={"colors":False})
```

```
sympy.utilities.runtests.run_in_subprocess_with_hash_randomization(function,           func-
                                         function_args=(),           func-
                                         function_kwargs={},         tion=
                                         command='/home/docs/checkouts/readthedocs.or-
                                         docs/bin/python', module='sympy.utilities.runtests',
                                         force=False)
```

Run a function in a Python subprocess with hash randomization enabled.

If hash randomization is not supported by the version of Python given, it returns False. Otherwise, it returns the exit value of the command. The function is passed to `sys.exit()`, so the return value of the function will be the return value.

The environment variable `PYTHONHASHSEED` is used to seed Python’s hash randomization. If it is set, this function will return False, because starting a new subprocess is unnecessary in that case. If it is not set, one is set at random, and the tests are run. Note that if this environment variable is set when Python starts, hash randomization is automatically enabled. To force a subprocess to be created even if `PYTHONHASHSEED` is set, pass `force=True`. This flag will not force a subprocess in Python versions that do not support hash randomization (see below), because those versions of Python do not support the `-R` flag.

`function` should be a string name of a function that is importable from the module `module`, like “`_test`”. The default for `module` is “`sympy.utilities.runtests`”. `function_args` and `function_kwargs`

should be a repr-able tuple and dict, respectively. The default Python command is `sys.executable`, which is the currently running Python command.

This function is necessary because the seed for hash randomization must be set by the environment variable before Python starts. Hence, in order to use a predetermined seed for tests, we must start Python in a separate subprocess.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

## Examples

```
>>> from sympy.utilities.runtests import (
... run_in_subprocess_with_hash_randomization)
>>> # run the core tests in verbose mode
>>> run_in_subprocess_with_hash_randomization("_test",
... function_args={"core":},
... function_kwargs={'verbose': True})
# Will return 0 if sys.executable supports hash randomization and tests
# pass, 1 if they fail, and False if it does not support hash
# randomization.
```

`sympy.utilities.runtests.split_list(l, split)`

Splits a list into part a of b

`split` should be a string of the form ‘a/b’. For instance, ‘1/3’ would give the split one of three.

If the length of the list is not divisible by the number of splits, the last split will have more items.

```
>>> from sympy.utilities.runtests import split_list
>>> a = list(range(10))
>>> split_list(a, '1/3')
[0, 1, 2]
>>> split_list(a, '2/3')
[3, 4, 5]
>>> split_list(a, '3/3')
[6, 7, 8, 9]
```

`sympy.utilities.runtests.sympytestfile(filename, module_relative=True, name=None, package=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, parser=<doctest.DocTestParser instance at 0x7fb9b0607a0>, encoding=None)`

Test examples in the given file. Return (#failures, #tests).

Optional keyword arg `module_relative` specifies how filenames should be interpreted:

- If `module_relative` is True (the default), then `filename` specifies a module-relative path. By default, this path is relative to the calling module’s directory; but if the `package` argument is specified, then it is relative to that package. To ensure os-independence, `filename` should use “/” characters to separate path segments, and should not be an absolute path (i.e., it may not begin with “/”).
- If `module_relative` is False, then `filename` specifies an os-specific path. The path may be absolute or relative (to the current working directory).

Optional keyword arg `name` gives the name of the test; by default use the file’s basename.

Optional keyword argument `package` is a Python package or the name of a Python package whose directory should be used as the base directory for a module relative filename. If no package is specified,

then the calling module's directory is used as the base directory for module relative filenames. It is an error to specify `package` if `module_relative` is False.

Optional keyword arg `globs` gives a dict to be used as the globals when executing examples; by default, use `{}`. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg `extraglobs` gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used.

Optional keyword arg `verbose` prints lots of stuff if true, prints only failures if false; by default, it's true iff “-v” is in `sys.argv`.

Optional keyword arg `report` prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg `optionflags` or's together module constants, and defaults to 0. Possible values (see the docs for details):

- `DONT_ACCEPT_TRUE_FOR_1`
- `DONT_ACCEPT_BLANKLINE`
- `NORMALIZE_WHITESPACE`
- `ELLIPSIS`
- `SKIP`
- `IGNORE_EXCEPTION_DETAIL`
- `REPORT_UDIFF`
- `REPORT_CDIFF`
- `REPORT_NDIFF`
- `REPORT_ONLY_FIRST_FAILURE`

Optional keyword arg `raise_on_error` raises an exception on the first unexpected exception or failure. This allows failures to be post-mortem debugged.

Optional keyword arg `parser` specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files.

Optional keyword arg `encoding` specifies an encoding that should be used to convert the file to unicode.

Advanced tomfoolery: `testmod` runs methods of a local instance of class `doctest.Tester`, then merges the results into (or creates) global `Tester` instance `doctest.master`. Methods of `doctest.master` can be called directly too, if you want to do something unusual. Passing `report=0` to `testmod` is especially useful then, to delay displaying a summary. Invoke `doctest.master.summarize(verbose)` when you're done fiddling.

```
sympy.utilities.runtests.test(*paths, **kwargs)
```

Run tests in the specified `test_*.py` files.

Tests in a particular `test_*.py` file are run if any of the given strings in `paths` matches a part of the test file's path. If `paths=[]`, tests in all `test_*.py` files are run.

Notes:

- If `sort=False`, tests are run in random order (not default).
- Paths can be entered in native system format or in unix, forward-slash format.

- Files that are on the blacklist can be tested by providing their path; they are only excluded if no paths are given.

### Explanation of test results

Out-put	Meaning
.	passed
F	failed
X	XPassed (expected to fail but passed)
f	XFAILED (expected to fail and indeed failed)
s	skipped
w	slow
T	timeout (e.g., when <code>--timeout</code> is used)
K	KeyboardInterrupt (when running the slow tests with <code>--slow</code> , you can interrupt one of them without killing the test runner)

Colors have no additional meaning and are used just to facilitate interpreting the output.

### Examples

```
>>> import sympy
```

Run all tests:

```
>>> sympy.test()
```

Run one file:

```
>>> sympy.test("sympy/core/tests/test_basic.py")
>>> sympy.test("_basic")
```

Run all tests in sympy/functions/ and some particular file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...             "sympy/functions")
```

Run all tests in sympy/core and sympy/utilities:

```
>>> sympy.test("/core", "/util")
```

Run specific test from a file:

```
>>> sympy.test("sympy/core/tests/test_basic.py",
...             kw="test_equality")
```

Run specific test from any file:

```
>>> sympy.test(kw="subs")
```

Run the tests with verbose mode on:

```
>>> sympy.test(verbose=True)
```

Don't sort the test output:

```
>>> sympy.test(sort=False)
```

Turn on post-mortem pdb:

```
>>> sympy.test(pdb=True)
```

Turn off colors:

```
>>> sympy.test(colors=False)
```

Force colors, even when the output is not to a terminal (this is useful, e.g., if you are piping to `less -r` and you still want colors)

```
>>> sympy.test(force_colors=False)
```

The traceback verboseness can be set to “short” or “no” (default is “short”)

```
>>> sympy.test(tb='no')
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form ‘`a/b`’, which will run part `a` of `b`. For instance, to run the first half of the test suite:

```
>>> sympy.test(split='1/2')
```

You can disable running the tests in a separate subprocess using `subprocess=False`. This is done to support seeding hash randomization, which is enabled by default in the Python versions where it is supported. If `subprocess=False`, hash randomization is enabled/disabled according to whether it has been enabled or not in the calling Python process. However, even if it is enabled, the seed cannot be printed unless it is called from a new Python process.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

If hash randomization is not supported `subprocess=False` is used automatically.

```
>>> sympy.test(subprocess=False)
```

To set the hash randomization seed, set the environment variable `PYTHONHASHSEED` before running the tests. This can be done from within Python using

```
>>> import os  
>>> os.environ['PYTHONHASHSEED'] = '42'
```

Or from the command line using

```
$ PYTHONHASHSEED=42 ./bin/test
```

If the seed is not set, a random seed will be chosen.

Note that to reproduce the same hash values, you must use both the same seed as well as the same architecture (32-bit vs. 64-bit).

### 3.31.13 Source Code Inspection

This module adds several functions for interactive source code inspection.

```
sympy.utilities.source.get_class(lookup_view)
```

Convert a string version of a class name to the object.

For example, `get_class('sympy.core.Basic')` will return class `Basic` located in module `sympy.core`

```
sympy.utilities.source.get_mod_func(callback)
```

splits the string path to a class into a string path to the module and the name of the class. For example:

```
>>> from sympy.utilities.source import get_mod_func
>>> get_mod_func('sympy.core.basic.Basic')
('sympy.core.basic', 'Basic')

sympy.utilities.source(object)
Prints the source code of a given object.
```

### 3.31.14 Timing Utilities

Simple tools for timing functions' execution, when IPython is not available.

```
sympy.utilities.timeutils.timed(func, setup='pass', limit=None)
Adaptively measure execution time of a function.
```

## 3.32 Parsing input

### 3.32.1 Parsing Functions Reference

```
sympy.parsing.sympy_parser.parse_expr(s, local_dict=None, transformations=(<function
lambda_notation at 0x7fb993465f0>, <function
auto_symbol at 0x7fb99346578>, <function auto_number
at 0x7fb9197b938>, <function factorial_notation at
0x7fb9197bd70>), global_dict=None, evaluate=True)
```

Converts the string `s` to a SymPy expression, in `local_dict`

**Parameters** `s` : str

The string to parse.

`local_dict` : dict, optional

A dictionary of local variables to use when parsing.

`global_dict` : dict, optional

A dictionary of global variables. By default, this is initialized with `from sympy import *`; provide this parameter to override this behavior (for instance, to parse "`Q & S`").

`transformations` : tuple, optional

A tuple of transformation functions used to modify the tokens of the parsed expression before evaluation. The default transformations convert numeric literals into their SymPy equivalents, convert undefined variables into SymPy symbols, and allow the use of standard mathematical factorial notation (e.g. `x!`).

`evaluate` : bool, optional

When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)

**See also:**

`sympy.parsing.sympy_parser.stringify_expr` (page 1162), `sympy.parsing.sympy_parser.eval_expr` (page 1162), `sympy.parsing.sympy_parser.standard_transformations` (page 1163),  
`sympy.parsing.sympy_parser.implicit_multiplication_application` (page 1164)

## Examples

```
>>> from sympy.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'sympy.core.numbers.Half'>
>>> from sympy.parsing.sympy_parser import standard_transformations,\n... implicit_multiplication_application
>>> transformations = (standard_transformations +
...     (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When evaluate=False, some automatic simplifications will not occur:

```
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

`sympy.parsing.sympy_parser.stringify_expr(s, local_dict, global_dict, transformations)`

Converts the string s to Python code, in local\_dict

Generally, `parse_expr` should be used.

`sympy.parsing.sympy_parser.eval_expr(code, local_dict, global_dict)`

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

`sympy.parsing.sympy_tokenize.printtoken(type, token, srow_scol, erow_ecol, line)`

```
sympy.parsing.sympy_tokenize.tokenize(readline,      tokeneater=<function      printtoken      at
                                         0x7fb91b2eb90>)
```

The `tokenize()` function accepts two parameters: one representing the input stream, and one providing an output mechanism for `tokenize()`.

The first parameter, `readline`, must be a callable object which provides the same interface as the `readline()` method of built-in file objects. Each call to the function should return one line of input as a string.

The second parameter, `tokeneater`, must also be a callable object. It is called once for each token, with five arguments, corresponding to the tuples generated by `generate_tokens()`.

`sympy.parsing.sympy_tokenize.untokenize(iterable)`

Transform tokens back into Python source code.

Each element returned by the iterable must be a token sequence with at least two elements, a token number and token value. If only two tokens are passed, the resulting output is poor.

**Round-trip invariant for full input:** Untokenized source will match input source exactly

Round-trip invariant for limited input:

```
# Output text will tokenize the back to the input
t1 = [tok[:2] for tok in generate_tokens(f.readline)]
newcode = untokenize(t1)
readline = iter(newcode.splitlines(1)).next
t2 = [tok[:2] for tok in generate_tokens(readline)]
if t1 != t2:
    raise ValueError("t1 should be equal to t2")

sympy.parsing.sympy_tokenize.generate_tokens(readline)
```

The `generate_tokens()` generator requires one argument, `readline`, which must be a callable object which provides the same interface as the `readline()` method of built-in file objects. Each call to the function should return one line of input as a string. Alternately, `readline` can be a callable function terminating with `StopIteration`:

```
readline = open(myfile).next      # Example of alternate readline
```

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the logical line; continuation lines are included.

```
sympy.parsing.sympy_tokenize.group(*choices)
sympy.parsing.sympy_tokenize.any(*choices)
sympy.parsing.sympy_tokenize.maybe(*choices)
sympy.parsing.maxima.parse_maxima(str, globals=None, name_dict={})
sympy.parsing.mathematica.mathematica(s)
```

### 3.32.2 Parsing Exceptions Reference

```
class sympy.parsing.sympy_tokenize.TokenError
class sympy.parsing.sympy_tokenize.StopTokenizing
```

### 3.32.3 Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to `parse_expr()` (page 1161) and are applied in the order given.

```
sympy.parsing.sympy_parser.standard_transformations = (<function lambda_notation at 0x7fb9d993465f0>, <
Standard transformations for parse_expr() (page 1161). Inserts calls to Symbol (page 95), Integer (page 103), and other SymPy datatypes and allows the use of standard factorial notation (e.g. x!).
```

```
sympy.parsing.sympy_parser.split_symbols(tokens, local_dict, global_dict)
```

Splits symbol names for implicit multiplication.

Intended to let expressions like `xyz` be parsed as `x*y*z`. Does not split Greek character names, so `theta` will *not* become `t*h*e*t*a`. Generally this should be used with `implicit_multiplication`.

```
sympy.parsing.sympy_parser.split_symbols_custom(predicate)
```

Creates a transformation that splits symbol names.

`predicate` should return True if the symbol name is to be split.

For instance, to retain the default behavior but avoid splitting certain symbol names, a predicate like this would work:

```
>>> from sympy.parsing.sympy_parser import (parse_expr, _tokenSplittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...         return _tokenSplittable(symbol)
...     return False
...
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
... (transformation, implicit_multiplication))
unsplittable
```

`sympy.parsing.sympy_parser.implicit_multiplication(result, local_dict, global_dict)`

Makes the multiplication operator optional in most cases.

Use this before `implicit_application()` (page 1164), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

## Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

`sympy.parsing.sympy_parser.implicit_application(result, local_dict, global_dict)`

Makes parentheses optional in some cases for function calls.

Use this after `implicit_multiplication()` (page 1164), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

## Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

`sympy.parsing.sympy_parser.function_exponentiation(tokens, local_dict, global_dict)`

Allows functions to be exponentiated, e.g. `cos**2(x)`.

## Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

---

```
sympy.parsing.sympy_parser.implicit_multiplication_application(result, local_dict,
                                                               global_dict)
```

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.

### Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10sin**2 x**2 + 3xyz + tan theta",
... transformations=(standard_transformations +
... (implicit_multiplication_application,)))
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

```
sympy.parsing.sympy_parser.rationalize(tokens, local_dict, global_dict)
Converts floats into Rational. Run AFTER auto_number.
```

```
sympy.parsing.sympy_parser.convert_xor(tokens, local_dict, global_dict)
Treats XOR, ^, as exponentiation, **.
```

These are included in :data:sympy.parsing.sympy\_parser.standard\_transformations and generally don't need to be manually added by the user.

```
sympy.parsing.sympy_parser.factorial_notation(tokens, local_dict, global_dict)
Allows standard notation for factorial.
```

```
sympy.parsing.sympy_parser.auto_symbol(tokens, local_dict, global_dict)
Inserts calls to Symbol for undefined variables.
```

```
sympy.parsing.sympy_parser.auto_number(tokens, local_dict, global_dict)
Converts numeric literals to use SymPy equivalents.
```

Complex numbers use `I`; integer literals use `Integer`, float literals use `Float`, and repeating decimals use `Rational`.

## 3.33 Calculus

Some calculus-related methods waiting to find a better place in the SymPy modules tree.

```
sympy.calculus.euler.euler_equations(L, funcs=(), vars=())
Find the Euler-Lagrange equations [R1] (page 1249) for a given Lagrangian.
```

### Parameters L : Expr

The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

For example, in the case of two functions  $f(x, y)$ ,  $g(x, y)$  and two independent variables  $x, y$  the Lagrangian would have the form:

$$L \left( f(x, y), g(x, y), \frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y}, \frac{\partial g(x, y)}{\partial x}, \frac{\partial g(x, y)}{\partial y}, x, y \right)$$

In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this couldn't be done).

**funcs** : Function or an iterable of Functions

The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

**vars** : Symbol or an iterable of Symbols

The Symbols that are the independent variables of the functions.

**Returns eqns** : list of Eq

The list of differential equations, one for each function.

## References

[R1] (page 1249)

## Examples

```
>>> from sympy import Symbol, Function
>>> from sympy.calculus.euler import euler_equations
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), t, t), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), t, t) + Derivative(u(t, x), x, x), 0)]
```

`sympy.calculus.singularities.singularities(expr, sym)`

Finds singularities for a function. Currently supported functions are: - univariate real rational functions

## References

[R2] (page 1249)

## Examples

```
>>> from sympy.calculus.singularities import singularities
>>> from sympy import Symbol
>>> x = Symbol('x', extended_real=True)
>>> singularities(x**2 + x + 1, x)
()
```

```
>>> singularities(1/(x + 1), x)
(-1,)
```

### 3.33.1 Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order.

The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- estimating a derivative (or interpolate) directly from a series of points is also provided (`apply_finite_diff`).
- making a finite difference approximation of a Derivative instance (`as_finite_diff`).

`sympy.calculus.finite_diff.apply_finite_diff(order, x_list, y_list, x0)`

Calculates the finite difference approximation of the derivative of requested order at  $x_0$  from points provided in `x_list` and `y_list`.

**Parameters** `order: int`

order of derivative to approximate. 0 corresponds to interpolation.

`x_list: sequence`

Strictly monotonically increasing sequence of values for the independent variable.

`y_list: sequence`

The function value at corresponding values for the independent variable in `x_list`.

`x0: Number or Symbol`

At what value of the independent variable the derivative should be evaluated.

**Returns** `sympy.core.add.Add` or `sympy.core.numbers.Number`

The finite difference expression approximating the requested derivative order at  $x_0$ .

**See also:**

[sympy.calculus.finite\\_diff.finite\\_diff\\_weights](#) (page 1169)

#### Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around  $x_0$  when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge's phenomenon.

#### References

Fortran 90 implementation with Python interface for numerics: [fitediff](#)

## Examples

```
>>> from sympy.calculus import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, map(cube, xlist), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. `apply_finite_diff` can also be used on more abstract objects:

```
>>> from sympy import IndexedBase, Idx
>>> from sympy.calculus import apply_finite_diff
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
(-1 + (x[i + 1] - x[i])/(-x[i - 1] + x[i]))*y[i]/(x[i + 1] - x[i]) + (-x[i - 1] + x[i])*y[i + 1]/((-x[i - 1] +
```

`sympy.calculus.finite_diff.as_finite_diff(derivative, points=1, x0=None, wrt=None)`

Returns an approximation of a derivative of a function in the form of a finite difference formula. The expression is a weighted sum of the function at a number of discrete values of (one of) the independent variable(s).

**Parameters** `derivative`: a `Derivative` instance (needs to have an `variables`

and `expr` attribute).

**points**: sequence or coefficient, optional

If sequence: discrete values ( $\geq \text{order}+1$ ) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length `order+1` centered around `x0`. default: 1 (step-size 1)

**x0**: number or Symbol, optional

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

**wrt**: Symbol, optional

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

**See also:**

`sympy.calculus.finite_diff.apply_finite_diff` (page 1167), `sympy.calculus.finite_diff.finite_diff_weight` (page 1169)

## Examples

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

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

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

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

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

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around  $x_0$ , but we can get an expression estimating the derivative at an offset:

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

Partial derivatives are also supported:

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

`sympy.calculus.finite_diff.finite_diff_weights(order, x_list, x0)`

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at '`x0`' of order 0, 1, ..., up to '`order`' using a recursive formula.

#### Parameters `order` : int

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

#### `x_list: sequence`

Strictly monotonically increasing sequence of values for the independent variable.

#### `x0: Number or Symbol`

At what value of the independent variable the finite difference weights should be generated.

#### >Returns list

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing accuracy.

#### See also:

`sympy.calculus.finite_diff.apply_finite_diff` (page 1167)

#### Notes

If weights for a finite difference approximation of the 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated “for free”, so are formulae using fewer and fewer of the parameters. This is something one can take advantage of to save computational cost.

## References

[R3] (page 1250)

## Examples

```
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> finite_diff_weights(1, [-S(1)/2, S(1)/2, S(3)/2, S(5)/2], 0)
[[[1, 0, 0, 0],
 [1/2, 1/2, 0, 0],
 [3/8, 3/4, -1/8, 0],
 [5/16, 15/16, -5/16, 1/16]],
 [[0, 0, 0, 0], [-1, 1, 0, 0], [-1, 1, 0, 0], [-23/24, 7/8, 1/8, -1/24]]]
```

the result is two sublists, the first is for the 0:th derivative (interpolation) and the second for the first derivative (we gave 1 as the parameter of order so this is why we get no list for a higher order derivative). Each sublist contains the most accurate formula in the end (all points used).

Beware of the offset in the lower accuracy formulae when looking at a centered difference:

```
>>> from sympy import S
>>> from sympy.calculus import finite_diff_weights
>>> finite_diff_weights(1, [-S(5)/2, -S(3)/2, -S(1)/2, S(1)/2,
...     S(3)/2, S(5)/2], 0)
[[[1, 0, 0, 0, 0, 0],
 [-3/2, 5/2, 0, 0, 0, 0],
 [3/8, -5/4, 15/8, 0, 0, 0],
 [1/16, -5/16, 15/16, 5/16, 0, 0],
 [3/128, -5/32, 45/64, 15/32, -5/128, 0],
 [3/256, -25/256, 75/128, 75/128, -25/256, 3/256]],
 [[0, 0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0, 0],
 [1, -3, 2, 0, 0, 0],
 [1/24, -1/8, -7/8, 23/24, 0, 0],
 [0, 1/24, -9/8, 9/8, -1/24, 0],
 [-3/640, 25/384, -75/64, 75/64, -25/384, 3/640]]]
```

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge's phenomenon by using Chebyshev nodes:

```
>>> from sympy import cos, symbols, pi, simplify
>>> from sympy.calculus import finite_diff_weights
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x+h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev nodes
>>> print(x_list)
[-h + x, -sqrt(2)*h/2 + x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4,
 (-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 6*x/h**2 - 8*x**3/h**4,
 (sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 (-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```

## 3.34 Category Theory Module

### 3.34.1 Introduction

The category theory module for SymPy will allow manipulating diagrams within a single category, including drawing them in TikZ and deciding whether they are commutative or not.

The general reference work this module tries to follow is

[JoyOfCats] J. Adamek, H. Herrlich, G. E. Strecker: Abstract and Concrete Categories. The Joy of Cats.

The latest version of this book should be available for free download from

[katmat.math.uni-bremen.de/acc/acc.pdf](http://katmat.math.uni-bremen.de/acc/acc.pdf)

The module is still in its pre-embryonic stage.

### 3.34.2 Base Class Reference

This section lists the classes which implement some of the basic notions in category theory: objects, morphisms, categories, and diagrams.

`class sympy.categories.Object`

The base class for any kind of object in an abstract category.

While technically any instance of [Basic](#) (page 62) will do, this class is the recommended way to create abstract objects in abstract categories.

`class sympy.categories.Morphism`

The base class for any morphism in an abstract category.

In abstract categories, a morphism is an arrow between two category objects. The object where the arrow starts is called the domain, while the object where the arrow ends is called the codomain.

Two morphisms between the same pair of objects are considered to be the same morphisms. To distinguish between morphisms between the same objects use [NamedMorphism](#) (page 1172).

It is prohibited to instantiate this class. Use one of the derived classes instead.

**See also:**

[IdentityMorphism](#) (page 1174), [NamedMorphism](#) (page 1172), [CompositeMorphism](#) (page 1173)

`codomain`

Returns the codomain of the morphism.

#### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.codomain
Object("B")
```

`compose(other)`

Composes self with the supplied morphism.

The order of elements in the composition is the usual order, i.e., to construct  $g \circ f$  use `g.compose(f)`.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> (g * f).domain
Object("A")
>>> (g * f).codomain
Object("C")
```

#### domain

Returns the domain of the morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.domain
Object("A")
```

## class sympy.categories.NamedMorphism

Represents a morphism which has a name.

Names are used to distinguish between morphisms which have the same domain and codomain: two named morphisms are equal if they have the same domains, codomains, and names.

#### See also:

[Morphism](#) (page 1171)

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f
NamedMorphism(Object("A"), Object("B"), "f")
>>> f.name
'f'
```

#### name

Returns the name of the morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.name
'f'
```

**class sympy.categories.CompositeMorphism**

Represents a morphism which is a composition of other morphisms.

Two composite morphisms are equal if the morphisms they were obtained from (components) are the same and were listed in the same order.

The arguments to the constructor for this class should be listed in diagram order: to obtain the composition  $g \circ f$  from the instances of `Morphism` (page 1171) `g` and `f` use `CompositeMorphism(f, g)`.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, CompositeMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> CompositeMorphism(f, g) == g * f
True
```

#### codomain

Returns the codomain of this composite morphism.

The codomain of the composite morphism is the codomain of its last component.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).codomain
Object("C")
```

#### components

Returns the components of this composite morphism.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).components
(NamedMorphism(Object("A"), Object("B"), "f"),
 NamedMorphism(Object("B"), Object("C"), "g"))
```

#### domain

Returns the domain of this composite morphism.

The domain of the composite morphism is the domain of its first component.

#### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).domain
Object("A")
```

#### flatten(*new\_name*)

Forgets the composite structure of this morphism.

If *new\_name* is not empty, returns a [NamedMorphism](#) (page 1172) with the supplied name, otherwise returns a [Morphism](#) (page 1171). In both cases the domain of the new morphism is the domain of this composite morphism and the codomain of the new morphism is the codomain of this composite morphism.

#### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).flatten("h")
NamedMorphism(Object("A"), Object("C"), "h")
```

### class sympy.categories.IdentityMorphism

Represents an identity morphism.

An identity morphism is a morphism with equal domain and codomain, which acts as an identity with respect to composition.

See also:

[Morphism](#) (page 1171)

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, IdentityMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> f * id_A == f
True
>>> id_B * f == f
True
```

class `sympy.categories.Category`  
An (abstract) category.

A category [JoyOfCats] is a quadruple  $K = (O, \text{hom}, \text{id}, \circ)$  consisting of

- a (set-theoretical) class  $O$ , whose members are called  $K$ -objects,
- for each pair  $(A, B)$  of  $K$ -objects, a set  $\text{hom}(A, B)$  whose members are called  $K$ -morphisms from  $A$  to  $B$ ,
- for each  $K$ -object  $A$ , a morphism  $\text{id} : A \rightarrow A$ , called the  $K$ -identity of  $A$ ,
- a composition law  $\circ$  associating with every  $K$ -morphisms  $f : A \rightarrow B$  and  $g : B \rightarrow C$  a  $K$ -morphism  $g \circ f : A \rightarrow C$ , called the composite of  $f$  and  $g$ .

Composition is associative,  $K$ -identities are identities with respect to composition, and the sets  $\text{hom}(A, B)$  are pairwise disjoint.

This class knows nothing about its objects and morphisms. Concrete cases of (abstract) categories should be implemented as classes derived from this one.

Certain instances of [Diagram](#) (page 1176) can be asserted to be commutative in a [Category](#) (page 1175) by supplying the argument `commutative_diagrams` in the constructor.

See also:

[Diagram](#) (page 1176)

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

### commutative\_diagrams

Returns the [FiniteSet](#) (page 910) of diagrams which are known to be commutative in this category.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

#### name

Returns the name of this category.

#### Examples

```
>>> from sympy.categories import Category
>>> K = Category("K")
>>> K.name
'K'
```

#### objects

Returns the class of objects of this category.

#### Examples

```
>>> from sympy.categories import Object, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> K = Category("K", FiniteSet(A, B))
>>> K.objects
Class({Object("A"), Object("B")})
```

### class sympy.categories.Diagram

Represents a diagram in a certain category.

Informally, a diagram is a collection of objects of a category and certain morphisms between them. A diagram is still a monoid with respect to morphism composition; i.e., identity morphisms, as well as all composites of morphisms included in the diagram belong to the diagram. For a more formal approach to this notion see [Pare1970].

The components of composite morphisms are also added to the diagram. No properties are assigned to such morphisms by default.

A commutative diagram is often accompanied by a statement of the following kind: “if such morphisms with such properties exist, then such morphisms which such properties exist and the diagram is commutative”. To represent this, an instance of `Diagram` (page 1176) includes a collection of morphisms which are the premises and another collection of conclusions. `premises` and `conclusions` associate morphisms belonging to the corresponding categories with the `FiniteSet` (page 910)’s of their properties.

The set of properties of a composite morphism is the intersection of the sets of properties of its components. The domain and codomain of a conclusion morphism should be among the domains and codomains of the morphisms listed as the premises of a diagram.

No checks are carried out of whether the supplied object and morphisms do belong to one and the same category.

## References

[Pare1970] B. Pareigis: Categories and functors. Academic Press, 1970.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet, pprint, default_sort_key
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> premises_keys = sorted(d.premises.keys(), key=default_sort_key)
>>> pprint(premises_keys, use_unicode=False)
[g*f:A-->C, id:A-->A, id:B-->B, id:C-->C, f:A-->B, g:B-->C]
>>> pprint(d.premises, use_unicode=False)
{g*f:A-->C: EmptySet(), id:A-->A: EmptySet(), id:B-->B: EmptySet(), id:C-->C:
EmptySet(), f:A-->B: EmptySet(), g:B-->C: EmptySet()}
>>> d = Diagram([f, g], {g * f: "unique"})
>>> pprint(d.conclusions)
{g*f:A-->C: {unique}}
```

### conclusions

Returns the conclusions of this diagram.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> IdentityMorphism(A) in d.premises.keys()
True
>>> g * f in d.premises.keys()
True
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d.conclusions[g * f] == FiniteSet("unique")
True
```

### hom( $A, B$ )

Returns a 2-tuple of sets of morphisms between objects A and B: one set of morphisms listed as premises, and the other set of morphisms listed as conclusions.

See also:

`Object` (page 1171), `Morphism` (page 1171)

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> print(pretty(d.hom(A, C), use_unicode=False))
{g*f:A-->C}, {g*f:A-->C}
```

### `is_subdiagram(diagram)`

Checks whether `diagram` is a subdiagram of `self`. Diagram  $D'$  is a subdiagram of  $D$  if all premises (conclusions) of  $D'$  are contained in the premises (conclusions) of  $D$ . The morphisms contained both in  $D'$  and  $D$  should have the same properties for  $D'$  to be a subdiagram of  $D$ .

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d1 = Diagram([f])
>>> d.is_subdiagram(d1)
True
>>> d1.is_subdiagram(d)
False
```

### `objects`

Returns the `FiniteSet` (page 910) of objects that appear in this diagram.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> d.objects
{Object("A"), Object("B"), Object("C")}
```

### `premises`

Returns the premises of this diagram.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> d = Diagram([f])
>>> print(pretty(d.premises, use_unicode=False))
{id:A-->A: EmptySet(), id:B-->B: EmptySet(), f:A-->B: EmptySet()}

subdiagram_from_objects(objects)
```

If `objects` is a subset of the objects of `self`, returns a diagram which has as premises all those premises of `self` which have a domains and codomains in `objects`, likewise for conclusions. Properties are preserved.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

### 3.34.3 Diagram Drawing

This section lists the classes which allow automatic drawing of diagrams.

```
class sympy.categories.diagram_drawing.DiagramGrid(diagram, groups=None, **hints)
```

Constructs and holds the fitting of the diagram into a grid.

The mission of this class is to analyse the structure of the supplied diagram and to place its objects on a grid such that, when the objects and the morphisms are actually drawn, the diagram would be “readable”, in the sense that there will not be many intersections of morphisms. This class does not perform any actual drawing. It does strive nevertheless to offer sufficient metadata to draw a diagram.

Consider the following simple diagram.

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> from sympy import pprint
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
```

The simplest way to have a diagram laid out is the following:

```
>>> grid = DiagramGrid(diagram)
>>> (grid.width, grid.height)
(2, 2)
>>> pprint(grid)
A  B
      C
```

Sometimes one sees the diagram as consisting of logical groups. One can advise `DiagramGrid` as to such groups by employing the `groups` keyword argument.

Consider the following diagram:

```
>>> D = Object("D")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
```

Lay it out with generic layout:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B  D
      C
```

Now, we can group the objects *A* and *D* to have them near one another:

```
>>> grid = DiagramGrid(diagram, groups=[[A, D], B, C])
>>> pprint(grid)
B      C
```

A D

Note how the positioning of the other objects changes.

Further indications can be supplied to the constructor of `DiagramGrid` (page 1179) using keyword arguments. The currently supported hints are explained in the following paragraphs.

`DiagramGrid` (page 1179) does not automatically guess which layout would suit the supplied diagram better. Consider, for example, the following linear diagram:

```
>>> E = Object("E")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(C, D, "h")
>>> i = NamedMorphism(D, E, "i")
>>> diagram = Diagram([f, g, h, i])
```

When laid out with the generic layout, it does not get to look linear:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B
      C  D
```

E

To get it laid out in a line, use `layout="sequential"`:

```
>>> grid = DiagramGrid(diagram, layout="sequential")
>>> pprint(grid)
A B C D E
```

One may sometimes need to transpose the resulting layout. While this can always be done by hand, `DiagramGrid` (page 1179) provides a hint for that purpose:

```
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A
B
C
D
E
```

Separate hints can also be provided for each group. For an example, refer to `tests/test_drawing.py`, and see the different ways in which the five lemma [FiveLemma] can be laid out.

**See also:**

`sympy.categories.Diagram` (page 1176)

## References

[FiveLemma] [http://en.wikipedia.org/wiki/Five\\_lemma](http://en.wikipedia.org/wiki/Five_lemma)

### height

Returns the number of rows in this diagram layout.

### Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.height
2
```

### morphisms

Returns those morphisms (and their properties) which are sufficiently meaningful to be drawn.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.morphisms
{NamedMorphism(Object("A"), Object("B"), "f"): EmptySet(),
 NamedMorphism(Object("B"), Object("C"), "g"): EmptySet()}

width
```

Returns the number of columns in this diagram layout.

## Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.width
2
```

```
class sympy.categories.diagram_drawing.ArrowStringDescription(unit, curving, curving_amount, looping_start, looping_end, horizontal_direction, vertical_direction, label_position, label)
```

Stores the information necessary for producing an Xy-pic description of an arrow.

The principal goal of this class is to abstract away the string representation of an arrow and to also provide the functionality to produce the actual Xy-pic string.

`unit` sets the unit which will be used to specify the amount of curving and other distances. `horizontal_direction` should be a string of "r" or "l" specifying the horizontal offset of the target cell of the arrow relatively to the current one. `vertical_direction` should specify the vertical offset using a series of either "d" or "u". `label_position` should be either "^", "\_", or "|" to specify that the label should be positioned above the arrow, below the arrow or just over the arrow, in a break. Note that the notions "above" and "below" are relative to arrow direction. `label` stores the morphism label.

This works as follows (disregard the yet unexplained arguments):

```
>>> from sympy.categories.diagram_drawing import ArrowStringDescription
>>> astr = ArrowStringDescription(
...     unit="mm", curving=None, curving_amount=None,
...     looping_start=None, looping_end=None, horizontal_direction="d",
```

---

```

... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar[dr]_f

```

`curving` should be one of "`^`", "`_`" to specify in which direction the arrow is going to curve. `curving_amount` is a number describing how many unit's the morphism is going to curve:

```

>>> astr = ArrowStringDescription(
...     unit="mm", curving="^", curving_amount=12,
...     looping_start=None, looping_end=None, horizontal_direction="d",
...     vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar@/^12mm/[dr]_f

```

`looping_start` and `looping_end` are currently only used for loop morphisms, those which have the same domain and codomain. These two attributes should store a valid Xy-pic direction and specify, correspondingly, the direction the arrow gets out into and the direction the arrow gets back from:

```

>>> astr = ArrowStringDescription(
...     unit="mm", curving=None, curving_amount=None,
...     looping_start="u", looping_end="l", horizontal_direction="",
...     vertical_direction="", label_position="_", label="f")
>>> print(str(astr))
\ar@{u,l}[]_f

```

`label_displacement` controls how far the arrow label is from the ends of the arrow. For example, to position the arrow label near the arrow head, use "`>`:

```

>>> astr = ArrowStringDescription(
...     unit="mm", curving="^", curving_amount=12,
...     looping_start=None, looping_end=None, horizontal_direction="d",
...     vertical_direction="r", label_position="_", label="f")
>>> astr.label_displacement = ">"
>>> print(str(astr))
\ar@/^12mm/[dr]>f

```

Finally, `arrow_style` is used to specify the arrow style. To get a dashed arrow, for example, use "`{-->}`" as arrow style:

```

>>> astr = ArrowStringDescription(
...     unit="mm", curving="^", curving_amount=12,
...     looping_start=None, looping_end=None, horizontal_direction="d",
...     vertical_direction="r", label_position="_", label="f")
>>> astr.arrow_style = "{-->}"
>>> print(str(astr))
\ar@/^12mm/{-->}[dr]_f

```

## See also:

[XypicDiagramDrawer](#) (page 1184)

## Notes

Instances of `ArrowStringDescription` (page 1182) will be constructed by `XypicDiagramDrawer` (page 1184) and provided for further use in formatters. The user is not expected to construct instances of `ArrowStringDescription` (page 1182) themselves.

To be able to properly utilise this class, the reader is encouraged to checkout the Xy-pic user guide, available at [Xypic].

## References

[Xypic] <http://www.tug.org/applications/Xy-pic/>

`class sympy.categories.diagram_drawing.XypicDiagramDrawer`

Given a `Diagram` (page 1176) and the corresponding `DiagramGrid` (page 1179), produces the Xy-pic representation of the diagram.

The most important method in this class is `draw`. Consider the following triangle diagram:

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a `DiagramGrid` (page 1179):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\ymatrix{
A \ar[d]_{{g}\circ{f}} \ar[r]^{{f}} & B \ar[l]^{{g}} \\
C &
}
```

For further details see the docstring of this method.

To control the appearance of the arrows, formatters are used. The dictionary `arrow_formatters` maps morphisms to formatter functions. A formatter accepts an `ArrowStringDescription` (page 1182) and is allowed to modify any of the arrow properties exposed thereby. For example, to have all morphisms with the property `unique` appear as dashed arrows, and to have their names prepended with `!`, the following should be done:

```
>>> def formatter(astr):
...     astr.label = "\exists ! " + astr.label
...     astr.arrow_style = "[-->]"
>>> drawer.arrow_formatters["unique"] = formatter
>>> print(drawer.draw(diagram, grid))
\ymatrix{
A \ar@{-->}[d]_{{\exists ! {g}\circ{f}}} \ar[r]^{{f}} & B \ar[l]^{{g}} \\
C &
}
```

To modify the appearance of all arrows in the diagram, set `default_arrow_formatter`. For example, to place all morphism labels a little bit farther from the arrow head so that they look more centred, do as follows:

```
>>> def default_formatter(astr):
...     astr.label_displacement = "(0.45)"
>>> drawer.default_arrow_formatter = default_formatter
>>> print(drawer.draw(diagram, grid))
\ymatrix{
A \ar@{-->}[d]_{(0.45){\exists ! {g}\circ{f}}} \ar[r]^{(0.45){f}} & B \ar[l]^{(0.45){g}} \\
```

```
C &
}
```

In some diagrams some morphisms are drawn as curved arrows. Consider the following diagram:

```
>>> D = Object("D")
>>> E = Object("E")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
>>> grid = DiagramGrid(diagram)
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^g & D \ar[l]^k \ar@/_3mm/[ll]_h \\
& C &
}
```

To control how far the morphisms are curved by default, one can use the `unit` and `default_curving_amount` attributes:

```
>>> drawer.unit = "cm"
>>> drawer.default_curving_amount = 1
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} & B \ar[d]^g & D \ar[l]^k \ar@/_1cm/[ll]_h \\
& C &
}
```

In some diagrams, there are multiple curved morphisms between the same two objects. To control by how much the curving changes between two such successive morphisms, use `default_curving_step`:

```
>>> drawer.default_curving_step = 1
>>> h1 = NamedMorphism(A, D, "h1")
>>> diagram = Diagram([f, g, h, k, h1])
>>> grid = DiagramGrid(diagram)
>>> print(drawer.draw(diagram, grid))
\xymatrix{
A \ar[r]_{f} \ar@/^1cm/[rr]^{h_1} & B \ar[d]^g & D \ar[l]^k \ar@/_2cm/[ll]_h \\
& C &
}
```

The default value of `default_curving_step` is 4 units.

**See also:**

`draw` (page 1185), `ArrowStringDescription` (page 1182)

`draw(diagram, grid, masked=None, diagram_format='')`

Returns the Xy-pic representation of `diagram` laid out in `grid`.

Consider the following simple triangle diagram.

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import DiagramGrid, XypicDiagramDrawer
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
```

To draw this diagram, its objects need to be laid out with a [DiagramGrid](#) (page 1179):

```
>>> grid = DiagramGrid(diagram)
```

Finally, the drawing:

```
>>> drawer = XypicDiagramDrawer()
>>> print(drawer.draw(diagram, grid))
\ymatrix{
A \ar[d]_{{g}\circ{f}} \ar[r]^{{f}} & B \ar[ld]^{{g}} \\
C &
}
```

The argument `masked` can be used to skip morphisms in the presentation of the diagram:

```
>>> print(drawer.draw(diagram, grid, masked=[g * f]))
\ymatrix{
A \ar[r]^{{f}} & B \ar[ld]^{{g}} \\
C &
}
```

Finally, the `diagram_format` argument can be used to specify the format string of the diagram. For example, to increase the spacing by 1 cm, proceeding as follows:

```
>>> print(drawer.draw(diagram, grid, diagram_format="@+1cm"))
\ymatrix@+1cm{
A \ar[d]_{{g}\circ{f}} \ar[r]^{{f}} & B \ar[ld]^{{g}} \\
C &
}

sympy.categories.diagram_drawing.xypic_draw_diagram(diagram,      masked=None,      dia-
                                                     gram_format=' ', groups=None, **hints)
```

Provides a shortcut combining [DiagramGrid](#) (page 1179) and [XypicDiagramDrawer](#) (page 1184). Returns an Xy-pic presentation of `diagram`. The argument `masked` is a list of morphisms which will be not be drawn. The argument `diagram_format` is the format string inserted after “`\ymatrix`”. `groups` should be a set of logical groups. The `hints` will be passed directly to the constructor of [DiagramGrid](#) (page 1179).

For more information about the arguments, see the docstrings of [DiagramGrid](#) (page 1179) and [XypicDiagramDrawer.draw](#).

See also:

[XypicDiagramDrawer](#) (page 1184), [DiagramGrid](#) (page 1179)

## Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import xypic_draw_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g], {g * f: "unique"})
>>> print(xypic_draw_diagram(diagram))
\ymatrix{
A \ar[d]_{{g}\circ{f}} \ar[r]^{{f}} & B \ar[ld]^{{g}} \\
C &
}
```

```
C &
}

sympy.categories.diagram_drawing.preview_diagram(diagram,           masked=None,           dia-
                                                 gram_format=';',   groups=None,   output=
                                                 put='png',   viewer=None,   euler=True,
                                                 **hints)
```

Combines the functionality of `xypic_draw_diagram` and `sympy.printing.preview`. The arguments `masked`, `diagram_format`, `groups`, and `hints` are passed to `xypic_draw_diagram`, while `output`, `viewer`, and ‘‘`euler`’’ are passed to `preview`.

See also:

`xypic_draw_diagram` (page 1186)

### Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy.categories import preview_diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> preview_diagram(d)
```

## 3.35 Differential Geometry Module

### 3.35.1 Introduction

### 3.35.2 Base Class Reference

```
class sympy.diffgeom.Manifold
```

Object representing a mathematical manifold.

The only role that this object plays is to keep a list of all patches defined on the manifold. It does not provide any means to study the topological characteristics of the manifold that it represents.

```
class sympy.diffgeom.Patch
```

Object representing a patch on a manifold.

On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parametrization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This object serves as a container/parent for all coordinate system charts that can be defined on the patch it represents.

### Examples

Define a Manifold and a Patch on that Manifold:

```
>>> from sympy.diffgeom import Manifold, Patch
>>> m = Manifold('M', 3)
>>> p = Patch('P', m)
>>> p in m.patches
True
```

```
class sympy.diffgeom.CoordSystem
    Contains all coordinate transformation logic.
```

### Examples

Define a Manifold and a Patch, and then define two coord systems on that patch:

```
>>> from sympy import symbols, sin, cos, pi
>>> from sympy.diffgeom import Manifold, Patch, CoordSystem
>>> from sympy.simplify import simplify
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
>>> patch = Patch('P', m)
>>> rect = CoordSystem('rect', patch)
>>> polar = CoordSystem('polar', patch)
>>> rect in patch.coord_systems
True
```

Connect the coordinate systems. An inverse transformation is automatically found by `solve` when possible:

```
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
>>> polar.coord_tuple_transform_to(rect, [0, 2])
Matrix([
[0],
[0]])
>>> polar.coord_tuple_transform_to(rect, [2, pi/2])
Matrix([
[0],
[2]])
>>> rect.coord_tuple_transform_to(polar, [1, 1]).applyfunc(simplify)
Matrix([
[sqrt(2)],
[-pi/4]])
```

Calculate the jacobian of the polar to cartesian transformation:

```
>>> polar.jacobian(rect, [r, theta])
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta), r*cos(theta)]])
```

Define a point using coordinates in one of the coordinate systems:

```
>>> p = polar.point([1, 3*pi/4])
>>> rect.point_to_coords(p)
Matrix([
[-sqrt(2)/2],
[ sqrt(2)/2]])
```

Define a basis scalar field (i.e. a coordinate function), that takes a point and returns its coordinates. It is an instance of `BaseScalarField`.

```
>>> rect.coord_function(0)(p)
-sqrt(2)/2
>>> rect.coord_function(1)(p)
sqrt(2)/2
```

Define a basis vector field (i.e. a unit vector field along the coordinate line). Vectors are also differential operators on scalar fields. It is an instance of `BaseVectorField`.

```
>>> v_x = rect.base_vector(0)
>>> x = rect.coord_function(0)
>>> v_x(x)
1
>>> v_x(v_x(x))
0
```

Define a basis oneform field:

```
>>> dx = rect.base_oneform(0)
>>> dx(v_x)
1
```

If you provide a list of names the fields will print nicely: - without provided names:

```
>>> x, v_x, dx
(rect_0, e_rect_0, drect_0)
```

•with provided names

```
>>> rect = CoordSystem('rect', patch, [x, y])
>>> rect.coord_function(0), rect.base_vector(0), rect.base_oneform(0)
(x, e_x, dx)
```

`base_oneform(coord_index)`

Return a basis 1-form field.

The basis one-form field for this coordinate system. It is also an operator on vector fields.

See the docstring of `CoordSystem` for examples.

`base_oneforms()`

Returns a list of all base oneforms.

For more details see the `base_oneform` method of this class.

`base_vector(coord_index)`

Return a basis vector field.

The basis vector field for this coordinate system. It is also an operator on scalar fields.

See the docstring of `CoordSystem` for examples.

`base_vectors()`

Returns a list of all base vectors.

For more details see the `base_vector` method of this class.

`connect_to(to_sys, from_coords, to_exprs, inverse=True, fill_in_gaps=False)`

Register the transformation used to switch to another coordinate system.

#### Parameters `to_sys`

another instance of `CoordSystem`

**from\_coords**  
list of symbols in terms of which `to_exprs` is given

**to\_exprs**  
list of the expressions of the new coordinate tuple

**inverse**  
try to deduce and register the inverse transformation

**fill\_in\_gaps**  
try to deduce other transformation that are made possible by composing the present transformation with other already registered transformation

**coord\_function(coord\_index)**  
Return a `BaseScalarField` that takes a point and returns one of the coords.  
Takes a point and returns its coordinate in this coordinate system.  
See the docstring of `CoordSystem` for examples.

**coord\_functions()**  
Returns a list of all coordinate functions.  
For more details see the `coord_function` method of this class.

**coord\_tuple\_transform\_to(to\_sys, coords)**  
Transform coords to coord system `to_sys`.  
See the docstring of `CoordSystem` for examples.

**jacobian(to\_sys, coords)**  
Return the jacobian matrix of a transformation.

**point(coords)**  
Create a `Point` with coordinates given in this coord system.  
See the docstring of `CoordSystem` for examples.

**point\_to\_coords(point)**  
Calculate the coordinates of a point in this coord system.  
See the docstring of `CoordSystem` for examples.

**class sympy.diffgeom.Point(coord\_sys, coords)**  
Point in a Manifold object.  
To define a point you must supply coordinates and a coordinate system.  
The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

## Examples

Define the boilerplate Manifold, Patch and coordinate systems:

```
>>> from sympy import symbols, sin, cos, pi
>>> from sympy.diffgeom import (
...     Manifold, Patch, CoordSystem, Point)
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
```

---

```
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
```

Define a point using coordinates from one of the coordinate systems:

```
>>> p = Point(polar, [r, 3*pi/4])
```

```
>>> p.coords()
```

```
Matrix([
```

```
[ r],
```

```
[3*pi/4]])
```

```
>>> p.coords(rect)
```

```
Matrix([
```

```
[-sqrt(2)*r/2],
```

```
[ sqrt(2)*r/2]])
```

```
coords(to_sys=None)
```

Coordinates of the point in a given coordinate system.

If `to_sys` is `None` it returns the coordinates in the system in which the point was defined.

`class sympy.diffgeom.BaseScalarField`

Base Scalar Field over a Manifold for a given Coordinate System.

A scalar field takes a point as an argument and returns a scalar.

A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems.

You can build complicated scalar fields by just building up SymPy expressions containing `BaseScalarField` instances.

## Examples

Define boilerplate Manifold, Patch and coordinate systems:

```
>>> from sympy import symbols, sin, cos, pi, Function
>>> from sympy.diffgeom import (
...     Manifold, Patch, CoordSystem, Point, BaseScalarField)
>>> r0, theta0 = symbols('r0, theta0')
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r0, theta0], [r0*cos(theta0), r0*sin(theta0)])
```

Point to be used as an argument for the filed:

```
>>> point = polar.point([r0, 0])
```

Examples of fields:

```
>>> fx = BaseScalarField(rect, 0)
>>> fy = BaseScalarField(rect, 1)
>>> (fx**2+fy**2).rcall(point)
r0**2

>>> g = Function('g')
>>> ftheta = BaseScalarField(polar, 1)
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)

class sympy.diffgeom.BaseVectorField
Vector Field over a Manifold.
```

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field).

A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems.

## Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy import symbols, pi, Function
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import BaseVectorField
>>> from sympy import pprint
>>> x0, y0, r0, theta0 = symbols('x0, y0, r0, theta0')
```

Points to be used as arguments for the field:

```
>>> point_p = R2_p.point([r0, theta0])
>>> point_r = R2_r.point([x0, y0])
```

Scalar field to operate on:

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
>>> s_field.rcall(point_r)
g(x0, y0)
>>> s_field.rcall(point_p)
g(r0*cos(theta0), r0*sin(theta0))
```

Vector field:

```
>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field))
/   d          \
|-----(g(x, xi_2))||_
\dx_i_2          /|xi_2=y
>>> pprint(v(s_field).rcall(point_r).doit())
d
```

```

---(g(x0, y0))
dy0
>>> pprint(v(s_field).rcall(point_p).doit())
/ d
|-----(g(r0*cos(theta0), xi_2))||
\dx{xi_2}                                /|xi_2=r0*sin(theta0)

class sympy.diffgeom.Commutator(v1, v2)
Commutator of two vector fields.

```

The commutator of two vector fields  $v_1$  and  $v_2$  is defined as the vector field  $[v_1, v_2]$  that evaluated on each scalar field  $f$  is equal to  $v_1(v_2(f)) - v_2(v_1(f))$ .

### Examples

Use the predefined R2 manifold, setup some boilerplate.

```

>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import Commutator
>>> from sympy import pprint
>>> from sympy.simplify import simplify

```

Vector fields:

```

>>> e_x, e_y, e_r = R2.e_x, R2.e_y, R2.e_r
>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0

```

Unfortunately, the current code is not able to compute everything:

```

>>> c_xr
Commutator(e_x, e_r)

```

```

>>> simplify(c_xr(R2.y**2).doit())
-2*cos(theta)*y**2/(x**2 + y**2)

```

```
class sympy.diffgeom.Differential(form_field)
```

Return the differential (exterior derivative) of a form field.

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case.

The differential  $df$  of the 0-form  $f$  is defined for any vector field  $v$  as  $df(v) = v(f)$ .

### Examples

Use the predefined R2 manifold, setup some boilerplate.

```

>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import Differential
>>> from sympy import pprint

```

Scalar field (0-forms):

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
```

Vector fields:

```
>>> e_x, e_y, = R2.e_x, R2.e_y
```

Differentials:

```
>>> dg = Differential(s_field)
>>> dg
d(g(x, y))
>>> pprint(dg(e_x))
/ d
|-----(g(xi_1, y))||
\dx{xi_1}           /|xi_1=x
>>> pprint(dg(e_y))
/ d
|-----(g(x, xi_2))||
\dx{xi_2}           /|xi_2=y
```

Applying the exterior derivative operator twice always results in:

```
>>> Differential(dg)
0
```

```
class sympy.diffgeom.TensorProduct(*args)
Tensor product of forms.
```

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order forms (e.g. 1-forms). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

## Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import TensorProduct
>>> from sympy import pprint

>>> TensorProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> TensorProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
0
>>> TensorProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
```

You can nest tensor products.

```
>>> tp1 = TensorProduct(R2.dx, R2.dy)
>>> TensorProduct(tp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
1
```

You can make partial contraction for instance when ‘raising an index’. Putting `None` in the second argument of `rcall` means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(R2.dx, R2.dx) + 3*TP(R2.dy, R2.dy)
>>> metric.rcall(R2.e_y, None)
3*dy
```

Or automatically pad the args with `None` without specifying them.

```
>>> metric.rcall(R2.e_y)
3*dy
```

```
class sympy.diffgeom.WedgeProduct(*args)
Wedge product of forms.
```

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

## Examples

Use the predefined `R2` manifold, setup some boilerplate.

```
>>> from sympy import Function
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import WedgeProduct
>>> from sympy import pprint

>>> WedgeProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> WedgeProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
-1
>>> WedgeProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
```

You can nest wedge products.

```
>>> wp1 = WedgeProduct(R2.dx, R2.dy)
>>> WedgeProduct(wp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
0
```

```
class sympy.diffgeom.LieDerivative(v_field, expr)
Lie derivative with respect to a vector field.
```

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

## Examples

```
>>> from sympy.diffgeom import (LieDerivative, TensorProduct)
>>> from sympy.diffgeom.rn import R2
>>> LieDerivative(R2.e_x, R2.y)
0
>>> LieDerivative(R2.e_x, R2.x)
1
>>> LieDerivative(R2.e_x, R2.e_x)
0
```

The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(R2.e_x, R2.e_r)
Commutator(e_x, e_r)
>>> LieDerivative(R2.e_x + R2.e_y, R2.x)
1
>>> tp = TensorProduct(R2.dx, R2.dy)
>>> LieDerivative(R2.e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
>>> LieDerivative(R2.e_x, tp).doit()
LieDerivative(e_x, TensorProduct(dx, dy))
```

```
class sympy.diffgeom.BaseCovarDerivativeOp(coord_sys, index, christoffel)
    Covariant derivative operator with respect to a base vector.
```

### Examples

```
>>> from sympy.diffgeom.rn import R2, R2_r
>>> from sympy.diffgeom import BaseCovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
(((0, 0), (0, 0)), ((0, 0), (0, 0)))
>>> cvd = BaseCovarDerivativeOp(R2_r, 0, ch)
>>> cvd(R2.x)
1
>>> cvd(R2.x*R2.e_x)
e_x
```

```
class sympy.diffgeom.CovarDerivativeOp(wrt, christoffel)
    Covariant derivative operator.
```

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import CovarDerivativeOp
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
(((0, 0), (0, 0)), ((0, 0), (0, 0)))
>>> cvd = CovarDerivativeOp(R2.x*R2.e_x, ch)
>>> cvd(R2.x)
x
>>> cvd(R2.x*R2.e_x)
x*e_x
```

```
sympy.diffgeom.intcurve_series(vector_field, param, start_point, n=6, coord_sys=None, coeffs=False)
```

Return the series expansion for an integral curve of the field.

Integral curve is a function  $\gamma$  taking a parameter in  $R$  to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt} f(\gamma(t))$$

where the given `vector_field` is denoted as  $V$ . This holds for any value  $t$  for the parameter and any scalar field  $f$ .

This equation can also be decomposed of a basis of coordinate functions

$$V(f_i)(\gamma(t)) = \frac{d}{dt} f_i(\gamma(t)) \quad \forall i$$

This function returns a series expansion of  $\gamma(t)$  in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

#### Parameters `vector_field`

the vector field for which an integral curve will be given

#### `param`

the argument of the function  $\gamma$  from R to the curve

#### `start_point`

the point which corresponds to  $\gamma(0)$

#### `n`

the order to which to expand

#### `coord_sys`

the coordinate system in which to expand coeffs (default False) - if True return a list of elements of the expansion

#### See also:

[intcurve\\_diffequ](#) (page 1198)

#### Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t, x, y
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([
[t + x],
[y]])
```

Or get the elements of the expansion in a list:

```
>>> series = intcurve_series(vector_field, t, start_point, n=3, coeffs=True)
>>> series[0]
Matrix([
[x],
[y]])
>>> series[1]
Matrix([
```

```
[t],  
[0]))  
>>> series[2]  
Matrix([  
[0],  
[0]])
```

The series in the polar coordinate system:

```
>>> series = intcurve_series(vector_field, t, start_point,  
...                           n=3, coord_sys=R2_p, coeffs=True)  
>>> series[0]  
Matrix([  
[sqrt(x**2 + y**2)],  
[atan2(y, x)])  
>>> series[1]  
Matrix([  
[t*x/sqrt(x**2 + y**2)],  
[-t*y/(x**2 + y**2)])  
>>> series[2]  
Matrix([  
[t**2*(-x**2/(x**2 + y**2))**(3/2) + 1/sqrt(x**2 + y**2))/2],  
[t**2*x*y/(x**2 + y**2)**2]])
```

`sympy.diffgeom.intcurve_diffequ(vector_field, param, start_point, coord_sys=None)`

Return the differential equation for an integral curve of the field.

Integral curve is a function  $\gamma$  taking a parameter in  $R$  to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt} f(\gamma(t))$$

where the given `vector_field` is denoted as  $V$ . This holds for any value  $t$  for the parameter and any scalar field  $f$ .

This function returns the differential equation of  $\gamma(t)$  in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

**Parameters** `vector_field`

the vector field for which an integral curve will be given

**param**

the argument of the function  $\gamma$  from  $R$  to the curve

**start\_point**

the point which corresponds to  $\gamma(0)$

**coord\_sys**

the coordinate system in which to give the equations

**Returns** a tuple of (equations, initial conditions)

**See also:**

[intcurve\\_series](#) (page 1196)

## Examples

Use the predefined R2 manifold:

```
>>> from sympy.abc import t
>>> from sympy.diffgeom.rn import R2, R2_p, R2_r
>>> from sympy.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point, R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]
```

`sympy.diffgeom.vectors_in_basis(expr, to_sys)`

Transform all base vectors in base vectors of a specified coord basis.

While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

## Examples

```
>>> from sympy.diffgeom import vectors_in_basis
>>> from sympy.diffgeom.rn import R2_r, R2_p
>>> vectors_in_basis(R2_r.e_x, R2_p)
-y*e_theta/(x**2 + y**2) + x*e_r/sqrt(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x
```

`sympy.diffgeom.twoform_to_matrix(expr)`

Return the matrix representing the twoform.

For the twoform  $w$  return the matrix  $M$  such that  $M[i, j] = w(e_i, e_j)$ , where  $e_i$  is the i-th base vector field for the coordinate system in which the expression of  $w$  is given.

## Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
```

```
Matrix([
[1, 0],
[0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[x, 0],
[0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.dy)/2)
Matrix([
[ 1, 0],
[-1/2, 1]]))
```

`sympy.diffgeom.metric_to_Christoffel_1st(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
((0, 0), (0, 0)), ((0, 0), (0, 0)))
>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
((1/2, 0), (0, 0)), ((0, 0), (0, 0)))
```

`sympy.diffgeom.metric_to_Christoffel_2nd(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

### Examples

```
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
((0, 0), (0, 0)), ((0, 0), (0, 0)))
>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
((1/(2*x), 0), (0, 0)), ((0, 0), (0, 0)))
```

`sympy.diffgeom.metric_to_Riemann_components(expr)`

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

### Examples

```
>>> from sympy import pprint, exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Riemann_components, TensorProduct
```

```
>>> TP = TensorProduct
>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
(((0, 0), (0, 0)), ((0, 0), (0, 0))), (((0, 0), (0, 0)), ((0, 0),
(0, 0)))

>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +           R2.r**2*TP(R2.dtheta, R2.dtheta)
>>> non_trivial_metric
exp(2*r)*TensorProduct(dr, dr) + r**2*TensorProduct(dtheta, dtheta)
>>> riemann = metric_to_Riemann_components(non_trivial_metric)
>>> riemann[0]
(((0, 0), (0, 0)), ((0, exp(-2*r)*r), (-exp(-2*r)*r, 0)))
>>> riemann[1]
(((0, -1/r), (1/r, 0)), ((0, 0), (0, 0)))

sympy.diffgeom.metric_to_Ricci_components(expr)
```

Return the components of the Ricci tensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

### Examples

```
>>> from sympy import pprint, exp
>>> from sympy.diffgeom.rn import R2
>>> from sympy.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
((0, 0), (0, 0))

>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) +           R2.r**2*TP(R2.dtheta, R2.
>>> non_trivial_metric
exp(2*r)*TensorProduct(dr, dr) + r**2*TensorProduct(dtheta, dtheta)
>>> metric_to_Ricci_components(non_trivial_metric)
((1/r, 0), (0, exp(-2*r)*r))
```

## 3.36 Contributions to docs

All contributions are welcome. If you'd like to improve something, look into the sources if they contain the information you need (if not, please fix them), otherwise the documentation generation needs to be improved (look in the doc/ directory).



---

## Special Topics

---

### 4.1 Introduction

The purpose of this collection of documents is to provide users of SymPy with topics which are not strictly tutorial or are longer than tutorials and tests. The documents will hopefully fill a gap as SymPy matures and users find more ways to show how SymPy can be used in more advanced topics.

### 4.2 Finite Difference Approximations to Derivatives

#### 4.2.1 Introduction

Finite difference approximations to derivatives is quite important in numerical analysis and in computational physics. In this tutorial we show how to use SymPy to compute approximations of varying accuracy. The hope is that these notes could be useful for the practicing researcher who is developing code in some language and needs to be able to efficiently generate finite difference formulae for various approximations.

In order to establish notation, we first state that we envision that there exists a continuous function  $F$  of a single variable  $x$ , with  $F$  having as many derivatives as desired. We sample  $x$  values uniformly at points along the real line separated by  $h$ . In most cases we want  $h$  to be small in some sense.  $F(x)$  may be expanded about some point  $x_0$  via the usual Taylor series expansion. Let  $x = x_0 + h$ . Then the Taylor expansion is

$$F(x_0 + h) = F(x_0) + \left(\frac{dF}{dx}\right)_{x_0} * h + \frac{1}{2!} \left(\frac{d^2F}{dx^2}\right)_{x_0} * h^2 + \frac{1}{3!} \left(\frac{d^3F}{dx^3}\right)_{x_0} * h^3 + \dots$$

In order to simplify the notation, we now define a set of coefficients  $c_n$ , where

$$c_n := \frac{1}{n!} \left(\frac{d^n F}{dx^n}\right)_{x_0}.$$

So now our series has the form:

$$F(x_0 + h) = F(x_0) + c_1 * h + c_2 * h^2 + c_3 * h^3 + \dots$$

In the following we will only use a finite grid of values  $x_i$  with  $i$  running from  $1, \dots, N$  and the corresponding values of our function  $F$  at these grid points denoted by  $F_i$ . So the problem is how to generate approximate values for the derivatives of  $F$  with the constraint that we use a subset of the finite set of pairs  $(x_i, F_i)$  of size  $N$ .

What follows are manipulations using SymPy to formulate approximations for derivatives of a given order and to assess its accuracy. First, we use SymPy to derive the approximations by using a rather brute force method frequently covered in introductory treatments. Later we shall make use of other SymPy functions which get the job done with more efficiency.

## 4.2.2 A Direct Method Using SymPy Matrices

If we let  $x_0 = x_i$ , evaluate the series at  $x_{i+1} = x_i + h$  and truncate all terms above  $O(h^1)$  we can solve for the single coefficient  $c_1$  and obtain an approximation to the first derivative:

$$\left(\frac{dF}{dx}\right)_{x_0} \approx \frac{F_{i+1} - F_i}{h} + O(h)$$

where the  $O(h)$  refers to the lowest order term in the series in  $h$ . This establishes that the derivative approximation is of first order accuracy. Put another way, if we decide that we can only use the two pairs  $(x_i, F_i)$  and  $(x_{i+1}, F_{i+1})$  we obtain a “first order accurate” derivative.

In addition to  $(x_i, F_i)$  we next use the two points  $(x_{i+1}, F_{i+1})$  and  $(x_{i+2}, F_{i+2})$ . Then we have two equations:

$$F_{i+1} = F_i + c_1 * h + \frac{1}{2} * c_2 * h^2 + \frac{1}{3!} * c_3 * h^3 + \dots$$

$$F_{i+2} = F_i + c_1 * (2h) + \frac{1}{2} * c_2 * (2h)^2 + \frac{1}{3!} * c_3 * (2h)^3 + \dots$$

If we again want to find the first derivative ( $c_1$ ), we can do that by eliminating the term involving  $c_2$  from the two equations. We show how to do it using SymPy.

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_0, h')
>>> Fi, Fip1, Fip2 = symbols('F_{i}, F_{i+1}, F_{i+2}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/dx, c_2=d**2F/dx**2
>>> c = symbols('c:3')
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Vector of right hand sides:

```
>>> R = Matrix([[Fi], [Fip1], [Fip2]])
```

Now we make a matrix consisting of the coefficients of the  $c_i$  in the nth degree polynomial  $P$ .

Coefficients of  $c_i$  evaluated at  $x_i$ :

```
>>> m11 = P(x0 , x0, c, n).diff(c[0])
>>> m12 = P(x0 , x0, c, n).diff(c[1])
>>> m13 = P(x0 , x0, c, n).diff(c[2])
```

Coefficients of  $c_i$  evaluated at  $x_i + h$ :

```
>>> m21 = P(x0+h, x0, c, n).diff(c[0])
>>> m22 = P(x0+h, x0, c, n).diff(c[1])
>>> m23 = P(x0+h, x0, c, n).diff(c[2])
```

Coefficients of  $c_i$  evaluated at  $x_i + 2 * h$ :

```
>>> m31 = P(x0+2*h, x0, c, n).diff(c[0])
>>> m32 = P(x0+2*h, x0, c, n).diff(c[1])
>>> m33 = P(x0+2*h, x0, c, n).diff(c[2])
```

Matrix of the coefficients is 3x3 in this case:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Matrix form of the three equations for the  $c_i$  is  $M^*X = R$ :

The solution is obtained by directly inverting the 3x3 matrix M:

```
>>> X = M.inv() * R
```

Note that all three coefficients make up the solution. The desired first derivative is coefficient  $c_1$  which is  $X[1]$ .

```
>>> print(together(X[1]))
(4*F_{i+1} - F_{i+2} - 3*F_{i})/(2*h)
```

It is instructive to compute another three-point approximation to the first derivative, except centering the approximation at  $x_i$  and thus using points at  $x_{i-1}$ ,  $x_i$ , and  $x_{i+1}$ . So here is how this can be done using the ‘brute force’ method:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, x0, h = symbols('x, x_0, h')
>>> Fi, Fim1, Fip1 = symbols('F_{i}, F_{i-1}, F_{i+1}')
>>> n = 3 # there are the coefficients c_0=Fi, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:3')
>>> # define a polynomial of degree n
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
>>> # now we make a matrix consisting of the coefficients
>>> # of the c_i in the nth degree polynomial P
>>> # coefficients of c_i evaluated at x_i
>>> m11 = P(x0 , x0, c, n).diff(c[0])
>>> m12 = P(x0 , x0, c, n).diff(c[1])
>>> m13 = P(x0 , x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(x0-h, x0, c, n).diff(c[0])
>>> m22 = P(x0-h, x0, c, n).diff(c[1])
>>> m23 = P(x0-h, x0, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(x0+h, x0, c, n).diff(c[0])
>>> m32 = P(x0+h, x0, c, n).diff(c[1])
>>> m33 = P(x0+h, x0, c, n).diff(c[2])
>>> # matrix of the coefficients is 3x3 in this case
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
```

Now that we have the matrix of coefficients we next form the right-hand-side and solve by inverting  $M$ :

```
>>> # matrix of the function values...actually a vector of right hand sides
>>> R = Matrix([[Fi], [Fim1], [Fip1]])
>>> # matrix form of the three equations for the c_i is M*X = R
>>> # solution directly inverting the 3x3 matrix M:
>>> X = M.inv() * R
>>> # note that all three coefficients make up the solution
>>> # the first derivative is coefficient c_1 which is X[1].
>>> print("The second-order accurate approximation for the first derivative is: ")
The second-order accurate approximation for the first derivative is:
>>> print(together(X[1]))
(F_{i+1} - F_{i-1})/(2*h)
```

These two examples serve to show how one can directly find second order accurate first derivatives using SymPy. The first example uses values of  $x$  and  $F$  at all three points  $x_i$ ,  $x_{i+1}$ , and  $x_{i+2}$  whereas the second example only uses values of  $x$  at the two points  $x_{i-1}$  and  $x_{i+1}$  and thus is a bit more efficient.

From these two simple examples a general rule is that if one wants a first derivative to be accurate to  $O(h^n)$  then one needs  $n+1$  function values in the approximating polynomial (here provided via the function  $P(x, x_0, c, n)$ ).

Now let's assess the question of the accuracy of the centered difference result to see how we determine that it is really second order. To do this we take the result for  $dF/dx$  and substitute in the polynomial expansion for a higher order polynomial and see what we get. To this end, we make a set of eight coefficients  $d$  and use them to perform the check:

```
>>> d = symbols('c:8')
>>> dfdxcheck = (P(x0+h, x0, d, 8) - P(x0-h, x0, d, 8))/(2*h)
>>> print(simplify(dfdxcheck)) # so the appropriate cancellation of terms involving 'h' happens
c1 + c3*h**2/6 + c5*h**4/120 + c7*h**6/5040
```

Thus we see that indeed the derivative is  $c_1$  with the next term in the series of order  $h^2$ .

However, it can quickly become rather tedious to generalize the direct method as presented above when attempting to generate a derivative approximation to high order, such as 6 or 8 although the method certainly works and using the present method is certainly less tedious than performing the calculations by hand.

As we have seen in the discussion above, the simple centered approximation for the first derivative only uses two point values of the  $(x_i, F_i)$  pairs. This works fine until one encounters the last point in the domain, say at  $i = N$ . Since our centered derivative approximation would use data at the point  $(x_{N+1}, F_{N+1})$  we see that the derivative formula will not work. So, what to do? Well, a simple way to handle this is to devise a different formula for this last point which uses points for which we do have values. This is the so-called backward difference formula. To obtain it, we can use the same direct approach, except now us the three points  $(x_N, F_N)$ ,  $(x_{N-1}, F_{N-1})$ , and  $(x_{N-2}, F_{N-2})$  and center the approximation at  $(x_N, F_N)$ . Here is how it can be done using SymPy:

```
>>> from __future__ import print_function
>>> from sympy import *
>>> x, xN, h = symbols('x, x_N, h')
>>> FN, FNm1, FNm2 = symbols('F_{N}, F_{N-1}, F_{N-2}')
>>> n = 8 # there are the coefficients c_0=F_i, c_1=dF/h, c_2=d**2F/h**2
>>> c = symbols('c:8')
>>> # define a polynomial of degree d
>>> def P(x, x0, c, n):
...     return sum( ((1/factorial(i))*c[i] * (x-x0)**i for i in range(n)) )
```

Now we make a matrix consisting of the coefficients of the  $c_i$  in the  $d$ th degree polynomial  $P$  coefficients of  $c_i$  evaluated at  $x_i, x_{i-1}$ , and  $x_{i+1}$ :

```
>>> m11 = P(xN , xN, c, n).diff(c[0])
>>> m12 = P(xN, xN, c, n).diff(c[1])
>>> m13 = P(xN , xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i - h
>>> m21 = P(xN-h, xN, c, n).diff(c[0])
>>> m22 = P(xN-h, xN, c, n).diff(c[1])
>>> m23 = P(xN-h, xN, c, n).diff(c[2])
>>> # coefficients of c_i evaluated at x_i + h
>>> m31 = P(xN-2*h, xN, c, n).diff(c[0])
>>> m32 = P(xN-2*h, xN, c, n).diff(c[1])
>>> m33 = P(xN-2*h, xN, c, n).diff(c[2])
```

Next we construct the  $3 \times 3$  matrix of the coefficients:

```
>>> M = Matrix([[m11, m12, m13], [m21, m22, m23], [m31, m32, m33]])
>>> # matrix of the function values...actually a vector of right hand sides
```

```
>>> R = Matrix([[FN], [FNm1], [FNm2]])
```

Then we invert  $M$  and write the solution to the  $3 \times 3$  system.

The matrix form of the three equations for the  $c_i$  is  $M * C = R$ . The solution is obtained by directly inverting  $M$ :

```
>>> X = M.inv() * R
```

The first derivative is coefficient  $c_1$  which is  $X[1]$ . Thus the second order accurate approximation for the first derivative is:

```
>>> print("The first derivative centered at the last point on the right is:")
The first derivative centered at the last point on the right is:
>>> print(together(X[1]))
(-4*F_{N-1} + F_{N-2} + 3*F_N)/(2*h)
```

Of course, we can devise a similar formula for the value of the derivative at the left end of the set of points at  $(x_1, F_1)$  in terms of values at  $(x_2, F_2)$  and  $(x_3, F_3)$ .

Also, we note that output of formats appropriate to Fortran, C, etc. may be done in the examples given above.

Next we show how to perform these and many other discretizations of derivatives, but using a much more efficient approach originally due to Bengt Fornberg and now incorporated into SymPy.



---

## User's Guide

---

### 5.1 Introduction

If you are new to SymPy, start with the [Tutorial](#) (page 3). If you went through it, now it's time to learn how SymPy works internally, which is what this guide is about. Once you grasp the ideas behind SymPy, you will be able to use it effectively and also know how to extend it and fix it. You may also be just interested in [SymPy Modules Reference](#) (page 59).

### 5.2 Learning SymPy

Everyone has different ways of understanding the code written by others.

#### 5.2.1 Ondej's approach

Let's say I'd like to understand how `x + y + x` works and how it is possible that it gets simplified to `2*x + y`.

I write a simple script, I usually call it `t.py` (I don't remember anymore why I call it that way):

```
from sympy.abc import x, y  
  
e = x + y + x  
  
print e
```

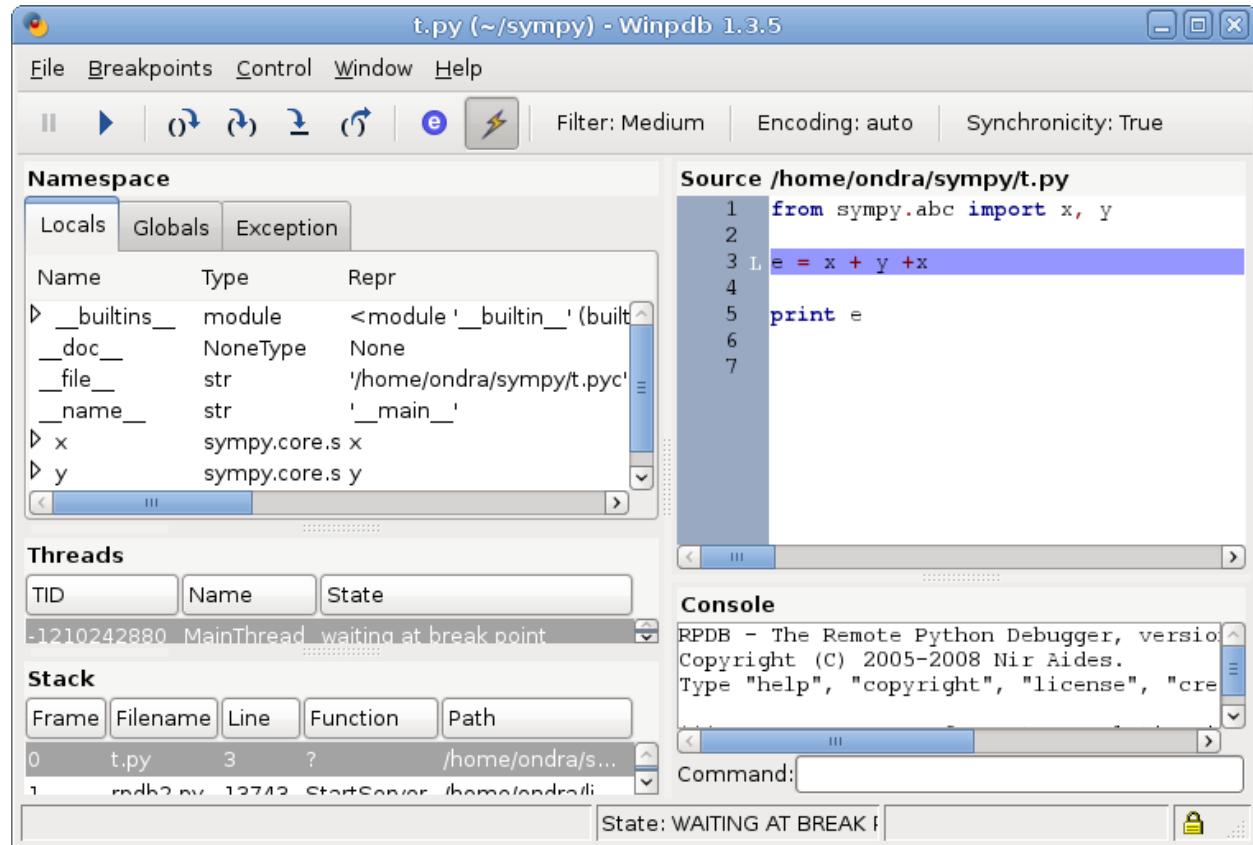
And I try if it works

```
$ python t.py  
y + 2*x
```

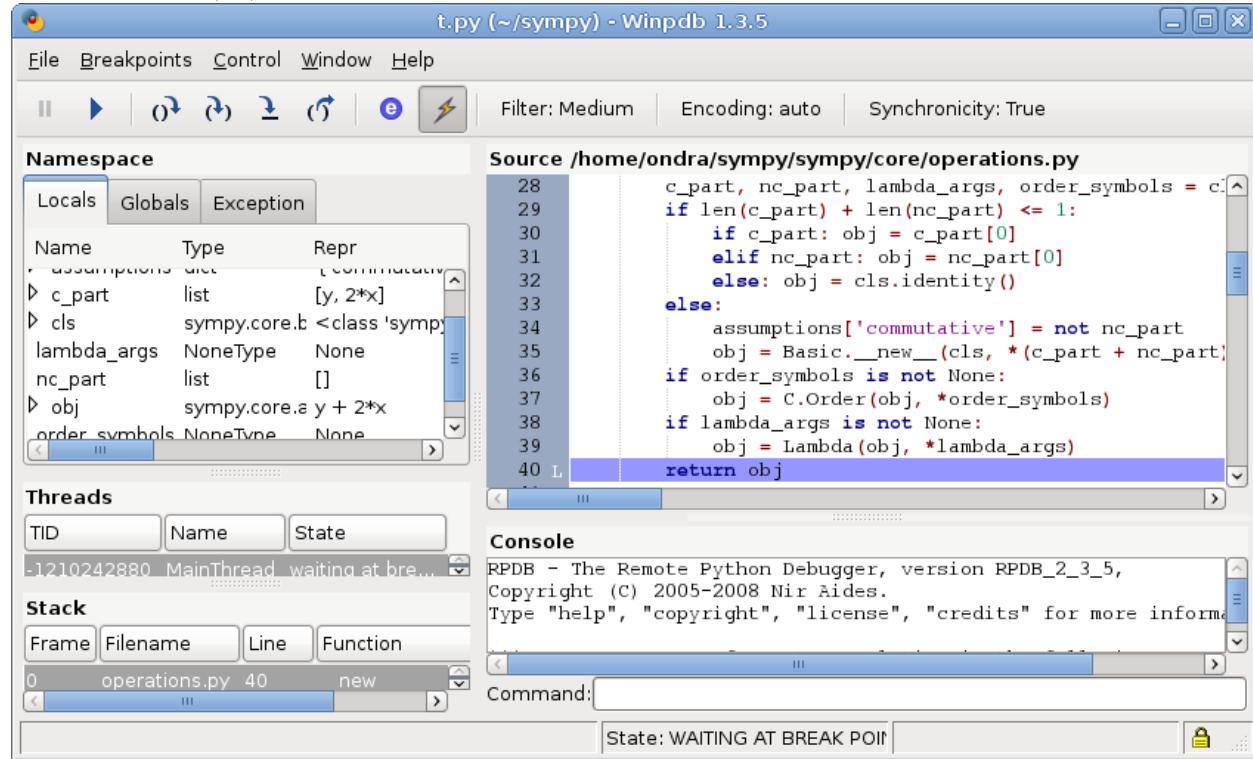
Now I start `winpdb` on it (if you've never used `winpdb` – it's an excellent multiplatform debugger, works on Linux, Windows and Mac OS X):

```
$ winpdb t.py  
y + 2*x
```

and a `winpdb` window will popup, I move to the next line using F6:



Then I step into (F7) and after a little debugging I get for example:



**Tip:** Make the winpdb window larger on your screen, it was just made smaller to fit in this guide.

I see values of all local variables in the left panel, so it's very easy to see what's happening. You can see, that the `y + 2*x` is emerging in the `obj` variable. Observing that `obj` is constructed from `c_part` and `nc_part` and seeing what `c_part` contains (`y` and `2*x`). So looking at the line 28 (the whole line is not visible on the screenshot, so here it is):

```
c_part, nc_part, lambda_args, order_symbols = cls.flatten(map(_sympify, args))
```

you can see that the simplification happens in `cls.flatten`. Now you can set the breakpoint on the line 28, quit wmpdb (it will remember the breakpoint), start it again, hit F5, this will stop at this breakpoint, hit F7, this will go into the function `Add.flatten()`:

```
@classmethod
def flatten(cls, seq):
    """
    Takes the sequence "seq" of nested Adds and returns a flatten list.

    Returns: (commutative_part, noncommutative_part, lambda_args,
              order_symbols)

    Applies associativity, all terms are commutable with respect to
    addition.
    """
    terms = {}      # term -> coeff
    # e.g. x**2 -> 5   for ... + 5*x**2 + ...
    #       # e.g. 3 + ...

    coeff = S.Zero  # standalone term
    # e.g. 3 + ...
    lambda_args = None
    order_factors = []
    while seq:
        o = seq.pop(0)
```

and then you can study how it works. I am going to stop here, this should be enough to get you going – with the above technique, I am able to understand almost any Python code.

## 5.3 SymPy's Architecture

We try to make the sources easily understandable, so you can look into the sources and read the doctests, it should be well documented and if you don't understand something, ask on the [issues](#).

You can find all the decisions archived in the [issues](#) or in the [oldissues](#) (up to issue 9035), to see rationale for doing this and that.

### 5.3.1 Basics

All symbolic things are implemented using subclasses of the `Basic` class. First, you need to create symbols using `Symbol("x")` or numbers using `Integer(5)` or `Float(34.3)`. Then you construct the expression using any class from SymPy. For example `Add(Symbol("a"), Symbol("b"))` gives an instance of the `Add` class. You can call all methods, which the particular class supports.

For easier use, there is a syntactic sugar for expressions like:

```
cos(x) + 1 is equal to cos(x).__add__(1) is equal to Add(cos(x), Integer(1))
```

or

$2/\cos(x)$  is equal to  $\cos(x)._rdiv_(2)$  is equal to `Mul(Rational(2), Pow(cos(x), Rational(-1)))`.

So, you can write normal expressions using python arithmetics like this:

```
a = Symbol("a")
b = Symbol("b")
e = (a + b)**2
print e
```

but from the SymPy point of view, we just need the classes `Add`, `Mul`, `Rational`, `Integer`.

### 5.3.2 Automatic evaluation to canonical form

For computation, all expressions need to be in a canonical form, this is done during the creation of the particular instance and only inexpensive operations are performed, necessary to put the expression in the canonical form. So the canonical form doesn't mean the simplest possible expression. The exact list of operations performed depend on the implementation. Obviously, the definition of the canonical form is arbitrary, the only requirement is that all equivalent expressions must have the same canonical form. We tried the conversion to a canonical (standard) form to be as fast as possible and also in a way so that the result is what you would write by hand - so for example  $b*a + -4 + b + a*b + 4 + (a + b)^2$  becomes  $2*a*b + b + (a + b)^2$ .

Whenever you construct an expression, for example `Add(x, x)`, the `Add.__new__()` is called and it determines what to return. In this case:

```
>>> from sympy import Add
>>> from sympy.abc import x
>>> e = Add(x, x)
>>> e
2*x

>>> type(e)
<class 'sympy.core.mul.Mul'>
```

`e` is actually an instance of `Mul(2, x)`, because `Add.__new__()` returned `Mul`.

### 5.3.3 Comparisons

Expressions can be compared using a regular python syntax:

```
>>> from sympy.abc import x, y
>>> x + y == y + x
True

>>> x + y == y - x
False
```

We made the following decision in SymPy: `a = Symbol("x")` and another `b = Symbol("x")` (with the same string "x") is the same thing, i.e `a == b` is `True`. We chose `a == b`, because it is more natural - `exp(x) == exp(x)` is also `True` for the same instance of `x` but different instances of `exp`, so we chose to have `exp(x) == exp(x)` even for different instances of `x`.

Sometimes, you need to have a unique symbol, for example as a temporary one in some calculation, which is going to be substituted for something else at the end anyway. This is achieved using `Dummy("x")`. So, to sum it up:

```
>>> from sympy import Symbol, Dummy
>>> Symbol("x") == Symbol("x")
True

>>> Dummy("x") == Dummy("x")
False
```

### 5.3.4 Debugging

Starting with 0.6.4, you can turn on/off debug messages with the environment variable `SYMPY_DEBUG`, which is expected to have the values True or False. For example, to turn on debugging, you would issue:

```
[user@localhost]: SYMPY_DEBUG=True ./bin/isympy
```

### 5.3.5 Functionality

There are no given requirements on classes in the library. For example, if they don't implement the `fdiff()` method and you construct an expression using such a class, then trying to use the `Basic.series()` method will raise an exception of not finding the `fdiff()` method in your class. This "duck typing" has an advantage that you just implement the functionality which you need.

You can define the `cos` class like this:

```
class cos(Function):
    pass
```

and use it like `1 + cos(x)`, but if you don't implement the `fdiff()` method, you will not be able to call `(1 + cos(x)).series()`.

The symbolic object is characterized (defined) by the things which it can do, so implementing more methods like `fdiff()`, `subs()` etc., you are creating a "shape" of the symbolic object. Useful things to implement in new classes are: `hash()` (to use the class in comparisons), `fdiff()` (to use it in series expansion), `subs()` (to use it in expressions, where some parts are being substituted) and `series()` (if the series cannot be computed using the general `Basic.series()` method). When you create a new class, don't worry about this too much - just try to use it in your code and you will realize immediately which methods need to be implemented in each situation.

All objects in SymPy are immutable - in the sense that any operation just returns a new instance (it can return the same instance only if it didn't change). This is a common mistake to change the current instance, like `self.arg = self.arg + 1` (wrong!). Use `arg = self.arg + 1; return arg` instead. The object is immutable in the sense of the symbolic expression it represents. It can modify itself to keep track of, for example, its hash. Or it can recalculate anything regarding the expression it contains. But the expression cannot be changed. So you can pass any instance to other objects, because you don't have to worry that it will change, or that this would break anything.

### 5.3.6 Conclusion

Above are the main ideas behind SymPy that we try to obey. The rest depends on the current implementation and may possibly change in the future. The point of all of this is that the interdependencies inside SymPy should be kept to a minimum. If one wants to add new functionality to SymPy, all that is necessary is to create a subclass of `Basic` and implement what you want.

### 5.3.7 Functions

How to create a new function with one variable:

```
class sign(Function):

    nargs = 1

    @classmethod
    def eval(cls, arg):
        if isinstance(arg, Basic.NaN):
            return S.NaN
        if isinstance(arg, Basic.Zero):
            return S.Zero
        if arg.is_positive:
            return S.One
        if arg.is_negative:
            return S.NegativeOne
        if isinstance(arg, Basic.Mul):
            coeff, terms = arg.as_coeff_mul()
            if not isinstance(coeff, Basic.One):
                return cls(coeff) * cls(Basic.Mul(*terms))

    is_finite = True

    def _eval_conjugate(self):
        return self

    def _eval_is_zero(self):
        return isinstance(self[0], Basic.Zero)
```

and that's it. The `_eval_*` functions are called when something is needed. The `eval` is called when the class is about to be instantiated and it should return either some simplified instance of some other class or if the class should be unmodified, return `None` (see `core/function.py` in `Function.__new__` for implementation details). See also tests in `sympy/functions/elementary/tests/test_interface.py` that test this interface. You can use them to create your own new functions.

The applied function `sign(x)` is constructed using

```
sign(x)
```

both inside and outside of SymPy. Unapplied functions `sign` is just the class itself:

```
sign
```

both inside and outside of SymPy. This is the current structure of classes in SymPy:

```
class BasicType(type):
    pass
class MetaBasicMeths(BasicType):
    ...
class BasicMeths(AssumeMeths):
    __metaclass__ = MetaBasicMeths
    ...
class Basic(BasicMeths):
    ...
class FunctionClass(MetaBasicMeths):
    ...
class Function(Basic, RelMeths, ArithMeths):
```

---

```
__metaclass__ = FunctionClass
...
```

The exact names of the classes and the names of the methods and how they work can be changed in the future.

This is how to create a function with two variables:

```
class chebyshev_root(Function):
    nargs = 2

    @classmethod
    def eval(cls, n, k):
        if not 0 <= k < n:
            raise ValueError("must have 0 <= k < n")
        return cos(S.Pi*(2*k + 1)/(2*n))
```

---

**Note:** the first argument of a `@classmethod` should be `cls` (i.e. not `self`).

---

Here it's how to define a derivative of the function:

```
>>> from sympy import Function, sympify, cos
>>> class my_function(Function):
...     nargs = 1
...
...     def fdiff(self, argindex = 1):
...         return cos(self.args[0])
...
...     @classmethod
...     def eval(cls, arg):
...         arg = sympify(arg)
...         if arg == 0:
...             return sympify(0)
```

So guess what this `my_function` is going to be? Well, it's derivative is `cos` and the function value at 0 is 0, but let's pretend we don't know:

```
>>> from sympy import pprint
>>> pprint(my_function(x).series(x, 0, 10))
      3      5      7      9
      x      x      x      x      / 10\
x - --- + ---- - ----- + ----- + 0\x / 
      6      120     5040    362880
```

Looks familiar indeed:

```
>>> from sympy import sin
>>> pprint(sin(x).series(x, 0, 10))
      3      5      7      9
      x      x      x      x      / 10\
x - --- + ---- - ----- + ----- + 0\x / 
      6      120     5040    362880
```

Let's try a more complicated example. Let's define the derivative in terms of the function itself:

```
>>> class what_am_i(Function):
...     nargs = 1
...
...     def fdiff(self, argindex = 1):
```

```
...         return 1 - what_am_i(self.args[0])**2
...
...     @classmethod
...     def eval(cls, arg):
...         arg = sympify(arg)
...         if arg == 0:
...             return sympify(0)
```

So what is `what_am_i`? Let's try it:

```
>>> pprint(what_am_i(x).series(x, 0, 10))
      3      5      7      9
x - -- + ----- - ----- + ----- + 0\x / 
      3      15     315    2835
```

Well, it's `tanh`:

```
>>> from sympy import tanh
>>> pprint(tanh(x).series(x, 0, 10))
      3      5      7      9
x - -- + ----- - ----- + ----- + 0\x / 
      3      15     315    2835
```

The new functions we just defined are regular SymPy objects, you can use them all over SymPy, e.g.:

```
>>> from sympy import limit
>>> limit(what_am_i(x)/x, x, 0)
1
```

### 5.3.8 Common tasks

Please use the same way as is shown below all across SymPy.

**accessing parameters:**

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

>>> e = sign(x**2)
>>> e.args
(x**2,)

>>> e.args[0]
x**2
```

Number arguments (in Adds and Muls) will always be the first argument;  
other arguments might be in arbitrary order:

```
>>> (1 + x + y*z).args[0]
1
>>> (1 + x + y*z).args[1] in (x, y*z)
True

>>> (y*z).args
(y, z)

>>> sin(y*z).args
(y*z,)
```

Never use internal methods or variables, prefixed with “\_” (example: don’t use `_args`, use `.args` instead).

### testing the structure of a SymPy expression

Applied functions:

```
>>> from sympy import sign, exp, Function
>>> e = sign(x**2)

>>> isinstance(e, sign)
True

>>> isinstance(e, exp)
False

>>> isinstance(e, Function)
True
```

So `e` is a `sign(z)` function, but not `exp(z)` function.

Unapplied functions:

```
>>> from sympy import sign, exp, FunctionClass
>>> e = sign

>>> f = exp

>>> g = Add

>>> isinstance(e, FunctionClass)
True

>>> isinstance(f, FunctionClass)
True

>>> isinstance(g, FunctionClass)
False

>>> g is Add
True
```

So `e` and `f` are functions, `g` is not a function.

## 5.4 Contributing

We welcome every SymPy user to participate in it’s development.

### 5.4.1 Improving the code

Go to [issues](#) that are sorted by priority and simply find something that you would like to get fixed and fix it. If you find something odd, please report it into [issues](#) first before fixing it.

Please read our excellent [SymPy Patches Tutorial](#) at our wiki for a guide on how to write patches to SymPy, how to work with Git, and how to make your life easier as you get started with SymPy.

### 5.4.2 Improving the docs

Please see [\*the documentation\*](#) (page 59) how to fix and improve SymPy's documentation. All contribution is very welcome.

---

## About

---

### 6.1 SymPy Development Team

SymPy is a team project and it was developed by a lot of people.

Here is a list of contributors together with what they do, (and in some cases links to their wiki pages), where they describe in more details what they do and what they are interested in (some people didn't want to be mentioned here, so see our repository history for a full list).

1. Ondej Čertk: started the project in 2006, on Jan 4, 2011 passed the project leadership to Aaron Meurer
2. Fabian Pedregosa: everything, reviewing patches, releases, general advice (issues and mailinglist), GSoC 2009
3. Jurjen N.E. Bos: pretty printing and other patches
4. Mateusz Paprocki: GSoC 2007, concrete math module, integration module, new core integration, a lot of patches, general advice, new polynomial module, improvements to solvers, simplifications, patch review
5. Marc-Etienne M.Leveille: matrix patch
6. Brian Jorgensen: GSoC 2007, plotting module and related things, patches
7. Jason Gedge: GSoC 2007, geometry module, a lot of patches and fixes, new core integration
8. Robert Schwarz: GSoC 2007, polynomials module, patches
9. Pearu Peterson: new core, sympycore project, general advice (issues and mailinglist)
10. Fredrik Johansson: mpmath project and its integration in SymPy, number theory, combinatorial functions, products & summation, statistics, units, patches, documentation, general advice (issues and mailinglist)
11. Chris Wu: GSoC 2007, linear algebra module
12. Ulrich Hecht: pattern matching and other patches
13. Goutham Lakshminarayan: number theory functions
14. David Lawrence: GHOP, Mathematica parser, square root denesting
15. Jaroslaw Tworek: GHOP, sympify AST implementation, sqrt() refactoring, maxima parser and other patches
16. David Marek: GHOP, derivative evaluation patch, int(NumberSymbol) fix
17. Bernhard R. Link: documentation patch

18. Andrej Tokark: GHOP, python printer
19. Or Dvory: GHOP, documentation
20. Saroj Adhikari: bug fixes
21. Pauli Virtanen: bug fix
22. Robert Kern: bug fix, common subexpression elimination
23. James Aspnes: bug fixes
24. Nimish Telang: multivariate lambdas
25. Abderrahim Kitouni: pretty printing + complex expansion bug fixes
26. Pan Peng: ode solvers patch
27. Friedrich Hagedorn: many bug fixes, refactorings and new features added
28. Elrond der Elbenfuerst: pretty printing fix
29. Rizgar Mella: BBP formula for pi calculating algorithm
30. Felix Kaiser: documentation + whitespace testing patches
31. Roberto Nobrega: several pretty printing patches
32. David Roberts: latex printing patches
33. Sebastian Krmer: implemented lambdify/numpy/mpmath cooperation, bug fixes, refactoring, lambdifying of matrices, large printing refactoring and bugfixes
34. Vinzent Steinberg: docstring patches, a lot of bug fixes, nsolve (nonlinear equation systems solver), compiling functions to machine code, patches review
35. Riccardo Gori: improvements and speedups to matrices, many bug fixes
36. Case Van Horsen: implemented optional support for gmpy in mpmath
37. tpn Rouka: a lot of bug fixes all over SymPy (matrix, simplification, limits, series, ...)
38. Ali Raza Syed: pretty printing/isympy on windows fix
39. Stefano Maggiolo: many bug fixes, polishings and improvements
40. Robert Cimrman: matrix patches
41. Bastian Weber: latex printing patches
42. Sebastian Krause: match patches
43. Sebastian Kreft: latex printing patches, Dirac delta function, other fixes
44. Dan (coolg49964): documentation fixes
45. Alan Bromborsky: geometric algebra modules
46. Boris Timokhin: matrix fixes
47. Robert (average.programmer): initial piecewise function patch
48. Andy R. Terrel: piecewise function polishing and other patches
49. Hubert Tsang: LaTeX printing fixes
50. Konrad Meyer: policy patch
51. Henrik Johansson: matrix zero finder patch
52. Priit Laes: line integrals, cleanups in ODE, tests added

53. Freddie Witherden: trigonometric simplifications fixes, LaTeX printer improvements
54. Brian E. Granger: second quantization physics module
55. Andrew Straw: lambdify() improvements
56. Kaifeng Zhu: factorint() fix
57. Ted Horst: Basic.\_call\_() fix, pretty printing fix
58. Andrew Docherty: Fix to series
59. Akshay Srinivasan: printing fix, improvements to integration
60. Aaron Meurer: ODE solvers (GSoC 2009), The Risch Algorithm for integration (GSoC 2010), other fixes, project leader as of Jan 4, 2011
61. Barry Wardell: implement series as function, several tests added
62. Tomasz Buchert: ccode printing fixes, code quality concerning exceptions, documentation
63. Vinay Kumar: polygamma tests
64. Johann Cohen-Tanugi: commutative diff tests
65. Jochen Voss: a test for the @cachit decorator and several other fixes
66. Luke Peterson: improve solve() to handle Derivatives
67. Chris Smith: improvements to solvers, many bug fixes, documentation and test improvements
68. Thomas Sidoti: MathML printer improvements
69. Florian Mickler: reimplementation of convex\_hull, several geometry module fixes
70. Nicolas Pourcelot: Infinity comparison fixes, latex fixes
71. Ben Goodrich: Matrix.jacobian() function improvements and fixes
72. Toon Verstraelen: code generation module, latex printing improvements
73. Ronan Lamy: test coverage script; limit, expansion and other fixes and improvements; cleanup
74. James Abbatello: fixes tests on windows
75. Ryan Krauss: fixes could\_extract\_minus\_sign(), latex fixes and improvements
76. Bill Flynn: substitution fixes
77. Kevin Goodsell: Fix to Integer/Rational
78. Jorn Baayen: improvements to piecewise functions and latex printing, bug fixes
79. Eh Tan: improve trigonometric simplification
80. Renato Coutinho: derivative improvements
81. Oscar Benjamin: latex printer fix, gcd bug fix
82. yvind Jensen: implemented coupled cluster expansion and wick theorem, improvements to assumptions, bugfixes
83. Julio Idichekop Filho: indentation fixes, docstring improvements
84. ukasz Pankowski: fix matrix multiplication with numpy scalars
85. Chu-Ching Huang: fix 3d plotting example
86. Fernando Perez: symarray() implemented
87. Raffaele De Feo: fix non-commutative expansion

88. Christian Muise: fixes to logic module
89. Matt Curry: GSoC 2010 project (symbolic quantum mechanics)
90. Kazuo Thow: cleanup pretty printing tests
91. Christian Schubert: Fix to sympify()
92. Jezreel Ng: fix hyperbolic functions rewrite
93. James Pearson: Py3k related fixes
94. Matthew Brett: fixes to lambdify
95. Addison Cugini: GSoC 2010 project (quantum computing)
96. Nicholas J.S. Kinar: improved documentation about “Immutability of Expressions”
97. Harold Erbin: Geometry related work
98. Thomas Dixon: fix a limit bug
99. Cristvo Sousa: implements `_sage_` method for sign function.
100. Andre de Fortier Smit: doctests in matrices improved
101. Mark Dewing: Fixes to Integral/Sum, MathML work
102. Alexey U. Gudchenko: various work in matrices
103. Gary Kerr: fix examples, docs
104. Sherjil Ozair: fixes to SparseMatrix
105. Oleksandr Gituliar: fixes to Matrix
106. Sean Vig: Quantum improvement
107. Prafullkumar P. Tale: fixed plotting documentation
108. Vladimir Peri: fix some Python 3 issues
109. Tom Bachmann: fixes to Real
110. Yuri Karadzhov: improvements to hyperbolic identities
111. Vladimir Lagunov: improvements to the geometry module
112. Matthew Rocklin: stats, sets, matrix expressions
113. Saptarshi Mandal: Test for an integral
114. Gilbert Gede: Tests for solvers
115. Anatolii Koval: Infinite 1D box example
116. Tomo Lazovich: fixes to quantum operators
117. Pavel Fedotov: fix to `is_number`
118. Kiboom Kim: fixes to cse function and `preorder_traversal`
119. Gregory Ksionda: fixes to Integral instantiation
120. Tom Bambas: prettier printing of examples
121. Jeremias Yehdegho: fixes to the nthoery module
122. Jack McCaffery: fixes to `asin` and `acos`
123. Raymond Wong: Quantum work

124. Luca Weihs: improvements to the geometry module
125. Shai ‘Deshe’ Wyborski: fix to numeric evaluation of hypergeometric sums
126. Thomas Wiecki: Fix Sum.diff
127. scar Njera: better Laguerre polynomial generator
128. Mario Pernici: faster algorithm for computing Groebner bases
129. Benjamin McDonald: Fix bug in geometry
130. Sam Magura: Improvements to Plot.savefig
131. Stefan Krastanov: Make Pyglet an external dependency
132. Bradley Froehle: Fix shell command to generate modules in setup.py
133. Min Ragan-Kelley: Fix isympy to work with IPython 0.11
134. Nikhil Sarda: Fix to combinatorics/prufer
135. Emma Hogan: Fixes to the documentation
136. Jason Moore: Fixes to the mechanics module
137. Julien Rioux: Fix behavior of some deprecated methods
138. Roberto Colistete, Jr.: Add num\_columns option to the pretty printer
139. Raoul Bourquin: Implement Euler numbers
140. Gert-Ludwig Ingold: Correct derivative of coth
141. Srinivas Vasudevan: Implementation of Catalan numbers
142. Miha Marolt: Add numpydoc extension to the Sphinx docs, fix ode\_order
143. Tim Lahey: Rotation matrices
144. Luis Garcia: update examples in symarry
145. Matt Rajca: Code quality fixes
146. David Li: Documentation fixes
147. David Ju: Increase test coverage, fixes to KroneckerDelta
148. Alexandr Gudulin: Code quality fixes
149. Bilal Akhtar: isympy man page
150. Grzegorz wirski: Fix to latex(), MacPorts portfile
151. Matt Habel: SymPy-wide pyflakes editing
152. Nikolay Lazarov: Translation of the tutorial to Bulgarian
153. Nichita Utiu: Add pretty printing to Product()
154. Tristan Hume: Fixes to test\_lambdify.py
155. Imran Ahmed Manzoor: Fixes to the test runner
156. Steve Anton: pyflakes cleanup of various modules, documentation fixes for logic
157. Sam Sleight: Fixes to geometry documentation
158. tsmars15: Fixes to code quality
159. Chancellor Arkantos: Fixes to the logo

160. Stepan Simska: Translation of the tutorial to Czech
161. Tobias Lenz: Unicode pretty printer for Sum
162. Siddhanathan Shanmugam: Documentation fixes for the Physics module
163. Tiffany Zhu: Improved the latex() docstring
164. Alexey Subach: Translation of the tutorial to Russian
165. Joan Creus: Improvements to the test runner
166. Geoffry Song: Improve code coverage
167. Puneeth Chaganti: Fix for the tutorial
168. Marcin Kostrzewa: SymPy Cheatsheet
169. Jim Zhang: Fixes to AUTHORS and .mailmap
170. Natalia Nawara: Fixes to Product()
171. vishal: Update the Czech tutorial translation to use a .po file
172. Shruti Mangipudi: added See Also documentation to Matrices
173. Davy Mao: Documentation
174. Swapnil Agarwal: added See Also documentation to ntheory, functions
175. Kendhia: Add XFAIL tests
176. jerryma1121: See Also documentation for geometry
177. Joachim Durchholz: corrected spacing issues identified by PyDev
178. Martin Povier: fix problem with rationaltools
179. Siddhant Jain: See Also documentation for functions/special
180. Kevin Hunter: Improvements to the inequality classes
181. Michael Mayorov: Improvement to series
182. Nathan Alison: Additions to the stats module
183. Christian Bhler: remove use of set\_main() in GA
184. Carsten Knoll: improvement to preview()
185. M R Bharath: modified use of int\_tested, improvement to Permutation
186. Matthias Toews: File permissions
187. Sergiu Ivanov: Fixes to zoo, SymPyDeprecationWarning
188. Jorge E. Cardona: Cleanup in polys
189. Sanket Agarwal: Rewrite coverage\_doctest.py script
190. Manoj Babu K.: Improve gamma function
191. Sai Nikhil: Fix to Heavyside with complex arguments
192. Aleksandar Makelov: Fixes regarding the dihedral group generator
193. Raphael Michel: German translation of the tutorial
194. Sachin Irukula: Changes to allow Dict sorting
195. Ashwini Oruganti: Changes to Pow printing

196. Andreas Kloeckner: Fix to cse()
197. Prateek Papriwal: improve summation documentation
198. Arpit Goyal: Improvements to Integral and Sum
199. Angadh Nanjangud: in physics.mechanics, added a function and tests for the parallel axis theorem
200. Comer Duncan: added dual, is\_antisymmetric, and det\_lu\_decomposition to matrices.py
201. Jens H. Nielsen: added sets to modules listing, update IPython printing extension
202. Joseph Dougherty: modified whitespace cleaning to remove multiple newlines at eof
203. marshall2389: Spelling correction
204. Guru Devanla: Implemented quantum density operator
205. George Waksman: Implemented JavaScript code printer and MathML printer
206. Angus Griffith: Fix bug in rsolve
207. Timothy Reluga: Rewrite trigonometric functions as rationals
208. Brian Stephanik: Test for a bug in fcode
209. Ljubia Moi: Serbian translation of the tutorial
210. Piotr Korgul: Polish translation of the tutorial
211. Rom le Clair: French translation of the tutorial
212. Alexandr Popov: Fixes to Pauli algebra
213. Saurabh Jha: Work on Kauers algorithm
214. Tarun Gaba: Implemented some trigonometric integrals
215. Takafumi Arakaki: Add info target to the doc Makefile
216. Alexander Eberspcher: correct typo in aboutus.rst
217. Sachin Joglekar: Simplification of logic expressions to SOP and POS forms
218. Tyler Pirtle: Fix improperly formatted error message
219. Vasily Povalyaev: Fix latex(Min)
220. Colleen Lee: replace uses of fnan with S.NaN
221. Niklas Thrne: Fix links in the docs
222. Huijun Mai: Chinese translation of the tutorial
223. Marek uppa: Improvements to symbols, tests
224. Prasoon Shukla: Bug fixes
225. Sergey B Kirpichev: Bug fixes
226. Stefen Yin: Fixes to the mechanics module
227. Thomas Hisch: Improvements to the printing module
228. Matthew Hoff: Addition to quantum module
229. Madeleine Ball: Bug fix
230. Case Van Horsen: Fixes to gmpy support
231. Mary Clark: Improvements to the group theory module

232. Rishabh Dixit: Bug fixes
233. Acebulf: Typos
234. Manoj Kumar: Bug fix
235. Akshit Agarwal: improvements to range handling in symbols
236. CJ Carey: Fix for limits of factorials
237. Patrick Lacasse: Fix for Piecewise.subs
238. Ananya H: Bug fix
239. Tarang Patel: added test for issue 4739
240. Christopher Dembia: improvements to mechanics documentation
241. Benjamin Fishbein: added rank method to Matrix
242. Sean Ge: made KroneckerDelta arguments canonically ordered
243. Ankit Agrawal: Statistical moments
244. Amit Jamadagni: qapply Rotation to spin states
245. Bjrn Dahlgren: Implemented finite difference module, minor fixes
246. Christophe Saint-Jean: fixed and added metrics to galgebra
247. Demian Wassermann: fix to ccode printer for Piecewise
248. Khagesh Patel: Addition to matrix expressions
249. Stephen Loo: Update minimum gmpy2 version
250. hm: Fixes to printing
251. Katja Sophie Hotz: use expansion in minpoly
252. Varun Joshi: Addition to functions
253. Chetna Gupta: Improvements to the Risch integration algorithm
254. Thilina Rathnayake: Fix to the matrices
255. Shravas K Rao: Implement prev\_lexicographic and next\_lexicographic
256. Max Hutchinson: Fix to HadamardProduct
257. Matthew Tadd: fix definition in units module
258. Alexander Hirzel: Updates to ODE docs
259. Randy Heydon: improve collinear point detection
260. Ramana Venkata: improvements to special functions
261. Oliver Lee: improvements to mechanics
262. Seshagiri Prabhu: hardcoded 3x3 determinant
263. Pradyumna: Fix to printing
264. Erik Welch: Fix a warning
265. Eric Nelson: Fixes to printing
266. Roland Puntaier: Improve App Engine support
267. Chris Conley: Use warnings instead of prints

268. Tim Swast: Help with pull requests and IPython
269. Dmitry Batkovich: Fix to series
270. Francesco Bonazzi: Improvements to matrices and tensors
271. Yuriy Demidov: Add examples from “Review of CAS mathematical capabilities”
272. Rick Muller: Implementation of quantum circuit plotting
273. Manish Gill: Fix infinite loop in Matrix constructor
274. Markus Mller: Add Jordan form for matrices
275. Amit Saha: Fixes to documentation
276. Jeremy: Bug fix
277. QuaBoo: Optimizations in ntheory
278. Stefan van der Walt: Fixes to mechanics module
279. David Joyner: Cryptography module
280. Lars Buitinck: Bug fix
281. Alkiviadis G. Akritas: Add a reference
282. Vinit Ravishankar: fix iterables documentation
283. Mike Boyle: Additions to the printing system
284. Heiner Kirchhoffer: PythonRational int conversion fix
285. Pablo Puente: Test cases from Wester paper
286. James Fiedler: Bug fix
287. Harsh Gupta: Bug fix
288. Tuomas Airaksinen: is\_real for besselx
289. rathmann: fix some Python 2/3 issues
290. Paul Strickland: documentation fix
291. James Goppert: correct code in form\_lagranges\_equations
292. Avichal Dayal: removed duplicate solutions obtained from solve
293. Paul Scott: pass along keywords for plots
294. shiprabanga: code quality fixes
295. Pramod Ch: fix problem with (I\*oo).expand\_complex()
296. Akshay: added normal\_lines to Ellipse
297. Buck Shlegeris: add inverses to error functions
298. Jonathan Miller: documentation
299. Edward: update license year
300. Rajath S: corrected iteration values in gaunt, racah, and wigner\_9j
301. Aditya Shah: bug fix, improvement to parsers
302. Sambuddha Basu: documentation fix
303. Zeel Shah: fixes to logic module

304. Abhinav Chanda: wigner\_3j and clebsch\_gordan fixes
305. Jim Crist: lambdify improvments
306. Sudhanshu Mishra: Added tests for racah() of sympy.physics.wigner module
307. Rajat Aggarwal: improvements to integration module
308. Soumya Dipta Biswas: Add support for Equivalent with multiple arguments
309. Anurag Sharma: improvements to Risch algorithm
310. Sushant Hiray: taylor\_term for sec
311. Ben Lucato: documentation fixes
312. Kunal Arora: documentation fixes
313. Henry Gebhardt: bugfix to ufuncify
314. Dammina Sahabandu: implemented continued fractions functions
315. Shukla: particle on a ring functions
316. Ralph Bean: typo
317. richierichrawr: documentation fixes
318. John Connor: fixes to nttheory
319. Juan Luis Cano Rodrguez: fixes to mechanics module
320. Sahil Shekhawat: fixes to core
321. Kundan Kumar: fixes to ODE identification of separable\_reduced
322. Stas Kelvich: fixes to quantum module
323. sevaader: fix to assumptions
324. Dhruvesh Vijay Parikh: implement Mbius function
325. Venkatesh Halli: improvement to matrices
326. Lennart Fricke: make sure that cse avoids symbol collision
327. Vlad Seghete: fix an issue with autowrap
328. shashank-agg: ensure that the range is correct for multiple plots
329. carstimon: make Abs(polar\_lift(arg)) -> abs(arg)
330. Pierre Haessig: fix typos
331. Maciej Baranski: refactor code in compilef
332. Zamrath Nizam: replace atoms(Symbol) with free\_symbols where appropriate
333. Benjamin Gudehus: add sampling\_density to stats
334. Faisal Anees: Egyptian Fractions
335. Robert Johansson: quantum field operators
336. Kalevi Suominen: fix to assumptions
337. Kaushik Varanasi: fix a typo
338. Fawaz Alazemi: combinatorics cheat sheet
339. Ambar Mehrotra: Fix to documentation

- 340. Mark Shoulson: Enhanced Egyptian fractions
- 341. David P. Sanders: Fix to latex printer
- 342. Peter Brady: avoid caching problem by using function instead of lambda
- 343. John V. Siratt: Fix to documentation
- 344. Sarwar Chahal: Fix to documentation
- 345. Nathan Woods: Fix to C code printer
- 346. Colin B. Macdonald: Fix to documentation
- 347. Marcus Nslund: Fix to documentation
- 348. Clemens Novak: improved docstring for apart
- 349. Craig A. Stoudt: correct addition of TWave objects
- 350. Mridul Seth: correct is\_tangent for Ellipse non-intersection
- 351. Raj: fixed typos in four files
- 352. Mihai A. Ionescu: Improvement to LaplaceTransform
- 353. immerr: add check of SYMPY\_DEBUG value
- 354. Leonid Blouvshtein: make integrals aware of both limits being +/-oo
- 355. Peleg Michaeli: implement the Rademacher distribution
- 356. Chai Wah Wu: Implement divisor\_sigma function
- 357. ck Lux: handle zoo, oo, nan in as\_int and round
- 358. zsc347: fixed a bug in crypto.rsa\_private\_key
- 359. Hamish Dickson: improve qubit tests
- 360. Michael Gallaspay: improve handling of inequalities involving RootOf
- 361. Roman Inflianskas: add svg support to preview
- 362. Duane Nykamp: improved function handling in parse\_expr
- 363. Ted Dokos: implemented interleaving for unions
- 364. Sunny Aggarwal: fix Integral.transform method
- 365. Victor Brebenar: fix the flickering problem
- 366. Akshat Jain: nsimplify returns ints quickly
- 367. WANG Longqi: some fixes for code quality, don't assign to self
- 368. Lukas Zorich: fix issue #6988
- 369. Juan Felipe Osorio: refine support for atan2 function
- 370. GitRay: fix incorrect term rewriting using cse
- 371. Eric Miller: Changed unicode references from hex to names
- 372. Shivam Vats: Added ReciprocalHyperbolicFunction class and misc bug fixes
- 373. Cody Herbst: modified Euler-MacLaurin to not exit when term == 0
- 374. AMiT Kumar: fix sign error in unrad
- 375. Nishith Shah: fix solving of Piecewise functions

- 376. Yury G. Kudryashov: Add README to examples/notebooks
- 377. Guillaume Gay: bugfix for LagrangesMethod
- 378. Ray Cathcart: improve error handling in \_random
- 379. Mihir Wadwekar: watch for duplicate bases in powsimp
- 380. Tuan Manh Lai: correct SparseMatrix.is\_Identity property
- 381. Asish Panda: implement SymmetricDifference class
- 382. Darshan Chaudhary: cross platform support in setup.py options
- 383. Alec Kalinin: fixed simplify and expand\_log so base of log is not dropped
- 384. Ralf Stephan: provide Function.\_sage\_; remove superfluous \_sage\_ methods
- 385. Aaditya Nair: import fix
- 386. Jayesh Lahori: remove uses of xrange
- 387. harshil goel: simplify roots by removing some powers from radicals
- 388. Luv Agarwal: change the return string of str(Eq) and str(Ne)
- 389. Jason Ly: fix typo in README.rst
- 390. Lokesh Sharma: reorder setup.py file imports to correct NameError
- 391. Sartaj Singh: use left | instead of lvert for latex Abs
- 392. Chris Swierczewski: RootOf.evalf watches for root on interval boundary
- 393. vizietto: fixed typo
- 394. Juha Remes: typos/pep8 fixes
- 395. Peter Schmidt: remove redundant tests
- 396. Jiaxing Liang: fix test\_f
- 397. Lucas Jones: Add test
- 398. Greg Ashton - LianLi: Allow use of symbols when solving system of eqns
- 399. jennifercw: Added checks into polynomials.py
- 400. Michael Boyle: fix formatting for Examples sections of docstrings

Up-to-date list in the order of the first contribution is given in the [AUTHORS](#) file.

You can use git to see the biggest developers:

```
git shortlog -ns
```

This will show the top developers from the last year:

```
git shortlog -ns --since="1 year"
```

## 6.2 Brief History

Sympy was started by Ondej Čertík in 2005, he wrote some code during the summer, then he wrote some more code during the summer 2006. In February 2007, Fabian Pedregosa joined the project and helped fix many things, contributed documentation and made it alive again. 5 students (Mateusz Paprocki, Brian Jorgensen, Jason Gedge, Robert Schwarz and Chris Wu) improved SymPy incredibly during the summer 2007 as part

of the Google Summer of Code (GSoC). Pearu Peterson joined the development during the summer 2007 and he has made SymPy much more competitive by rewriting the core from scratch, that has made it from 10x to 100x faster. Jurjen N.E. Bos has contributed pretty printing and other patches. Fredrik Johansson has wrote mpmath and contributed a lot of patches.

SymPy has participated in every GSoC since 2007. Moderate amount of SymPy's development has come from GSoC students.

In 2011, Ondej Čertk stepped down as lead developer, with Aaron Meurer, who also started as a GSoC student, taking his place.

Ondej Čertk is still active in the community, but is too busy with work and family to play a lead development role. Unfortunately, his remaining activity neither constructive nor productive anymore and SymPy just slowly dying now.

This project is a fork of the SymPy, last SymPy's commit is cbdd072 (22 Feb 2015). The git history goes back to 2007, when development was in svn and then in hg.

## 6.3 Financial and Infrastructure Support

- Google: SymPy has received generous financial support from Google in various years through the Google Summer of Code program by providing stipends:
  - in 2007 for 5 students ([GSoC 2007](#))
  - in 2008 for 1 student ([GSoC 2008](#))
  - in 2009 for 5 students ([GSoC 2009](#))
  - in 2010 for 5 students ([GSoC 2010](#))
  - in 2011 for 9 students ([GSoC 2011](#))
  - in 2012 for 6 students ([GSoC 2012](#))
  - in 2013 for 7 students ([GSoC 2013](#))
- Python Software Foundation (PSF) has hosted various GSoC students over the years:
  - 3 GSoC 2007 students (Brian, Robert and Jason)
  - 1 GSoC 2008 student (Fredrik)
  - 2 GSoC 2009 students (Freddie and Priit)
  - 4 GSoC 2010 students (Aaron, Christian, Matthew and yvind)
- Portland State University (PSU) has hosted following GSoC students:
  - 1 student (Chris) in 2007
  - 3 students (Aaron, Dale and Fabian) in 2009
  - 1 student (Addison) in 2010
- The Space Telescope Science Institute: STScI hosted 1 GSoC 2007 student (Mateusz)
- Several 13-17 year old pre-university students contributed as part of Google's [Code-In 2011](#). ([GCI 2011](#))
- Simula Research Laboratory: supports Pearu Peterson work in SymPy/SymPy Core projects
- GitHub is providing us with development and collaboration tools

## 6.4 License

Unless stated otherwise, all files in the SymPy project, SymPy's webpage (and wiki), all images and all documentation including this User's Guide are licensed using the new BSD license:

Copyright (c) 2006-2015 SymPy Development Team

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---

## Bibliography

---

- [R27] [http://en.wikipedia.org/wiki/Negative\\_number](http://en.wikipedia.org/wiki/Negative_number)
- [R28] [http://en.wikipedia.org/wiki/Parity\\_%28mathematics%29](http://en.wikipedia.org/wiki/Parity_%28mathematics%29)
- [R29] [http://en.wikipedia.org/wiki/Imaginary\\_number](http://en.wikipedia.org/wiki/Imaginary_number)
- [R30] [http://en.wikipedia.org/wiki/Composite\\_number](http://en.wikipedia.org/wiki/Composite_number)
- [R31] [http://en.wikipedia.org/wiki/Irrational\\_number](http://en.wikipedia.org/wiki/Irrational_number)
- [R32] [http://en.wikipedia.org/wiki/Prime\\_number](http://en.wikipedia.org/wiki/Prime_number)
- [R33] <http://en.wikipedia.org/wiki/Finite>
- [R34] <https://docs.python.org/3/library/math.html#math.isfinite>
- [R35] <http://docs.scipy.org/doc/numpy/reference/generated/numpy.isfinite.html>
- [R36] A New Algorithm for Computing Asymptotic Series - Dominik Gruntz
- [R37] Gruntz thesis - p90
- [R38] [http://en.wikipedia.org/wiki/Asymptotic\\_expansion](http://en.wikipedia.org/wiki/Asymptotic_expansion)
- [R39] <http://en.wikipedia.org/wiki/Zero>
- [R40] [http://en.wikipedia.org/wiki/1\\_%28number%29](http://en.wikipedia.org/wiki/1_%28number%29)
- [R41] [http://en.wikipedia.org/wiki/%E2%88%921\\_%28number%29](http://en.wikipedia.org/wiki/%E2%88%921_%28number%29)
- [R42] [http://en.wikipedia.org/wiki/One\\_half](http://en.wikipedia.org/wiki/One_half)
- [R43] <http://en.wikipedia.org/wiki/NaN>
- [R44] <http://en.wikipedia.org/wiki/Infinity>
- [R45] [http://en.wikipedia.org/wiki/E\\_%28mathematical\\_constant%29](http://en.wikipedia.org/wiki/E_%28mathematical_constant%29)
- [R46] [http://en.wikipedia.org/wiki/Imaginary\\_unit](http://en.wikipedia.org/wiki/Imaginary_unit)
- [R47] <http://en.wikipedia.org/wiki/Pi>
- [R48] [http://en.wikipedia.org/wiki/Euler%20%93Mascheroni\\_constant](http://en.wikipedia.org/wiki/Euler%20%93Mascheroni_constant)
- [R49] [http://en.wikipedia.org/wiki/Catalan%27s\\_constant](http://en.wikipedia.org/wiki/Catalan%27s_constant)
- [R50] [http://en.wikipedia.org/wiki/Golden\\_ratio](http://en.wikipedia.org/wiki/Golden_ratio)
- [R51] <http://en.wikipedia.org/wiki/Exponentiation>

[R52] [http://en.wikipedia.org/wiki/Exponentiation#Zero\\_to\\_the\\_power\\_of\\_zero](http://en.wikipedia.org/wiki/Exponentiation#Zero_to_the_power_of_zero)

[R53] [http://en.wikipedia.org/wiki/Indeterminate\\_forms](http://en.wikipedia.org/wiki/Indeterminate_forms)

[R54] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The `and` operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being `Symbols`, Python converts the statement (roughly) into these steps:

1.  $x > y > z$
2.  $(x > y)$  and  $(y > z)$
3. `(GreaterThanObject)` and  $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and  $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like  $x > y > z$  work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R55] For more information, see these two bug reports:

“Separate boolean and symbolic relational” [Issue 4986](#)

“It right  $0 < x < 1$ ?” [Issue 6059](#)

[R56] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The `and` operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being `Symbols`, Python converts the statement (roughly) into these steps:

1.  $x > y > z$
2.  $(x > y)$  and  $(y > z)$
3. `(GreaterThanObject)` and  $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and  $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like  $x > y > z$  work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R57] For more information, see these two bug reports:

- “Separate boolean and symbolic relational” [Issue 4986](#)
- “It right  $0 < x < 1$ ?” [Issue 6059](#)

[R58] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The `and` operator coerces each side into a `bool`, returning the object itself when it short-circuits. The `bool` of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being `Symbols`, Python converts the statement (roughly) into these steps:

1.  $x > y > z$
2.  $(x > y)$  and  $(y > z)$
3. `(GreaterThanObject)` and  $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and  $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like  $x > y > z$  work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R59] For more information, see these two bug reports:

- “Separate boolean and symbolic relational” [Issue 4986](#)
- “It right  $0 < x < 1$ ?” [Issue 6059](#)

[R60] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <http://docs.python.org/2/reference/expressions.html#notin>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example,  $1 > 2 > 3$  is evaluated by Python as  $(1 > 2)$  and  $(2 > 3)$ . The `and` operator coerces each side into a `bool`, returning the object itself when it short-circuits. The `bool` of the `-Than` operators will raise `TypeError` on purpose, because SymPy cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute  $x > y > z$ , with  $x$ ,  $y$ , and  $z$  being `Symbols`, Python converts the statement (roughly) into these steps:

1.  $x > y > z$
2.  $(x > y)$  and  $(y > z)$
3. `(GreaterThanObject)` and  $(y > z)$
4. `(GreaterThanObject.__nonzero__())` and  $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__nonzero__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like `x > y > z` work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R61] For more information, see these two bug reports:

“Separate boolean and symbolic relational” [Issue 4986](#)

“It right  $0 < x < 1$ ?” [Issue 6059](#)

[R4] [http://en.wikipedia.org/wiki/Partition\\_%28number\\_theory%29](http://en.wikipedia.org/wiki/Partition_%28number_theory%29)

[R8] Skiena, S. ‘Permutations.’ 1.1 in Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 3-16, 1990.

[R9] Knuth, D. E. The Art of Computer Programming, Vol. 4: Combinatorial Algorithms, 1st ed. Reading, MA: Addison-Wesley, 2011.

[R10] Wendy Myrvold and Frank Ruskey. 2001. Ranking and unranking permutations in linear time. Inf. Process. Lett. 79, 6 (September 2001), 281-284. DOI=10.1016/S0020-0190(01)00141-7

[R11] D. L. Kreher, D. R. Stinson ‘Combinatorial Algorithms’ CRC Press, 1999

[R12] Graham, R. L.; Knuth, D. E.; and Patashnik, O. Concrete Mathematics: A Foundation for Computer Science, 2nd ed. Reading, MA: Addison-Wesley, 1994.

[R13] [http://en.wikipedia.org/wiki/Permutation#Product\\_and\\_inverse](http://en.wikipedia.org/wiki/Permutation#Product_and_inverse)

[R14] [http://en.wikipedia.org/wiki/Lehmer\\_code](http://en.wikipedia.org/wiki/Lehmer_code)

[R15] <http://mathworld.wolfram.com/LabeledTree.html>

[R16] Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

[R336] [http://en.wikipedia.org/wiki/Square-free\\_integer#Squarefree\\_core](http://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core)

[R337] [http://en.wikipedia.org/wiki/Continued\\_fraction](http://en.wikipedia.org/wiki/Continued_fraction)

[R338] [http://en.wikipedia.org/wiki/Periodic\\_continued\\_fraction](http://en.wikipedia.org/wiki/Periodic_continued_fraction)

[R339] K. Rosen. Elementary Number theory and its applications. Addison-Wesley, 3 Sub edition, pages 379-381, January 1992.

[R340] [http://en.wikipedia.org/wiki/M%C3%B6bius\\_function](http://en.wikipedia.org/wiki/M%C3%B6bius_function)

[R341] Thomas Koshy “Elementary Number Theory with Applications”

[R342] [http://en.wikipedia.org/wiki/Egyptian\\_fraction](http://en.wikipedia.org/wiki/Egyptian_fraction)

[R343] [https://en.wikipedia.org/wiki/Greedy\\_algorithm\\_for\\_Egyptian\\_fractions](https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions)

[R344] <http://www.ics.uci.edu/~eppstein/numth/egypt/conflict.html>

[R345] [http://ami.ektf.hu/uploads/papers/finalpdf/AMI\\_42\\_from129to134.pdf](http://ami.ektf.hu/uploads/papers/finalpdf/AMI_42_from129to134.pdf)

[R62] [http://en.wikipedia.org/wiki/Vigenere\\_cipher](http://en.wikipedia.org/wiki/Vigenere_cipher)

[R63] [en.wikipedia.org/wiki/Hill\\_cipher](http://en.wikipedia.org/wiki/Hill_cipher)

[R64] Lester S. Hill, Cryptography in an Algebraic Alphabet, The American Mathematical Monthly Vol.36, June-July 1929, pp.306-312.

[R65] [http://en.wikipedia.org/wiki/Morse\\_code](http://en.wikipedia.org/wiki/Morse_code)

[R66] [http://en.wikipedia.org/wiki/Morse\\_code](http://en.wikipedia.org/wiki/Morse_code)

[G66] Solomon Golomb, Shift register sequences, Aegean Park Press, Laguna Hills, Ca, 1967

- [M67] James L. Massey, “Shift-Register Synthesis and BCH Decoding.” IEEE Trans. on Information Theory, vol. 15(1), pp. 122-127, Jan 1969.
- [R17] Michael Karr, “Summation in Finite Terms”, Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R18] [http://en.wikipedia.org/wiki/Summation#Capital-sigma\\_notation](http://en.wikipedia.org/wiki/Summation#Capital-sigma_notation)
- [R19] [http://en.wikipedia.org/wiki/Empty\\_sum](http://en.wikipedia.org/wiki/Empty_sum)
- [R20] 13. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996, Ch. 4.
- [R21] Michael Karr, “Summation in Finite Terms”, Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R22] Michael Karr, “Summation in Finite Terms”, Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R23] [http://en.wikipedia.org/wiki/Multiplication#Capital\\_Pi\\_notation](http://en.wikipedia.org/wiki/Multiplication#Capital_Pi_notation)
- [R24] [http://en.wikipedia.org/wiki/Empty\\_product](http://en.wikipedia.org/wiki/Empty_product)
- [R25] Michael Karr, “Summation in Finite Terms”, Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R26] Marko Petkovsek, Herbert S. Wilf, Doron Zeilberger, A = B, AK Peters, Ltd., Wellesley, MA, USA, 1997, pp. 73–100
- [R100] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R101] <http://dlmf.nist.gov/4.23>
- [R102] <http://functions.wolfram.com/ElementaryFunctions/ArcCos>
- [R103] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R104] <http://dlmf.nist.gov/4.23>
- [R105] <http://functions.wolfram.com/ElementaryFunctions/ArcCot>
- [R106] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R107] <http://dlmf.nist.gov/4.23>
- [R108] <http://functions.wolfram.com/ElementaryFunctions/ArcCsc>
- [R109] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R110] <http://dlmf.nist.gov/4.23>
- [R111] <http://functions.wolfram.com/ElementaryFunctions/ArcSin>
- [R112] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R113] <http://dlmf.nist.gov/4.23>
- [R114] <http://functions.wolfram.com/ElementaryFunctions/ArcSec>
- [R115] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R116] <http://dlmf.nist.gov/4.23>
- [R117] <http://functions.wolfram.com/ElementaryFunctions/ArcTan>
- [R118] [http://en.wikipedia.org/wiki/Inverse\\_trigonometric\\_functions](http://en.wikipedia.org/wiki/Inverse_trigonometric_functions)
- [R119] <http://en.wikipedia.org/wiki/Atan2>
- [R120] <http://functions.wolfram.com/ElementaryFunctions/ArcTan2>

- [R121] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R122] <http://dlmf.nist.gov/4.14>
- [R123] <http://functions.wolfram.com/ElementaryFunctions/Cos>
- [R124] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R125] <http://dlmf.nist.gov/4.14>
- [R126] <http://functions.wolfram.com/ElementaryFunctions/Cot>
- [R127] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R128] <http://dlmf.nist.gov/4.14>
- [R129] <http://functions.wolfram.com/ElementaryFunctions/Csc>
- [R130] [http://en.wikipedia.org/wiki/Lambert\\_W\\_function](http://en.wikipedia.org/wiki/Lambert_W_function)
- [R131] [http://en.wikipedia.org/wiki/Directed\\_complete\\_partial\\_order](http://en.wikipedia.org/wiki/Directed_complete_partial_order)
- [R132] [http://en.wikipedia.org/wiki/Lattice\\_%28order%29](http://en.wikipedia.org/wiki/Lattice_%28order%29)
- [R133] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R134] <http://dlmf.nist.gov/4.14>
- [R135] <http://functions.wolfram.com/ElementaryFunctions/Sin>
- [R136] <http://mathworld.wolfram.com/TrigonometryAngles.html>
- [R137] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R138] <http://dlmf.nist.gov/4.14>
- [R139] <http://functions.wolfram.com/ElementaryFunctions/Sec>
- [R140] [http://en.wikipedia.org/wiki/Trigonometric\\_functions](http://en.wikipedia.org/wiki/Trigonometric_functions)
- [R141] <http://dlmf.nist.gov/4.14>
- [R142] <http://functions.wolfram.com/ElementaryFunctions/Tan>
- [R69] [http://en.wikipedia.org/wiki/Bell\\_number](http://en.wikipedia.org/wiki/Bell_number)
- [R70] <http://mathworld.wolfram.com/BellNumber.html>
- [R71] <http://mathworld.wolfram.com/BellPolynomial.html>
- [R72] [http://en.wikipedia.org/wiki/Bernoulli\\_number](http://en.wikipedia.org/wiki/Bernoulli_number)
- [R73] [http://en.wikipedia.org/wiki/Bernoulli\\_polynomial](http://en.wikipedia.org/wiki/Bernoulli_polynomial)
- [R74] <http://mathworld.wolfram.com/BernoulliNumber.html>
- [R75] <http://mathworld.wolfram.com/BernoulliPolynomial.html>
- [R76] [http://en.wikipedia.org/wiki/Catalan\\_number](http://en.wikipedia.org/wiki/Catalan_number)
- [R77] <http://mathworld.wolfram.com/CatalanNumber.html>
- [R78] <http://functions.wolfram.com/GammaBetaErf/CatalanNumber/>
- [R79] <http://geometer.org/mathcircles/catalan.pdf>
- [R80] [http://en.wikipedia.org/wiki/Euler\\_numbers](http://en.wikipedia.org/wiki/Euler_numbers)
- [R81] <http://mathworld.wolfram.com/EulerNumber.html>
- [R82] [http://en.wikipedia.org/wiki/Alternating\\_permutation](http://en.wikipedia.org/wiki/Alternating_permutation)

- [R83] <http://mathworld.wolfram.com/AlternatingPermutation.html>
- [R84] <http://en.wikipedia.org/wiki/Subfactorial>
- [R85] <http://mathworld.wolfram.com/Subfactorial.html>
- [R86] [https://en.wikipedia.org/wiki/Double\\_factorial](https://en.wikipedia.org/wiki/Double_factorial)
- [R87] [http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)
- [R88] <http://mathworld.wolfram.com/FibonacciNumber.html>
- [R89] [http://en.wikipedia.org/wiki/Harmonic\\_number](http://en.wikipedia.org/wiki/Harmonic_number)
- [R90] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/>
- [R91] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/>
- [R92] [http://en.wikipedia.org/wiki/Lucas\\_number](http://en.wikipedia.org/wiki/Lucas_number)
- [R93] <http://mathworld.wolfram.com/LucasNumber.html>
- [R94] [http://en.wikipedia.org/wiki/Stirling\\_numbers\\_of\\_the\\_first\\_kind](http://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind)
- [R95] [http://en.wikipedia.org/wiki/Stirling\\_numbers\\_of\\_the\\_second\\_kind](http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind)
- [R96] <http://en.wikipedia.org/wiki/Combination>
- [R97] <http://tinyurl.com/cep849r>
- [R98] <http://en.wikipedia.org/wiki/Permutation>
- [R99] <http://undergraduate.csse.uwa.edu.au/units/CITS7209/partition.pdf>
- [R143] <http://mathworld.wolfram.com/DeltaFunction.html>
- [R144] <http://mathworld.wolfram.com/HeavisideStepFunction.html>
- [R145] [http://en.wikipedia.org/wiki/Gamma\\_function](http://en.wikipedia.org/wiki/Gamma_function)
- [R146] <http://dlmf.nist.gov/5>
- [R147] <http://mathworld.wolfram.com/GammaFunction.html>
- [R148] <http://functions.wolfram.com/GammaBetaErf/Gamma/>
- [R149] [http://en.wikipedia.org/wiki/Gamma\\_function](http://en.wikipedia.org/wiki/Gamma_function)
- [R150] <http://dlmf.nist.gov/5>
- [R151] <http://mathworld.wolfram.com/LogGammaFunction.html>
- [R152] <http://functions.wolfram.com/GammaBetaErf/LogGamma/>
- [R153] [http://en.wikipedia.org/wiki/Polygamma\\_function](http://en.wikipedia.org/wiki/Polygamma_function)
- [R154] <http://mathworld.wolfram.com/PolygammaFunction.html>
- [R155] <http://functions.wolfram.com/GammaBetaErf/PolyGamma/>
- [R156] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R157] [http://en.wikipedia.org/wiki/Digamma\\_function](http://en.wikipedia.org/wiki/Digamma_function)
- [R158] <http://mathworld.wolfram.com/DigammaFunction.html>
- [R159] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R160] [http://en.wikipedia.org/wiki/Trigamma\\_function](http://en.wikipedia.org/wiki/Trigamma_function)
- [R161] <http://mathworld.wolfram.com/TrigammaFunction.html>

- [R162] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R163] [http://en.wikipedia.org/wiki/Incomplete\\_gamma\\_function#Upper\\_Incomplete\\_Gamma\\_Function](http://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_Incomplete_Gamma_Function)
- [R164] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R165] <http://dlmf.nist.gov/8>
- [R166] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R167] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R168] [http://en.wikipedia.org/wiki/Exponential\\_integral#Relation\\_with\\_other\\_functions](http://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions)
- [R169] [http://en.wikipedia.org/wiki/Incomplete\\_gamma\\_function#Lower\\_Incomplete\\_Gamma\\_Function](http://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_Incomplete_Gamma_Function)
- [R170] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R171] <http://dlmf.nist.gov/8>
- [R172] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R173] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R174] [http://en.wikipedia.org/wiki/Beta\\_function](http://en.wikipedia.org/wiki/Beta_function)
- [R175] <http://mathworld.wolfram.com/BetaFunction.html>
- [R176] <http://dlmf.nist.gov/5.12>
- [R177] [http://en.wikipedia.org/wiki/Error\\_function](http://en.wikipedia.org/wiki/Error_function)
- [R178] <http://dlmf.nist.gov/7>
- [R179] <http://mathworld.wolfram.com/Erf.html>
- [R180] <http://functions.wolfram.com/GammaBetaErf/Erf>
- [R181] [http://en.wikipedia.org/wiki/Error\\_function](http://en.wikipedia.org/wiki/Error_function)
- [R182] <http://dlmf.nist.gov/7>
- [R183] <http://mathworld.wolfram.com/Erfc.html>
- [R184] <http://functions.wolfram.com/GammaBetaErf/Erfc>
- [R185] [http://en.wikipedia.org/wiki/Error\\_function](http://en.wikipedia.org/wiki/Error_function)
- [R186] <http://mathworld.wolfram.com/Erfi.html>
- [R187] <http://functions.wolfram.com/GammaBetaErf/Erfi>
- [R188] <http://functions.wolfram.com/GammaBetaErf/Erf2/>
- [R189] [http://en.wikipedia.org/wiki/Error\\_function#Inverse\\_functions](http://en.wikipedia.org/wiki/Error_function#Inverse_functions)
- [R190] <http://functions.wolfram.com/GammaBetaErf/InverseErf/>
- [R191] [http://en.wikipedia.org/wiki/Error\\_function#Inverse\\_functions](http://en.wikipedia.org/wiki/Error_function#Inverse_functions)
- [R192] <http://functions.wolfram.com/GammaBetaErf/InverseErfc/>
- [R193] <http://functions.wolfram.com/GammaBetaErf/InverseErf2/>
- [R194] [http://en.wikipedia.org/wiki/Fresnel\\_integral](http://en.wikipedia.org/wiki/Fresnel_integral)
- [R195] <http://dlmf.nist.gov/7>
- [R196] <http://mathworld.wolfram.com/FresnelIntegrals.html>

- [R197] <http://functions.wolfram.com/GammaBetaErf/FresnelS>
- [R198] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R199] [http://en.wikipedia.org/wiki/Fresnel\\_integral](http://en.wikipedia.org/wiki/Fresnel_integral)
- [R200] <http://dlmf.nist.gov/7>
- [R201] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R202] <http://functions.wolfram.com/GammaBetaErf/FresnelC>
- [R203] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R204] <http://dlmf.nist.gov/6.6>
- [R205] [http://en.wikipedia.org/wiki/Exponential\\_integral](http://en.wikipedia.org/wiki/Exponential_integral)
- [R206] Abramowitz & Stegun, section 5: [http://people.math.sfu.ca/~cbm/aands/page\\_228.htm](http://people.math.sfu.ca/~cbm/aands/page_228.htm)
- [R207] <http://dlmf.nist.gov/8.19>
- [R208] <http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/>
- [R209] [http://en.wikipedia.org/wiki/Exponential\\_integral](http://en.wikipedia.org/wiki/Exponential_integral)
- [R210] [http://en.wikipedia.org/wiki/Logarithmic\\_integral](http://en.wikipedia.org/wiki/Logarithmic_integral)
- [R211] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R212] <http://dlmf.nist.gov/6>
- [R213] <http://mathworld.wolfram.com/SoldnersConstant.html>
- [R214] [http://en.wikipedia.org/wiki/Logarithmic\\_integral](http://en.wikipedia.org/wiki/Logarithmic_integral)
- [R215] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R216] <http://dlmf.nist.gov/6>
- [R217] [http://en.wikipedia.org/wiki/Trigonometric\\_integral](http://en.wikipedia.org/wiki/Trigonometric_integral)
- [R218] [http://en.wikipedia.org/wiki/Trigonometric\\_integral](http://en.wikipedia.org/wiki/Trigonometric_integral)
- [R219] [http://en.wikipedia.org/wiki/Trigonometric\\_integral](http://en.wikipedia.org/wiki/Trigonometric_integral)
- [R220] [http://en.wikipedia.org/wiki/Trigonometric\\_integral](http://en.wikipedia.org/wiki/Trigonometric_integral)
- [R221] Abramowitz, Milton; Stegun, Irene A., eds. (1965), “Chapter 9”, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R222] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R223] [http://en.wikipedia.org/wiki/Bessel\\_function](http://en.wikipedia.org/wiki/Bessel_function)
- [R224] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/>
- [R225] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/>
- [R226] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/>
- [R227] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/>
- [R228] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/>
- [R229] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/>
- [R230] [http://en.wikipedia.org/wiki/Airy\\_function](http://en.wikipedia.org/wiki/Airy_function)
- [R231] <http://dlmf.nist.gov/9>

- [R232] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R233] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R234] [http://en.wikipedia.org/wiki/Airy\\_function](http://en.wikipedia.org/wiki/Airy_function)
- [R235] <http://dlmf.nist.gov/9>
- [R236] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R237] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R238] [http://en.wikipedia.org/wiki/Airy\\_function](http://en.wikipedia.org/wiki/Airy_function)
- [R239] <http://dlmf.nist.gov/9>
- [R240] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R241] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R242] [http://en.wikipedia.org/wiki/Airy\\_function](http://en.wikipedia.org/wiki/Airy_function)
- [R243] <http://dlmf.nist.gov/9>
- [R244] [http://www.encyclopediaofmath.org/index.php/Airy\\_functions](http://www.encyclopediaofmath.org/index.php/Airy_functions)
- [R245] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R246] <http://en.wikipedia.org/wiki/B-spline>
- [R247] <http://dlmf.nist.gov/25.11>
- [R248] [http://en.wikipedia.org/wiki/Hurwitz\\_zeta\\_function](http://en.wikipedia.org/wiki/Hurwitz_zeta_function)
- [R249] [http://en.wikipedia.org/wiki/Dirichlet\\_eta\\_function](http://en.wikipedia.org/wiki/Dirichlet_eta_function)
- [R250] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill.  
Section 1.11.
- [R251] <http://dlmf.nist.gov/25.14>
- [R252] [http://en.wikipedia.org/wiki/Lerch\\_transcendent](http://en.wikipedia.org/wiki/Lerch_transcendent)
- [R253] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R254] [http://en.wikipedia.org/wiki/Generalized\\_hypergeometric\\_function](http://en.wikipedia.org/wiki/Generalized_hypergeometric_function)
- [R255] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R256] [http://en.wikipedia.org/wiki/Meijer\\_G-function](http://en.wikipedia.org/wiki/Meijer_G-function)
- [R257] [http://en.wikipedia.org/wiki/Elliptic\\_integrals](http://en.wikipedia.org/wiki/Elliptic_integrals)
- [R258] <http://functions.wolfram.com/EllipticIntegrals/EllipticK>
- [R259] [http://en.wikipedia.org/wiki/Elliptic\\_integrals](http://en.wikipedia.org/wiki/Elliptic_integrals)
- [R260] <http://functions.wolfram.com/EllipticIntegrals/EllipticF>
- [R261] [http://en.wikipedia.org/wiki/Elliptic\\_integrals](http://en.wikipedia.org/wiki/Elliptic_integrals)
- [R262] <http://functions.wolfram.com/EllipticIntegrals/EllipticE2>
- [R263] <http://functions.wolfram.com/EllipticIntegrals/EllipticE>
- [R264] [http://en.wikipedia.org/wiki/Elliptic\\_integrals](http://en.wikipedia.org/wiki/Elliptic_integrals)
- [R265] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi3>
- [R266] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi>

- [R267] [http://en.wikipedia.org/wiki/Jacobi\\_polynomials](http://en.wikipedia.org/wiki/Jacobi_polynomials)
- [R268] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R269] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R270] [http://en.wikipedia.org/wiki/Jacobi\\_polynomials](http://en.wikipedia.org/wiki/Jacobi_polynomials)
- [R271] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R272] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R273] [http://en.wikipedia.org/wiki/Gegenbauer\\_polynomials](http://en.wikipedia.org/wiki/Gegenbauer_polynomials)
- [R274] <http://mathworld.wolfram.com/GegenbauerPolynomial.html>
- [R275] <http://functions.wolfram.com/Polynomials/GegenbauerC3/>
- [R276] [http://en.wikipedia.org/wiki/Chebyshev\\_polynomial](http://en.wikipedia.org/wiki/Chebyshev_polynomial)
- [R277] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R278] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R279] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R280] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R281] [http://en.wikipedia.org/wiki/Chebyshev\\_polynomial](http://en.wikipedia.org/wiki/Chebyshev_polynomial)
- [R282] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R283] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R284] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R285] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R286] [http://en.wikipedia.org/wiki/Legendre\\_polynomial](http://en.wikipedia.org/wiki/Legendre_polynomial)
- [R287] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R288] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R289] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R290] [http://en.wikipedia.org/wiki/Associated\\_Legendre\\_polynomials](http://en.wikipedia.org/wiki/Associated_Legendre_polynomials)
- [R291] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R292] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R293] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R294] [http://en.wikipedia.org/wiki/Hermite\\_polynomial](http://en.wikipedia.org/wiki/Hermite_polynomial)
- [R295] <http://mathworld.wolfram.com/HermitePolynomial.html>
- [R296] <http://functions.wolfram.com/Polynomials/HermiteH/>
- [R297] [http://en.wikipedia.org/wiki/Laguerre\\_polynomial](http://en.wikipedia.org/wiki/Laguerre_polynomial)
- [R298] <http://mathworld.wolfram.com/LaguerrePolynomial.html>
- [R299] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R300] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R301] [http://en.wikipedia.org/wiki/Laguerre\\_polynomial#Assoc\\_laguerre\\_polynomials](http://en.wikipedia.org/wiki/Laguerre_polynomial#Assoc_laguerre_polynomials)
- [R302] <http://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html>

- [R303] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R304] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R305] [http://en.wikipedia.org/wiki/Spherical\\_harmonics](http://en.wikipedia.org/wiki/Spherical_harmonics)
- [R306] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R307] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R308] <http://dlmf.nist.gov/14.30>
- [R309] [http://en.wikipedia.org/wiki/Spherical\\_harmonics](http://en.wikipedia.org/wiki/Spherical_harmonics)
- [R310] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R311] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R312] [http://en.wikipedia.org/wiki/Spherical\\_harmonics](http://en.wikipedia.org/wiki/Spherical_harmonics)
- [R313] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R314] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R315] [http://en.wikipedia.org/wiki/Kronecker\\_delta](http://en.wikipedia.org/wiki/Kronecker_delta)
- [WikiPappus] “Pappus’s Hexagon Theorem” Wikipedia, the Free Encyclopedia. Web. 26 Apr. 2013.  
[<http://en.wikipedia.org/wiki/Pappus%27s\\_hexagon\\_theorem>](http://en.wikipedia.org/wiki/Pappus%27s_hexagon_theorem)
- [BlogPost] <http://nessgrh.wordpress.com/2011/07/07/tricky-branch-cuts/>
- [Bro05] M. Bronstein, Symbolic Integration I: Transcendental Functions, Second Edition, Springer-Verlag, 2005, pp. 35-70
- [R316] [http://en.wikipedia.org/wiki/Gaussian\\_quadrature](http://en.wikipedia.org/wiki/Gaussian_quadrature)
- [R317] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/legendre\\_rule/legendre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/legendre_rule/legendre_rule.html)
- [R318] [http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre\\_quadrature](http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature)
- [R319] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/laguerre\\_rule/laguerre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.html)
- [R320] [http://en.wikipedia.org/wiki/Gauss-Hermite\\_Quadrature](http://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature)
- [R321] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/hermite\\_rule/hermite\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/hermite_rule/hermite_rule.html)
- [R322] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gen\\_hermite\\_rule/gen\\_hermite\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_hermite_rule/gen_hermite_rule.html)
- [R323] [http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre\\_quadrature](http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature)
- [R324] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gen\\_laguerre\\_rule/gen\\_laguerre\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_laguerre_rule/gen_laguerre_rule.html)
- [R325] [http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss\\_quadrature](http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature)
- [R326] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/chebyshev1\\_rule/chebyshev1\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev1_rule/chebyshev1_rule.html)
- [R327] [http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss\\_quadrature](http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature)
- [R328] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/chebyshev2\\_rule/chebyshev2\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev2_rule/chebyshev2_rule.html)
- [R329] [http://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi\\_quadrature](http://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature)
- [R330] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/jacobi\\_rule/jacobi\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_rule/jacobi_rule.html)
- [R331] [http://people.sc.fsu.edu/~jburkardt/cpp\\_src/gegenbauer\\_rule/gegenbauer\\_rule.html](http://people.sc.fsu.edu/~jburkardt/cpp_src/gegenbauer_rule/gegenbauer_rule.html)
- [R332] [en.wikipedia.org/wiki/Quine-McCluskey\\_algorithm](http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm)
- [R333] [en.wikipedia.org/wiki/Quine-McCluskey\\_algorithm](http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm)
- [R334] [https://en.wikipedia.org/wiki/Moore-Penrose\\_pseudoinverse](https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse)

- [R335] [https://en.wikipedia.org/wiki/Moore-Penrose\\_pseudoinverse#Obtaining\\_all\\_solutions\\_of\\_a\\_linear\\_system](https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system)
- [Wester1999] Michael J. Wester, A Critique of the Mathematical Abilities of CA Systems, 1999, <http://www.math.unm.edu/~wester/cas/book/Wester.pdf>
- [Kozen89] D. Kozen, S. Landau, Polynomial decomposition algorithms, Journal of Symbolic Computation 7 (1989), pp. 445–456
- [Liao95] Hsin-Chao Liao, R. Fateman, Evaluation of the heuristic polynomial GCD, International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, Montreal, Quebec, Canada, 1995, pp. 240–247
- [Gathen99] J. von zur Gathen, J. Gerhard, Modern Computer Algebra, First Edition, Cambridge University Press, 1999
- [Weisstein09] Eric W. Weisstein, Cyclotomic Polynomial, From MathWorld - A Wolfram Web Resource, <http://mathworld.wolfram.com/CyclotomicPolynomial.html>
- [Wang78] P. S. Wang, An Improved Multivariate Polynomial Factoring Algorithm, Math. of Computation 32, 1978, pp. 1215–1231
- [Geddes92] K. Geddes, S. R. Czapor, G. Labahn, Algorithms for Computer Algebra, Springer, 1992
- [Monagan93] Michael Monagan, In-place Arithmetic for Polynomials over  $Z_n$ , Proceedings of DISCO ‘92, Springer-Verlag LNCS, 721, 1993, pp. 22–34
- [Kaltofen98] E. Kaltofen, V. Shoup, Subquadratic-time Factoring of Polynomials over Finite Fields, Mathematics of Computation, Volume 67, Issue 223, 1998, pp. 1179–1197
- [Shoup95] V. Shoup, A New Polynomial Factorization Algorithm and its Implementation, Journal of Symbolic Computation, Volume 20, Issue 4, 1995, pp. 363–397
- [Gathen92] J. von zur Gathen, V. Shoup, Computing Frobenius Maps and Factoring Polynomials, ACM Symposium on Theory of Computing, 1992, pp. 187–224
- [Shoup91] V. Shoup, A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic, In Proceedings of International Symposium on Symbolic and Algebraic Computation, 1991, pp. 14–21
- [Cox97] D. Cox, J. Little, D. O’Shea, Ideals, Varieties and Algorithms, Springer, Second Edition, 1997
- [Ajwa95] I.A. Ajwa, Z. Liu, P.S. Wang, Groebner Bases Algorithm, <https://citeseer.ist.psu.edu/myciteseer/login>, 1995
- [Bose03] N.K. Bose, B. Buchberger, J.P. Guiver, Multidimensional Systems Theory and Applications, Springer, 2003
- [Giovini91] A. Giovini, T. Mora, “One sugar cube, please” or Selection strategies in Buchberger algorithm, ISSAC ‘91, ACM
- [Bronstein93] M. Bronstein, B. Salvy, Full partial fraction decomposition of rational functions, Proceedings ISSAC ‘93, ACM Press, Kiev, Ukraine, 1993, pp. 157–160
- [Buchberger01] B. Buchberger, Groebner Bases: A Short Introduction for Systems Theorists, In: R. Moreno-Diaz, B. Buchberger, J. L. Freire, Proceedings of EUROCAST‘01, February, 2001
- [Davenport88] J.H. Davenport, Y. Siret, E. Tournier, Computer Algebra Systems and Algorithms for Algebraic Computation, Academic Press, London, 1988, pp. 124–128
- [Greuel2008] G.-M. Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer, 2008
- [Atiyah69] M.F. Atiyah, I.G. MacDonald, Introduction to Commutative Algebra, Addison-Wesley, 1969

- [Collins67] G.E. Collins, Subresultants and Reduced Polynomial Remainder Sequences. *J. ACM* 14 (1967) 128-142
- [BrownTraub71] W.S. Brown, J.F. Traub, On Euclid's Algorithm and the Theory of Subresultants. *J. ACM* 18 (1971) 505-514
- [Brown78] W.S. Brown, The Subresultant PRS Algorithm. *ACM Transaction of Mathematical Software* 4 (1978) 237-249
- [Monagan00] M. Monagan and A. Wittkopf, On the Design and Implementation of Browns Algorithm over the Integers and Number Fields, Proceedings of ISSAC 2000, pp. 225-233, ACM, 2000.
- [Brown71] W.S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, *J. ACM* 18, 4, pp. 478-504, 1971.
- [Hoeij04] M. van Hoeij and M. Monagan, Algorithms for polynomial GCD computation over algebraic function fields, Proceedings of ISSAC 2004, pp. 297-304, ACM, 2004.
- [Wang81] P.S. Wang, A p-adic algorithm for univariate partial fractions, Proceedings of SYMSAC 1981, pp. 212-217, ACM, 1981.
- [Hoeij02] M. van Hoeij and M. Monagan, A modular GCD algorithm over number fields presented with multiple extensions, Proceedings of ISSAC 2002, pp. 109-116, ACM, 2002
- [ManWright94] Yiu-Kwong Man and Francis J. Wright, "Fast Polynomial Dispersion Computation and its Application to Indefinite Summation", Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1994, Pages 175-180 <http://dl.acm.org/citation.cfm?doid=190347.190413>
- [Koepf98] Wolfram Koepf, "Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities", Advanced lectures in mathematics, Vieweg, 1998
- [Abramov71] S. A. Abramov, "On the Summation of Rational Functions", USSR Computational Mathematics and Mathematical Physics, Volume 11, Issue 4, 1971, Pages 324-330
- [Man93] Yiu-Kwong Man, "On Computing Closed Forms for Indefinite Summations", Journal of Symbolic Computation, Volume 16, Issue 4, 1993, Pages 355-376 <http://www.sciencedirect.com/science/article/pii/S0747717183710539>
- [R346] Big O notation
- [R347] [http://en.wikipedia.org/wiki/Residue\\_%28complex\\_analysis%29](http://en.wikipedia.org/wiki/Residue_%28complex_analysis%29)
- [R348] [http://en.wikipedia.org/wiki/Residue\\_theorem](http://en.wikipedia.org/wiki/Residue_theorem)
- [R349] Gruntz Thesis
- [R350] [http://en.wikipedia.org/wiki/Disjoint\\_sets](http://en.wikipedia.org/wiki/Disjoint_sets)
- [R351] [http://en.wikipedia.org/wiki/Power\\_set](http://en.wikipedia.org/wiki/Power_set)
- [R352] [http://en.wikipedia.org/wiki/Interval\\_%28mathematics%29](http://en.wikipedia.org/wiki/Interval_%28mathematics%29)
- [R353] [http://en.wikipedia.org/wiki/Finite\\_set](http://en.wikipedia.org/wiki/Finite_set)
- [R354] [http://en.wikipedia.org/wiki/Union\\_%28set\\_theory%29](http://en.wikipedia.org/wiki/Union_%28set_theory%29)
- [R355] [http://en.wikipedia.org/wiki/Intersection\\_%28set\\_theory%29](http://en.wikipedia.org/wiki/Intersection_%28set_theory%29)
- [R356] [http://en.wikipedia.org/wiki/Cartesian\\_product](http://en.wikipedia.org/wiki/Cartesian_product)
- [R357] [http://en.wikipedia.org/wiki/Empty\\_set](http://en.wikipedia.org/wiki/Empty_set)
- [R358] [http://en.wikipedia.org/wiki/Universal\\_set](http://en.wikipedia.org/wiki/Universal_set)
- [Roach1996] Kelly B. Roach. Hypergeometric Function Representations. In: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, pages 301-308, New York, 1996. ACM.

- [Roach1997] Kelly B. Roach. Meijer G Function Representations. In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.
- [Luke1969] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1.
- [Prudnikov1990] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990). Integrals and Series: More Special Functions, Vol. 3, Gordon and Breach Science Publisher.
- [R383] [http://en.wikipedia.org/wiki/Arcsine\\_distribution](http://en.wikipedia.org/wiki/Arcsine_distribution)
- [R384] [http://en.wikipedia.org/wiki/Benini\\_distribution](http://en.wikipedia.org/wiki/Benini_distribution)
- [R385] <http://reference.wolfram.com/legacy/v8/ref/BeniniDistribution.html>
- [R386] [http://en.wikipedia.org/wiki/Beta\\_distribution](http://en.wikipedia.org/wiki/Beta_distribution)
- [R387] <http://mathworld.wolfram.com/BetaDistribution.html>
- [R388] [http://en.wikipedia.org/wiki/Beta\\_prime\\_distribution](http://en.wikipedia.org/wiki/Beta_prime_distribution)
- [R389] <http://mathworld.wolfram.com/BetaPrimeDistribution.html>
- [R390] [http://en.wikipedia.org/wiki/Cauchy\\_distribution](http://en.wikipedia.org/wiki/Cauchy_distribution)
- [R391] <http://mathworld.wolfram.com/CauchyDistribution.html>
- [R392] [http://en.wikipedia.org/wiki/Chi\\_distribution](http://en.wikipedia.org/wiki/Chi_distribution)
- [R393] <http://mathworld.wolfram.com/ChiDistribution.html>
- [R394] [http://en.wikipedia.org/wiki/Noncentral\\_chi\\_distribution](http://en.wikipedia.org/wiki/Noncentral_chi_distribution)
- [R395] [http://en.wikipedia.org/wiki/Chi\\_squared\\_distribution](http://en.wikipedia.org/wiki/Chi_squared_distribution)
- [R396] <http://mathworld.wolfram.com/Chi-SquaredDistribution.html>
- [R397] [http://en.wikipedia.org/wiki/Dagum\\_distribution](http://en.wikipedia.org/wiki/Dagum_distribution)
- [R398] [http://en.wikipedia.org/wiki/Erlang\\_distribution](http://en.wikipedia.org/wiki/Erlang_distribution)
- [R399] <http://mathworld.wolfram.com/ErlangDistribution.html>
- [R400] [http://en.wikipedia.org/wiki/Exponential\\_distribution](http://en.wikipedia.org/wiki/Exponential_distribution)
- [R401] <http://mathworld.wolfram.com/ExponentialDistribution.html>
- [R402] <http://en.wikipedia.org/wiki/F-distribution>
- [R403] <http://mathworld.wolfram.com/F-Distribution.html>
- [R404] [http://en.wikipedia.org/wiki/Fisher%27s\\_z-distribution](http://en.wikipedia.org/wiki/Fisher%27s_z-distribution)
- [R405] <http://mathworld.wolfram.com/Fishersz-Distribution.html>
- [R406] [http://en.wikipedia.org/wiki/Fr%C3%A9chet\\_distribution](http://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution)
- [R407] [http://en.wikipedia.org/wiki/Gamma\\_distribution](http://en.wikipedia.org/wiki/Gamma_distribution)
- [R408] <http://mathworld.wolfram.com/GammaDistribution.html>
- [R409] [http://en.wikipedia.org/wiki/Inverse-gamma\\_distribution](http://en.wikipedia.org/wiki/Inverse-gamma_distribution)
- [R410] [http://en.wikipedia.org/wiki/Kumaraswamy\\_distribution](http://en.wikipedia.org/wiki/Kumaraswamy_distribution)
- [R411] [http://en.wikipedia.org/wiki/Laplace\\_distribution](http://en.wikipedia.org/wiki/Laplace_distribution)
- [R412] <http://mathworld.wolfram.com/LaplaceDistribution.html>
- [R413] [http://en.wikipedia.org/wiki/Logistic\\_distribution](http://en.wikipedia.org/wiki/Logistic_distribution)
- [R414] <http://mathworld.wolfram.com/LogisticDistribution.html>

- [R415] <http://en.wikipedia.org/wiki/Lognormal>
- [R416] <http://mathworld.wolfram.com/LogNormalDistribution.html>
- [R417] [http://en.wikipedia.org/wiki/Maxwell\\_distribution](http://en.wikipedia.org/wiki/Maxwell_distribution)
- [R418] <http://mathworld.wolfram.com/MaxwellDistribution.html>
- [R419] [http://en.wikipedia.org/wiki/Nakagami\\_distribution](http://en.wikipedia.org/wiki/Nakagami_distribution)
- [R420] [http://en.wikipedia.org/wiki/Normal\\_distribution](http://en.wikipedia.org/wiki/Normal_distribution)
- [R421] <http://mathworld.wolfram.com/NormalDistributionFunction.html>
- [R422] [http://en.wikipedia.org/wiki/Pareto\\_distribution](http://en.wikipedia.org/wiki/Pareto_distribution)
- [R423] <http://mathworld.wolfram.com/ParetoDistribution.html>
- [R424] [http://en.wikipedia.org/wiki/U-quadratic\\_distribution](http://en.wikipedia.org/wiki/U-quadratic_distribution)
- [R425] [http://en.wikipedia.org/wiki/Raised\\_cosine\\_distribution](http://en.wikipedia.org/wiki/Raised_cosine_distribution)
- [R426] [http://en.wikipedia.org/wiki/Rayleigh\\_distribution](http://en.wikipedia.org/wiki/Rayleigh_distribution)
- [R427] <http://mathworld.wolfram.com/RayleighDistribution.html>
- [R428] [http://en.wikipedia.org/wiki/Student\\_t-distribution](http://en.wikipedia.org/wiki/Student_t-distribution)
- [R429] <http://mathworld.wolfram.com/Studentst-Distribution.html>
- [R430] [http://en.wikipedia.org/wiki/Triangular\\_distribution](http://en.wikipedia.org/wiki/Triangular_distribution)
- [R431] <http://mathworld.wolfram.com/TriangularDistribution.html>
- [R432] [http://en.wikipedia.org/wiki/Uniform\\_distribution\\_%28continuous%29](http://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29)
- [R433] <http://mathworld.wolfram.com/UniformDistribution.html>
- [R434] [http://en.wikipedia.org/wiki/Uniform\\_sum\\_distribution](http://en.wikipedia.org/wiki/Uniform_sum_distribution)
- [R435] <http://mathworld.wolfram.com/UniformSumDistribution.html>
- [R436] [http://en.wikipedia.org/wiki/Von\\_Mises\\_distribution](http://en.wikipedia.org/wiki/Von_Mises_distribution)
- [R437] <http://mathworld.wolfram.com/vonMisesDistribution.html>
- [R438] [http://en.wikipedia.org/wiki/Weibull\\_distribution](http://en.wikipedia.org/wiki/Weibull_distribution)
- [R439] <http://mathworld.wolfram.com/WeibullDistribution.html>
- [R440] [http://en.wikipedia.org/wiki/Wigner\\_semicircle\\_distribution](http://en.wikipedia.org/wiki/Wigner_semicircle_distribution)
- [R441] <http://mathworld.wolfram.com/WignersSemicircleLaw.html>
- [R377] S. A. Abramov, M. Bronstein and M. Petkovsek, On polynomial solutions of linear operator equations, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 290-296.
- [R378] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R379] 13. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.
- [R380] S. A. Abramov, Rational solutions of linear difference and q-difference equations with polynomial coefficients, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 285-289
- [R381] M. Petkovsek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R382] 13. Petkovsek, H. S. Wilf, D. Zeilberger, A = B, 1996.

- [R359] Methods to solve  $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$ , [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>
- [R360] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , [online], Available: <http://www.jpr2718.org/ax2p.pdf>
- [R361] Solving the generalized Pell equation  $x^{**2} - D*y^{**2} = N$ , John P. Robertson, July 31, 2004, Pages 16 - 17. [online], Available: <http://www.jpr2718.org/pell.pdf>
- [R362] 1. Nitaj, "L'algorithme de Cornacchia"
- [R363] Solving the diophantine equation  $ax^{**2} + by^{**2} = m$  by Cornacchia's method, [online], Available: <http://www.numbertheory.org/php/cornacchia.html>
- [R364] Solving the generalized Pell equation  $x^{**2} - D*y^{**2} = N$ , John P. Robertson, July 31, 2004, Page 15. <http://www.jpr2718.org/pell.pdf>
- [R365] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>
- [R366] Solving the equation  $ax^2 + bxy + cy^2 + dx + ey + f = 0$ , John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>
- [R367] Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R368] Representing a number as a sum of three squares, [online], Available: <http://www.schorn.ch/howto.html>
- [R369] Representing a number as a sum of four squares, [online], Available: <http://www.schorn.ch/howto.html>
- [R370] Solving the generalized Pell equation  $x^2 - Dy^2 = N$ , John P. Robertson, July 31, 2004, Pages 4 - 8. <http://www.jpr2718.org/pell.pdf>
- [R371] Solving the generalized Pell equation  $x^{**2} - D*y^{**2} = N$ , John P. Robertson, July 31, 2004, Page 12. <http://www.jpr2718.org/pell.pdf>
- [R372] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R373] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R374] Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R375] Gaussian lattice Reduction [online]. Available: <http://home.ie.cuhk.edu.hk/~wkshum/wordpress/?p=404>
- [R376] Efficient Solution of Rational Conices, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [AOCP] Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.
- [Factorisatio] On a Problem of Oppenheim concerning "Factorisatio Numerorum" E. R. Canfield, Paul Erdos, Carl Pomerance, JOURNAL OF NUMBER THEORY, Vol. 17, No. 1. August 1983. See section 7 for a description of an algorithm similar to Knuth's.
- [Yorgey] Generating Multiset Partitions, Brent Yorgey, The Monad.Reader, Issue 8, September 2007.
- [R1] [http://en.wikipedia.org/wiki/Euler%20%93Lagrange\\_equation](http://en.wikipedia.org/wiki/Euler%20%93Lagrange_equation)
- [R2] [http://en.wikipedia.org/wiki/Mathematical\\_singularity](http://en.wikipedia.org/wiki/Mathematical_singularity)

[R3] Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0

## Symbols

<code>_TensorManager</code> (class in <code>sympy.tensor.tensor</code> ), 1096	<code>_linear_2eq_order1_type7()</code>	(in <code>sympy.solvers.ode</code> ), 1018	module
<code>_eq_()</code> ( <code>sympy.combinatorics.perm_groups.PermutationGroup</code> )	<code>_linear_2eq_order2_type1()</code>	(in <code>sympy.solvers.ode</code> ), 1019	module
method), 186			
<code>_getitem_()</code> ( <code>sympy.matrices.dense.DenseMatrix</code> )	<code>_linear_2eq_order2_type10()</code>	(in <code>sympy.solvers.ode</code> ), 1025	module
method), 623			
<code>_mul_()</code> ( <code>sympy.combinatorics.perm_groups.PermutationGroup</code> )	<code>_linear_2eq_order2_type11()</code>	(in <code>sympy.solvers.ode</code> ), 1025	module
method), 186			
<code>_mul_()</code> ( <code>sympy.matrices.dense.DenseMatrix</code> )	<code>_linear_2eq_order2_type2()</code>	(in <code>sympy.solvers.ode</code> ), 1020	module
method), 624			
<code>_new_()</code> ( <code>sympy.combinatorics.perm_groups.PermutationGroup</code> )	<code>_linear_2eq_order2_type3()</code>	(in <code>sympy.solvers.ode</code> ), 1021	module
static method), 186			
<code>_weakref_</code> ( <code>sympy.combinatorics.perm_groups.PermutationGroup</code> )	<code>_linear_2eq_order2_type4()</code>	(in <code>sympy.solvers.ode</code> ), 1021	module
attribute), 186			
<code>_af_parity()</code>	<code>_linear_2eq_order2_type5()</code>	(in <code>sympy.solvers.ode</code> ), 1022	module
(in <code>sympy.combinatorics.permutations</code> ), 183			
<code>_base_ordering()</code>	<code>_linear_2eq_order2_type6()</code>	(in <code>sympy.solvers.ode</code> ), 1022	module
(in <code>sympy.combinatorics.util</code> ), 232			
<code>_check_cycles_alt_sym()</code>	<code>_linear_2eq_order2_type7()</code>	(in <code>sympy.solvers.ode</code> ), 1023	module
(in <code>sympy.combinatorics.util</code> ), 233			
<code>_cmp_perm_lists()</code>	<code>_linear_2eq_order2_type8()</code>	(in <code>sympy.solvers.ode</code> ), 1023	module
(in <code>sympy.combinatorics.testutil</code> ), 238			
<code>_distribute_gens_by_base()</code>	<code>_linear_2eq_order2_type9()</code>	(in <code>sympy.solvers.ode</code> ), 1024	module
(in <code>sympy.combinatorics.util</code> ), 233			
<code>_handle_Integral()</code> (in module <code>sympy.solvers.ode</code> ), 1036	<code>_linear_3eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code> ), 1025	module
<code>_handle_precomputed_bsgs()</code>	<code>_linear_3eq_order1_type2()</code>	(in <code>sympy.solvers.ode</code> ), 1026	module
(in <code>sympy.combinatorics.util</code> ), 234			
<code>_linear_2eq_order1_type1()</code>	<code>_linear_3eq_order1_type3()</code>	(in <code>sympy.solvers.ode</code> ), 1027	module
(in <code>sympy.solvers.ode</code> ), 1015			
<code>_linear_2eq_order1_type2()</code>	<code>_linear_3eq_order1_type4()</code>	(in <code>sympy.solvers.ode</code> ), 1027	module
(in <code>sympy.solvers.ode</code> ), 1016			
<code>_linear_2eq_order1_type3()</code>	<code>_linear_neq_order1_type1()</code>	(in <code>sympy.solvers.ode</code> ), 1028	module
(in <code>sympy.solvers.ode</code> ), 1017			
<code>_linear_2eq_order1_type4()</code>	<code>_modgcd_multivariate_p()</code>	(in <code>sympy.polys.modulargcd</code> ), 844	module
(in <code>sympy.solvers.ode</code> ), 1017			
<code>_linear_2eq_order1_type5()</code>	<code>_naive_list_centralizer()</code>	(in <code>sympy.combinatorics.testutil</code> ), 238	module
(in <code>sympy.solvers.ode</code> ), 1018			
<code>_linear_2eq_order1_type6()</code>	<code>_nonlinear_2eq_order1_type1()</code>	(in <code>sympy.solvers.ode</code> ), 1029	module
(in <code>sympy.solvers.ode</code> ), 1018			

_nonlinear_2eq_order1_type2()	(in	module	accepted_latex_functions	(in	module
sympy.solvers.ode),	1029		sympy.printing.latex),	861	
_nonlinear_2eq_order1_type3()	(in	module	acos (class in sympy.functions.elementary.trigonometric),		
sympy.solvers.ode),	1030		314		
_nonlinear_2eq_order1_type4()	(in	module	acosh (class in sympy.functions.elementary.hyperbolic),		
sympy.solvers.ode),	1030		315		
_nonlinear_2eq_order1_type5()	(in	module	acot (class in sympy.functions.elementary.trigonometric),		
sympy.solvers.ode),	1030		315		
_nonlinear_3eq_order1_type1()	(in	module	acoth (class in sympy.functions.elementary.hyperbolic),		
sympy.solvers.ode),	1031		315		
_nonlinear_3eq_order1_type2()	(in	module	acsc (class in sympy.functions.elementary.trigonometric),		
sympy.solvers.ode),	1031		316		
_nonlinear_3eq_order1_type3()	(in	module	Add (class in sympy.core.add),	115	
sympy.solvers.ode),	1032		add() (sympy.assumptions.assume.AssumptionsContext		
_nonlinear_3eq_order1_type4()	(in	module	method),	882	
sympy.solvers.ode),	1032		Add() (sympy.assumptions.handlers.calculus.AskFiniteHandler		
_nonlinear_3eq_order1_type5()	(in	module	static method),	884	
sympy.solvers.ode),	1033		Add() (sympy.assumptions.handlers.calculus.AskInfinitesimalHandler		
_orbits_transversals_from_bsgs()	(in	module	static method),	886	
sympy.combinatorics.util),	235		Add() (sympy.assumptions.handlers.order.AskNegativeHandler		
_print() (sympy.printing.printer.Printer method),	853		static method),	886	
_random_pr_init() (sympy.combinatorics.perm_groups.Permutat	ion method),	186	Add() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler		
_remove_gens()	(in	module	static method),	887	
sympy.combinatorics.util),	235		Add() (sympy.assumptions.handlers.sets.AskHermitianHandler		
_strip() (in module sympy.combinatorics.util),	236		static method),	887	
_strong_gens_from_distr()	(in	module	Add() (sympy.assumptions.handlers.sets.AskImaginaryHandler		
sympy.combinatorics.util),	237		static method),	887	
_undetermined_coefficients_match()	(in	module	Add() (sympy.assumptions.handlers.sets.AskIntegerHandler		
sympy.solvers.ode),	1036		static method),	888	
_union_find_merge() (sympy.combinatorics.perm_groups.Permutat	ion method),	188	Add() (sympy.assumptions.handlers.sets.AskRationalHandler		
_union_find.rep() (sympy.combinatorics.perm_groups.Permutat	ion method),	187	ion method),	888	
_union_find.rep()	(sympy.combinatorics.perm_groups.Permutat		Add() (sympy.assumptions.handlers.sets.AskRealHandler		
ion method),	187		ion method),	888	
_verify_bsgs()	(in	module	add() (sympy.matrices.matrices.MatrixBase		
sympy.combinatorics.testutil),	238		method),	585	
_verify_centralizer()	(in	module	add() (sympy.matrices.sparse.SparseMatrix method),		
sympy.combinatorics.testutil),	239		633		
_verify_normal_closure()	(in	module	add() (sympy.polys.domains.domain.Domain		
sympy.combinatorics.testutil),	239		method),	752	
A			add() (sympy.polys.polyclasses.DMF method),	767	
a2idx() (in module sympy.matrices.matrices),	623		add() (sympy.polys.polyclasses.DMP method),	762	
AbelianGroup() (in	module		add() (sympy.polys.polytools.Poly method),	681	
sympy.combinatorics.named_groups),	232		add() (sympy.polys.rings.PolyRing method),	814	
above() (sympy.printing.pretty.stringPict.stringPict			add_formulae() (in	module	
method),	869		sympy.simplify.hyperexpand),	947	
Abs (class in sympy.functions.elementary.complexes),			add_ground() (sympy.polys.polyclasses.DMP		
313			method),	762	
abs() (sympy.polys.domains.domain.Domain			add_ground() (sympy.polys.polytools.Poly method),		
method),	752		681		
abs() (sympy.polys.polyclasses.DMP method),	762		adjoint() (sympy.matrices.immutable.ImmutableMatrix		
abs() (sympy.polys.polytools.Poly method),	681		method),	645	

**adjugate()** (sympy.matrices.matrices.MatrixBase method), 585  
**airyai** (class in sympy.functions.special.bessel), 394  
**airyaiprime** (class in sympy.functions.special.bessel), 397  
**AiryBase** (class in sympy.functions.special.bessel), 394  
**airybi** (class in sympy.functions.special.bessel), 395  
**airybiprime** (class in sympy.functions.special.bessel), 398  
**algebraic**, 61  
**algebraic\_field()** (sympy.polys.domains.AlgebraicField method), 759  
**algebraic\_field()** (sympy.polys.domains.domain.Domain method), 752  
**algebraic\_field()** (sympy.polys.domains.IntegerRing method), 757  
**algebraic\_field()** (sympy.polys.domains.RationalField method), 758  
**AlgebraicField** (class in sympy.polys.domains), 759  
**AlgebraicNumber** (class in sympy.polys.numberfields), 720  
**all\_coeffs()** (sympy.polys.polyclasses.DMP method), 762  
**all\_coeffs()** (sympy.polys.polytools.Poly method), 681  
**all\_monoms()** (sympy.polys.polyclasses.DMP method), 762  
**all\_monoms()** (sympy.polys.polytools.Poly method), 682  
**all\_roots()** (sympy.polys.polytools.Poly method), 682  
**all\_terms()** (sympy.polys.polyclasses.DMP method), 762  
**all\_terms()** (sympy.polys.polytools.Poly method), 682  
**allhints** (in module sympy.solvers.ode), 989  
**almosteq()** (sympy.polys.domains.domain.Domain method), 752  
**almosteq()** (sympy.polys.domains.RealField method), 760  
**almosteq()** (sympy.polys.rings.PolyElement method), 814  
**alphabet\_of\_cipher()** (in sympy.crypto.crypto module), 272  
**alternating()** (sympy.combinatorics.generators static method), 184  
**AlternatingGroup()** (in sympy.combinatorics.named\_groups), 231  
**altitudes** (sympy.geometry.polygon.Triangle attribute), 510  
**an** (sympy.functions.special.hyper.meijerg attribute), 409  
**And** (class in sympy.logic.boolalg), 565  
**angle\_between()** (sympy.geometry.line.LinearEntity method), 449  
**angle\_between()** (sympy.geometry.line3d.LinearEntity3D method), 466  
**angle\_between()** (sympy.geometry.plane.Plane method), 516  
**angles** (sympy.geometry.polygon.Polygon attribute), 497  
**angles** (sympy.geometry.polygon.RegularPolygon attribute), 503  
**annotated()** (in sympy.printing.pretty.pretty\_symbology), 869  
**ANP** (class in sympy.polys.polyclasses), 768  
**antihermitian**, 61  
**aother** (sympy.functions.special.hyper.meijerg attribute), 409  
**ap** (sympy.functions.special.hyper.hyper attribute), 407  
**ap** (sympy.functions.special.hyper.meijerg attribute), 409  
**apart()** (in module sympy.polys.partfrac), 725  
**apart()** (sympy.core.expr.Expr method), 75  
**apart\_list()** (in module sympy.polys.partfrac), 725  
**apoapsis** (sympy.geometry.ellipse.Ellipse attribute), 483  
**apothem** (sympy.geometry.polygon.RegularPolygon attribute), 503  
**append()** (sympy.plotting.plot.Plot method), 874  
**AppliedPredicate** (class in sympy.assumptions.assume), 881  
**apply()** (sympy.printing.pretty.stringpict.prettyForm static method), 871  
**apply()** (sympy.simplify.epathtools.EPath method), 936  
**apply\_finite\_diff()** (in sympy.calculus.finite\_diff), 1167  
**applyfunc()** (sympy.matrices.dense.DenseMatrix method), 624  
**applyfunc()** (sympy.matrices.sparse.SparseMatrix method), 633  
**approximation()** (sympy.core.numbers.NumberSymbol method), 103  
**arbitrary\_point()** (sympy.geometry.curve.Curve method), 479  
**arbitrary\_point()** (sympy.geometry.ellipse.Ellipse method), 483  
**arbitrary\_point()** (sympy.geometry.line.LinearEntity method), 449  
**arbitrary\_point()** (sympy.geometry.line3d.LinearEntity3D method), 467  
**arbitrary\_point()** (sympy.geometry.plane.Plane method), 516

arbitrary\_point() (sympy.geometry.polygon.Polygon method), 497  
Arcsin() (in module sympy.stats), 952  
are\_collinear() (sympy.geometry.point3d.Point3D static method), 444  
are\_concurrent() (sympy.geometry.line.LinearEntity static method), 450  
are\_concurrent() (sympy.geometry.line3d.LinearEntity3D static method), 467  
are\_concurrent() (sympy.geometry.plane.Plane static method), 517  
are\_coplanar() (sympy.geometry.point3d.Point3D static method), 444  
are\_similar() (in module sympy.geometry.util), 436  
area (sympy.geometry.ellipse.Ellipse attribute), 484  
area (sympy.geometry.polygon.Polygon attribute), 498  
area (sympy.geometry.polygon.RegularPolygon attribute), 503  
arg (class in sympy.functions.elementary.complexes), 316  
arg (sympy.assumptions.assume.AppliedPredicate attribute), 881  
args (sympy.core.basic.Basic attribute), 62  
args (sympy.geometry.polygon.RegularPolygon attribute), 503  
args (sympy.polys.polytools.Poly attribute), 682  
args (sympy.tensor.indexed.IndexedBase attribute), 1092  
args\_cnc() (sympy.core.expr.Expr method), 75  
Argument (class in sympy.utilitiescodegen), 1117  
argument (sympy.functions.special.bessel.BesselBase attribute), 389  
argument (sympy.functions.special.hyper.hyper attribute), 407  
argument (sympy.functions.special.hyper.meijerg attribute), 409  
array\_form (sympy.combinatorics.permutations.Permutation attribute), 165  
array\_form (sympy.combinatorics.polyhedron.Polyhedron attribute), 213  
ArrowStringDescription (class in sympy.categories.diagram\_drawing), 1182  
as\_base\_exp() (sympy.core.function.Function method), 141  
as\_base\_exp() (sympy.core.power.Pow method), 111  
as\_base\_exp() (sympy.functions.elementary.exponential.exp method), 329  
as\_coeff\_Add() (sympy.core.add.Add method), 115  
as\_coeff\_add() (sympy.core.add.Add method), 115  
as\_coeff\_Add() (sympy.core.expr.Expr method), 75  
as\_coeff\_add() (sympy.core.expr.Expr method), 75  
as\_coeff\_Add() (sympy.core.numbers.Number method), 99  
as\_coeff\_exponent() (sympy.core.expr.Expr method), 76  
as\_coeff\_Mul() (sympy.core.expr.Expr method), 75  
as\_coeff\_mul() (sympy.core.expr.Expr method), 76  
as\_coeff\_Mul() (sympy.core.mul.Mul method), 113  
as\_coeff\_Mul() (sympy.core.numbers.Number method), 99  
as\_coefficient() (sympy.core.expr.Expr method), 76  
as\_coefficients\_dict() (sympy.core.add.Add method), 115  
as\_coefficients\_dict() (sympy.core.expr.Expr method), 77  
as\_content\_primitive() (sympy.core.add.Add method), 116  
as\_content\_primitive() (sympy.core.basic.Basic method), 63  
as\_content\_primitive() (sympy.core.expr.Expr method), 78  
as\_content\_primitive() (sympy.core.mul.Mul method), 113  
as\_content\_primitive() (sympy.core.numbers.Rational method), 102  
as\_content\_primitive() (sympy.core.power.Pow method), 112  
as\_dict() (sympy.combinatorics.partitions.IntegerPartition method), 161  
as\_dict() (sympy.polys.polytools.Poly method), 683  
as\_dummy() (sympy.concrete.expr\_with\_limits.ExprWithLimits method), 296  
as\_dummy() (sympy.core.symbol.Symbol method), 95  
as\_explicit() (sympy.matrices.expressions.MatrixExpr method), 647  
as\_expr() (sympy.core.expr.Expr method), 78  
as\_expr() (sympy.polys.numberfields.AlgebraicNumber method), 720  
as\_expr() (sympy.polys.polytools.Poly method), 683  
as\_finite\_diff() (in module sympy.calculus.finite\_diff), 1168  
as\_immutable() (sympy.matrices.dense.DenseMatrix method), 624  
as\_immutable() (sympy.matrices.sparse.SparseMatrix method), 633  
as\_independent() (sympy.core.expr.Expr method), 79  
as\_integ() (in module sympy.core.compatibility), 151, 157  
as\_leading\_term() (sympy.core.expr.Expr method), 81  
as\_list() (sympy.polys.polytools.Poly method), 683  
asMutable() (sympy.matrices.dense.DenseMatrix method), 624

asMutable() (sympy.matrices.expressions.MatrixExpr method),	648	ask_full_inference() (in module sympy.assumptions.ask),	880	module
asMutable() (sympy.matrices.immutable.ImmutableMatrix method),	645	AskAlgebraicHandler (class in sympy.assumptions.handlers.sets),	887	in
asMutable() (sympy.matrices.sparse.SparseMatrix method),	633	AskAntiHermitianHandler (class in sympy.assumptions.handlers.sets),	887	in
as_numer_denom() (sympy.core.expr.Expr method),	81	AskComplexHandler (class in sympy.assumptions.handlers.sets),	887	in
as_ordered_factors() (sympy.core.expr.Expr method),	81	AskExtendedRealHandler (class in sympy.assumptions.handlers.sets),	887	in
as_ordered_factors() (sympy.core.mul.Mul method),	113	AskFiniteHandler (class in sympy.assumptions.handlers.calculus),	884	in
as_ordered_terms() (sympy.core.expr.Expr method),	81	AskHermitianHandler (class in sympy.assumptions.handlers.sets),	887	in
as_poly() (sympy.core.basic.Basic method),	63	AskImaginaryHandler (class in sympy.assumptions.handlers.sets),	887	in
as_poly() (sympy.polys.numberfields.AlgebraicNumber method),	720	AskInfinitesimalHandler (class in sympy.assumptions.handlers.calculus),	886	in
as_powers_dict() (sympy.core.expr.Expr method),	81	AskNonZeroHandler (class in sympy.assumptions.handlers.order),	886	in
as_real_imag() (sympy.core.add.Add method),	116	AskOddHandler (class in sympy.assumptions.handlers.order),	886	in
as_real_imag() (sympy.core.expr.Expr method),	82	AskIntegerHandler (class in sympy.assumptions.handlers.sets),	887	in
as_real_imag() (sympy.functions.elementary.complexes.infinity method),	328	AskNegativeHandler (class in sympy.assumptions.handlers.order),	886	in
as_real_imag() (sympy.functions.elementary.complexes.reals method),	333	AskNonZeroHandler (class in sympy.assumptions.handlers.order),	886	in
as_real_imag() (sympy.functions.elementary.exponential.exp method),	325	AskPrimeHandler (class in sympy.assumptions.handlers.order),	886	in
as_real_imag() (sympy.functions.elementary.exponential.exp method),	329	AskOddHandler (class in sympy.assumptions.handlers.ntheory),	886	in
as_real_imag() (sympy.functions.elementary.hyperbolic.sinh method),	336	AskPositiveHandler (class in sympy.assumptions.handlers.order),	886	in
as_relational() (sympy.sets.sets.FiniteSet method),	910	AskPrimeHandler (class in sympy.assumptions.handlers.order),	886	in
as_relational() (sympy.sets.sets.Intersection method),	911	AskRationalHandler (class in sympy.assumptions.handlers.sets),	888	in
as_relational() (sympy.sets.sets.Interval method),	908	AskRealHandler (class in sympy.assumptions.handlers.sets),	888	in
as_relational() (sympy.sets.sets.Union method),	911	assemble_partfrac_list() (in module sympy.polys.partfrac),	727	module
as_set() (sympy.core.relational.Relational method),	119	AssignmentError, 868		
as_sum() (sympy.integrals.integrals.Integral method),	551	assoc_laguerre (class in sympy.functions.special.polynomials),	420	in
as_terms() (sympy.core.expr.Expr method),	82	assoc_legendre (class in sympy.functions.special.polynomials),	420	in
as_two_terms() (sympy.core.add.Add method),	116	assoc_recurrence_memo() (in module sympy.utilities.memoization),	1150	module
as_two_terms() (sympy.core.mul.Mul method),	113	assuming() (in module sympy.assumptions.assume),	882	
ascents() (sympy.combinatorics.permutations.Permutation method),	165	assumptions0 (sympy.core.basic.Basic attribute),	63	
asec (class in sympy.functions.elementary.trigonometric),	318	AssumptionsContext (class in sympy.assumptions.assume),	881	in
aseries() (sympy.core.expr.Expr method),	82			
asin (class in sympy.functions.elementary.trigonometric),	317			
asinh (class in sympy.functions.elementary.hyperbolic),	317			
ask() (in module sympy.assumptions.ask),	880			

atan (class in `sympy.functions.elementary.trigonometric`),  
    **318**

atan2 (class in `sympy.functions.elementary.trigonometric`),  
    **319**

atanh (class in `sympy.functions.elementary.hyperbolic`),  
    **321**

Atom (class in `sympy.core.basic`), **74**

AtomicExpr (class in `sympy.core.expr`), **95**

atoms() (sympy.combinatorics.permutations.Permutat**ionGroup**ions),  
    **165**

atoms() (sympy.core.basic.Basic method), **64**

atoms() (sympy.matrices.matrices.MatrixBase  
    method), **585**

atoms\_table (in module  
    `sympy.printing.pretty.pretty_symbology`),  
    **869**

auto\_number() (in module  
    `sympy.parsing.sympy_parser`), **1165**

auto\_symbol() (in module  
    `sympy.parsing.sympy_parser`), **1165**

autowrap() (in module `sympy.utilities.autowrap`),  
    **1112**

**B**

base (sympy.combinatorics.perm\_groups.PermutationGroup  
    attribute), **188**

base (sympy.functions.elementary.exponential.exp  
    attribute), **325**

base (sympy.tensor.indexed.Indexed attribute), **1090**

base\_oneform() (sympy.diffgeom.CoordSystem  
    method), **1189**

base\_oneforms() (sympy.diffgeom.CoordSystem  
    method), **1189**

base\_solution\_linear() (in module  
    `sympy.solvers.diophantine`), **1066**

base\_vector() (sympy.diffgeom.CoordSystem  
    method), **1189**

base\_vectors() (sympy.diffgeom.CoordSystem  
    method), **1189**

BaseCovarDerivativeOp (class in `sympy.diffgeom`),  
    **1196**

BasePolynomialError (class in  
    `sympy.polys.polyerrors`), **838**

BaseScalarField (class in `sympy.diffgeom`), **1191**

BaseSeries (class in `sympy.plotting.plot`), **878**

baseswap() (sympy.combinatorics.perm\_groups.Permutat**ionGroup**ions),  
    **188**

BaseVectorField (class in `sympy.diffgeom`), **1192**

Basic (class in `sympy.core.basic`), **62**

basic\_orbits (sympy.combinatorics.perm\_groups.Permutat**ionGroup**ions),  
    **189**

basic\_stabilizers (sympy.combinatorics.perm\_groups.Permutat**ionGroup**ions),  
    **189**

basic\_transversals (sympy.combinatorics.perm\_groups.Permutat**ionGroup**ions),  
    **190**

below() (sympy.printing.pretty.stringPict.stringPict  
    method), **869**

Benini() (in module `sympy.stats`), **952**

berkowitz() (sympy.matrices.matrices.MatrixBase  
    method), **586**

berkowitz\_charpoly() (sympy.matrices.matrices.MatrixBase  
    method), **586**

berkowitz\_det() (sympy.matrices.matrices.MatrixBase  
    method), **587**

berkowitz\_eigenvals() (sympy.matrices.matrices.MatrixBase  
    method), **587**

berkowitz\_minors() (sympy.matrices.matrices.MatrixBase  
    method), **587**

bernoulli (class in `sympy.functions.combinatorial.numbers`),  
    **343**

Bernoulli() (in module `sympy.stats`), **949**

BesselBase (class in `sympy.functions.special.bessel`),  
    **389**

besselj (class in `sympy.functions.special.bessel`), **389**

bessellk (class in `sympy.functions.special.bessel`), **391**

besselsimp() (in module `sympy.simplify.simplify`), **925**

bessely (class in `sympy.functions.special.bessel`), **390**

beta (class in `sympy.functions.special(beta_functions)`),  
    **367**

Beta() (in module `sympy.stats`), **953**

BetaPrime() (in module `sympy.stats`), **954**

bifid5\_square() (in module `sympy.crypto.crypto`), **279**

bifid6\_square() (in module `sympy.crypto.crypto`), **280**

bifid7\_square() (in module `sympy.crypto.crypto`), **281**

bin\_to\_gray() (sympy.combinatorics.graycode static  
    method), **229**

binary\_function() (in module  
    `sympy.utilities.autowrap`), **1113**

binary\_partitions() (in module  
    `sympy.utilities.iterables`), **1131**

binomial (class in `sympy.functions.combinatorial.factorials`),  
    **344**

Binomial() (in module `sympy.stats`), **949**

binomial\_coefficients() (in module  
    `sympy.nttheory.multinomial`), **262**

binomial\_coefficients\_list() (in module  
    `sympy.nttheory.multinomial`), **262**

binionGroup (sympy.geometry.polygon.Triangle  
    method), **510**

bitlist\_inCrsubset() (sympy.combinatorics.subsets.Subset  
    class method), **220**

block\_collapse() (in module sympy.matrices.expressions.blockmatrix), 651  
 BlockDiagMatrix (class in sympy.matrices.expressions.blockmatrix), 651  
 BlockMatrix (class in sympy.matrices.expressions.blockmatrix), 650  
 bm (sympy.functions.special.hyper.meijerg attribute), 409  
 bool\_map() (in module sympy.logic.boolalg), 571  
 BooleanFalse (class in sympy.logic.boolalg), 565  
 BooleanTrue (class in sympy.logic.boolalg), 564  
 bother (sympy.functions.special.hyper.meijerg attribute), 409  
 boundary (sympy.sets.sets.Set attribute), 903  
 bq (sympy.functions.special.hyper.hyper attribute), 407  
 bq (sympy.functions.special.hyper.meijerg attribute), 409  
 bracelets() (in module sympy.utilities.iterables), 1131  
 bsgs\_direct\_product() (in module sympy.combinatorics.tensor\_can), 244  
 bspline\_basis() (in module sympy.functions.special.bsplines), 399  
 bspline\_basis\_set() (in module sympy.functions.special.bsplines), 400  
 build\_expression\_tree() (in module sympy.series.gruntz), 902

**C**

C (sympy.matrices.immutable.ImmutableMatrix attribute), 645  
 C (sympy.matrices.matrices.MatrixBase attribute), 581  
 cacheit() (in module sympy.core.cache), 62  
 calculate\_series() (in module sympy.series.gruntz), 902  
 cancel() (in module sympy.polys.polytools), 678  
 cancel() (sympy.core.expr.Expr method), 83  
 cancel() (sympy.polys.polyclasses.DMF method), 768  
 cancel() (sympy.polys.polyclasses.DMP method), 763  
 cancel() (sympy.polys.polytools.Poly method), 683  
 cancel() (sympy.polys.rings.PolyElement method), 815  
 canon\_bp() (in module sympy.tensor.tensor), 1109  
 canon\_bp() (sympy.tensor.tensor.TensAdd method), 1106  
 canon\_bp() (sympy.tensor.tensor.TensMul method), 1107  
 canonical (sympy.core.relational.Relational attribute), 119  
 canonical\_variables (sympy.core.basic.Basic attribute), 65  
 canonicalize() (in module sympy.combinatorics.tensor\_can), 239  
 capture() (in module sympy.utilities.iterables), 1132  
 cardinality (sympy.combinatorics.permutations.Permutation attribute), 165  
 cardinality (sympy.combinatorics.subsets.Subset attribute), 220  
 casoratian() (in module sympy.matrices.dense), 619  
 Catalan (class in sympy.core.numbers), 110  
 catalan (class in sympy.functions.combinatorial.numbers), 345  
 Category (class in sympy.categories), 1175  
 Cauchy() (in module sympy.stats), 954  
 ccode() (in module sympy.printing.ccode), 854  
 CCodeGen (class in sympy.utilitiescodegen), 1118  
 CCodePrinter (class in sympy.printing.ccode), 854  
 ceiling (class in sympy.functions.elementary.integers), 321  
 center (sympy.geometry.ellipse.Ellipse attribute), 484  
 center (sympy.geometry.polygon.RegularPolygon attribute), 504  
 center() (sympy.combinatorics.perm\_groups.PermutationGroup method), 190  
 centralizer() (sympy.combinatorics.perm\_groups.PermutationGroup method), 191  
 centroid (sympy.geometry.polygon.Polygon attribute), 498  
 centroid (sympy.geometry.polygon.RegularPolygon attribute), 504  
 centroid() (in module sympy.geometry.util), 436  
 change\_index() (sympy.concrete.expr\_with\_intlimits.ExprWithIntLimits method), 297  
 characteristic() (sympy.polys.domains.domain.Domain method), 752  
 characteristic() (sympy.polys.domains.FiniteField method), 757  
 charpoly() (sympy.matrices.matrices.MatrixBase method), 587  
 chebyshevt (class in sympy.functions.special.polynomials), 415  
 chebyshevt\_poly() (in module sympy.polys.orthopolys), 723  
 chebyshevt\_root (class in sympy.functions.special.polynomials), 416  
 chebyshev (class in sympy.functions.special.polynomials), 416  
 chebyshev\_poly() (in module sympy.polys.orthopolys), 723

chebyshev.u\_root (class in `sympy.functions.special.polynomials`), [417](#)  
check\_assumptions() (in module `sympy.solvers.solvers`), [1053](#)  
check\_output() (`sympy.utilities.runtests.SymPyOutput` method), [1155](#)  
checkinfsol() (in module `sympy.solvers.ode`), [989](#)  
checkodesol() (in module `sympy.solvers.ode`), [986](#)  
checkpdesol() (in module `sympy.solvers.pde`), [1040](#)  
checksol() (in module `sympy.solvers.solvers`), [1054](#)  
Chi (class in `sympy.functions.special.error_functions`), [388](#)  
Chi() (in module `sympy.stats`), [955](#)  
ChiNoncentral() (in module `sympy.stats`), [955](#)  
ChiSquared() (in module `sympy.stats`), [956](#)  
cholesky() (`sympy.matrices.matrices.MatrixBase` method), [588](#)  
cholesky() (`sympy.matrices.sparse.SparseMatrix` method), [634](#)  
cholesky\_solve() (`sympy.matrices.matrices.MatrixBase` method), [588](#)  
Ci (class in `sympy.functions.special.error_functions`), [385](#)  
Circle (class in `sympy.geometry.ellipse`), [493](#)  
circumcenter (`sympy.geometry.polygon.RegularPolygon`.cofactor() attribute), [504](#)  
circumcenter (`sympy.geometry.polygon.Triangle` attribute), [510](#)  
circumcircle (`sympy.geometry.polygon.RegularPolygon` attribute), [504](#)  
circumcircle (`sympy.geometry.polygon.Triangle` attribute), [511](#)  
circumference (`sympy.geometry.ellipse.Circle` attribute), [493](#)  
circumference (`sympy.geometry.ellipse.Ellipse` attribute), [484](#)  
circumradius (`sympy.geometry.polygon.RegularPolygon` attribute), [505](#)  
circumradius (`sympy.geometry.polygon.Triangle` attribute), [511](#)  
CL (`sympy.matrices.sparse.SparseMatrix` attribute), [632](#)  
class\_key() (`sympy.core.add.Add` class method), [116](#)  
class\_key() (`sympy.core.basic.Basic` class method), [65](#)  
classify\_diop() (in module `sympy.solvers.diophantine`), [1065](#)  
classify\_ode() (in module `sympy.solvers.ode`), [985](#)  
classify\_pde() (in module `sympy.solvers.pde`), [1039](#)  
classof() (in module `sympy.matrices.matrices`), [615](#)  
clear() (`sympy.tensor.tensor_.TensorManager` method), [1096](#)  
clear\_denoms() (`sympy.polys.polyclasses.DMP` method), [763](#)  
clear\_denoms() (in `sympy.polys.polytools.Poly` method), [683](#)  
CodeGen (class in `sympy.utilitiescodegen`), [1117](#)  
codegen() (in module `sympy.utilitiescodegen`), [1121](#)  
CodePrinter (class in `sympy.printing.codeprinter`), [867](#)  
CodeWrapper (class in `sympy.utilities.autowrap`), [1112](#)  
codomain (`sympy.categories.CompositeMorphism` attribute), [1173](#)  
codomain (`sympy.categories.Morphism` attribute), [1171](#)  
coeff() (`sympy.core.expr.Expr` method), [83](#)  
coeff() (`sympy.polys.rings.PolyElement` method), [815](#)  
coeff\_monomial() (`sympy.polys.polytools.Poly` method), [684](#)  
coefficients (`sympy.geometry.line.LinearEntity` attribute), [450](#)  
coeffs() (`sympy.polys.numberfields.AlgebraicNumber` method), [720](#)  
coeffs() (`sympy.polys.polyclasses.DMP` method), [763](#)  
coeffs() (`sympy.polys.polytools.Poly` method), [684](#)  
coeffs() (`sympy.polys.rings.PolyElement` method), [815](#)  
CoercionFailed (class in `sympy.polys.polyerrors`), [838](#)  
cofactorMatrix() (`sympy.matrices.matrices.MatrixBase` method), [589](#)  
cofactors() (in module `sympy.polys.polytools`), [671](#)  
cofactors() (`sympy.core.numbers.Number` method), [99](#)  
cofactors() (`sympy.polys.domains.domain.Domain` method), [752](#)  
cofactors() (`sympy.polys.polyclasses.DMP` method), [763](#)  
cofactors() (`sympy.polys.polytools.Poly` method), [685](#)  
Coin() (in module `sympy.stats`), [949](#)  
col() (`sympy.matrices.dense.DenseMatrix` method), [624](#)  
col() (`sympy.matrices.sparse.SparseMatrix` method), [634](#)  
col\_del() (`sympy.matrices.dense.MutableDenseMatrix` method), [626](#)  
col\_del() (`sympy.matrices.sparse.MutableSparseMatrix` method), [640](#)  
col\_insert() (`sympy.matrices.matrices.MatrixBase` method), [589](#)  
col\_join() (`sympy.matrices.matrices.MatrixBase` method), [589](#)  
col\_join() (`sympy.matrices.sparse.MutableSparseMatrix` method), [640](#)  
col\_list() (`sympy.matrices.sparse.SparseMatrix` method), [634](#)

col\_op() (sympy.matrices.dense.MutableDenseMatrix method), 627  
 col\_op() (sympy.matrices.sparse.MutableSparseMatrix method), 641  
 col\_swap() (sympy.matrices.dense.MutableDenseMatrix method), 627  
 col\_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 641  
 collect() (in module sympy.simplify.simplify), 918  
 collect() (sympy.core.expr.Expr method), 85  
 collect\_const() (in module sympy.simplify.simplify), 929  
 collect\_sqrt() (in module sympy.simplify.simplify), 929  
 combsimp() (in module sympy.simplify.simplify), 927  
 combsimp() (sympy.core.expr.Expr method), 85  
 comm\_i2symbol() (sympy.tensor.tensor.\_TensorManager method), 1096  
 comm\_symbols2i() (sympy.tensor.tensor.\_TensorManager method), 1096  
 common\_prefix() (in module sympy.utilities.iterables), 1132  
 common\_suffix() (in module sympy.utilities.iterables), 1132  
 commutative, 61  
 commutative\_diagrams (sympy.categories.Category attribute), 1175  
 Commutator (class in sympy.diffgeom), 1193  
 commutator() (sympy.combinatorics.perm\_groups.PermutationGroup method), 191  
 commutator() (sympy.combinatorics.permutations.PermutationGroup method), 166  
 commutes\_with() (sympy.combinatorics.permutations.PermutationGroup method), 166  
 commutes\_with() (sympy.tensor.tensor.TensorHead method), 1104  
 compare() (in module sympy.series.gruntz), 902  
 compare() (sympy.core.basic.Basic method), 65  
 Complement (class in sympy.sets.sets), 912  
 complement() (sympy.sets.sets.Set method), 904  
 complex, 61  
 ComplexInfinity (class in sympy.core.numbers), 107  
 components (sympy.categories.CompositeMorphism attribute), 1173  
 components() (in module sympy.integrals.heurisch), 548  
 compose() (in module sympy.polys.polytools), 673  
 compose() (sympy.categories.Morphism method), 1171  
 compose() (sympy.polys.polyclasses.DMP method), 763  
 compose() (sympy.polys.polytools.Poly method), 685  
 compose() (sympy.polys.rings.PolyElement method), 815  
 composite, 61  
 CompositeDomain (class in sympy.polys.domains.compositedomain), 757  
 CompositeMorphism (class in sympy.categories), 1173  
 ComputationFailed (class in sympy.polys.polyerrors), 839  
 compute\_known\_facts() (in module sympy.assumptions.ask), 880  
 compute\_leading\_term() (sympy.core.expr.Expr method), 85  
 conclusions (sympy.categories.Diagram attribute), 1177  
 cond (sympy.functions.elementary.piecewise.ExprCondPair attribute), 326  
 condition\_number() (sympy.matrices.matrices.MatrixBase method), 589  
 ConditionalDomain (class in sympy.stats.rv), 981  
 conjugate (class in sympy.functions.elementary.complexes), 322  
 conjugate (sympy.combinatorics.partitions.IntegerPartition attribute), 162  
 conjugate() (sympy.matrices.immutable.ImmutableMatrix method), 645  
 conjugate() (sympy.matrices.matrices.MatrixBase method), 590  
 connect\_to() (sympy.diffgeom.CoordSystem method), 1089  
 conserve\_mpmath\_dps() (in module sympy.utilities.decorator), 1124  
 const() (sympy.polys.rings.PolyElement method), 15  
 constant\_renumber() (in module sympy.solvers.ode), 990  
 constantsimp() (in module sympy.solvers.ode), 991  
 construct\_domain() (in module sympy.polys.constructor), 719  
 contains() (sympy.combinatorics.perm\_groups.PermutationGroup method), 192  
 contains() (sympy.geometry.line.Line method), 457  
 contains() (sympy.geometry.line.LinearEntity method), 451  
 contains() (sympy.geometry.line.Ray method), 459  
 contains() (sympy.geometry.line.Segment method), 462  
 contains() (sympy.geometry.line3d.Line3D method), 465  
 contains() (sympy.geometry.line3d.LinearEntity3D method), 468  
 contains() (sympy.geometry.line3d.Ray3D method), 474  
 contains() (sympy.geometry.line3d.Segment3D method), 477

contains() (sympy.polys.agca.ideals.Ideal method), 741  
contains() (sympy.polys.agca.modules.Module method), 734  
contains() (sympy.polys.polytools.GroebnerBasis method), 715  
contains() (sympy.series.order.Order method), 899  
contains() (sympy.sets.sets.Set method), 904  
content() (in module sympy.polys.polytools), 672  
content() (sympy.polys.polyclasses.DMP method), 763  
content() (sympy.polys.polytools.Poly method), 685  
content() (sympy.polys.rings.PolyElement method), 815  
continued\_fraction\_convergents() (in module sympy.nttheory.continued\_fraction), 267  
continued\_fraction\_iterator() (in module sympy.nttheory.continued\_fraction), 268  
continued\_fraction\_periodic() (in module sympy.nttheory.continued\_fraction), 268  
continued\_fraction\_reduce() (in module sympy.nttheory.continued\_fraction), 269  
ContinuousDomain (class in sympy.stats.crv), 981  
ContinuousPSpace (class in sympy.stats.crv), 981  
ContinuousRV() (in module sympy.stats), 976  
contract\_metric() (sympy.tensor.tensor.TensAdd method), 1106  
contract\_metric() (sympy.tensor.tensor.TensMul method), 1108  
convergence\_statement (sympy.functions.special.hyper.hyper class in sympy.functions.elementary.trigonometric), 407  
convert() (sympy.polys.agca.modules.FreeModule method), 735  
convert() (sympy.polys.agca.modules.Module method), 734  
convert() (sympy.polys.agca.modules.QuotientModule method), 743  
convert() (sympy.polys.agca.modules.SubModule method), 737  
convert() (sympy.polys.domains.domain.Domain method), 752  
convert() (sympy.polys.polyclasses.DMP method), 763  
convert\_from() (sympy.polys.domains.domain.Domain method), 752  
convert\_to\_native\_paths() (in module sympy.utilities.runtests), 1155  
convert\_xor() (in module sympy.parsing.sympy\_parser), 1165  
convex\_hull() (in module sympy.geometry.util), 435  
coord\_function() (sympy.diffgeom.CoordSystem method), 1190  
coord\_functions() (sympy.diffgeom.CoordSystem method), 1190  
coord\_tuple\_transform\_to() (sympy.diffgeom.CoordSystem method), 1190  
coords() (sympy.diffgeom.Point method), 1191  
CoordSystem (class in sympy.diffgeom), 1188  
copy() (sympy.polys.rings.PolyElement method), 816  
copyin\_list() (sympy.matrices.dense.MutableDenseMatrix method), 627  
copyin\_matrix() (sympy.matrices.dense.MutableDenseMatrix method), 628  
core() (in module sympy.nttheory.factor\_), 259  
cornacchia() (in module sympy.solvers.diophantine), 1068  
corners (sympy.combinatorics.polyhedron.Polyhedron attribute), 213  
cos (class in sympy.functions.elementary.trigonometric), 322  
coset\_factor() (sympy.combinatorics.perm\_groups.PermutationGroup method), 193  
coset\_rank() (sympy.combinatorics.perm\_groups.PermutationGroup method), 194  
coset\_unrank() (sympy.combinatorics.perm\_groups.PermutationGroup method), 194  
cosh (class in sympy.functions.elementary.hyperbolic), 323  
cosine\_transform() (in module sympy.integrals.transforms), 526  
CosineTransform (class in sympy.integrals.transforms), 556  
cosh (class in sympy.functions.elementary.hyperbolic), 324  
could\_extract\_minus\_sign() (sympy.core.expr.Expr method), 85  
count() (sympy.core.basic.Basic method), 65  
count\_complex\_roots() (sympy.polys.polyclasses.DMP method), 763  
count\_ops() (in module sympy.core.function), 145  
count\_ops() (sympy.core.basic.Basic method), 65  
count\_ops() (sympy.core.expr.Expr method), 85  
count\_partitions() (sympy.utilities.enumerative.MultisetPartitionTrav method), 1127  
count\_real\_roots() (sympy.polys.polyclasses.DMP method), 763  
count\_roots() (in module sympy.polys.polytools), 676  
count\_roots() (sympy.polys.polytools.Poly method), 685  
CovarDerivativeOp (class in sympy.diffgeom), 1196  
cross() (sympy.matrices.matrices.MatrixBase method), 590  
crt() (in module sympy.nttheory.modular), 260  
crt1() (in module sympy.nttheory.modular), 260  
crt2() (in module sympy.nttheory.modular), 261

csc (class in `sympy.functions.elementary.trigonometric`), `decompose()` (`sympy.polys.polytools.Poly` method), 686  
`cse()` (in module `sympy.simplify.cse_main`), 933  
`current` (`sympy.combinatorics.graycode.GrayCode` attribute), 226  
`Curve` (class in `sympy.geometry.curve`), 478  
`Cycle` (class in `sympy.combinatorics.permutations`), 181  
`cycle.length()` (in module `sympy.ntheory.generate`), 249  
`cycle_list()` (in module `sympy.crypto.crypto`), 272  
`cycle_structure` (`sympy.combinatorics.permutations.Permutation` attribute), 166  
`cycles` (`sympy.combinatorics.permutations.Permutation` attribute), 167  
`cyclic()` (`sympy.combinatorics.generators` static method), 183  
`cyclic_form` (`sympy.combinatorics.permutations.Permutation` attribute), 167  
`cyclic_form` (`sympy.combinatorics.polyhedron.Polyhedron` attribute), 214  
`CyclicGroup()` (in module `sympy.combinatorics.named_groups`), 230  
`cyclotomic_poly()` (in module `sympy.polys.specialpolys`), 723  
`CythonCodeWrapper` (class in `sympy.utilities.autowrap`), 1112

## D

`D` (`sympy.matrices.matrices.MatrixBase` attribute), 581  
`Dagum()` (in module `sympy.stats`), 957  
`DataType` (class in `sympy.utilitiescodegen`), 1117  
`debug()` (in module `sympy.utilities.misc`), 1150  
`debug_decorator()` (in module `sympy.utilities.misc`), 1150  
`decipher_bifid5()` (in module `sympy.crypto.crypto`), 279  
`decipher_bifid6()` (in module `sympy.crypto.crypto`), 280  
`decipher_elgamal()` (in module `sympy.crypto.crypto`), 287  
`decipher_hill()` (in module `sympy.crypto.crypto`), 278  
`decipher_kid_rsa()` (in module `sympy.crypto.crypto`), 283  
`decipher_rsa()` (in module `sympy.crypto.crypto`), 282  
`decipher_vigenere()` (in module `sympy.crypto.crypto`), 276  
`decode_morse()` (in module `sympy.crypto.crypto`), 284  
`decompose()` (in module `sympy.polys.polytools`), 673  
`decompose()` (`sympy.polys.polyclasses.DMP` method), 763  
`default_sort_key()` (in module `sympy.core.compatibility`), 152  
`deflate()` (`sympy.polys.polyclasses.DMP` method), 763  
`deflate()` (`sympy.polys.polytools.Poly` method), 686  
`degree` (`sympy.combinatorics.perm_groups.PermutationGroup` attribute), 194  
`degree()` (in module `sympy.polys.polytools`), 663  
`degree()` (`sympy.polys.polyclasses.DMP` method), 763  
`degree()` (`sympy.polys.polytools.Poly` method), 686  
`degree()` (`sympy.polys.rings.PolyElement` method), 816  
`degree_list()` (in module `sympy.polys.polytools`), 663  
`degree_list()` (`sympy.polys.polyclasses.DMP` method), 763  
`degree_list()` (`sympy.polys.polytools.Poly` method), 686  
`degrees()` (`sympy.polys.rings.PolyElement` method), 816  
`delta` (`sympy.functions.special.hyper.meijerg` attribute), 409  
`deltaintegrate()` (in module `sympy.integrals.deltafunctions`), 546  
`denom()` (`sympy.polys.domains.AlgebraicField` method), 759  
`denom()` (`sympy.polys.domains.domain.Domain` method), 752  
`denom()` (`sympy.polys.domains.ExpressionDomain` method), 761  
`denom()` (`sympy.polys.domains.FractionField` method), 759  
`denom()` (`sympy.polys.domains.ring.Ring` method), 756  
`denom()` (`sympy.polys.polyclasses.DMF` method), 768  
`DenseMatrix` (class in `sympy.matrices.dense`), 623  
`density()` (in module `sympy.stats`), 978  
`dependent()` (in module `sympy.stats.rv`), 983  
`depth()` (`sympy.polys.agca.ideals.Ideal` method), 741  
`Derivative` (class in `sympy.core.function`), 135  
`derived_series()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 195  
`derived_subgroup()` (`sympy.combinatorics.perm_groups.PermutationGroup` method), 195  
`descent()` (in module `sympy.solvers.diophantine`), 1071  
`descents()` (`sympy.combinatorics.permutations.Permutation` method), 167  
`det()` (`sympy.matrices.matrices.MatrixBase` method), 590

det\_bareis() (sympy.matrices.matrices.MatrixBase method), 590  
det\_LU\_decomposition() (sympy.matrices.matrices.MatrixBase method), 590  
diag() (in module sympy.matrices.dense), 616  
diagonal\_solve() (sympy.matrices.matrices.MatrixBase method), 590  
diagonalize() (sympy.matrices.matrices.MatrixBase method), 591  
Diagram (class in sympy.categories), 1176  
DiagramGrid (class in sympy.categories.diagram\_drawing), 1179  
Dict (class in sympy.core.containers), 151  
dict\_merge() (in module sympy.utilities.iterables), 1132  
Die() (in module sympy.stats), 949  
DiePSpace (class in sympy.stats.frv\_types), 982  
diff() (in module sympy.core.function), 138  
diff() (sympy.matrices.matrices.MatrixBase method), 591  
diff() (sympy.polys.polyclasses.DMP method), 763  
diff() (sympy.polys.polytools.Poly method), 687  
diff() (sympy.polys.rings.PolyElement method), 816  
Differential (class in sympy.diffgeom), 1193  
digamma() (in module sympy.functions.special.gamma\_functions), 365  
digit\_2txt (in module sympy.printing.pretty.pretty\_symbology), 868  
dihedral() (sympy.combinatorics.generators static method), 184  
DihedralGroup() (in module sympy.combinatorics.named\_groups), 231  
diop\_bf\_DN() (in module sympy.solvers.diophantine), 1068  
diop\_DN() (in module sympy.solvers.diophantine), 1067  
diop\_general\_pythagorean() (in module sympy.solvers.diophantine), 1072  
diop\_general\_sum\_of\_squares() (in module sympy.solvers.diophantine), 1072  
diop\_linear() (in module sympy.solvers.diophantine), 1066  
diop\_quadratic() (in module sympy.solvers.diophantine), 1067  
diop\_solve() (in module sympy.solvers.diophantine), 1065  
diop\_ternary\_quadratic() (in module sympy.solvers.diophantine), 1070  
diop\_ternary\_quadratic\_normal() (in module sympy.solvers.diophantine), 1076  
diophantine() (in module sympy.solvers.diophantine), 1064  
DiracDelta (class in sympy.functions.special.delta\_functions), 358  
direction\_cosine (sympy.geometry.line3d.LinearEntity3D attribute), 468  
direction\_cosine() (sympy.geometry.point3d.Point3D method), 444  
direction\_ratio (sympy.geometry.line3d.LinearEntity3D attribute), 468  
direction\_ratio() (sympy.geometry.point3d.Point3D method), 445  
DirectProduct() (in module sympy.combinatorics.group\_constructs), 238  
dirichlet\_eta (class in sympy.functions.special.zeta\_functions), 402  
DiscreteUniform() (in module sympy.stats), 948  
discriminant() (in module sympy.polys.polytools), 667  
discriminant() (sympy.polys.polyclasses.DMP method), 763  
discriminant() (sympy.polys.polytools.Poly method), 687  
dispersion() (in module sympy.polys.dispersion), 667, 729  
dispersion() (sympy.polys.polytools.Poly method), 687  
dispersionset() (in module sympy.polys.dispersion), 668, 728  
dispersionset() (sympy.polys.polytools.Poly method), 688  
distance() (sympy.geometry.line.Line method), 457  
distance() (sympy.geometry.line.Ray method), 460  
distance() (sympy.geometry.line.Segment method), 462  
distance() (sympy.geometry.line3d.Line3D method), 465  
distance() (sympy.geometry.line3d.Ray3D method), 474  
distance() (sympy.geometry.line3d.Segment3D method), 477  
distance() (sympy.geometry.plane.Plane method), 517  
distance() (sympy.geometry.point.Point method), 438  
distance() (sympy.geometry.point3d.Point3D method), 445  
distance() (sympy.geometry.polygon.Polygon method), 498  
div() (in module sympy.polys.polytools), 665

div() (sympy.polys.domains.domain.Domain method), 752  
 div() (sympy.polys.domains.field.Field method), 755  
 div() (sympy.polys.domains.ring.Ring method), 756  
 div() (sympy.polys.polyclasses.DMP method), 763  
 div() (sympy.polys.polytools.Poly method), 689  
 div() (sympy.polys.rings.PolyElement method), 816  
 divisible() (in module sympy.solvers.diophantine), 1074  
 divisor\_count() (in module sympy.nttheory.factor\_), 258  
 divisors() (in module sympy.nttheory.factor\_), 258  
 DMF (class in sympy.polys.polyclasses), 767  
 DMP (class in sympy.polys.polyclasses), 762  
 dmp\_abs() (in module sympy.polys.densearith), 783  
 dmp\_add() (in module sympy.polys.densearith), 783  
 dmp\_add\_ground() (in module sympy.polys.densearith), 781  
 dmp\_add\_mul() (in module sympy.polys.densearith), 784  
 dmp\_add\_term() (in module sympy.polys.densearith), 780  
 dmp\_apply\_pairs() (in module sympy.polys.densebasic), 780  
 dmp\_cancel() (in module sympy.polys.euclidtools), 832  
 dmp\_clear\_denoms() (in module sympy.polys.densetools), 796  
 dmp\_compose() (in module sympy.polys.densetools), 794  
 dmp\_content() (in module sympy.polys.euclidtools), 831  
 dmp\_convert() (in module sympy.polys.densebasic), 773  
 dmp\_copy() (in module sympy.polys.densebasic), 772  
 dmp\_deflate() (in module sympy.polys.densebasic), 778  
 dmp\_degree() (in module sympy.polys.densebasic), 770  
 dmp\_degree\_in() (in module sympy.polys.densebasic), 771  
 dmp\_degree\_list() (in module sympy.polys.densebasic), 771  
 dmp\_diff() (in module sympy.polys.densetools), 789  
 dmp\_diff\_eval\_in() (in module sympy.polys.densetools), 790  
 dmp\_diff\_in() (in module sympy.polys.densetools), 789  
 dmp\_discriminant() (in module sympy.polys.euclidtools), 829  
 dmp\_div() (in module sympy.polys.densearith), 786  
 dmp\_eject() (in module sympy.polys.densebasic), 779  
 dmp\_euclidean\_prs() (in module sympy.polys.euclidtools), 825  
 dmp\_eval() (in module sympy.polys.densetools), 789  
 dmp\_eval\_in() (in module sympy.polys.densetools), 789  
 dmp\_eval\_tail() (in module sympy.polys.densetools), 790  
 dmp\_exclude() (in module sympy.polys.densebasic), 778  
 dmp\_expand() (in module sympy.polys.densearith), 788  
 dmp\_exquo() (in module sympy.polys.densearith), 787  
 dmp\_exquo\_ground() (in module sympy.polys.densearith), 782  
 dmp\_ext\_factor() (in module sympy.polys.factortools), 836  
 dmp\_factor\_list() (in module sympy.polys.factortools), 836  
 dmp\_factor\_list\_include() (in module sympy.polys.factortools), 836  
 dmp\_ff\_div() (in module sympy.polys.densearith), 786  
 dmp\_ff\_prs\_gcd() (in module sympy.polys.euclidtools), 829  
 dmp\_from\_dict() (in module sympy.polys.densebasic), 776  
 dmp\_from\_sympy() (in module sympy.polys.densebasic), 773  
 dmp\_gcd() (in module sympy.polys.euclidtools), 831  
 dmp\_gcdex() (in module sympy.polys.euclidtools), 824  
 dmp\_ground() (in module sympy.polys.densebasic), 775  
 dmp\_ground\_content() (in module sympy.polys.densetools), 792  
 dmp\_ground\_extract() (in module sympy.polys.densetools), 793  
 dmp\_ground\_LC() (in module sympy.polys.densebasic), 770  
 dmp\_ground\_monic() (in module sympy.polys.densetools), 791  
 dmp\_ground\_nth() (in module sympy.polys.densebasic), 774  
 dmp\_ground\_p() (in module sympy.polys.densebasic), 775  
 dmp\_ground\_primitive() (in module sympy.polys.densetools), 792  
 dmp\_ground\_TC() (in module sympy.polys.densebasic), 770  
 dmp\_ground\_trunc() (in module sympy.polys.densetools), 791  
 dmp\_grounds() (in module sympy.polys.densebasic), 775  
 dmp\_half\_gcdex() (in module sympy.polys.euclidtools), 824

dmp\_include() (in module `sympy.polys.densebasic`), 779  
dmp\_inflate() (in module `sympy.polys.densebasic`), 778  
dmp\_inject() (in module `sympy.polys.densebasic`), 779  
dmp\_inner\_gcd() (in module `sympy.polys.euclidtools`), 831  
dmp\_inner\_subresultants() (in module `sympy.polys.euclidtools`), 827  
dmp\_integrate() (in module `sympy.polys.densetools`), 788  
dmp\_integrate\_in() (in module `sympy.polys.densetools`), 788  
dmp\_invert() (in module `sympy.polys.euclidtools`), 824  
dmp\_irreducible\_p() (in module `sympy.polys.factortools`), 836  
dmp\_l1\_norm() (in module `sympy.polys.densearith`), 788  
dmp\_LC() (in module `sympy.polys.densebasic`), 769  
dmp\_lcm() (in module `sympy.polys.euclidtools`), 831  
dmp\_lift() (in module `sympy.polys.densetools`), 795  
dmp\_list\_terms() (in module `sympy.polys.densebasic`), 780  
dmp\_max\_norm() (in module `sympy.polys.densearith`), 787  
dmp\_mul() (in module `sympy.polys.densearith`), 784  
dmp\_mul\_ground() (in module `sympy.polys.densearith`), 782  
dmp\_mul\_term() (in module `sympy.polys.densearith`), 781  
dmp\_multi\_deflate() (in module `sympy.polys.densebasic`), 778  
dmp\_neg() (in module `sympy.polys.densearith`), 783  
dmp\_negative\_p() (in module `sympy.polys.densebasic`), 776  
dmp\_nest() (in module `sympy.polys.densebasic`), 777  
dmp\_normal() (in module `sympy.polys.densebasic`), 772  
dmp\_nth() (in module `sympy.polys.densebasic`), 773  
dmp\_one() (in module `sympy.polys.densebasic`), 774  
dmp\_one\_p() (in module `sympy.polys.densebasic`), 774  
dmp\_pdiv() (in module `sympy.polys.densearith`), 785  
dmp\_permute() (in module `sympy.polys.densebasic`), 777  
dmp\_pexquo() (in module `sympy.polys.densearith`), 785  
dmp\_positive\_p() (in module `sympy.polys.densebasic`), 776  
dmp\_pow() (in module `sympy.polys.densearith`), 784  
dmp\_pquo() (in module `sympy.polys.densearith`), 785  
dmp\_prem() (in module `sympy.polys.densearith`), 785  
dmp\_primitive() (in module `sympy.polys.euclidtools`), 832  
dmp\_primitive\_prs() (in module `sympy.polys.euclidtools`), 825  
dmp\_prs\_resultant() (in module `sympy.polys.euclidtools`), 827  
dmp\_qq\_collins\_resultant() (in module `sympy.polys.euclidtools`), 828  
dmp\_qq\_heu\_gcd() (in module `sympy.polys.euclidtools`), 830  
dmp\_quo() (in module `sympy.polys.densearith`), 787  
dmp\_quo\_ground() (in module `sympy.polys.densearith`), 782  
dmp\_raise() (in module `sympy.polys.densebasic`), 777  
dmp\_rem() (in module `sympy.polys.densearith`), 786  
dmp\_resultant() (in module `sympy.polys.euclidtools`), 829  
dmp\_revert() (in module `sympy.polys.densetools`), 796  
dmp\_rr\_div() (in module `sympy.polys.densearith`), 786  
dmp\_rr\_prs\_gcd() (in module `sympy.polys.euclidtools`), 829  
dmp\_slice() (in module `sympy.polys.densebasic`), 780  
dmp\_sqr() (in module `sympy.polys.densearith`), 784  
dmp\_strip() (in module `sympy.polys.densebasic`), 771  
dmp\_sub() (in module `sympy.polys.densearith`), 783  
dmp\_sub\_ground() (in module `sympy.polys.densearith`), 781  
dmp\_sub\_mul() (in module `sympy.polys.densearith`), 784  
dmp\_sub\_term() (in module `sympy.polys.densearith`), 781  
dmp\_subresultants() (in module `sympy.polys.euclidtools`), 827  
dmp\_swap() (in module `sympy.polys.densebasic`), 777  
dmp\_TC() (in module `sympy.polys.densebasic`), 769  
dmp\_terms\_gcd() (in module `sympy.polys.densebasic`), 779  
dmp\_to\_dict() (in module `sympy.polys.densebasic`), 776  
dmp\_to\_tuple() (in module `sympy.polys.densebasic`), 772  
dmp\_trial\_division() (in module `sympy.polys.factortools`), 832  
dmp\_true\_LT() (in module `sympy.polys.densebasic`), 770  
dmp\_trunc() (in module `sympy.polys.densetools`), 790  
dmp\_validate() (in module `sympy.polys.densebasic`), 771  
dmp\_zero() (in module `sympy.polys.densebasic`), 774  
dmp\_zero\_p() (in module `sympy.polys.densebasic`), 774  
dmp\_zeros() (in module `sympy.polys.densebasic`), 775

dmp_zz_collins_resultant()	(in module <code>sympy.polys.euclidtools</code> ), 828	module	<code>dotprint()</code> (in module <code>sympy.printing.dot</code> ), 871
dmp_zz_diophantine()	(in module <code>sympy.polys.factorTools</code> ), 835	module	<code>double_coset_can_rep()</code> (in module <code>sympy.combinatorics.tensor_can</code> ), 241
dmp_zz_factor()	(in module <code>sympy.polys.factorTools</code> ), 835	module	<code>draw()</code> ( <code>sympy.categories.diagram_drawing.XypicDiagramDrawer</code> method), 1185
dmp_zz_heu_gcd()	(in module <code>sympy.polys.euclidtools</code> ), 830	module	<code>drop()</code> ( <code>sympy.polys.rings.PolyRing</code> method), 814
dmp_zz_mignotte_bound()	(in module <code>sympy.polys.factorTools</code> ), 832	module	<code>drop_to_ground()</code> ( <code>sympy.polys.rings.PolyRing</code> method), 814
dmp_zz_modular_resultant()	(in module <code>sympy.polys.euclidtools</code> ), 828	module	<code>dsolve()</code> (in module <code>sympy.solvers.ode</code> ), 984
dmp_zz_wang()	(in module <code>sympy.polys.factorTools</code> ), 835	module	<code>dtype</code> ( <code>sympy.polys.agca.modules.FreeModule</code> attribute), 735
dmp_zz_wang_hensel_lifting()	(in module <code>sympy.polys.factorTools</code> ), 835	module	<code>dtype</code> ( <code>sympy.polys.agca.modules.QuotientModule</code> attribute), 743
dmp_zz_wang_lead_coeffs()	(in module <code>sympy.polys.factorTools</code> ), 835	module	<code>dtype</code> ( <code>sympy.polys.domains.AlgebraicField</code> attribute), 759
dmp_zz_wang_non_divisors()	(in module <code>sympy.polys.factorTools</code> ), 834	module	<code>dtype</code> ( <code>sympy.polys.domains.ExpressionDomain</code> attribute), 761
dmp_zz_wang_test_points()	(in module <code>sympy.polys.factorTools</code> ), 834	module	<code>dual()</code> ( <code>sympy.matrices.matrices.MatrixBase</code> method), 592
doctest()	(in module <code>sympy.utilities.runtests</code> ), 1155	module	<code>Dummy</code> (class in <code>sympy.core.symbol</code> ), 96
doctest_depends_on()	(in module <code>sympy.utilities.decorator</code> ), 1124	module	<code>dummy_eq()</code> ( <code>sympy.core.basic.Basic</code> method), 66
doit()	( <code>sympy.core.basic.Basic</code> method), 65	module	<code>DummyWrapper</code> (class in <code>sympy.utilities.autowrap</code> ), 1112
doit()	( <code>sympy.functions.elementary.piecewise.Piecewise</code> method), 332	module	<code>dump_c()</code> ( <code>sympy.utilities.autowrap.UfuncifyCodeWrapper</code> method), 1112
doit()	( <code>sympy.integrals.integrals.Integral</code> method), 552	module	<code>dump_c()</code> ( <code>sympy.utilities.codegen.CCodeGen</code> method), 1118
doit()	( <code>sympy.integrals.transforms.IntegralTransform</code> method), 555	module	<code>dump_code()</code> ( <code>sympy.utilities.codegen.CodeGen</code> method), 1117
doit()	( <code>sympy.series.limits.Limit</code> method), 897	module	<code>dump_f95()</code> ( <code>sympy.utilities.codegen.FCodeGen</code> method), 1119
doit_numerically()	( <code>sympy.core.function.Derivative</code> method), 138	module	<code>dump_h()</code> ( <code>sympy.utilities.codegen.CCodeGen</code> method), 1119
Domain	(class in <code>sympy.polys.domains.domain</code> ), 752	module	<code>dump_h()</code> ( <code>sympy.utilities.codegen.FCodeGen</code> method), 1120
domain	( <code>sympy.categories.CompositeMorphism</code> attribute), 1174	module	<code>dump_m()</code> ( <code>sympy.utilities.codegen.OctaveCodeGen</code> method), 1120
domain	( <code>sympy.categories.Morphism</code> attribute), 1172	module	<code>dump_pyx()</code> ( <code>sympy.utilities.autowrap.CythonCodeWrapper</code> method), 1112
domain	( <code>sympy.polys.polytools.Poly</code> attribute), 690	module	<code>dup_content()</code> (in module <code>sympy.polys.denseTools</code> ), 791
domain_check()	(in module <code>sympy.solvers.solveset</code> ), 1085	module	<code>dup_cyclotomic_p()</code> (in module <code>sympy.polys.factorTools</code> ), 833
DomainError	(class in <code>sympy.polys.polyerrors</code> ), 839	module	<code>dup_decompose()</code> (in module <code>sympy.polys.denseTools</code> ), 794
doprint()	( <code>sympy.printing.mathematica.MCodePrinter</code> method), 860	module	<code>dup_extract()</code> (in module <code>sympy.polys.denseTools</code> ), 793
doprint()	( <code>sympy.printing.mathml.MathMLPrinter</code> method), 863	module	<code>dup_gf_factor()</code> (in module <code>sympy.polys.factorTools</code> ), 836
doprint()	( <code>sympy.printing.printer.Printer</code> method), 853	module	<code>dup_lshift()</code> (in module <code>sympy.polys.densearith</code> ), 782
dot()	( <code>sympy.geometry.point.Point</code> method), 438	module	<code>dup_mirror()</code> (in module <code>sympy.polys.denseTools</code> ), 793
dot()	( <code>sympy.geometry.point3d.Point3D</code> method), 445	module	<code>dup_monic()</code> (in module <code>sympy.polys.denseTools</code> ), 791
dot()	( <code>sympy.matrices.matrices.MatrixBase</code> method), 592	module	

dup\_primitive() (in module sympy.polys.densetools), 792  
dup\_random() (in module sympy.polys.densebasic), 780  
dup\_real\_imag() (in module sympy.polys.densetools), 793  
dup\_reverse() (in module sympy.polys.densebasic), 772  
dup\_rshift() (in module sympy.polys.densearith), 783  
dup\_scale() (in module sympy.polys.densetools), 794  
dup\_shift() (in module sympy.polys.densetools), 794  
dup\_sign\_variations() (in module sympy.polys.densetools), 795  
dup\_transform() (in module sympy.polys.densetools), 794  
dup\_zz\_cyclotomic\_factor() (in module sympy.polys.factor tools), 833  
dup\_zz\_cyclotomic\_poly() (in module sympy.polys.factor tools), 833  
dup\_zz\_factor() (in module sympy.polys.factor tools), 834  
dup\_zz\_factor\_sqf() (in module sympy.polys.factor tools), 834  
dup\_zz\_hensel\_lift() (in module sympy.polys.factor tools), 833  
dup\_zz\_hensel\_step() (in module sympy.polys.factor tools), 832  
dup\_zz\_irreducible\_p() (in module sympy.polys.factor tools), 833  
dup\_zz\_zassenhaus() (in module sympy.polys.factor tools), 833

**E**

E() (in module sympy.stats), 977  
E1() (in module sympy.functions.special.error\_functions), 382  
EC() (sympy.polys.polytools.Poly method), 679  
eccentricity (sympy.geometry.ellipse.Ellipse attribute), 484  
edges (sympy.combinatorics.polyhedron.Polyhedron attribute), 214  
edges() (sympy.combinatorics.prufer.Prufer static method), 216  
egyptian.fraction() (in module sympy.nttheory.egyptian\_fraction), 270  
Ei (class in sympy.functions.special.error\_functions), 378  
eigenvals() (sympy.matrices.matrices.MatrixBase method), 592  
eigenvecs() (sympy.matrices.matrices.MatrixBase method), 592  
Eijk() (in module sympy.functions.special.tensor\_functions), 424  
eject() (sympy.polys.polyclasses.DMP method), 763  
eject() (sympy.polys.polytools.Poly method), 690  
elgamal\_private\_key() (in module sympy.crypto.crypto), 287  
elgamal\_public\_key() (in module sympy.crypto.crypto), 286  
Ellipse (class in sympy.geometry.ellipse), 482  
elliptic\_e (class in sympy.functions.special.elliptic\_integrals), 411  
elliptic\_f (class in sympy.functions.special.elliptic\_integrals), 410  
elliptic\_k (class in sympy.functions.special.elliptic\_integrals), 410  
elliptic\_pi (class in sympy.functions.special.elliptic\_integrals), 411  
EM() (sympy.polys.polytools.Poly method), 679  
emptyPrinter() (sympy.printing.repr.ReprPrinter method), 864  
EmptySet (class in sympy.sets.sets), 913  
encipher\_affine() (in module sympy.crypto.crypto), 273  
encipher\_bifid5() (in module sympy.crypto.crypto), 278  
encipher\_bifid6() (in module sympy.crypto.crypto), 280  
encipher\_bifid7() (in module sympy.crypto.crypto), 281  
encipher\_elgamal() (in module sympy.crypto.crypto), 287  
encipher\_hill() (in module sympy.crypto.crypto), 276  
encipher\_kid\_rsa() (in module sympy.crypto.crypto), 283  
encipher\_rsa() (in module sympy.crypto.crypto), 282  
encipher\_shift() (in module sympy.crypto.crypto), 273  
encipher\_substitution() (in module sympy.crypto.crypto), 274  
encipher\_vigenere() (in module sympy.crypto.crypto), 275  
encloses() (sympy.geometry.entity.GeometryEntity method), 432  
encloses\_point() (sympy.geometry.ellipse.Ellipse method), 485  
encloses\_point() (sympy.geometry.polygon.Polygon method), 499  
encloses\_point() (sympy.geometry.polygon.RegularPolygon method), 505  
encode\_morse() (in module sympy.crypto.crypto), 284  
end (sympy.sets.sets.Interval attribute), 908  
enum\_all() (sympy.utilities.enumerative.MultisetPartitionTraverser method), 1128  
enum\_large() (sympy.utilities.enumerative.MultisetPartitionTraverser method), 1128

enum\_range() (sympy.utilities.enumerative.MultisetPartition),  
     method), 1129  
 enum\_small() (sympy.utilities.enumerative.MultisetPartition),  
     method), 1129  
 EPath (class in sympy.simplify.epathtools), 936  
 epath() (in module sympy.simplify.epathtools), 937  
 Eq (in module sympy.core.relational), 118  
 eq() (sympy.polys.agca.modules.QuotientModuleElement),  
     method), 745  
 equal() (sympy.geometry.line.Line method), 457  
 Equality (class in sympy.core.relational), 120  
 equals() (sympy.core.expr.Expr method), 85  
 equals() (sympy.core.relational.Relational method),  
     119  
 equals() (sympy.geometry.line.Ray method), 460  
 equals() (sympy.geometry.line3d.Line3D method),  
     465  
 equals() (sympy.geometry.line3d.Ray3D method),  
     474  
 equals() (sympy.matrices.dense.DenseMatrix  
     method), 625  
 equals() (sympy.matrices.expressions.MatrixExpr  
     method), 648  
 equals() (sympy.matrices.immutable.ImmutableMatrix  
     method), 646  
 equation() (sympy.geometry.ellipse.Circle method),  
     494  
 equation() (sympy.geometry.ellipse.Ellipse method),  
     485  
 equation() (sympy.geometry.line.Line method), 458  
 equation() (sympy.geometry.line3d.Line3D method),  
     465  
 equation() (sympy.geometry.plane.Plane method),  
     518  
 Equivalent (class in sympy.logic.boolalg), 569  
 equivalent() (in module sympy.solvers.diophantine),  
     1075  
 erf (class in sympy.functions.special.error\_functions),  
     369  
 erf2 (class in sympy.functions.special.error\_functions),  
     372  
 erf2inv (class in sympy.functions.special.error\_functions),  
     375  
 erfc (class in sympy.functions.special.error\_functions),  
     370  
 erfcinv (class in sympy.functions.special.error\_functions),  
     374  
 erfi (class in sympy.functions.special.error\_functions),  
     371  
 erfinv (class in sympy.functions.special.error\_functions),  
     373  
 Erlang() (in module sympy.stats), 957  
 ET() (sympy.polys.polytools.Poly method), 680  
  
 Function (sympy.functions.special.hyper.hyper attribute),  
     407  
 Function (class in sympy.functions.combinatorial.numbers),  
     347  
 euler\_equations() (in module sympy.calculus.euler),  
     1165  
 euler\_maclaurin() (sympy.concrete.summations.Sum  
     method), 290  
 EulerGamma (class in sympy.core.numbers), 109  
 eval() (sympy.assumptions.assume.Predicate  
     method), 882  
 eval() (sympy.functions.special.tensor\_functions.KroneckerDelta  
     class method), 425  
 eval() (sympy.polys.polyclasses.DMP method), 763  
 eval() (sympy.polys.polytools.Poly method), 690  
 eval\_expr() (in module sympy.parsing.sympy\_parser),  
     1162  
 eval\_levi civita() (in module  
     sympy.functions.special.tensor\_functions),  
     424  
 evalf() (sympy.core.evalf.EvalfMixin method), 149  
 evalf() (sympy.geometry.point.Point method), 438  
 evalf() (sympy.geometry.point3d.Point3D method),  
     445  
 evalf() (sympy.matrices.matrices.MatrixBase  
     method), 592  
 evalf() (sympy.polys.domains.domain.Domain  
     method), 752  
 EvalfMixin (class in sympy.core.evalf), 149  
 EvaluationFailed (class in sympy.polys.polyerrors),  
     838  
 even, 61  
 evolve() (sympy.geometry.ellipse.Ellipse method),  
     486  
 ExactQuotientFailed (class in  
     sympy.polys.polyerrors), 838  
 exclude() (sympy.polys.polyclasses.DMP method),  
     763  
 exclude() (sympy.polys.polytools.Poly method), 691  
 exec\_() (in module sympy.core.compatibility), 153  
 exp (class in sympy.functions.elementary.exponential),  
     325  
 exp() (sympy.matrices.matrices.MatrixBase  
     method), 592  
 Exp1 (class in sympy.core.numbers), 108  
 exp\_polar (class in sympy.functions.elementary.exponential),  
     325  
 expand() (in module sympy.core.function), 142  
 expand() (sympy.core.expr.Expr method), 85  
 expand() (sympy.matrices.matrices.MatrixBase  
     method), 592  
 expand\_complex() (in module sympy.core.function),  
     147  
 expand\_func() (in module sympy.core.function), 146

expand\_log() (in module sympy.core.function), 146  
expand\_mul() (in module sympy.core.function), 146  
expand\_multinomial() (in module sympy.core.function), 147  
expand\_power\_base() (in module sympy.core.function), 148  
expand\_power\_exp() (in module sympy.core.function), 148  
expand\_trig() (in module sympy.core.function), 147  
expint (class in sympy.functions.special.error\_functions), 380  
Exponential() (in module sympy.stats), 958  
Expr (class in sympy.core.expr), 74  
expr (sympy.core.function.Lambda attribute), 134  
expr (sympy.core.function.Subs attribute), 142  
expr (sympy.functions.elementary.piecewise.ExprCondPair attribute), 326  
ExprCondPair (class in sympy.functions.elementary.piecewise), 326  
ExpressionDomain (class in sympy.polys.domains), 761  
ExpressionDomain.Expression (class in sympy.polys.domains), 761  
ExprWithIntLimits (class in sympy.concrete.expr\_with\_intlimits), 297  
ExprWithLimits (class in sympy.concrete.expr\_with\_limits), 295  
exquo() (in module sympy.polys.polytools), 665  
exquo() (sympy.polys.domains.domain.Domain method), 752  
exquo() (sympy.polys.domains.field.Field method), 755  
exquo() (sympy.polys.domains.ring.Ring method), 756  
exquo() (sympy.polys.polyclasses.DMF method), 768  
exquo() (sympy.polys.polyclasses.DMP method), 764  
exquo() (sympy.polys.polytools.Poly method), 691  
exquo\_ground() (sympy.polys.polyclasses.DMP method), 764  
exquo\_ground() (sympy.polys.polytools.Poly method), 691  
extend() (sympy.nttheory.generate.Sieve method), 244  
extend() (sympy.plotting.plot.Plot method), 874  
extend\_to\_no() (sympy.nttheory.generate.Sieve method), 245  
extended\_euclid() (in module sympy.solvers.diophantine), 1074  
extended\_real, 61  
exterior\_angle (sympy.geometry.polygon.RegularPolygon attribute), 506  
extract() (sympy.matrices.matrices.MatrixBase method), 593  
extract() (sympy.matrices.sparse.SparseMatrix method), 635  
extract\_additively() (sympy.core.expr.Expr method), 85  
extract\_branch\_factor() (sympy.core.expr.Expr method), 86  
extract\_leading\_order() (sympy.core.add.Add method), 117  
extract\_multiplicatively() (sympy.core.expr.Expr method), 86  
ExtraneousFactors (class in sympy.polys.polyerrors), 838  
eye() (in module sympy.matrices.dense), 616  
eye() (sympy.matrices.dense.DenseMatrix class method), 625  
eye() (sympy.matrices.sparse.SparseMatrix class method), 635

## F

F2PyCodeWrapper (class in sympy.utilities.autowrap), 1112  
faces (sympy.combinatorics.polyhedron.Polyhedron attribute), 214  
factor() (in module sympy.polys.polytools), 675  
factor() (sympy.core.expr.Expr method), 87  
factor\_list() (in module sympy.polys.polytools), 675  
factor\_list() (sympy.polys.polyclasses.DMP method), 764  
factor\_list() (sympy.polys.polytools.Poly method), 691  
factor\_list\_include() (sympy.polys.polyclasses.DMP method), 764  
factor\_list\_include() (sympy.polys.polytools.Poly method), 692  
factor.terms() (in module sympy.core.exprtools), 159  
factorial (class in sympy.functions.combinatorial.factorials), 347  
factorial() (sympy.polys.domains.FractionField method), 759  
factorial() (sympy.polys.domains.PolynomialRing method), 758  
factorial2 (class in sympy.functions.combinatorial.factorials), 349  
factorial\_notation() (in module sympy.parsing.sympy\_parser), 1165  
factoring\_visitor() (in module sympy.utilities.enumerative), 1126  
factorint() (in module sympy.nttheory.factor\_), 255  
factors() (sympy.core.numbers.Rational method), 103  
FallingFactorial (class in sympy.functions.combinatorial.factorials), 350  
fcode() (in module sympy.printing.fcode), 856

FCodeGen (class in `sympy.utilities.codegen`), 1119  
FCodePrinter (class in `sympy.printing.fcode`), 858  
fdiff() (`sympy.core.function.Function` method), 141  
fdiff() (`sympy.functions.elementary.complexes.Abs` method), 314  
fdiff() (`sympy.functions.elementary.exponential.exp` method), 325  
fdiff() (`sympy.functions.elementary.exponential.LambertW` method), 328  
fdiff() (`sympy.functions.elementary.exponential.log` method), 329  
fdiff() (`sympy.functions.elementary.hyperbolic.sinh` method), 336  
FDistribution() (in module `sympy.stats`), 959  
fglm() (`sympy.polys.polytools.GroebnerBasis` method), 715  
fibonacci (class in `sympy.functions.combinatorial.numbers`), 350  
Field (class in `sympy.polys.domains.field`), 755  
field\_isomorphism() (in module `sympy.polys.numberfields`), 720  
fill() (`sympy.matrices.dense.MutableDenseMatrix` method), 628  
fill() (`sympy.matrices.sparse.MutableSparseMatrix` method), 641  
filter\_symbols() (in module `sympy.utilities.iterables`), 1132  
find() (`sympy.core.basic.Basic` method), 66  
find\_DN() (in module `sympy.solvers.diophantine`), 1070  
find\_executable() (in module `sympy.utilities.misc`), 1151  
findrecur() (`sympy.concrete.summations.Sum` method), 291  
finite, 61  
finite\_diff\_weights() (in module `sympy.calculus.finite_diff`), 1169  
FiniteDomain (class in `sympy.stats.frv`), 981  
FiniteField (class in `sympy.polys.domains`), 757  
FinitePSpace (class in `sympy.stats.frv`), 981  
FiniteRV() (in module `sympy.stats`), 950  
FiniteSet (class in `sympy.sets.sets`), 910  
FisherZ() (in module `sympy.stats`), 960  
FlagError (class in `sympy.polys.polyerrors`), 839  
flatten() (in module `sympy.utilities.iterables`), 1132  
flatten() (`sympy.categories.CompositeMorphism` method), 1174  
flatten() (`sympy.core.add.Add` class method), 117  
flatten() (`sympy.core.mul.Mul` class method), 113  
Float (class in `sympy.core.numbers`), 99  
floor (class in `sympy.functions.elementary.integers`), 326  
foci (`sympy.geometry.ellipse.Ellipse` attribute), 486  
focus\_distance (`sympy.geometry.ellipse.Ellipse` attribute), 486  
fourier\_transform() (in module `sympy.integrals.transforms`), 525  
FourierTransform (class in `sympy.integrals.transforms`), 555  
frac (in module `sympy.printing.pretty.pretty_symbology`), 869  
frac\_field() (`sympy.polys.domains.domain.Domain` method), 752  
frac\_unify() (`sympy.polys.polyclasses.DMF` method), 768  
fraction() (in module `sympy.simplify.simplify`), 923  
FractionField (class in `sympy.polys.domains`), 759  
Frechet() (in module `sympy.stats`), 961  
free\_module() (`sympy.polys.domains.ring.Ring` method), 756  
free\_symbols (`sympy.concrete.expr_with_limits.ExprWithLimits` attribute), 296  
free\_symbols (`sympy.core.basic.Basic` attribute), 66  
free\_symbols (`sympy.functions.elementary.piecewise.ExprCondPair` attribute), 326  
free\_symbols (`sympy.geometry.curve.Curve` attribute), 480  
free\_symbols (`sympy.integrals.integrals.Integral` attribute), 553  
free\_symbols (`sympy.integrals.transforms.IntegralTransform` attribute), 555  
free\_symbols (`sympy.matrices.matrices.MatrixBase` attribute), 593  
free\_symbols (`sympy.polys.polytools.Poly` attribute), 692  
free\_symbols (`sympy.polys.polytools.PurePoly` attribute), 715  
free\_symbols\_in\_domain (`sympy.polys.polytools.Poly` attribute), 692  
FreeModule (class in `sympy.polys.agca.modules`), 735  
FreeModuleElement (class in `sympy.polys.agca.modules`), 740  
fresnelc (class in `sympy.functions.special.error_functions`), 377  
FresnelIntegral (class in `sympy.functions.special.error_functions`), 376  
fresnels (class in `sympy.functions.special.error_functions`), 376  
from\_AlgebraicField() (`sympy.polys.domains.domain.Domain` method), 752  
from\_AlgebraicField() (`sympy.polys.domains.FractionField` method), 760  
from\_AlgebraicField() (`sympy.polys.domains.IntegerRing` method), 757  
from\_AlgebraicField() (`sympy.polys.domains.PolynomialRing` method), 758

```
from_AlgebraicField() (sympy.polys.domains.RationalField.PolynomialRing())
    method), 759
from_ComplexField() (sympy.polys.domains.domain.Domain
    method), 752
from_dict() (sympy.polys.polyclasses.DMP  class
    method), 764
from_dict() (sympy.polys.polytools.Poly  class
    method), 693
from_expr() (sympy.polys.polytools.Poly  class
    method), 693
from_ExpressionDomain()
    (sympy.polys.domains.domain.Domain
        method), 752
from_ExpressionDomain()
    (sympy.polys.domains.ExpressionDomain
        method), 761
from_FF_gmpy() (sympy.polys.domains.domain.Domain
    method), 752
from_FF_gmpy() (sympy.polys.domains.FiniteField
    method), 757
from_FF_python() (sympy.polys.domains.domain.Domain
    method), 752
from_FF_python() (sympy.polys.domains.FiniteField
    method), 757
from_FractionField() (sympy.polys.domains.domain.Domain
    method), 752
from_FractionField() (sympy.polys.domains.ExpressionDomain
    method), 761
from_FractionField() (sympy.polys.domains.FractionField
    method), 760
from_FractionField() (sympy.polys.domains.PolynomialRing
    method), 758
from_GlobalPolynomialRing()
    (sympy.polys.domains.domain.Domain
        method), 752
from_inversion_vector()
    (sympy.combinatorics.permutations.Permutation
        class method), 167
from_list() (sympy.polys.polyclasses.DMP  class
    method), 764
from_list() (sympy.polys.polytools.Poly  class
    method), 693
from_poly() (sympy.polys.polytools.Poly  class
    method), 693
from_PolynomialRing()
    (sympy.polys.domains.domain.Domain
        method), 752
from_PolynomialRing()
    (sympy.polys.domains.ExpressionDomain
        method), 761
from_PolynomialRing()
    (sympy.polys.domains.FractionField
        method), 760
from_QQ_gmpy() (sympy.polys.domains.AlgebraicField
    method), 759
from_QQ_gmpy() (sympy.polys.domains.domain.Domain
    method), 753
from_QQ_gmpy() (sympy.polys.domains.ExpressionDomain
    method), 761
from_QQ_gmpy() (sympy.polys.domains.FiniteField
    method), 757
from_QQ_gmpy() (sympy.polys.domains.FractionField
    method), 760
from_QQ_gmpy() (sympy.polys.domains.PolynomialRing
    method), 758
from_QQ_python() (sympy.polys.domains.AlgebraicField
    method), 759
from_QQ_python() (sympy.polys.domains.domain.Domain
    method), 753
from_QQ_python() (sympy.polys.domains.ExpressionDomain
    method), 761
from_QQ_python() (sympy.polys.domains.FiniteField
    method), 757
from_QQ_python() (sympy.polys.domains.FractionField
    method), 760
from_QQ_python() (sympy.polys.domains.PolynomialRing
    method), 758
from_RealField() (sympy.polys.domains.AlgebraicField
    method), 759
from_RealField() (sympy.polys.domains.domain.Domain
    method), 753
from_RealField() (sympy.polys.domains.ExpressionDomain
    method), 761
from_RealField() (sympy.polys.domains.FiniteField
    method), 757
from_RealField() (sympy.polys.domains.FractionField
    method), 760
from_RealField() (sympy.polys.domains.PolynomialRing
    method), 758
from_rgs() (sympy.combinatorics.partitions.Partition
    class method), 160
from_sequence() (sympy.combinatorics.permutations.Permutation
    class method), 168
from_sympy() (sympy.polys.domains.AlgebraicField
    method), 759
from_sympy() (sympy.polys.domains.domain.Domain
    method), 753
from_sympy() (sympy.polys.domains.ExpressionDomain
    method), 761
from_sympy() (sympy.polys.domains.FiniteField
    method), 757
from_sympy() (sympy.polys.domains.FractionField
    method), 760
```

from\_sympy() (sympy.polys.domains.PolynomialRing  
method), 758

from\_sympy() (sympy.polys.domains.RealField  
method), 760

from\_sympy\_list() (sympy.polys.polyclasses.DMP  
class method), 764

from\_TIDS\_list() (sympy.tensor.tensor.TensAdd  
static method), 1106

from\_ZZ\_gmpy() (sympy.polys.domains.AlgebraicField  
method), 759

from\_ZZ\_gmpy() (sympy.polys.domains.domain.DomainGamma()  
method), 753

from\_ZZ\_gmpy() (sympy.polys.domains.ExpressionDomain  
method), 761

from\_ZZ\_gmpy() (sympy.polys.domains.FiniteField  
method), 757

from\_ZZ\_gmpy() (sympy.polys.domains.FractionField  
method), 760

from\_ZZ\_gmpy() (sympy.polys.domains.PolynomialRing  
method), 758

from\_ZZ\_python() (sympy.polys.domains.AlgebraicField  
method), 759

from\_ZZ\_python() (sympy.polys.domains.domain.Domain  
method), 753

from\_ZZ\_python() (sympy.polys.domains.ExpressionDomain  
method), 761

from\_ZZ\_python() (sympy.polys.domains.FiniteField  
method), 757

from\_ZZ\_python() (sympy.polys.domains.FractionField  
method), 760

from\_ZZ\_python() (sympy.polys.domains.PolynomialRing  
method), 758

fromiter() (sympy.core.basic.Basic class method), 66

full\_cyclic\_form (sympy.combinatorics.permutations.Permutation  
attribute), 168

fun\_eval() (sympy.tensor.tensor.TensAdd method), 1106

fun\_eval() (sympy.tensor.tensor.TensMul method), 1108

func (sympy.core.basic.Basic attribute), 66

func\_field\_modgcd() (in module sympy.polys.modular)  
gcd(), 842

Function (class in sympy.core.function), 140

function (sympy.concrete.expr\_with\_limits.ExprWithLimits  
attribute), 296

function (sympy.integrals.transforms.IntegralTransform  
attribute), 555

function\_exponentiation() (in module sympy.parsing.sympy\_parser), 1164

function\_variable (sympy.integrals.transforms.IntegralTransform  
attribute), 555

FunctionClass (class in sympy.core.function), 139

FunctionMatrix (class in sympy.matrices.expressions), 650

functions (sympy.geometry.curve.Curve attribute), 480

**G**

G() (in module sympy.printing.pretty.pretty\_symbology), 868

g() (in module sympy.printing.pretty.pretty\_symbology), 868

gamma (class in sympy.functions.special.gamma\_functions), 360

GammaInverse() (in module sympy.stats), 961

gauss\_chebyshev\_t() (in module sympy.integrals.quadrature), 560

gauss\_chebyshev\_u() (in module sympy.integrals.quadrature), 561

gauss\_gen\_laguerre() (in module sympy.integrals.quadrature), 559

gauss\_hermite() (in module sympy.integrals.quadrature), 558

gauss\_jacobi() (in module sympy.integrals.quadrature), 562

gauss\_laguerre() (in module sympy.integrals.quadrature), 557

gauss\_legendre() (in module sympy.integrals.quadrature), 557

gaussian\_reduce() (in module sympy.solvers.diophantine), 1077

gcd() (in module sympy.polys.polytools), 671

gcd() (sympy.core.numbers.Number method), 99

gcd() (sympy.polys.domains.PolynomialRing  
method), 753

gcd() (sympy.polys.domains.field.Field method), 755

gcd() (sympy.polys.domains.PolynomialRing  
method), 758

gcd() (sympy.polys.domains.RealField method), 760

gcd() (sympy.polys.polyclasses.DMP method), 764

gcd() (sympy.polys.polytools.Poly method), 693

gcd\_list() (in module sympy.polys.polytools), 671

gcd\_terms() (in module sympy.core.exprtools), 158

gcdex() (in module sympy.polys.polytools), 666

gcdex() (sympy.polys.domains.domain.Domain  
method), 753

gcdex() (sympy.polys.domains.PolynomialRing  
method), 758

gcdex() (sympy.polys.polytools.Poly method), 693

Ge (in module sympy.core.relational), 119

gegenbauer (class in sympy.functions.special.polynomials), 414

gegenbauer\_poly() (in module sympy.polys.orthopolys), 723

gen (sympy.polys.polytools.Poly attribute), 693

generate() (sympy.combinatorics.perm\_groups.PermutationGroup method),  
method), 195  
get\_field() (sympy.polys.domains.field.Field method),  
755  
generate\_bell() (in module sympy.utilities.iterables), get\_field() (sympy.polys.domains.FiniteField  
method), 757  
1133  
generate\_derangements() (in module sympy.polys.domains.IntegerRing  
sympy.utilities.iterables), 1134  
get\_field() (sympy.polys.domains.IntegerRing  
method), 757  
generate\_dimino() (sympy.combinatorics.perm\_groups.PermutationGroup method),  
196  
get\_field() (sympy.polys.domains.PolynomialRing  
method), 758  
generate\_gray() (sympy.combinatorics.graycode.GrayCode.get\_indices() (in module  
method), 226  
sympy.tensor.index\_methods), 1095  
generate\_involutions() (in module sympy.tensor.index\_methods),  
sympy.utilities.iterables), 1134  
get\_indices() (sympy.tensor.tensor.TensMul method),  
1108  
generate\_oriented\_forest() (in module sympy.utilities.codegen.FCodeGen  
sympy.utilities.iterables), 1135  
get\_interface() (sympy.utilities.codegen.FCodeGen  
method), 1120  
generate\_schreier\_sims()  
(sympy.combinatorics.perm\_groups.PermutationGroup method), 1104  
method), 197  
get\_matrix() (sympy.tensor.tensor.TensExpr  
method), 1160  
generate\_tokens() (in module  
sympy.parsing.sympy\_tokenize), 1163  
get\_mod\_func() (in module sympy.utilities.source),  
generators (sympy.combinatorics.perm\_groups.PermutationGroup  
attribute), 197  
1160  
get\_modulus() (sympy.polys.polytools.Poly method),  
694  
get\_period() (sympy.functions.special.hyper.meijerg  
method), 409  
GeneratorsError (class in sympy.polys.polyerrors),  
839  
GeneratorsNeeded (class in sympy.polys.polyerrors),  
839  
get\_positional\_distance()  
(sympy.combinatorics.permutations.Permutation  
method), 169  
get\_precedence\_distance()  
(sympy.combinatorics.permutations.Permutation  
method), 169  
get\_precedence\_matrix()  
(sympy.combinatorics.permutations.Permutation  
method), 170  
get\_prototype() (sympy.utilities.codegen.CCodeGen  
method), 1119  
get\_resource() (in module sympy.utilities.pkgdata),  
1152  
get\_class() (in module sympy.utilities.source), 1160  
get\_ring() (sympy.polys.domains.AlgebraicField  
method), 759  
get\_comm() (sympy.tensor.tensor.\_TensorManager  
method), 1096  
get\_ring() (sympy.polys.domains.domain.Domain  
method), 753  
get\_ring() (sympy.polys.domains.ExpressionDomain  
method), 761  
get\_ring() (sympy.polys.domains.field.Field method),  
755  
get\_ring() (sympy.polys.domains.FractionField  
method), 760  
get\_ring() (sympy.polys.domains.RealField method),  
761  
get\_ring() (sympy.polys.domains.ring.Ring method),  
756  
get\_ring() (sympy.polys.domains.ring.Ring method),  
761  
get\_segments() (sympy.plotting.plot.LineOver1DRangeSeries  
method), 878  
get\_segments() (sympy.plotting.plot.Parametric2DLineSeries  
method), 879  
get\_ring() (sympy.polys.domains.domain.Domain  
method), 753  
get\_ring() (sympy.polys.domains.ExpressionDomain  
method), 761  
get\_ring() (sympy.polys.domains.field.Field method),  
755  
get\_ring() (sympy.polys.domains.FractionField  
method), 760  
get\_ring() (sympy.polys.domains.RealField method),  
761  
get\_ring() (sympy.polys.domains.ring.Ring method),  
756  
get\_segments() (sympy.plotting.plot.LineOver1DRangeSeries  
method), 878  
get\_segments() (sympy.plotting.plot.Parametric2DLineSeries  
method), 879

get\_subset\_from\_bitstring()  
     (sympy.combinatorics.graycode static  
         method), 229

get\_symmetric\_group\_sgs()         (in module  
     sympy.combinatorics.tensor\_can), 243

get\_sympy\_dir() (in module sympy.utilities.runtests),  
     1156

getn() (sympy.core.expr.Expr method), 87

getO() (sympy.core.expr.Expr method), 87

gf.add() (in module sympy.polys.galoistools), 801

gf.add\_ground() (in module sympy.polys.galoistools),  
     800

gf.add\_mul() (in module sympy.polys.galoistools),  
     801

gf\_berlekamp() (in module sympy.polys.galoistools),  
     810

gf\_cofactors() (in module sympy.polys.galoistools),  
     805

gf\_compose() (in module sympy.polys.galoistools),  
     806

gf\_compose\_mod()                 (in module  
     sympy.polys.galoistools), 807

gf\_crt() (in module sympy.polys.galoistools), 796

gf\_crt1() (in module sympy.polys.galoistools), 797

gf\_crt2() (in module sympy.polys.galoistools), 797

gf\_csolve() (in module sympy.polys.galoistools), 812

gf\_degree() (in module sympy.polys.galoistools), 797

gf\_diff() (in module sympy.polys.galoistools), 806

gf\_div() (in module sympy.polys.galoistools), 802

gf\_eval() (in module sympy.polys.galoistools), 806

gf\_expand() (in module sympy.polys.galoistools), 802

gf\_exquo() (in module sympy.polys.galoistools), 803

gf\_factor() (in module sympy.polys.galoistools), 811

gf\_factor\_sqf() (in module sympy.polys.galoistools),  
     810

gf\_from\_dict() (in module sympy.polys.galoistools),  
     799

gf\_from\_int\_poly()                 (in module  
     sympy.polys.galoistools), 799

gf\_gcd() (in module sympy.polys.galoistools), 804

gf\_gcdex() (in module sympy.polys.galoistools), 805

gf\_int() (in module sympy.polys.galoistools), 797

gf\_irreducible() (in module sympy.polys.galoistools),  
     808

gf\_irreducible\_p()                 (in module  
     sympy.polys.galoistools), 808

gf\_LC() (in module sympy.polys.galoistools), 798

gf\_lcm() (in module sympy.polys.galoistools), 805

gf\_lshift() (in module sympy.polys.galoistools), 803

gf\_monic() (in module sympy.polys.galoistools), 806

gf\_mul() (in module sympy.polys.galoistools), 801

gf\_mul\_ground() (in module sympy.polys.galoistools),  
     800

gf\_multi\_eval() (in module sympy.polys.galoistools),  
     806

gf\_neg() (in module sympy.polys.galoistools), 800

gf\_normal() (in module sympy.polys.galoistools), 798

gf\_pow() (in module sympy.polys.galoistools), 804

gf\_pow\_mod() (in module sympy.polys.galoistools),  
     804

gf\_Qbasis() (in module sympy.polys.galoistools), 810

gf\_Qmatrix() (in module sympy.polys.galoistools),  
     809

gf\_quo() (in module sympy.polys.galoistools), 803

gf\_quo\_ground() (in module sympy.polys.galoistools),  
     800

gf\_random() (in module sympy.polys.galoistools), 807

gf\_rem() (in module sympy.polys.galoistools), 803

gf\_rshift() (in module sympy.polys.galoistools), 804

gf\_shoup() (in module sympy.polys.galoistools), 810

gf\_sqf\_list() (in module sympy.polys.galoistools), 808

gf\_sqf\_p() (in module sympy.polys.galoistools), 808

gf\_sqf\_part() (in module sympy.polys.galoistools), 808

gf\_sqr() (in module sympy.polys.galoistools), 801

gf\_strip() (in module sympy.polys.galoistools), 798

gf\_sub() (in module sympy.polys.galoistools), 801

gf\_sub\_ground() (in module sympy.polys.galoistools),  
     800

gf\_sub\_mul() (in module sympy.polys.galoistools),  
     802

gf\_TC() (in module sympy.polys.galoistools), 798

gf\_to\_dict() (in module sympy.polys.galoistools), 799

gf\_to\_int\_poly() (in module sympy.polys.galoistools),  
     799

gf\_trace\_map() (in module sympy.polys.galoistools),  
     807

gf\_trunc() (in module sympy.polys.galoistools), 798

gf\_value() (in module sympy.polys.galoistools), 811

gf\_zassenhaus() (in module sympy.polys.galoistools),  
     810

gff() (in module sympy.polys.polytools), 674

gff\_list() (in module sympy.polys.polytools), 673

gff\_list() (sympy.polys.polyclasses.DMP method),  
     764

gff\_list() (sympy.polys.polytools.Poly method), 694

given() (in module sympy.stats), 978

GMPYFiniteField (class in sympy.polys.domains),  
     762

GMPYIntegerRing (class in sympy.polys.domains),  
     762

GMPYRationalField (class in sympy.polys.domains),  
     762

GoldenRatio (class in sympy.core.numbers), 110

gosper\_normal() (in module sympy.concrete.gosper),  
     301

gosper\_sum() (in module sympy.concrete.gosper), 302

gosper\_term() (in module sympy.concrete.gosper), 302  
GradedLexOrder (class in sympy.polys.orderings), 721  
GramSchmidt() (in module sympy.matrices.dense), 618  
gray\_to\_bin() (sympy.combinatorics.graycode static method), 229  
GrayCode (class in sympy.combinatorics.graycode), 226  
graycode\_subsets() (sympy.combinatorics.graycode static method), 229  
GreaterThan (class in sympy.core.relational), 121  
greek\_letters (in module sympy.printing.pretty.pretty\_symbology), 868  
groebner() (in module sympy.polys.groebnertools), 836  
groebner() (in module sympy.polys.polytools), 678  
GroebnerBasis (class in sympy.polys.polytools), 715  
ground\_roots() (in module sympy.polys.polytools), 677  
ground\_roots() (sympy.polys.polytools.Poly method), 694  
group() (in module sympy.parsing.sympy\_tokenize), 1163  
group() (in module sympy.utilities.iterables), 1135  
gruntz() (in module sympy.series.gruntz), 902  
Gt (in module sympy.core.relational), 118

## H

H (sympy.matrices.matrices.MatrixBase attribute), 582  
Half (class in sympy.core.numbers), 105  
half\_gcdex() (in module sympy.polys.polytools), 666  
half\_gcdex() (sympy.polys.domains.domain.Domain method), 753  
half\_gcdex() (sympy.polys.polyclasses.DMP method), 764  
half\_gcdex() (sympy.polys.polytools.Poly method), 694  
half\_per() (sympy.polys.polyclasses.DMF method), 768  
hankel1 (class in sympy.functions.special.bessel), 392  
hankel2 (class in sympy.functions.special.bessel), 392  
hankel\_transform() (in module sympy.integrals.transforms), 527  
HankelTransform (class in sympy.integrals.transforms), 556  
harmonic (class in sympy.functions.combinatorial.numbers), 351  
has() (sympy.core.basic.Basic method), 67  
has() (sympy.matrices.matrices.MatrixBase method), 594  
has() (sympy.matrices.sparse.SparseMatrix method), 635  
has\_dups() (in module sympy.utilities.iterables), 1135  
has\_only\_gens() (sympy.polys.polytools.Poly method), 695  
has\_variety() (in module sympy.utilities.iterables), 1136  
Heaviside (class in sympy.functions.special.delta\_functions), 360  
height (sympy.categories.diagram\_drawing.DiagramGrid attribute), 1181  
height() (sympy.polys.agca.ideals.Ideal method), 741  
height() (sympy.printing.pretty.stringpict.stringPict method), 870  
hermite (class in sympy.functions.special.polynomials), 419  
hermite\_poly() (in module sympy.polys.orthopolys), 723  
hermitian, 61  
hessian() (in module sympy.matrices.dense), 618  
heurisch() (in module sympy.integrals.heurisch), 548  
heurisch\_wrapper() (in module sympy.integrals.heurisch), 549  
HeuristicGCDFailed (class in sympy.polys.polyerrors), 838  
hobj() (in module sympy.printing.pretty.pretty\_symbology), 869  
holzer() (in module sympy.solvers.diophantine), 1077  
hom() (sympy.categories.Diagram method), 1177  
homogeneous\_order() (in module sympy.solvers.ode), 987  
homogeneous\_order() (sympy.polys.polyclasses.DMP method), 764  
homogeneous\_order() (sympy.polys.polytools.Poly method), 695  
homogenize() (sympy.polys.polyclasses.DMP method), 764  
homogenize() (sympy.polys.polytools.Poly method), 695  
homomorphism() (in module sympy.polys.agca.homomorphisms), 746  
HomomorphismFailed (class in sympy.polys.polyerrors), 838  
horner() (in module sympy.polys.polyfuncs), 717  
hradius (sympy.geometry.ellipse.Ellipse attribute), 487  
hstack() (sympy.matrices.matrices.MatrixBase class method), 594  
hyper (class in sympy.functions.special.hyper), 405  
HyperbolicFunction (class in sympy.functions.elementary.hyperbolic), 327  
hyperexpand() (in module sympy.simplify.hyperexpand), 935

Hypergeometric() (in module sympy.stats), 950  
 hypersimilar() (in module sympy.simplify.simplify), 928  
 hypersimp() (in module sympy.simplify.simplify), 927  
 |  
 ibin() (in module sympy.utilities.iterables), 1136  
 Ideal (class in sympy.polys.agca.ideals), 740  
 ideal() (sympy.polys.domains.ring.Ring method), 756  
 Identity (class in sympy.matrices.expressions), 650  
 identity\_hom() (sympy.polys.agca.modules.FreeModule method), 735  
 identity\_hom() (sympy.polys.agca.modules.Module method), 734  
 identity\_hom() (sympy.polys.agca.modules.QuotientModule method), 743  
 identity\_hom() (sympy.polys.agca.modules.SubModule method), 737  
 IdentityFunction (class in sympy.functions.elementary.miscellaneous), 327  
 IdentityMorphism (class in sympy.categories), 1174  
 idiff() (in module sympy.geometry.util), 434  
 Idx (class in sympy.tensor.indexed), 1088  
 igcd() (in module sympy.core.numbers), 103  
 ilcm() (in module sympy.core.numbers), 104  
 im (class in sympy.functions.elementary.complexes), 327  
 image() (sympy.polys.agca.homomorphisms.ModuleHomeomorphism method), 748  
 ImageSet (class in sympy.sets.fancysets), 915  
 imageset() (in module sympy.sets.sets), 907  
 imaginary, 61  
 ImaginaryUnit (class in sympy.core.numbers), 108  
 ImmutableMatrix (class in sympy.matrices.immutable), 645  
 ImmutableSparseMatrix (class in sympy.matrices.immutable), 644  
 implemented\_function() (in module sympy.utilities.lambdify), 1148  
 implicit\_application() (in module sympy.parsing.sympy\_parser), 1164  
 implicit\_multiplication() (in module sympy.parsing.sympy\_parser), 1164  
 implicit\_multiplication\_application() (in module sympy.parsing.sympy\_parser), 1164  
 ImplicitSeries (class in sympy.plotting.plot\_implicit), 879  
 Implies (class in sympy.logic.boolalg), 568  
 imul\_num() (sympy.polys.rings.PolyElement method), 817  
 in\_terms\_of\_generators() (sympy.polys.agca.modules.SubModule method), 737  
 incenter (sympy.geometry.polygon.Triangle attribute), 511  
 incircle (sympy.geometry.polygon.RegularPolygon attribute), 506  
 incircle (sympy.geometry.polygon.Triangle attribute), 512  
 inclusion\_hom() (sympy.polys.agca.modules.SubModule method), 737  
 indent\_code() (sympy.printing.ccode.CCodePrinter method), 854  
 indent\_code() (sympy.printing.fcode.FCodePrinter method), 858  
 independent() (in module sympy.stats.rv), 983  
 index() (sympy.combinatorics.permutations.Permutation method), 170  
 index() (sympy.concrete.expr\_with\_intlimits.ExprWithIntLimits method), 299  
 index() (sympy.core.containers.Tuple method), 151  
 index() (sympy.polys.rings.PolyRing method), 814  
 Indexed (class in sympy.tensor.indexed), 1090  
 IndexedBase (class in sympy.tensor.indexed), 1091  
 indices (sympy.tensor.indexed.Indexed attribute), 1090  
 indices\_contain\_equal\_information (sympy.functions.special.tensor\_functions.KroneckerDelta attribute), 426  
 inf (sympy.sets.sets.Set attribute), 904  
 infinite, 61  
 infinitesimal() (in module sympy.solvers.ode), 988  
 Infinity (class in sympy.core.numbers), 106  
 inject() (sympy.polys.domains.compositedomain.CompositeDomain method), 757  
 inject() (sympy.polys.domains.domain.Domain method), 753  
 inject() (sympy.polys.domains.simpledomain.SimpleDomain method), 757  
 inject() (sympy.polys.polyclasses.DMP method), 764  
 inject() (sympy.polys.polytools.Poly method), 695  
 inradius (sympy.geometry.polygon.RegularPolygon attribute), 506  
 inradius (sympy.geometry.polygon.Triangle attribute), 512  
 intcurve\_diffequ() (in module sympy.diffgeom), 1198  
 intcurve\_series() (in module sympy.diffgeom), 1196  
 integer, 61  
 Integer (class in sympy.core.numbers), 103  
 integer\_nthroot() (in module sympy.core.power), 112  
 IntegerPartition (class in sympy.combinatorics.partitions), 161  
 IntegerRing (class in sympy.polys.domains), 757  
 Integers (class in sympy.sets.fancysets), 915  
 Integral (class in sympy.integrals.integrals), 551  
 Integral.is\_commutative (in module sympy.integrals.transforms), 551

integral\_steps() (in module sympy.integrals.manualintegrate), 550  
IntegralTransform (class in sympy.integrals.transforms), 554  
integrand() (sympy.functions.special.hyper.meijerg method), 410  
integrate() (in module sympy.integrals.integrals), 543  
integrate() (sympy.core.expr.Expr method), 87  
integrate() (sympy.matrices.matrices.MatrixBase method), 594  
integrate() (sympy.polys.polyclasses.DMP method), 764  
integrate() (sympy.polys.polytools.Poly method), 696  
interactive\_traversal() (in module sympy.utilities.iterables), 1137  
interior\_angle (sympy.geometry.polygon.RegularPolygon inverse attribute), 506  
interpolate() (in module sympy.polys.polyfuncs), 718  
interpolating\_poly() (in module sympy.polys.specialpolys), 723  
intersect() (sympy.polys.agca.ideals.Ideal method), 741  
intersect() (sympy.polys.agca.modules.SubModule method), 737  
intersect() (sympy.sets.sets.Set method), 904  
Intersection (class in sympy.sets.sets), 911  
intersection() (in module sympy.geometry.util), 435  
intersection() (sympy.geometry.ellipse.Circle method), 494  
intersection() (sympy.geometry.ellipse.Ellipse method), 487  
intersection() (sympy.geometry.entity.GeometryEntity method), 432  
intersection() (sympy.geometry.line.LinearEntity method), 451  
intersection() (sympy.geometry.line3d.LinearEntity3D method), 469  
intersection() (sympy.geometry.plane.Plane method), 518  
intersection() (sympy.geometry.point.Point method), 439  
intersection() (sympy.geometry.point3d.Point3D method), 446  
intersection() (sympy.geometry.polygon.Polygon method), 499  
intersection() (sympy.sets.sets.Set method), 904  
Interval (class in sympy.sets.sets), 908  
intervals() (in module sympy.polys.polytools), 676  
intervals() (sympy.polys.polyclasses.DMP method), 764  
intervals() (sympy.polys.polytools.Poly method), 696  
inv() (sympy.matrices.matrices.MatrixBase method), 595  
inv\_mod() (sympy.matrices.matrices.MatrixBase method), 595  
Inverse (class in sympy.matrices.expressions), 649  
inverse() (sympy.functions.elementary.exponential.log method), 329  
inverse() (sympy.functions.elementary.hyperbolic.acosh method), 315  
inverse() (sympy.functions.elementary.hyperbolic.acoth method), 315  
inverse() (sympy.functions.elementary.hyperbolic.asinh method), 317  
inverse() (sympy.functions.elementary.hyperbolic.atanh method), 321  
inverse() (sympy.functions.elementary.hyperbolic.coth method), 324  
inverse() (sympy.functions.elementary.hyperbolic.sinh method), 336  
inverse() (sympy.functions.elementary.hyperbolic.tanh method), 340  
inverse() (sympy.functions.elementary.trigonometric.acos method), 314  
inverse() (sympy.functions.elementary.trigonometric.acot method), 315  
inverse() (sympy.functions.elementary.trigonometric.acsc method), 316  
inverse() (sympy.functions.elementary.trigonometric.asec method), 318  
inverse() (sympy.functions.elementary.trigonometric.asin method), 317  
inverse() (sympy.functions.elementary.trigonometric.atan method), 319  
inverse() (sympy.functions.elementary.trigonometric.cot method), 324  
inverse() (sympy.functions.elementary.trigonometric.tan method), 340  
inverse\_ADJ() (sympy.matrices.matrices.MatrixBase method), 595  
inverse\_cosine\_transform() (in module sympy.integrals.transforms), 527  
inverse\_fourier\_transform() (in module sympy.integrals.transforms), 525  
inverse\_GE() (sympy.matrices.matrices.MatrixBase method), 595  
inverse\_hankel\_transform() (in module sympy.integrals.transforms), 528  
inverse\_laplace\_transform() (in module sympy.integrals.transforms), 524  
inverse\_LU() (sympy.matrices.matrices.MatrixBase method), 596  
inverse\_mellin\_transform() (in module sympy.integrals.transforms), 523  
inverse\_sine\_transform() (in module sympy.integrals.transforms), 526

InverseCosineTransform (class  
sympy.integrals.transforms), 556

InverseFourierTransform (class  
sympy.integrals.transforms), 556

InverseHankelTransform (class  
sympy.integrals.transforms), 556

InverseLaplaceTransform (class  
sympy.integrals.transforms), 555

InverseMellinTransform (class  
sympy.integrals.transforms), 555

InverseSineTransform (class  
sympy.integrals.transforms), 556

inversion\_vector() (sympy.combinatorics.permutations.  
method), 170

inversions() (sympy.combinatorics.permutations.Permutation  
method), 171

invert() (in module sympy.polys.polytools), 666

invert() (sympy.core.expr.Expr method), 87

invert() (sympy.polys.domains.domain.Domain  
method), 753

invert() (sympy.polys.domains.ring.Ring method),  
756

invert() (sympy.polys.polyclasses.DMF method), 768

invert() (sympy.polys.polyclasses.DMP method), 764

invert() (sympy.polys.polytools.Poly method), 696

invert\_complex() (in module sympy.solvers.solveset),  
1085

invert\_real() (in module sympy.solvers.solveset), 1084

irrational, 61

is\_abelian (sympy.combinatorics.perm\_groups.PermutationGroup  
attribute), 197

is\_above\_fermi (sympy.functions.special.tensor\_functions.KroneckerDeltoid), 198

is\_algebraic\_expr() (sympy.core.expr.Expr method),  
87

is\_aliased (sympy.numberfields.AlgebraicNumber  
attribute), 720

is\_alt\_sym() (sympy.combinatorics.perm\_groups.PermutationGroupDeltoid), 198

is\_anti\_symmetric() (sympy.matrices.matrices.MatrixBase  
method), 596

is\_below\_fermi (sympy.functions.special.tensor\_functions.KroneckerDeltoid), 427

is\_cnf() (in module sympy.logic.boolalg), 570

is\_collinear() (sympy.geometry.point.Point method),  
439

is\_commutative (sympy.core.function.Function attribute), 141

is\_comparable (sympy.core.basic.Basic attribute), 67

is\_concyclic() (sympy.geometry.point.Point method),  
440

is\_constant() (sympy.core.expr.Expr method), 88

is\_convex() (sympy.geometry.polygon.Polygon  
method), 500

in is\_coplanar() (sympy.geometry.plane.Plane method),  
518

in is\_cyclotomic (sympy.polys.polyclasses.DMP attribute), 764

in is\_cyclotomic (sympy.polys.polytools.Poly attribute),  
697

in is\_diagonal() (sympy.matrices.matrices.MatrixBase  
method), 597

in is\_diagonalizable() (sympy.matrices.matrices.MatrixBase  
method), 597

in is\_disjoint() (sympy.sets.sets.Set method), 904

is\_dnf() (in module sympy.logic.boolalg), 570

Permutation (sympy.combinatorics.permutations.Permutation  
attribute), 172

PermutationEquilateral() (sympy.geometry.polygon.Triangle  
method), 512

is\_even (sympy.combinatorics.permutations.Permutation  
attribute), 173

is\_full\_module() (sympy.polys.agca.modules.SubModule  
method), 738

is\_full\_module() (sympy.polys.agca.modules.SubQuotientModule  
method), 745

is\_groebner() (in module sympy.polys.groebnertools),  
836

is\_ground (sympy.polys.polyclasses.ANP attribute),  
768

is\_ground (sympy.polys.polyclasses.DMP attribute),  
764

is\_ground (sympy.polys.polytools.Poly attribute),  
697

is\_group() (sympy.combinatorics.perm\_groups.PermutationGroup  
attribute), 198

is\_hermitian (sympy.matrices.matrices.MatrixBase  
attribute), 598

is\_hermitian (sympy.matrices.sparse.SparseMatrix  
attribute), 636

is\_homogeneous (sympy.polys.polyclasses.DMP attribute), 765

is\_homogeneous (sympy.polys.polytools.Poly attribute),  
697

is.Identity (sympy.combinatorics.permutations.Permutation  
attribute), 172

is\_identity (sympy.core.function.Lambda attribute),  
134

is\_injective() (sympy.polys.agca.homomorphisms.ModuleHomomorphism  
method), 748

is\_irreducible (sympy.polys.polyclasses.DMP attribute), 765

is\_irreducible (sympy.polys.polytools.Poly attribute),  
697

is\_isomorphism() (sympy.polys.agca.homomorphisms.ModuleHomomorphism  
method), 748

is\_isosceles() (sympy.geometry.polygon.Triangle  
method), 513

is\_left\_unbounded (sympy.sets.sets.Interval attribute), 909  
is\_linear (sympy.polys.polyclasses.DMP attribute), 765  
is\_linear (sympy.polys.polytools.Poly attribute), 698  
is\_lower (sympy.matrices.matrices.MatrixBase attribute), 598  
is\_lower\_hessenberg (sympy.matrices.matrices.MatrixBase attribute), 599  
is\_maximal() (sympy.polys.agca.ideals.Ideal method), 741  
is\_minimal() (in module sympy.polys.groebnertools), 836  
is\_monic (sympy.polys.polyclasses.DMP attribute), 765  
is\_monic (sympy.polys.polytools.Poly attribute), 698  
is\_monomial (sympy.polys.polyclasses.DMP attribute), 765  
is\_monomial (sympy.polys.polytools.Poly attribute), 698  
is\_multivariate (sympy.polys.polytools.Poly attribute), 698  
is\_negative() (sympy.polys.domains.AlgebraicField method), 759  
is\_negative() (sympy.polys.domains.domain.Domain method), 753  
is\_negative() (sympy.polys.domains.ExpressionDomain method), 761  
is\_negative() (sympy.polys.domains.FractionField method), 760  
is\_negative() (sympy.polys.domains.PolynomialRing method), 758  
is\_nilpotent (sympy.combinatorics.perm\_groups.PermutationGroup attribute), 199  
is\_nilpotent() (sympy.matrices.matrices.MatrixBase method), 599  
is\_nonnegative() (sympy.polys.domains.AlgebraicField method), 759  
is\_nonnegative() (sympy.polys.domains.domain.Domain method), 753  
is\_nonnegative() (sympy.polys.domains.ExpressionDomain method), 761  
is\_nonnegative() (sympy.polys.domains.FractionField method), 760  
is\_nonnegative() (sympy.polys.domains.PolynomialRing method), 758  
is\_nonpositive() (sympy.polys.domains.AlgebraicField method), 759  
is\_nonpositive() (sympy.polys.domains.domain.Domain method), 753  
is\_nonpositive() (sympy.polys.domains.ExpressionDomain method), 762  
is\_nonpositive() (sympy.polys.domains.FractionField method), 760  
is\_nonpositive() (sympy.polys.domains.PolynomialRing method), 758  
is\_primary() (sympy.polys.agca.ideals.Ideal method), 741  
is\_prime() (sympy.polys.agca.ideals.Ideal method), 742  
is\_primitive (sympy.polys.polyclasses.DMP attribute), 765  
is\_primitive (sympy.polys.polytools.Poly attribute), 699  
is\_primitive() (sympy.combinatorics.perm\_groups.PermutationGroup method), 200

is\_primitive\_root() (in module sympy.nttheory.residue\_nttheory), 264  
 is\_principal() (sympy.polys.agca.ideals.Ideal method), 742  
 is\_proper\_subset() (sympy.sets.sets.Set method), 905  
 is\_proper\_superset() (sympy.sets.sets.Set method), 905  
 is\_quad\_residue() (in module sympy.nttheory.residue\_nttheory), 266  
 is\_quadratic (sympy.polys.polyclasses.DMP attribute), 765  
 is\_quadratic (sympy.polys.polytools.Poly attribute), 699  
 is\_radical() (sympy.polys.agca.ideals.Ideal method), 742  
 is\_rational\_function() (sympy.core.expr.Expr method), 90  
 is\_reduced() (in module sympy.polys.groebnertools), 836  
 is\_right() (sympy.geometry.polygon.Triangle method), 513  
 is\_right\_unbounded (sympy.sets.sets.Interval attribute), 909  
 is\_scalene() (sympy.geometry.polygon.Triangle method), 513  
 is\_sequence() (in module sympy.core.compatibility), 153, 157  
 is\_similar() (sympy.geometry.entity.GeometryEntity method), 433  
 is\_similar() (sympy.geometry.line.LinearEntity method), 452  
 is\_similar() (sympy.geometry.line3d.LinearEntity3D method), 470  
 is\_similar() (sympy.geometry.polygon.Triangle method), 514  
 is\_simple() (sympy.functions.special.delta\_functions.DiracDelta method), 359  
 is\_Singleton (sympy.combinatorics.permutations.Permutation attribute), 172  
 is\_solvable (sympy.combinatorics.perm\_groups.PermutationGroup attribute), 201  
 is\_sqf (sympy.polys.polyclasses.DMP attribute), 765  
 is\_sqf (sympy.polys.polytools.Poly attribute), 700  
 is\_square (sympy.matrices.matrices.MatrixBase attribute), 600  
 is\_subdiagram() (sympy.categories.Diagram method), 1178  
 is\_subgroup() (sympy.combinatorics.perm\_groups.PermutationGroup method), 201  
 is\_submodule() (sympy.polys.agca.modules.FreeModule method), 735  
 is\_submodule() (sympy.polys.agca.modules.Module method), 734  
 is\_submodule() (sympy.polys.agca.modules.QuotientModule method), 743  
 is\_submodule() (sympy.polys.agca.modules.SubModule method), 738  
 is\_subset() (sympy.sets.sets.Set method), 905  
 is\_superset() (sympy.sets.sets.Set method), 905  
 is\_surjective() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 748  
 is\_symbolic() (sympy.matrices.matrices.MatrixBase method), 600  
 is\_symmetric() (sympy.matrices.matrices.MatrixBase method), 600  
 is\_symmetric() (sympy.matrices.sparse.SparseMatrix method), 636  
 is\_tangent() (sympy.geometry.ellipse.Ellipse method), 488  
 is\_transitive() (sympy.combinatorics.perm\_groups.PermutationGroup method), 202  
 is\_trivial (sympy.combinatorics.perm\_groups.PermutationGroup attribute), 202  
 is\_univariate (sympy.polys.polytools.Poly attribute), 700  
 is\_upper (sympy.matrices.matrices.MatrixBase attribute), 601  
 is\_upper\_hessenberg (sympy.matrices.matrices.MatrixBase attribute), 602  
 is\_whole\_ring() (sympy.polys.agca.ideals.Ideal method), 742  
 is\_zero (sympy.matrices.immutable.ImmutableMatrix attribute), 646  
 is\_zero (sympy.matrices.matrices.MatrixBase attribute), 602  
 is\_zero (sympy.polys.polyclasses.ANP attribute), 769  
 is\_zero (sympy.polys.polyclasses.DMF attribute), 768  
 is\_zero (sympy.polys.polyclasses.DMP attribute), 765  
 isZeroGroup (sympy.polys.agca.modules.FreeModule method), 736  
 is\_zero() (sympy.polys.agca.modules.Module method), 734  
 is\_zero() (sympy.polys.agca.modules.QuotientModule method), 744  
 is\_zero() (sympy.polys.agca.modules.SubModule method), 738  
 is\_zero\_dimensional (sympy.polys.polytools.GroebnerBasis attribute), 716  
 is\_zero\_dimensional() (in module sympy.polys.polytools), 679  
 isdisjoint() (sympy.sets.sets.Set method), 906

isolate() (in module sympy.polys.numberfields), 720  
IsomorphismFailed (class in sympy.polys.polyerrors), 838  
isprime() (in module sympy.ntheory.primetest), 263  
issubset() (sympy.sets.sets.Set method), 906  
issuperset() (sympy.sets.sets.Set method), 906  
ITE (class in sympy.logic.boolalg), 569  
items() (sympy.core.containers.Dict method), 151  
iterable() (in module sympy.core.compatibility), 153, 156  
iterate\_binary() (sympy.combinatorics.subsets.Subset method), 220  
iterate\_graycode() (sympy.combinatorics.subsets.Subset method), 220  
itercoeffs() (sympy.polys.rings.PolyElement method), 817  
itermonomials() (in module sympy.polys.monomials), 720  
itermonoms() (sympy.polys.rings.PolyElement method), 817  
iterterms() (sympy.polys.rings.PolyElement method), 817

**J**

jacobi (class in sympy.functions.special.polynomials), 412  
jacobi\_normalized() (in module sympy.functions.special.polynomials), 413  
jacobi\_poly() (in module sympy.polys.orthopolys), 723  
jacobi\_symbol() (in module sympy.ntheory.residue\_nttheory), 266  
jacobian() (sympy.diffgeom.CoordSystem method), 1190  
jacobian() (sympy.matrices.matrices.MatrixBase method), 603  
jn (class in sympy.functions.special.bessel), 393  
jn\_zeros() (in module sympy.functions.special.bessel), 394  
jordan\_cell() (in module sympy.matrices.dense), 617  
jordan\_cells() (sympy.matrices.matrices.MatrixBase method), 603  
jordan\_form() (sympy.matrices.matrices.MatrixBase method), 604  
josephus() (sympy.combinatorics.permutations.Permutation class method), 173

**K**

kbins() (in module sympy.utilities.iterables), 1137  
kernel() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 749  
key2bounds() (sympy.matrices.matrices.MatrixBase method), 604  
key2ij() (sympy.matrices.matrices.MatrixBase method), 604  
keys() (sympy.core.containers.Dict method), 151  
kid\_rsa\_private\_key() (in module sympy.crypto.crypto), 283  
kid\_rsa\_public\_key() (in module sympy.crypto.crypto), 283  
killable\_index (sympy.functions.special.tensor\_functions.KroneckerDelta attribute), 428  
known\_functions (in module sympy.printing.ccode), 854  
known\_functions (in module sympy.printing.mathematica), 860  
KroneckerDelta (class in sympy.functions.special.tensor\_functions), 425  
ksubsets() (sympy.combinatorics.subsets static method), 225  
Kumaraswamy() (in module sympy.stats), 963

**L**

l1\_norm() (sympy.polys.polyclasses.DMP method), 765  
l1\_norm() (sympy.polys.polytools.Poly method), 700  
label (sympy.tensor.indexed.Idx attribute), 1089  
label (sympy.tensor.indexed.IndexedBase attribute), 1092  
laguerre (class in sympy.functions.special.polynomials), 419  
laguerre\_poly() (in module sympy.polys.orthopolys), 723  
Lambda (class in sympy.core.function), 134  
LambdaPrinter (class in sympy.printing.lambdarepr), 861  
lambdarepr() (in module sympy.printing.lambdarepr), 861  
lambdastr() (in module sympy.utilities.lambdify), 1148  
lambdify() (in module sympy.utilities.lambdify), 1149  
LambertW (class in sympy.functions.elementary.exponential), 328  
Laplace() (in module sympy.stats), 964  
laplace\_transform() (in module sympy.integrals.transforms), 524  
LaplaceTransform (class in sympy.integrals.transforms), 555  
latex() (in module sympy.printing.latex), 861  
LatexPrinter (class in sympy.printing.latex), 861  
LC() (in module sympy.polys.polytools), 663  
LC() (sympy.polys.polyclasses.ANP method), 768  
LC() (sympy.polys.polyclasses.DMP method), 762  
LC() (sympy.polys.polytools.Poly method), 680

lcm() (in module sympy.polys.polytools), 671  
 lcm() (sympy.core.numbers.Number method), 99  
 lcm() (sympy.polys.domains.domain.Domain method), 753  
 lcm() (sympy.polys.domains.field.Field method), 755  
 lcm() (sympy.polys.domains.PolynomialRing method), 758  
 lcm() (sympy.polys.domains.RealField method), 761  
 lcm() (sympy.polys.polyclasses.DMP method), 765  
 lcm() (sympy.polys.polytools.Poly method), 701  
 lcm\_list() (in module sympy.polys.polytools), 671  
 ldescent() (in module sympy.solvers.diophantine), 1076  
 LDLdecomposition() (sympy.matrices.matrices.MatrixBase method), 582  
 LDLdecomposition() (sympy.matrices.sparse.SparseMatrix method), 632  
 LDLsolve() (sympy.matrices.matrices.MatrixBase method), 583  
 Le (in module sympy.core.relational), 118  
 leading\_expv() (sympy.polys.rings.PolyElement method), 817  
 leading\_monom() (sympy.polys.rings.PolyElement method), 818  
 leading\_term() (sympy.polys.rings.PolyElement method), 818  
 leadterm() (sympy.core.expr.Expr method), 91  
 left (sympy.sets.sets.Interval attribute), 909  
 left() (sympy.printing.pretty.stringpict.stringPict method), 870  
 left\_open (sympy.sets.sets.Interval attribute), 909  
 leftslash() (sympy.printing.pretty.stringpict.stringPict method), 870  
 legendre (class in sympy.functions.special.polynomials), 417  
 legendre\_poly() (in module sympy.polys.orthopolys), 723  
 legendre\_symbol() (in module sympy.nttheory.residue\_nttheory), 266  
 length (sympy.geometry.line.LinearEntity attribute), 452  
 length (sympy.geometry.line.Segment attribute), 463  
 length (sympy.geometry.line3d.LinearEntity3D attribute), 470  
 length (sympy.geometry.line3d.Segment3D attribute), 477  
 length (sympy.geometry.point.Point attribute), 440  
 length (sympy.geometry.point3d.Point3D attribute), 446  
 length (sympy.geometry.polygon.RegularPolygon attribute), 507  
 length() (sympy.combinatorics.permutations.Permutation method), 174  
 length() (sympy.polys.polytools.Poly method), 701  
 lerchphi (class in sympy.functions.special.zeta\_functions), 404  
 LessThan (class in sympy.core.relational), 124  
 LeviCivita (class in sympy.functions.special.tensor\_functions), 424  
 LexOrder (class in sympy.polys.orderings), 721  
 lfsr\_autocorrelation() (in module sympy.crypto.crypto), 285  
 lfsr\_connection\_polynomial() (in module sympy.crypto.crypto), 286  
 lfsr\_sequence() (in module sympy.crypto.crypto), 284  
 lhs (sympy.core.relational.Relational attribute), 119  
 Li (class in sympy.functions.special.error\_functions), 383  
 li (class in sympy.functions.special.error\_functions), 382  
 lie\_heuristic\_abaco1\_product() (in module sympy.solvers.ode), 1012  
 lie\_heuristic\_abaco1\_simple() (in module sympy.solvers.ode), 1011  
 lie\_heuristic\_abaco2\_similar() (in module sympy.solvers.ode), 1013  
 lie\_heuristic\_abaco2\_unique\_general() (in module sympy.solvers.ode), 1014  
 lie\_heuristic\_abaco2\_unique\_unknown() (in module sympy.solvers.ode), 1014  
 lie\_heuristic\_bivariate() (in module sympy.solvers.ode), 1012  
 lie\_heuristic\_chi() (in module sympy.solvers.ode), 1012  
 lie\_heuristic\_function\_sum() (in module sympy.solvers.ode), 1013  
 lie\_heuristic\_linear() (in module sympy.solvers.ode), 1015  
 LieDerivative (class in sympy.diffgeom), 1195  
 lift() (sympy.polys.polyclasses.DMP method), 765  
 lift() (sympy.polys.polytools.Poly method), 701  
 Limit (class in sympy.series.limits), 897  
 limit() (in module sympy.series.limits), 897  
 limit() (sympy.core.expr.Expr method), 91  
 limit() (sympy.matrices.matrices.MatrixBase method), 605  
 limit\_denominator() (sympy.core.numbers.Rational method), 103  
 limitinf() (in module sympy.series.gruntz), 902  
 limits (sympy.concrete.expr\_with\_limits.ExprWithLimits attribute), 297  
 limits (sympy.geometry.curve.Curve attribute), 480  
 Line (class in sympy.geometry.line), 456  
 Line2DBaseSeries (class in sympy.plotting.plot), 878  
 Line3D (class in sympy.geometry.line3d), 464  
 Line3DBaseSeries (class in sympy.plotting.plot), 879  
 line\_integrate() (in module sympy.integrals.integrals), 545

LinearEntity (class in `sympy.geometry.line`), 448  
LinearEntity3D (class in `sympy.geometry.line3d`), 466  
LineOver1DRangeSeries (class in `sympy.plotting.plot`), 878  
list() (sympy.combinatorics.permutations.Cycle method), 182  
list() (sympy.combinatorics.permutations.Permutation method), 174  
list2numpy() (in module `sympy.matrices.dense`), 620  
list\_visitor() (in module `sympy.utilities.enumerative`), 1126  
listcoeffs() (sympy.polys.rings.PolyElement method), 818  
listmonoms() (sympy.polys.rings.PolyElement method), 818  
listterms() (sympy.polys.rings.PolyElement method), 818  
liupc() (sympy.matrices.sparse.SparseMatrix method), 636  
LM() (in module `sympy.polys.polytools`), 663  
LM() (sympy.polys.polytools.Poly method), 680  
log (class in `sympy.functions.elementary.exponential`), 329  
log() (sympy.polys.domains.domain.Domain method), 754  
log() (sympy.polys.domains.IntegerRing method), 757  
logcombine() (in module `sympy.simplify.simplify`), 932  
loggamma (class in `sympy.functions.special.gamma_functions`), 362  
Logistic() (in module `sympy.stats`), 964  
LogNormal() (in module `sympy.stats`), 965  
Lopen() (sympy.sets.sets.Interval class method), 908  
lower (sympy.tensor.indexed.Idx attribute), 1089  
lower\_central\_series() (sympy.combinatorics.perm\_groups method), 202  
lower\_triangular\_solve() (sympy.matrices.matrices.MatrixBase method), 605  
lowergamma (class in `sympy.functions.special.gamma_functions`), 367  
lru\_cache() (in module `sympy.core.compatibility`), 154  
lseries() (sympy.core.expr.Expr method), 91  
Lt (in module `sympy.core.relational`), 118  
LT() (in module `sympy.polys.polytools`), 663  
LT() (sympy.polys.polytools.Poly method), 680  
ltrim() (sympy.polys.polytools.Poly method), 701  
lucas (class in `sympy.functions.combinatorial.numbers`), 353  
LUdecomposition() (sympy.matrices.matrices.MatrixBase method), 583  
LUdecomposition\_Simple() (sympy.matrices.matrices.MatrixBase method), 584  
LUdecompositionFF() (sympy.matrices.matrices.MatrixBase method), 584  
LUsolve() (sympy.matrices.matrices.MatrixBase method), 584

## M

major (sympy.geometry.ellipse.Ellipse attribute), 488  
make\_perm() (sympy.combinatorics.perm\_groups.PermutationGroup method), 203  
make\_prime() (in module `sympy.solvers.diophantine`), 1078  
make\_routine() (in module `sympy.utilitiescodegen`), 1122  
Manifold (class in `sympy.diffgeom`), 1187  
manualintegrate() (in module `sympy.integrals.manualintegrate`), 550  
map() (sympy.polys.domains.domain.Domain method), 754  
MatAdd (class in `sympy.matrices.expressions`), 648  
match() (sympy.core.basic.Basic method), 67  
matches() (sympy.core.basic.Basic method), 68  
mathematica() (in module `sympy.parsing.mathematica`), 1163  
mathematica\_code() (in module `sympy.printing.mathematica`), 860  
mathml() (in module `sympy.printing.mathml`), 863  
mathml\_tag() (sympy.printing.mathml.MathMLPrinter method), 863  
MathMLPrinter (class in `sympy.printing.mathml`), 863  
MatMul (class in `sympy.matrices.expressions`), 648  
MatPow (class in `sympy.matrices.expressions`), 649  
matrix2lumpy() (in module `sympy.matrices.dense`), 620  
matrix\_fglm() (in module `sympy.polys.fglmtools`), 836  
matrix\_multiply\_elementwise() (in module `sympy.matrices.dense`), 615  
MatrixBase (class in `sympy.matrices.matrices`), 581  
MatrixError (class in `sympy.matrices.matrices`), 615  
MatrixExpr (class in `sympy.matrices.expressions`), 647  
MatrixSymbol (class in `sympy.matrices.expressions`), 648  
Max (class in `sympy.functions.elementary.miscellaneous`), 330  
max() (sympy.combinatorics.permutations.Permutation method), 175  
max\_div (sympy.combinatorics.perm\_groups.PermutationGroup attribute), 203

max_norm()	(sympy.polys.polyclasses.DMP method),	765	Mod (class in sympy.core.mod),	118
max_norm()	(sympy.polys.polytools.Poly method),	701	modgcd_bivariate() (in module sympy.polys.modular_gcd),	840
Maxwell()	(in module sympy.stats),	966	modgcd_multivariate() (in module sympy.polys.modular_gcd),	841
maybe()	(in module sympy.parsing.sympy_tokenize),	1163	modgcd_univariate() (in module sympy.polys.modular_gcd),	839
MCodePrinter	(class in sympy.printing.mathematica),	860	Module (class in sympy.polys.agca.modules),	734
measure	(sympy.sets.sets.Set attribute),	906	module_quotient() (sympy.polys.agca.modules.SubModule method),	738
medial	(sympy.geometry.polygon.Triangle attribute),	514	ModuleHomomorphism (class in sympy.polys.agca.homomorphisms),	747
medians	(sympy.geometry.polygon.Triangle attribute),	514	monic() (in module sympy.polys.polytools),	672
meets()	(sympy.series.gruntz.SubsSet method),	902	monic() (sympy.polys.polyclasses.DMP method),	765
meijerg	(class in sympy.functions.special.hyper),	407	monic() (sympy.polys.polytools.Poly method),	702
mellin_transform()	(in module sympy.integrals.transforms),	522	monic() (sympy.polys.rings.PolyElement method),	818
MellinTransform	(class in sympy.integrals.transforms),	555	Monomial (class in sympy.polys.monomials),	720
merge_solution()	(in module sympy.solvers.diophantine),	1074	monomial_basis() (sympy.polys.rings.PolyRing method),	814
metric_to_Christoffel_1st()	(in module sympy.diffgeom),	1200	monomial_count() (in module sympy.polys.monomials),	721
metric_to_Christoffel_2nd()	(in module sympy.diffgeom),	1200	monoms() (sympy.polys.polyclasses.DMP method),	765
metric_to_Ricci_components()	(in module sympy.diffgeom),	1201	monoms() (sympy.polys.polytools.Poly method),	702
metric_to_Riemann_components()	(in module sympy.diffgeom),	1200	monoms() (sympy.polys.rings.PolyElement method),	818
midpoint	(sympy.geometry.line.Segment attribute),	463	Morphism (class in sympy.categories),	1171
midpoint	(sympy.geometry.line3d.Segment3D attribute),	477	morphisms (sympy.categories.diagram_drawing.DiagramGrid attribute),	1181
midpoint()	(sympy.geometry.point.Point method),	441	mr() (in module sympy.ntheory.prime_test),	263
midpoint()	(sympy.geometry.point3d.Point3D method),	446	mrv() (in module sympy.series.gruntz),	902
Min	(class in sympy.functions.elementary.miscellaneous),	330	mrv_leadterm() (in module sympy.series.gruntz),	902
min()	(sympy.combinatorics.permutations.Permutation method),	175	mrv_max1() (in module sympy.series.gruntz),	902
minimal_block()	(sympy.combinatorics.perm_groups.PermGroup method),	204	mrv_max3() (in module sympy.series.gruntz),	903
minimal_polynomial()	(in module sympy.polys.numberfields),	719	Mul (class in sympy.core.mul),	113
minlex()	(in module sympy.utilities.iterables),	1138	Mul() (sympy.assumptions.handlers.calculus.AskFiniteHandler static method),	885
minor	(sympy.geometry.ellipse.Ellipse attribute),	489	Mul() (sympy.assumptions.handlers.calculus.AskInfinitesimalHandler static method),	886
minorEntry()	(sympy.matrices.matrices.MatrixBase method),	605	Mul() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler static method),	887
minorMatrix()	(sympy.matrices.matrices.MatrixBase method),	605	Mul() (sympy.assumptions.handlers.sets.AskHermitianHandler static method),	887
minpoly()	(in module sympy.polys.numberfields),	719	Mul() (sympy.assumptions.handlers.sets.AskImaginaryHandler static method),	887
mobius	(class in sympy.ntheory),	269	Mul() (sympy.assumptions.handlers.sets.AskIntegerHandler static method),	888
			Mul() (sympy.assumptions.handlers.sets.AskRationalHandler static method),	888
			Mul() (sympy.assumptions.handlers.sets.AskRealHandler static method),	888
			mul() (sympy.polys.domains.domain.Domain method),	754

mul() (sympy.polys.polyclasses.DMF method), 768  
mul() (sympy.polys.polyclasses.DMP method), 765  
mul() (sympy.polys.polytools.Poly method), 702  
mul() (sympy.polys.rings.PolyRing method), 814  
mul.ground() (sympy.polys.polyclasses.DMP method), 765  
mul\_ground() (sympy.polys.polytools.Poly method), 703  
mul\_inv() (sympy.combinatorics.permutations.Permutation method), 175  
MultiFactorial (class in sympy.functions.combinatorial.factorials), 354  
multinomial\_coefficients() (in module sympy.nttheory.multinomial), 262  
multinomial\_coefficients\_iterator() (in module sympy.nttheory.multinomial), 262  
multiplicity() (in module sympy.nttheory.factor\_), 251  
multiply() (sympy.matrices.matrices.MatrixBase method), 605  
multiply() (sympy.matrices.sparse.SparseMatrix method), 637  
multiply\_elementwise() (sympy.matrices.matrices.MatrixBase method), 605  
multiply\_ideal() (sympy.polys.agca.modules.FreeModule method), 736  
multiply\_ideal() (sympy.polys.agca.modules.Module method), 734  
multiply\_ideal() (sympy.polys.agca.modules.SubModule method), 739  
multiset() (in module sympy.utilities.iterables), 1138  
multiset\_combinations() (in module sympy.utilities.iterables), 1139  
multiset\_partitions() (in module sympy.utilities.iterables), 1139  
multiset\_partitions\_taocp() (in module sympy.utilities.enumerative), 1125  
multiset\_permutations() (in module sympy.utilities.iterables), 1140  
MultisetPartitionTraverser (class in sympy.utilities.enumerative), 1126  
MultivariatePolynomialError (class in sympy.polys.polyerrors), 839  
MutableDenseMatrix (class in sympy.matrices.dense), 626  
MutableSparseMatrix (class in sympy.matrices.sparse), 640

n() (sympy.geometry.point.Point method), 441  
n() (sympy.geometry.point3d.Point3D method), 447  
n() (sympy.matrices.matrices.MatrixBase method), 606  
n() (sympy.polys.domains.domain.Domain method), 754  
n\_order() (in module sympy.nttheory.residue\_nttheory), 264  
Nakagami() (in module sympy.stats), 966  
name (sympy.categories.Category attribute), 1176  
name (sympy.categories.NamedMorphism attribute), 1172  
NamedMorphism (class in sympy.categories), 1172  
NaN (class in sympy.core.numbers), 106  
Nand (class in sympy.logic.boolalg), 567  
nargs (sympy.core.function.FunctionClass attribute), 139  
native\_coeffs() (sympy.polys.numberfields.AlgebraicNumber method), 720  
Naturals (class in sympy.sets.fancysets), 914  
Naturals0 (class in sympy.sets.fancysets), 914  
nC() (in module sympy.functions.combinatorial.numbers), 356  
Ne (in module sympy.core.relational), 118  
necklaces() (in module sympy.utilities.iterables), 1140  
neg() (sympy.polys.domains.domain.Domain method), 754  
neg() (sympy.polys.polyclasses.DMF method), 768  
neg() (sympy.polys.polyclasses.DMP method), 765  
neg() (sympy.polys.polytools.Poly method), 703  
negative, 61  
NegativeInfinity (class in sympy.core.numbers), 107  
NegativeOne (class in sympy.core.numbers), 105  
new() (sympy.polys.polytools.Poly class method), 703  
next() (sympy.combinatorics.graycode.GrayCode method), 227  
next() (sympy.combinatorics.prufer.Prufer method), 216  
next() (sympy.printing.pretty.stringpict.stringPict static method), 870  
next\_binary() (sympy.combinatorics.subsets.Subset method), 221  
next\_gray() (sympy.combinatorics.subsets.Subset method), 221  
next\_lex() (sympy.combinatorics.partitions.IntegerPartition method), 162  
next\_lex() (sympy.combinatorics.permutations.Permutation method), 175  
next\_lexicographic() (sympy.combinatorics.subsets.Subset method), 221  
next\_nonlex() (sympy.combinatorics.permutations.Permutation method), 175

## N

n (sympy.combinatorics.graycode.GrayCode attribute), 227  
N() (in module sympy.core.evalf), 150  
n() (sympy.core.evalf.EvalfMixin method), 149

at-

next\_trotterjohnson() (sympy.combinatorics.permutations method), 176

nextprime() (in module sympy.ntheory.generate), 246

nfloat() (in module sympy.core.function), 149

nnz() (sympy.matrices.sparse.SparseMatrix method), 637

no\_attrs\_in\_subclass (class sympy.utilities.decorator), 1124

nodes (sympy.combinatorics.prufer.Prufer attribute), 216

nonnegative, 61

nonpositive, 61

NonSquareMatrixError (class sympy.matrices.matrices), 615

nonzero, 61

Nor (class in sympy.logic.boolalg), 568

norm() (sympy.matrices.matrices.MatrixBase method), 606

Normal() (in module sympy.stats), 967

normal() (sympy.core.expr.Expr method), 91

normal\_closure() (sympy.combinatorics.perm\_groups.Permutation (sympy.polys.polyclasses.DMF method), 768

normal\_lines() (sympy.geometry.ellipse.Ellipse method), 489

normal\_vector (sympy.geometry.plane.Plane attribute), 519

normalized() (sympy.matrices.matrices.MatrixBase method), 606

NormalPSpace (class in sympy.stats.crv\_types), 982

Not (class in sympy.logic.boolalg), 566

NotAlgebraic (class in sympy.polys.polyerrors), 839

NotInvertible (class in sympy.polys.polyerrors), 839

NotIterable (class in sympy.core.compatibility), 151

NotReversible (class in sympy.polys.polyerrors), 839

nP() (in module sympy.functions.combinatorial.numbers), 357

npartitions() (in module sympy.ntheory.partitions\_), 263

nroots() (in module sympy.polys.polytools), 677

nroots() (sympy.polys.polytools.Poly method), 703

nseries() (sympy.core.expr.Expr method), 91

nsimplify() (in module sympy.simplify.simplify), 928

nsimplify() (sympy.core.expr.Expr method), 92

nsolve() (in module sympy.solvers.solvers), 1052

nT() (in module sympy.functions.combinatorial.numbers), 357

nth() (sympy.polys.polyclasses.DMP method), 765

nth() (sympy.polys.polytools.Poly method), 703

nth\_power\_roots\_poly() (in module sympy.polys.polytools), 677

nth\_power\_roots\_poly() (sympy.polys.polytools.Poly method), 704

nthroot() (in module sympy.simplify.simplify), 921

nthPermutation (in module sympy.ntheory.residue\_nttheory), 265

nu (sympy.functions.special.hyper.meijerg attribute), 410

nullspace() (sympy.matrices.matrices.MatrixBase method), 607

Number (class in sympy.core.numbers), 98

numbered\_symbols() (in module sympy.utilities.iterables), 1141

NumberSymbol (class in sympy.core.numbers), 103

numer() (sympy.polys.domains.AlgebraicField method), 759

numer() (sympy.polys.domains.domain.Domain method), 754

numer() (sympy.polys.domains.ExpressionDomain method), 762

numer() (sympy.polys.domains.FractionField method), 760

numer() (sympy.polys.domains.ring.Ring method), 756

numerator (sympy.polys.polyclasses.DMF method), 768

## O

O (in module sympy.series.order), 898

Object (class in sympy.categories), 1171

objects (sympy.categories.Category attribute), 1176

objects (sympy.categories.Diagram attribute), 1178

OctaveCodeGen (class in sympy.utilities.codegen), 1120

odd, 61

ode\_1st\_exact() (in module sympy.solvers.ode), 993

ode\_1st\_homogeneous\_coeff\_best() (in module sympy.solvers.ode), 994

ode\_1st\_homogeneous\_coeff\_subs\_dep\_div\_indep() (in module sympy.solvers.ode), 995

ode\_1st\_homogeneous\_coeff\_subs\_indep\_div\_dep() (in module sympy.solvers.ode), 996

ode\_1st\_linear() (in module sympy.solvers.ode), 997

ode\_1st\_power\_series() (in module sympy.solvers.ode), 1009

ode\_2nd\_power\_series\_ordinary() (in module sympy.solvers.ode), 1009

ode\_2nd\_power\_series\_regular() (in module sympy.solvers.ode), 1010

ode\_almost\_linear() (in module sympy.solvers.ode), 1005

ode\_Bernoulli() (in module sympy.solvers.ode), 998

ode\_lie\_group() (in module sympy.solvers.ode), 1008

ode\_linear\_coefficients() (in module sympy.solvers.ode), 1006

ode\_Liouville() (in module sympy.solvers.ode), 999

ode\_nth\_linear\_constant\_coeff\_homogeneous() (in module sympy.solvers.ode), 1001

ode\_nth\_linear\_constant\_coeff\_undetermined\_coefficients() (sympy.geometry.line.LinearEntity attribute), 519  
    (in module sympy.solvers.ode), 1002

ode\_nth\_linear\_constant\_coeff\_variation\_of\_parameters() (sympy.geometry.line3d.LinearEntity3D attribute), 471  
    (in module sympy.solvers.ode), 1003

ode\_order() (in module sympy.solvers.deutils), 1056

ode\_Riccati\_special\_minus2() (in module sympy.solvers.ode), 1000

ode\_separable() (in module sympy.solvers.ode), 1004

ode\_separable\_reduced() (in module sympy.solvers.ode), 1007

ode\_sol\_simplicity() (in module sympy.solvers.ode), 992

odesimp() (in module sympy.solvers.ode), 989

of\_type() (sympy.polys.domains.domain.Domain method), 754

old\_frac\_field() (sympy.polys.domains.domain.Domain method), 754

old\_poly\_ring() (sympy.polys.domains.domain.Domain method), 754

One (class in sympy.core.numbers), 104

one (sympy.polys.polytools.Poly attribute), 704

ones() (in module sympy.matrices.dense), 616

open() (sympy.sets.sets.Interval class method), 909

OperationNotSupported (class in sympy.polys.polyerrors), 838

opt\_cse() (in module sympy.simplify.cse\_main), 934

OptionError (class in sympy.polys.polyerrors), 839

Options (class in sympy.polys.polyoptions), 848

Or (class in sympy.logic.boolalg), 565

orbit() (sympy.combinatorics.perm\_groups.Permutation class method), 205

orbit\_rep() (sympy.combinatorics.perm\_groups.Permutation class method), 205

orbit\_transversal() (sympy.combinatorics.perm\_groups.Permutation class method), 206

orbits() (sympy.combinatorics.perm\_groups.Permutation class method), 206

Order (class in sympy.polys.polyoptions), 848

Order (class in sympy.series.order), 898

order (sympy.functions.special.bessel.BesselBase attribute), 389

order() (sympy.combinatorics.perm\_groups.Permutation class method), 206

order() (sympy.combinatorics.permutations.Permutation class method), 176

ordered() (in module sympy.core.compatibility), 154

orthocenter (sympy.geometry.polygon.Triangle attribute), 515

P

P() (in module sympy.stats), 977

p1 (sympy.geometry.line.LinearEntity attribute), 453

p1 (sympy.geometry.line3d.LinearEntity3D attribute), 470

p2 (sympy.geometry.line.LinearEntity attribute), 453

p2 (sympy.geometry.line3d.LinearEntity3D attribute), 471

pairwise\_prime() (in module sympy.solvers.diophantine), 1077

parallel\_line() (sympy.geometry.line.LinearEntity method), 453

parallel\_line() (sympy.geometry.line3d.LinearEntity3D method), 471

parallel\_plane() (sympy.geometry.line3d.LinearEntity3D method), 520

parallel\_poly\_from\_expr() (in module sympy.polys.polytools), 662

parameter (sympy.geometry.curve.Curve attribute), 480

Parametric2DLineSeries (class in sympy.plotting.plot), 879

Parametric3DLineSeries (class in sympy.plotting.plot), 879

ParametricSurfaceSeries (class in sympy.plotting.plot), 879

parametrize\_ternary\_quadratic() (in module sympy.solvers.diophantine), 1075

parens() (sympy.printing.pretty.stringpict.StringPict method), 870

Pareto() (in module sympy.stats), 968

parity() (sympy.combinatorics.permutations.Permutation method), 176

parse\_expr() (in module sympy.parsing.sympy\_parser), 1161

parse\_Graphima() (in module sympy.parsing.maxima), 1163

Partition (class in sympy.combinatorics.partitions), 160

partition() (sympy.combinatorics.partitions.Partition attribute), 160

partition() (in module sympy.solvers.diophantine), 1072

partitions() (in module sympy.utilities.iterables), 1141

Patchp (class in sympy.diffgeom), 1187

pde\_1st\_linear\_constant\_coeff() (in module sympy.solvers.pde), 1042

pde\_1st\_linear\_constant\_coeff\_homogeneous() (in module sympy.solvers.pde), 1041

pde\_1st\_linear\_variable\_coeff() (in module sympy.solvers.pde), 1043

pde\_separate() (in module sympy.solvers.pde), 1037

pde\_separate\_add() (in module sympy.solvers.pde), 1037

pde\_separate\_mul() (in module sympy.solvers.pde), 1038

pdiv() (in module sympy.polys.polytools), 664

pdiv() (sympy.polys.polyclasses.DMP method), 765  
 pdiv() (sympy.polys.polytools.Poly method), 704  
 pdsolve() (in module sympy.solvers.pde), 1038  
 per() (sympy.polys.polyclasses.DMF method), 768  
 per() (sympy.polys.polyclasses.DMP method), 766  
 per() (sympy.polys.polytools.Poly method), 704  
 perfect\_power() (in module sympy.nttheory.factor\_), 251  
 periapsis (sympy.geometry.ellipse.Ellipse attribute), 490  
 perimeter (sympy.geometry.polygon.Polygon attribute), 500  
 periodic\_argument (class in sympy.functions.elementary.complexes), 341  
 perm2tensor() (sympy.tensor.tensor.TensMul method), 1109  
 Permutation (class in sympy.combinatorics.permutations), 164  
 PermutationGroup (class in sympy.combinatorics.perm\_groups), 184  
 permute() (sympy.polys.polyclasses.DMP method), 766  
 permuteBkwd() (sympy.matrices.matrices.MatrixBase method), 607  
 permuteFwd() (sympy.matrices.matrices.MatrixBase method), 607  
 perpendicular\_bisector() (sympy.geometry.line.Segment method), 463  
 perpendicular\_line() (sympy.geometry.line.LinearEntity method), 454  
 perpendicular\_line() (sympy.geometry.line3d.LinearEntity3D method), 471  
 perpendicular\_line() (sympy.geometry.plane.Plane method), 520  
 perpendicular\_plane() (sympy.geometry.plane.Plane method), 520  
 perpendicular\_segment() (sympy.geometry.line.LinearEntity method), 454  
 perpendicular\_segment() (sympy.geometry.line3d.LinearEntity3D method), 472  
 pexquo() (in module sympy.polys.polytools), 664  
 pexquo() (sympy.polys.polyclasses.DMP method), 766  
 pexquo() (sympy.polys.polytools.Poly method), 705  
 pgroup (sympy.combinatorics.polyhedron.Polyhedron attribute), 214  
 Pi (class in sympy.core.numbers), 108  
 Piecewise (class in sympy.functions.elementary.piecewise), 332  
 piecewise\_fold() (in module sympy.functions.elementary.piecewise), 332  
 pinv() (sympy.matrices.matrices.MatrixBase method), 607  
 pinv\_solve() (sympy.matrices.matrices.MatrixBase method), 608  
 Plane (class in sympy.geometry.plane), 516  
 Plot (class in sympy.plotting.plot), 872  
 plot() (in module sympy.plotting.plot), 874  
 plot3d() (in module sympy.plotting.plot), 876  
 plot3d\_parametric\_line() (in module sympy.plotting.plot), 876  
 plot3d\_parametric\_surface() (in module sympy.plotting.plot), 877  
 plot\_implicit() (in module sympy.plotting.plot\_implicit), 877  
 plot\_interval() (sympy.geometry.curve.Curve method), 481  
 in plot\_interval() (sympy.geometry.ellipse.Ellipse method), 490  
 plot\_interval() (sympy.geometry.line.Line method), 458  
 plot\_interval() (sympy.geometry.line.Ray method), 460  
 plot\_interval() (sympy.geometry.line.Segment method), 464  
 plot\_interval() (sympy.geometry.line3d.Line3D method), 466  
 plot\_interval() (sympy.geometry.line3d.Ray3D method), 474  
 plot\_interval() (sympy.geometry.line3d.Segment3D method), 478  
 plot\_interval() (sympy.geometry.polygon.Polygon method), 500  
 plot\_parametric() (in module sympy.plotting.plot), 875  
 Point (class in sympy.diffgeom), 1190  
 Point (class in sympy.geometry.point), 437  
 point (sympy.core.function.Subs attribute), 142  
 point() (sympy.diffgeom.CoordSystem method), 1190  
 Point3D (class in sympy.geometry.point3d), 443  
 point\_to\_coords() (sympy.diffgeom.CoordSystem method), 1190  
 points (sympy.geometry.line.LinearEntity attribute), 454  
 points (sympy.geometry.line3d.LinearEntity3D attribute), 472  
 pointwise\_stabilizer() (sympy.combinatorics.perm\_groups.Permutation method), 207  
 Poisson() (in module sympy.stats), 951  
 polar\_lift (class in sympy.functions.elementary.complexes), 340  
 PoleError (class in sympy.core.function), 145

PolificationFailed (class in `sympy.polys.polyerrors`), 839  
pollard\_pm1() (in module `sympy.ntheory.factor_`), 253  
pollard\_rho() (in module `sympy.ntheory.factor_`), 252  
Poly (class in `sympy.polys.polytools`), 679  
poly() (in module `sympy.polys.polytools`), 662  
poly\_from\_expr() (in module `sympy.polys.polytools`), 662  
poly\_ring() (sympy.polys.domains.domain.Domain method), 754  
poly\_unify() (sympy.polys.polyclasses.DMF method), 768  
PolyElement (class in `sympy.polys.rings`), 814  
polygamma (class in `sympy.functions.special.gamma_functions`), 363  
Polygon (class in `sympy.geometry.polygon`), 495  
Polyhedron (class in `sympy.combinatorics.polyhedron`), 213  
polylog (class in `sympy.functions.special.zeta_functions`), 403  
PolynomialError (class in `sympy.polys.polyerrors`), 839  
PolynomialRing (class in `sympy.polys.domains`), 757  
PolyRing (class in `sympy.polys.rings`), 814  
pos() (sympy.polys.domains.domain.Domain method), 754  
POSform() (in module `sympy.logic.boolalg`), 564  
posify() (in module `sympy.simplify.simplify`), 930  
positive, 61  
postfixes() (in module `sympy.utilities.iterables`), 1142  
postorder\_traversal() (in module `sympy.utilities.iterables`), 1143  
Pow (class in `sympy.core.power`), 111  
Pow() (sympy.assumptions.handlers.calculus.AskFiniteHandler static method), 885  
Pow() (sympy.assumptions.handlers.calculus.AskInfiniteHandler static method), 886  
Pow() (sympy.assumptions.handlers.ntheory.AskPrimeHandler static method), 886  
Pow() (sympy.assumptions.handlers.order.AskNegativeHandler static method), 886  
Pow() (sympy.assumptions.handlers.sets.AskAntiHermitianHandler static method), 887  
Pow() (sympy.assumptions.handlers.sets.AskHermitianHandler static method), 887  
Pow() (sympy.assumptions.handlers.sets.AskImaginaryHandler static method), 887  
Pow() (sympy.assumptions.handlers.sets.AskIntegerHandler static method), 888  
Pow() (sympy.assumptions.handlers.sets.AskRationalHandler static method), 888  
Pow() (sympy.assumptions.handlers.sets.AskRealHandler static method), 888  
pow() (sympy.polys.domains.domain.Domain method), 754  
pow() (sympy.polys.polyclasses.ANP method), 769  
pow() (sympy.polys.polyclasses.DMF method), 768  
pow() (sympy.polys.polyclasses.DMP method), 766  
pow() (sympy.polys.polytools.Poly method), 705  
powdenest() (in module `sympy.simplify.simplify`), 930  
powerset() (sympy.sets.sets.Set method), 906  
powsimp() (in module `sympy.simplify.simplify`), 925  
powsimp() (sympy.core.expr.Expr method), 92  
pprint\_nodes() (in module `sympy.printing.tree`), 864  
PQa() (in module `sympy.solvers.diophantine`), 1074  
pquo() (in module `sympy.polys.polytools`), 664  
pquo() (sympy.polys.polyclasses.DMP method), 766  
pquo() (sympy.polys.polytools.Poly method), 705  
PRECEDENCE (in module `sympy.printing.precedence`), 868  
precedence() (in module `sympy.printing.precedence`), 868  
PRECEDENCE\_FUNCTIONS (in module `sympy.printing.precedence`), 868  
PRECEDENCE\_VALUES (in module `sympy.printing.precedence`), 868  
PrecisionExhausted (class in `sympy.core.evalf`), 150  
Predicate (class in `sympy.assumptions.assume`), 882  
preferred\_index (`sympy.functions.special.tensor_functions.KroneckerD` attribute), 428  
prefixes() (in module `sympy.utilities.iterables`), 1143  
prem() (in module `sympy.polys.polytools`), 664  
prem() (sympy.polys.polyclasses.DMP method), 766  
prem() (sympy.polys.polytools.Poly method), 706  
premises (`sympy.categories.Diagram` attribute), 1178  
pretty() (in module `sympy.printing.pretty.pretty`), 869  
pretty\_atom() (in module `sympy.printing.pretty.pretty_symbology`), 869  
Handlerprint() (in module `sympy.printing.pretty.pretty`), 853  
Handlersymbol() (in module `sympy.printing.pretty.pretty_symbology`), 868  
Handlertry\_use\_unicode() (in module `sympy.printing.pretty.pretty_symbology`), 868  
sympy.printing.pretty.pretty\_symbology), 868  
PrettyForm (class in `sympy.printing.pretty.pretty`), 853  
PrettyPrinter (class in `sympy.printing.pretty.pretty`), 853

prev() (sympy.combinatorics.prufer.Prufer method), 217  
 prev\_binary() (sympy.combinatorics.subsets.Subset method), 222  
 prev\_gray() (sympy.combinatorics.subsets.Subset method), 222  
 prev\_lex() (sympy.combinatorics.partitions.IntegerPartitions method), 162  
 prev\_lexicographic() (sympy.combinatorics.subsets.Subsets method), 222  
 preview() (in module sympy.printing.preview), 866  
 preview\_diagram() (in module sympy.categories.diagram\_drawing), 1187  
 prevprime() (in module sympy.nttheory.generate), 247  
 prime, 61  
 prime() (in module sympy.nttheory.generate), 246  
 prime\_as\_sum\_of\_two\_squares() (in module sympy.solvers.diophantine), 1077  
 primefactors() (in module sympy.nttheory.factor\_), 257  
 primepi() (in module sympy.nttheory.generate), 246  
 primerange() (in module sympy.nttheory.generate), 247  
 primerange() (sympy.nttheory.generate.Sieve method), 245  
 primitive() (in module sympy.polys.polytools), 672  
 primitive() (sympy.core.add.Add method), 117  
 primitive() (sympy.core.expr.Expr method), 92  
 primitive() (sympy.polys.polyclasses.DMP method), 766  
 primitive() (sympy.polys.polytools.Poly method), 706  
 primitive() (sympy.polys.rings.PolyElement method), 819  
 primitive\_element() (in module sympy.polys.numberfields), 720  
 primitive\_root() (in module sympy.nttheory.residue\_nttheory), 264  
 primorial() (in module sympy.nttheory.generate), 248  
 principal\_branch (class in sympy.functions.elementary.complexes), 341  
 print\_ccode() (in module sympy.printing.ccode), 856  
 print\_fcode() (in module sympy.printing.fcode), 858  
 print\_gtk() (in module sympy.printing.gtk), 861  
 print\_latex() (in module sympy.printing.latex), 863  
 print\_mathml() (in module sympy.printing.mathml), 863  
 print\_node() (in module sympy.printing.tree), 865  
 print\_nonzero() (sympy.matrices.matrices.MatrixBase method), 609  
 print\_tree() (in module sympy.printing.tree), 865  
 Printer (class in sympy.printing.printer), 851  
 printmethod (sympy.printing.ccode.CCodePrinter attribute), 854  
 printmethod (sympy.printing.codeprinter.CodePrinter attribute), 868  
 printmethod (sympy.printing.fcode.FCodePrinter attribute), 858  
 printmethod (sympy.printing.lambdarepr.LambdaPrinter attribute), 861  
 printmethod (sympy.printing.latex.LatexPrinter attribute), 861  
 printmethod (sympy.printing.mathematica.MCodePrinter attribute), 860  
 printmethod (sympy.printing.mathml.MathMLPrinter attribute), 863  
 printmethod (sympy.printing.pretty.pretty.PrettyPrinter attribute), 853  
 printmethod (sympy.printing.printer.Printer attribute), 852  
 printmethod (sympy.printing.repr.ReprPrinter attribute), 864  
 printmethod (sympy.printing.str.StrPrinter attribute), 864  
 printtoken() (in module sympy.parsing.sympy\_tokenize), 1162  
 prod() (in module sympy.core.mul), 115  
 Product (class in sympy.concrete.products), 292  
 product() (in module sympy.concrete.products), 301  
 product() (sympy.polys.agca.ideals.Ideal method), 742  
 ProductDomain (class in sympy.stats.rv), 981  
 ProductPSpace (class in sympy.stats.rv), 981  
 ProductSet (class in sympy.sets.sets), 912  
 project() (sympy.matrices.matrices.MatrixBase method), 609  
 projection() (sympy.geometry.line.LinearEntity method), 455  
 projection() (sympy.geometry.line3d.LinearEntity3D method), 473  
 projection() (sympy.geometry.plane.Plane method), 520  
 projection\_line() (sympy.geometry.plane.Plane method), 521  
 Prufer (class in sympy.combinatorics.prufer), 216  
 prufer\_rank() (sympy.combinatorics.prufer.Prufer method), 217  
 prufer\_repr (sympy.combinatorics.prufer.Prufer attribute), 217  
 PSpace (class in sympy.stats.rv), 981  
 pspace() (in module sympy.stats.rv), 982  
 public() (in module sympy.utilities.decorator), 1124  
 PurePoly (class in sympy.polys.polytools), 715  
 PyTestReporter (class in sympy.utilities.runtests), 1154  
 Python Enhancement Proposals

PEP 335, 1234–1236

PythonFiniteField (class in `sympy.polys.domains`),  
762  
PythonIntegerRing (class in `sympy.polys.domains`),  
762  
PythonRationalField (class in `sympy.polys.domains`),  
762

## Q

Q (class in `sympy.assumptions.ask`), 880  
QRdecomposition() (`sympy.matrices.matrices.MatrixBase`  
method), 584  
QRsolve() (`sympy.matrices.matrices.MatrixBase`  
method), 585  
quadratic\_residues() (in module  
`sympy.nttheory.residue_nttheory`), 265  
QuadraticU() (in module `sympy.stats`), 969  
quo() (in module `sympy.polys.polytools`), 665  
quo() (`sympy.polys.domains.domain.Domain`  
method), 754  
quo() (`sympy.polys.domains.field.Field` method), 755  
quo() (`sympy.polys.domains.ring.Ring` method), 756  
quo() (`sympy.polys.polyclasses.DMF` method), 768  
quo() (`sympy.polys.polyclasses.DMP` method), 766  
quo() (`sympy.polys.polytools.Poly` method), 706  
quo\_ground() (`sympy.polys.polyclasses.DMP`  
method), 766  
quo\_ground() (`sympy.polys.polytools.Poly` method),  
706  
quotient() (`sympy.polys.agca.ideals.Ideal` method),  
742  
quotient\_codomain() (`sympy.polys.agca.homomorphisms`.  
method), 749  
quotient\_domain() (`sympy.polys.agca.homomorphisms`.  
method), 750  
quotient\_hom() (`sympy.polys.agca.modules.QuotientModule`  
method), 744  
quotient\_hom() (`sympy.polys.agca.modules.SubQuotient`  
method), 745  
quotient\_module() (`sympy.polys.agca.modules.FreeModule`  
method), 736  
quotient\_module() (`sympy.polys.agca.modules.Module`  
method), 735  
quotient\_module() (`sympy.polys.agca.modules.SubModule`  
method), 739  
quotient\_ring() (`sympy.polys.domains.ring.Ring`  
method), 756  
QuotientModule (class  
`sympy.polys.agca.modules`), 743  
QuotientModuleElement (class  
`sympy.polys.agca.modules`), 745

## R

rad\_rationalize() (in module  
`sympy.simplify.simplify`), 921  
radical() (`sympy.polys.agca.ideals.Ideal` method), 742  
radius (`sympy.geometry.ellipse.Circle` attribute), 494  
radius (`sympy.geometry.polygon.RegularPolygon` at-  
tribute), 507  
radius\_of\_convergence (`sympy.functions.special.hyper.hyper`  
attribute), 407  
radsimp() (in module `sympy.simplify.simplify`), 922  
radsimp() (`sympy.core.expr.Expr` method), 93  
RaisedCosine() (in module `sympy.stats`), 970  
raises() (in module `sympy.utilities.pytest`), 1152  
randMatrix() (in module `sympy.matrices.dense`), 619  
random() (`sympy.combinatorics.perm_groups.PermutationGroup`  
method), 208  
random() (`sympy.combinatorics.permutations.Permutation`  
class method), 177  
random\_bitstring() (`sympy.combinatorics.graycode`  
static method), 228  
random\_complex\_number() (in module  
`sympy.utilities.randtest`), 1153  
random\_integer\_partition() (in module  
`sympy.combinatorics.partitions`), 163  
random\_point() (`sympy.geometry.ellipse.Ellipse`  
method), 490  
random\_point() (`sympy.geometry.line.LinearEntity`  
method), 455  
random\_point() (`sympy.geometry.plane.Plane`  
method), 521  
random\_poly() (in module `sympy.polys.specialpolys`),  
~~ModuleHomomorphism~~, 160  
random\_pr() (`sympy.combinatorics.perm_groups.PermutationGroup`  
~~ModuleHomomorphism~~, 160  
random\_stab() (`sympy.combinatorics.perm_groups.PermutationGroup`  
method), 208  
random\_symbols() (in module `sympy.stats.rv`), 982  
RandomDomain (class in `sympy.stats.rv`), 981  
RandomSymbol (class in `sympy.stats.rv`), 981  
randprime() (in module `sympy.nttheory.generate`), 248  
ranges (`sympy.tensor.indexed.Indexed` attribute),  
1090  
rank (`sympy.combinatorics.graycode.GrayCode` at-  
tribute), 227  
rank (`sympy.combinatorics.partitions.Partition` at-  
tribute), 161  
rank (`sympy.combinatorics.prufer.Prufer` attribute),  
218  
rank (`sympy.tensor.indexed.Indexed` attribute), 1091  
rank() (`sympy.combinatorics.permutations.Permutation`  
method), 177  
rank() (`sympy.matrices.matrices.MatrixBase`  
method), 609

rank\_binary (sympy.combinatorics.subsets.Subset attribute), 223  
 rank\_gray (sympy.combinatorics.subsets.Subset attribute), 223  
 rank\_lexicographic (sympy.combinatorics.subsets.Subset attribute), 223  
 rank\_nonlex() (sympy.combinatorics.permutations.Permutations method), 177  
 rank\_trotterjohnson() (sympy.combinatorics.permutations method), 178  
 rat\_clear\_denoms() (sympy.polys.polytools.Poly method), 707  
 ratint() (in module sympy.integrals.rationaltools), 546  
 ratint\_logpart() (in module sympy.integrals.rationaltools), 547  
 ratint\_ratpart() (in module sympy.integrals.rationaltools), 547  
 rational, 61  
 Rational (class in sympy.core.numbers), 101  
 RationalField (class in sympy.polys.domains), 758  
 rationalize() (in module sympy.parsing.sympy\_parser), 1165  
 ratsimp() (in module sympy.simplify.simplify), 923  
 ratsimp() (sympy.core.expr.Expr method), 93  
 rawlines() (in module sympy.utilities.misc), 1151  
 Ray (class in sympy.geometry.line), 458  
 Ray3D (class in sympy.geometry.line3d), 473  
 Rayleigh() (in module sympy.stats), 970  
 rcall() (sympy.core.basic.Basic method), 68  
 rcollect() (in module sympy.simplify.simplify), 920  
 re (class in sympy.functions.elementary.complexes), 333  
 real, 61  
 real\_root() (in module sympy.functions.elementary.miscellaneous), 333  
 real\_roots() (in module sympy.polys.polytools), 677  
 real\_roots() (sympy.polys.polytools.Poly method), 707  
 RealField (class in sympy.polys.domains), 760  
 RealNumber (in module sympy.core.numbers), 103  
 reconstruct() (in module sympy.solvers.diophantine), 1078  
 recurrence\_memo() (in module sympy.utilities.memoization), 1150  
 red\_groebner() (in module sympy.polys.groebnertools), 836  
 reduce() (sympy.polys.polytools.GroebnerBasis method), 716  
 reduce() (sympy.sets.sets.Complement static method), 913  
 reduce() (sympy.sets.sets.Intersection static method), 911  
 reduce() (sympy.sets.sets.Union static method), 911  
 reduce\_abs\_inequalities() (in module sympy.solvers.inequalities), 1080  
 reduce\_abs\_inequality() (in module sympy.solvers.inequalities), 1080  
 reduce\_element() (sympy.polys.agca.ideals.IdealPermutation method), 742  
 reduce\_element() (sympy.polys.agca.modules.SubModule method), 739  
 reduce\_inequalities() (in module sympy.solvers.inequalities), 1080  
 reduce\_rational\_inequalities() (in module sympy.solvers.inequalities), 1079  
 reduced() (in module sympy.polys.polytools), 678  
 refine() (in module sympy.assumptions.refine), 882  
 refine() (sympy.core.expr.Expr method), 93  
 refine\_abs() (in module sympy.assumptions.refine), 883  
 refine\_atan2() (in module sympy.assumptions.refine), 883  
 refine\_exp() (in module sympy.assumptions.refine), 884  
 refine\_Pow() (in module sympy.assumptions.refine), 883  
 refine\_Relational() (in module sympy.assumptions.refine), 883  
 refine\_root() (in module sympy.polys.polytools), 676  
 refine\_root() (sympy.polys.polyclasses.DMP method), 766  
 refine\_root() (sympy.polys.polytools.Poly method), 707  
 RefinementFailed (class in sympy.polys.polyerrors), 838  
 reflect() (sympy.geometry.ellipse.Circle method), 495  
 reflect() (sympy.geometry.ellipse.Ellipse method), 491  
 reflect() (sympy.geometry.polygon.RegularPolygon method), 507  
 register\_handler() (in module sympy.assumptions.ask), 880  
 RegularPolygon (class in sympy.geometry.polygon), 502  
 Rel (in module sympy.core.relational), 118  
 Relational (class in sympy.core.relational), 119  
 rem() (in module sympy.polys.polytools), 665  
 rem() (sympy.polys.domains.domain.Domain method), 754  
 rem() (sympy.polys.domains.field.Field method), 755  
 rem() (sympy.polys.domains.ring.Ring method), 756  
 rem() (sympy.polys.polyclasses.DMP method), 766  
 rem() (sympy.polys.polytools.Poly method), 707  
 remove\_handler() (in module sympy.assumptions.ask), 881  
 removeO() (sympy.core.expr.Expr method), 93

render() (sympy.printing.pretty.stringPict.stringPict method), 870  
reorder() (sympy.concrete.expr\_with\_intlimits.ExprWithIntLimits method), 299  
reorder() (sympy.polys.polytools.Poly method), 708  
reorder\_limit() (sympy.concrete.expr\_with\_intlimits.ExprWithIntLimits method), 300  
replace() (sympy.core.basic.Basic method), 68  
replace() (sympy.matrices.matrices.MatrixBase method), 610  
replace() (sympy.polys.polytools.Poly method), 708  
Reporter (class in sympy.utilities.runtests), 1154  
reprify() (sympy.printing.repr.ReprPrinter method), 864  
ReprPrinter (class in sympy.printing.repr), 864  
reset() (sympy.combinatorics.polyhedron.Polyhedron method), 214  
reshape() (in module sympy.utilities.iterables), 1143  
reshape() (sympy.matrices.dense.DenseMatrix method), 625  
reshape() (sympy.matrices.sparse.SparseMatrix method), 637  
residue() (in module sympy.series.residues), 900  
restrict\_codomain() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 750  
restrict\_domain() (sympy.polys.agca.homomorphisms.ModuleHomomorphism method), 750  
Result (class in sympy.utilitiescodegen), 1117  
result\_variables (sympy.utilitiescodegen.Routine attribute), 1116  
resultant() (in module sympy.polys.polytools), 667  
resultant() (sympy.polys.polyclasses.DMP method), 766  
resultant() (sympy.polys.polytools.Poly method), 708  
retract() (sympy.polys.polytools.Poly method), 708  
reverse\_order() (sympy.concrete.products.Product method), 295  
reverse\_order() (sympy.concrete.summations.Sum method), 292  
reversed (sympy.core.relational.Relational attribute), 120  
ReversedGradedLexOrder (class in sympy.polys.orderings), 722  
revert() (sympy.polys.domains.domain.Domain method), 754  
revert() (sympy.polys.domains.field.Field method), 755  
revert() (sympy.polys.domains.ring.Ring method), 756  
revert() (sympy.polys.polyclasses.DMP method), 766  
revert() (sympy.polys.polytools.Poly method), 709  
rewrite() (in module sympy.series.gruntz), 903  
rewrite() (sympy.core.basic.Basic method), 70  
RGS (sympy.combinatorics.partitions.Partition attribute), 160  
RGS.items() (in module sympy.combinatorics.partitions), 163  
RGS\_generalized() (in module sympy.combinatorics.partitions), 163  
RGS\_rank() (in module sympy.combinatorics.partitions), 164  
RGS\_unrank() (in module sympy.combinatorics.partitions), 164  
rhs (sympy.core.relational.Relational attribute), 120  
riemann\_cyclic() (in module sympy.tensor.tensor), 1109  
riemann\_cyclic\_replace() (in module sympy.tensor.tensor), 1109  
right (sympy.sets.sets.Interval attribute), 909  
right() (sympy.printing.pretty.stringPict.stringPict method), 870  
right\_open (sympy.sets.sets.Interval attribute), 909  
Ring (class in sympy.polys.domains.ring), 755  
ring() (in module sympy.polys.rings), 812  
RisingFactorial (class in sympy.functions.combinatorial.factorials), 851  
RL (sympy.matrices.sparse.SparseMatrix attribute), 663  
rmul() (sympy.combinatorics.permutations.Permutation static method), 178  
rmul\_with\_af() (sympy.combinatorics.permutations.Permutation static method), 179  
root (in module sympy.printing.pretty.pretty\_symbology), 869  
root() (in module sympy.functions.elementary.miscellaneous), 334  
root() (sympy.polys.polytools.Poly method), 709  
root() (sympy.printing.pretty.stringPict.stringPict method), 870  
RootOf (class in sympy.polys.rootoftools), 722  
roots() (in module sympy.polys.polyroots), 722  
RootSum (class in sympy.polys.rootoftools), 722  
Ropen() (sympy.sets.sets.Interval class method), 908  
rot\_axis1() (in module sympy.matrices.dense), 621  
rot\_axis2() (in module sympy.matrices.dense), 622  
rot\_axis3() (in module sympy.matrices.dense), 622  
rotate() (sympy.combinatorics.polyhedron.Polyhedron method), 214  
rotate() (sympy.geometry.curve.Curve method), 481  
rotate() (sympy.geometry.ellipse.Ellipse method), 491  
rotate() (sympy.geometry.entity.GeometryEntity method), 433  
rotate() (sympy.geometry.point.Point method), 441  
rotate() (sympy.geometry.polygon.RegularPolygon method), 507

rotate\_left() (in module sympy.utilities.iterables), 1144  
 rotate\_right() (in module sympy.utilities.iterables), 1144  
 rotation (sympy.geometry.polygon.RegularPolygon attribute), 508  
 round() (sympy.core.expr.Expr method), 93  
 RoundFunction (class in sympy.functions.elementary.integers), 335  
 Routine (class in sympy.utilitiescodegen), 1116  
 routine() (sympy.utilitiescodegen.CodeGen method), 1117  
 routine() (sympy.utilitiescodegen.OctaveCodeGen method), 1121  
 row() (sympy.matrices.dense.DenseMatrix method), 625  
 row() (sympy.matrices.sparse.SparseMatrix method), 637  
 row\_del() (sympy.matrices.dense.MutableDenseMatrix method), 628  
 row\_del() (sympy.matrices.sparse.MutableSparseMatrix method), 642  
 row\_insert() (sympy.matrices.matrices.MatrixBase method), 610  
 row\_join() (sympy.matrices.matrices.MatrixBase method), 610  
 row\_join() (sympy.matrices.sparse.MutableSparseMatrix method), 642  
 row\_list() (sympy.matrices.sparse.SparseMatrix method), 638  
 row\_op() (sympy.matrices.dense.MutableDenseMatrix method), 629  
 row\_op() (sympy.matrices.sparse.MutableSparseMatrix method), 643  
 row\_structure\_symbolic\_cholesky() (sympy.matrices.sparse.SparseMatrix method), 638  
 row\_swap() (sympy.matrices.dense.MutableDenseMatrix method), 629  
 row\_swap() (sympy.matrices.sparse.MutableSparseMatrix method), 643  
 rref() (sympy.matrices.matrices.MatrixBase method), 611  
 rs\_compose.add() (in module sympy.polys.ring\_series), 848  
 rs\_exp() (in module sympy.polys.ring\_series), 847  
 rs\_hadamard\_exp() (in module sympy.polys.ring\_series), 848  
 rs\_integrate() (in module sympy.polys.ring\_series), 847  
 rs\_log() (in module sympy.polys.ring\_series), 847  
 rs\_mul() (in module sympy.polys.ring\_series), 845  
 rs\_newton() (in module sympy.polys.ring\_series), 847  
 rs\_pow() (in module sympy.polys.ring\_series), 846  
 rs\_series\_from\_list() (in module sympy.polys.ring\_series), 846  
 rs\_series\_inversion() (in module sympy.polys.ring\_series), 846  
 rs\_square() (in module sympy.polys.ring\_series), 845  
 rs\_swap() (in module sympy.stats.rv), 982  
 rs\_trunc() (in module sympy.polys.ring\_series), 845  
 rsa\_private\_key() (in module sympy.crypto.crypto), 282  
 rsa\_public\_key() (in module sympy.crypto.crypto), 282  
 rsolve() (in module sympy.solvers.recurr), 1056  
 rsolve\_hyper() (in module sympy.solvers.recurr), 1058  
 rsolve\_poly() (in module sympy.solvers.recurr), 1057  
 rsolve\_ratio() (in module sympy.solvers.recurr), 1058  
 run() (sympy.utilities.runtests.SymPyDocTestRunner method), 1154  
 run\_all\_tests() (in module sympy.utilities.runtests), 1156  
 run\_in\_subprocess\_with\_hash\_randomization() (in module sympy.utilities.runtests), 1156  
 runs() (in module sympy.utilities.iterables), 1144  
 runs() (sympy.combinatorics.permutations.Permutation method), 179

## S

S (in module sympy.core.singleton), 74  
 sample() (in module sympy.stats), 980  
 sample\_iter() (in module sympy.stats), 980  
 sample\_iter\_lambdify() (in module sympy.stats.rv), 983  
 sample\_iter\_subs() (in module sympy.stats.rv), 983  
 sampling\_density() (in module sympy.stats.rv), 983  
 sampling\_E() (in module sympy.stats.rv), 982  
 sampling\_P() (in module sympy.stats.rv), 982  
 satisfiable() (in module sympy.logic.inference), 572  
 saturate() (sympy.polys.agca.ideals.Ideal method), 742  
 scalar\_multiply() (sympy.matrices.sparse.SparseMatrix method), 638  
 scale() (sympy.geometry.curve.Curve method), 481  
 scale() (sympy.geometry.ellipse.Circle method), 495  
 scale() (sympy.geometry.ellipse.Ellipse method), 492  
 scale() (sympy.geometry.entity.GeometryEntity method), 433  
 scale() (sympy.geometry.point.Point method), 441  
 scale() (sympy.geometry.point3d.Point3D method), 447  
 scale() (sympy.geometry.polygon.RegularPolygon method), 508  
 schreier\_sims() (sympy.combinatorics.perm\_groups.PermutationGroup method), 208

schreier\_sims\_incremental()  
    (sympy.combinatorics.perm\_groups.PermutationGroup method), 208  
schreier\_sims\_random()  
    (sympy.combinatorics.perm\_groups.PermutationGroup method), 209  
schreier\_vector() (sympy.combinatorics.perm\_groups.PermutationGroup method), 210  
sdm\_add() (in module sympy.polys.distributedmodules), 821  
sdm\_deg() (in module sympy.polys.distributedmodules), 823  
sdm\_ecart() (in module sympy.polys.distributedmodules), 837  
sdm\_from\_dict() (in module sympy.polys.distributedmodules), 821  
sdm\_from\_vector() (in module sympy.polys.distributedmodules), 823  
sdm\_groebner() (in module sympy.polys.distributedmodules), 838  
sdm\_LC() (in module sympy.polys.distributedmodules), 821  
sdm\_LM() (in module sympy.polys.distributedmodules), 822  
sdm\_LT() (in module sympy.polys.distributedmodules), 822  
sdm\_monomial\_deg() (in module sympy.polys.distributedmodules), 820  
sdm\_monomial\_divides() (in module sympy.polys.distributedmodules), 820  
sdm\_monomial\_mul() (in module sympy.polys.distributedmodules), 820  
sdm\_mul\_term() (in module sympy.polys.distributedmodules), 822  
sdm\_nf\_mora() (in module sympy.polys.distributedmodules), 837  
sdm\_spoly() (in module sympy.polys.distributedmodules), 837  
sdm\_to\_dict() (in module sympy.polys.distributedmodules), 821  
sdm\_to\_vector() (in module sympy.polys.distributedmodules), 823  
sdm\_zero() (in module sympy.polys.distributedmodules), 823  
search() (sympy.ntheory.generate.Sieve method), 245  
sec (class in sympy.functions.elementary.trigonometric)  
    337  
Segment (class in sympy.geometry.line), 461  
Segment3D (class in sympy.geometry.line3d), 476  
select() (sympy.simplify.epathtools.EPath method), 937  
selections (sympy.combinatorics.graycode.GrayCode attribute), 228  
separate() (sympy.core.expr.Expr method), 94  
separatevars() (in module sympy.simplify.simplify), 920  
series() (in module sympy.series.series), 898  
series() (sympy.core.expr.Expr method), 94  
SGC(~~oops~~) in sympy.sets.sets), 903  
set\_comm() (sympy.tensor.tensor.\_TensorManager PermutationGroup method), 1096  
set\_comms() (sympy.tensor.tensor.\_TensorManager method), 1097  
set\_domain() (sympy.polys.polytools.Poly method), 709  
set\_global\_settings() (sympy.printing.printer.Printer class method), 853  
set\_modulus() (sympy.polys.polytools.Poly method), 709  
seterr() (in module sympy.core.numbers), 104  
setup() (in module sympy.polys.polyconfig), 849  
shape (sympy.matrices.matrices.MatrixBase attribute), 611  
shape (sympy.tensor.indexed.Indexed attribute), 1091  
shape (sympy.tensor.indexed.IndexedBase attribute), 1092  
ShapeError (class in sympy.matrices.matrices), 615  
Shi (class in sympy.functions.special.error\_functions), 387  
shift() (sympy.polys.polyclasses.DMP method), 766  
shift() (sympy.polys.polytools.Poly method), 709  
Si (class in sympy.functions.special.error\_functions), 384  
sides (sympy.geometry.polygon.Polygon attribute), 501  
Sieve (class in sympy.ntheory.generate), 244  
sift() (in module sympy.utilities.iterables), 1145  
sign (class in sympy.functions.elementary.complexes), 339  
sign() (in module sympy.series.gruntz), 903  
signature() (sympy.combinatorics.permutations.Permutation method), 179  
SimpleDomain (class in sympy.polys.domains.simpledomain), 756  
simplified() (in module sympy.solvers.diophantine), 1075  
simplify() (in module sympy.simplify.simplify), 916  
simplify() (sympy.core.expr.Expr method), 94  
simplify() (sympy.functions.special.delta\_functions.DiracDelta method), 359  
simplify() (sympy.matrices.dense.MutableDenseMatrix method), 629  
simplify() (sympy.matrices.matrices.MatrixBase method), 611  
simplify\_logic() (in module sympy.logic.boolalg), 571

sin (class in `sympy.functions.elementary.trigonometric`), `solve_rational_inequalities()` (in module `sympy.solvers.inequalities`), 1078  
335

sine\_transform() (in module `sympy.integrals.transforms`), 526

SineTransform (class in `sympy.integrals.transforms`), 556

SingleDomain (class in `sympy.stats.rv`), 981

SinglePSpace (class in `sympy.stats.rv`), 981

singular\_values() (`sympy.matrices.matrices.MatrixBase` method), 611

singularities() (in module `sympy.calculus.singularities`), 1166

sinh (class in `sympy.functions.elementary.hyperbolic`), 336

size (`sympy.combinatorics.permutations.Permutation` attribute), 179

size (`sympy.combinatorics.polyhedron.Polyhedron` attribute), 215

size (`sympy.combinatorics.prufer.Prufer` attribute), 218

size (`sympy.combinatorics.subsets.Subset` attribute), 223

skip() (`sympy.combinatorics.graycode.GrayCode` method), 228

slice() (`sympy.polys.polyclasses.DMP` method), 766

slice() (`sympy.polys.polytools.Poly` method), 710

slope (`sympy.geometry.line.LinearEntity` attribute), 456

smoothness() (in module `sympy.nttheory.factor_`), 250

smoothness\_p() (in module `sympy.nttheory.factor_`), 250

solve() (in module `sympy.solvers.solvers`), 1045

solve() (in module `sympy.matrices.matrices.MatrixBase` method), 612

solve() (`sympy.matrices.sparse.SparseMatrix` method), 638

solve\_congruence() (in module `sympy.nttheory.modular`), 261

solve\_least\_squares() (`sympy.matrices.matrices.MatrixBase` method), 612

solve\_least\_squares() (`sympy.matrices.sparse.SparseMatrix` method), 638

solve\_linear() (in module `sympy.solvers.solvers`), 1050

solve\_linear\_system() (in module `sympy.solvers.solvers`), 1051

solve\_linear\_system\_LU() (in module `sympy.solvers.solvers`), 1052

solve\_poly\_inequalities() (in module `sympy.solvers.inequalities`), 1079

solve\_poly\_inequality() (in module `sympy.solvers.inequalities`), 1079

solve\_poly\_system() (in module `sympy.solvers.polysys`), 1059

solve\_triangulated() (in module `sympy.solvers.polysys`), 1059

solve\_undetermined\_coeffs() (in module `sympy.solvers.solvers`), 1052

solve\_univariate\_inequality() (in module `sympy.solvers.inequalities`), 1081

solveset() (in module `sympy.solvers.solveset`), 1081

solveset\_complex() (in module `sympy.solvers.solveset`), 1084

solveset\_real() (in module `sympy.solvers.solveset`), 1083

SOPform() (in module `sympy.logic.boolalg`), 563

sort\_key() (`sympy.combinatorics.partitions.Partition` method), 161

sort\_key() (`sympy.core.basic.Basic` method), 71

sorted\_components() (`sympy.tensor.tensor.TensMul` method), 1109

source (`sympy.geometry.line.Ray` attribute), 460

source (`sympy.geometry.line3d.Ray3D` attribute), 475

source() (in module `sympy.utilities.source`), 1161

SparseMatrix (class in `sympy.matrices.sparse`), 632

spherical\_bessel\_fn() (in module `sympy.polys.orthopolys`), 723

spin() (`sympy.geometry.polygon.RegularPolygon` method), 508

split() (`sympy.tensor.tensor.TensMul` method), 1109

split\_list() (in module `sympy.utilities.runtests`), 1157

split\_super\_sub() (in module `sympy.printing.conventions`), 867

split\_symbols() (in module `sympy.parsing.sympy_parser`), 1163

split\_symbols\_custom() (in module `sympy.parsing.sympy_parser`), 1163

spoly() (in module `sympy.polys.groebnertools`), 836

sqf() (in module `sympy.polys.polytools`), 674

sqf\_list() (in module `sympy.polys.polytools`), 674

sqf\_list() (`sympy.polys.polyclasses.DMP` method), 766

sqf\_list() (`sympy.polys.polytools.Poly` method), 710

sqf\_list\_include() (in module `sympy.polys.polyclasses.DMP` method), 767

sqf\_list\_include() (`sympy.polys.polytools.Poly` method), 710

sqf\_norm() (in module `sympy.polys.polytools`), 674

sqf\_norm() (`sympy.polys.polyclasses.DMP` method), 767

sqf\_norm() (`sympy.polys.polytools.Poly` method), 711

sqf\_part() (in module `sympy.polys.polytools`), 674

sqf\_part() (`sympy.polys.polyclasses.DMP` method), 767

sqf\_part() (`sympy.polys.polytools.Poly` method), 711

sqr() (`sympy.polys.polyclasses.DMP` method), 767

sqr() (sympy.polys.polytools.Poly method), 711  
sqrt() (in module sympy.functions.elementary.miscellaneous), 712  
    337  
sqrt() (sympy.polys.domains.domain.Domain method), 754  
sqrt\_mod() (in module sympy.nttheory.residue\_nttheory), 265  
sqrtdenest() (in module sympy.simplify.sqrtdenest), 933  
square() (sympy.polys.rings.PolyElement method), 819  
square\_factor() (in module sympy.solvers.diophantine), 1071  
srepr() (in module sympy.printing.repr), 864  
string() (in module sympy.polys.rings), 813  
sstrrepr() (in module sympy.printing.str), 864  
stabilizer() (sympy.combinatorics.perm\_groups.PermutationGroup method), 210  
stack() (sympy.printing.pretty.stringpict.StringPict static method), 870  
standard\_transformations (in module sympy.parsing.sympy\_parser), 1163  
start (sympy.sets.sets.Interval attribute), 910  
std() (in module sympy.stats), 980  
stirling() (in module sympy.functions.combinatorial.numbers), 354  
StopTokenizing (class in sympy.parsing.sympy\_tokenize), 1163  
StrictGreaterThan (class in sympy.core.relational), 127  
StrictLessThan (class in sympy.core.relational), 130  
stringify\_expr() (in module sympy.parsing.sympy\_parser), 1162  
StringPict (class in sympy.printing.pretty.stringpict), 869  
strip\_zero() (sympy.polys.rings.PolyElement method), 819  
strong\_gens (sympy.combinatorics.perm\_groups.PermutationGroup attribute), 211  
StrPrinter (class in sympy.printing.str), 864  
StudentT() (in module sympy.stats), 971  
sturm() (in module sympy.polys.polytools), 673  
sturm() (sympy.polys.polyclasses.DMP method), 767  
sturm() (sympy.polys.polytools.Poly method), 711  
sub (in module sympy.printing.pretty.pretty\_symbology), 868  
sub() (sympy.polys.domains.domain.Domain method), 754  
sub() (sympy.polys.polyclasses.DMF method), 768  
sub() (sympy.polys.polyclasses.DMP method), 767  
sub() (sympy.polys.polytools.Poly method), 712  
sub\_ground() (sympy.polys.polyclasses.DMP method), 767  
sub\_ground() (sympy.polys.polytools.Poly method), 712  
subdiagram\_from\_objects() (sympy.categories.Diagram method), 1179  
subfactorial (class in sympy.functions.combinatorial.factorials), 348  
subgroup\_search() (sympy.combinatorics.perm\_groups.PermutationGroup method), 211  
SubModule (class in sympy.polys.agca.modules), 736  
submodule() (sympy.polys.agca.modules.Module method), 735  
submodule() (sympy.polys.agca.modules.QuotientModule method), 744  
submodule() (sympy.polys.agca.modules.SubModule method), 739  
SubQuotientModule (class in sympy.polys.agca.modules), 744  
subresultants() (in module sympy.polys.polytools), 666  
subresultants() (sympy.polys.polyclasses.DMP method), 767  
subresultants() (sympy.polys.polytools.Poly method), 712  
Subs (class in sympy.core.function), 142  
subs() (sympy.core.basic.Basic method), 71  
subs() (sympy.matrices.immutable.ImmutableSparseMatrix method), 644  
subs() (sympy.matrices.matrices.MatrixBase method), 613  
Subset (class in sympy.combinatorics.subsets), 219  
subset (sympy.combinatorics.subsets.Subset attribute), 224  
subset() (sympy.polys.agca.ideals.Ideal method), 742  
subset() (sympy.polys.agca.modules.Module method), 735  
subset\_from\_bitlist() (sympy.combinatorics.subsets.Subset class method), 224  
subset\_indices() (sympy.combinatorics.subsets.Subset class method), 224  
subsets() (in module sympy.utilities.iterables), 1145  
SubsSet (class in sympy.series.gruntz), 901  
substitute\_indices() (sympy.tensor.tensor.TensAdd method), 1107  
Sum (class in sympy.concrete.summations), 289  
    sum\_of\_four\_squares() (in module sympy.solvers.diophantine), 1073  
    sum\_of\_three\_squares() (in module sympy.solvers.diophantine), 1073  
summation() (in module sympy.concrete.summations), 300  
sup (in module sympy.printing.pretty.pretty\_symbology), 868

sup (sympy.sets.sets.Set attribute), 906  
 superset (sympy.combinatorics.subsets.Subset attribute), 224  
 superset\_size (sympy.combinatorics.subsets.Subset attribute), 225  
 support() (sympy.combinatorics.permutations.Permutation method), 180  
 SurfaceBaseSeries (class in sympy.plotting.plot), 879  
 SurfaceOver2DRangeSeries (class in sympy.plotting.plot), 879  
 swinnerton\_dyner\_poly() (in module sympy.polys.specialpolys), 723  
 symarray() (in module sympy.matrices.dense), 620  
 symb\_2txt (in module sympy.printing.pretty.pretty\_symbology), 868  
 Symbol (class in sympy.core.symbol), 95  
 Symbol() (sympy.assumptions.handlers.calculus.AskFinitely module), 885  
 symbols() (in module sympy.core.symbol), 96  
 symmetric() (sympy.combinatorics.generators static method), 183  
 symmetric\_poly() (in module sympy.polys.specialpolys), 723  
 symmetric\_residue() (in module sympy.nttheory.modular), 259  
 SymmetricGroup() (in module sympy.combinatorics.named\_groups), 230  
 symmetrize() (in module sympy.polys.polyfuncs), 717  
 sympify() (in module sympy.core.sympify), 59  
 sympy (module), 59  
 sympy.assumptions (module), 880  
 sympy.assumptions.ask (module), 880  
 sympy.assumptionsassume (module), 881  
 sympy.assumptions.handlers (module), 884  
 sympy.assumptions.handlers.calculus (module), 884  
 sympy.assumptions.handlers.nttheory (module), 886  
 sympy.assumptions.handlers.order (module), 886  
 sympy.assumptions.handlers.sets (module), 887  
 sympy.assumptions.refine (module), 882  
 sympy.calculus (module), 1165  
 sympy.calculus.euler (module), 1165  
 sympy.calculus.finite\_diff (module), 1167  
 sympy.calculus.singularities (module), 1166  
 sympy.categories (module), 1171  
 sympy.categories.diagram\_drawing (module), 1179  
 sympy.combinatorics.generators (module), 183  
 sympy.combinatorics.graycode (module), 226  
 sympy.combinatorics.group\_constructs (module), 238  
 sympy.combinatorics.named\_groups (module), 230  
 sympy.combinatorics.partitions (module), 160  
 sympy.combinatorics.perm\_groups (module), 184  
 sympy.combinatorics.permutations (module), 164  
 sympy.combinatorics.polyhedron (module), 213  
 sympy.combinatorics.prufer (module), 216  
 sympy.combinatorics.subsets (module), 219  
 sympy.combinatorics.tensor\_can (module), 239  
 sympy.combinatorics.util (module), 232  
 sympy.core.add (module), 115  
 sympy.core.assumptions (module), 60  
 sympy.core.basic (module), 62  
 sympy.core.cache (module), 62  
 sympy.core.compatibility (module), 151  
 sympy.core.containers (module), 150  
 sympy.core.core (module), 74  
 sympy.core.evalf (module), 149  
 sympy.core.expr (module), 74  
 sympy.core.exptools (module), 158  
 sympy.core.function (module), 134  
 sympy.core.HypergeometricMod (module), 118  
 sympy.core.mul (module), 113  
 sympy.core.multidimensional (module), 133  
 sympy.core.numbers (module), 98  
 sympy.core.power (module), 111  
 sympy.core.relational (module), 118  
 sympy.core.singleton (module), 74  
 sympy.core.symbol (module), 95  
 sympy.core.sympify (module), 59  
 sympy.crypto.crypto (module), 272  
 sympy.diffgeom (module), 1187  
 sympy.functions (module), 311  
 sympy.functions.special.bessel (module), 389  
 sympy.functions.special.beta\_functions (module), 367  
 sympy.functions.special.elliptic\_integrals (module), 410  
 sympy.functions.special.error\_functions (module), 369  
 sympy.functions.special.gamma\_functions (module), 360  
 sympy.functions.special.polynomials (module), 412  
 sympy.functions.special.zeta\_functions (module), 401  
 sympy.geometry.curve (module), 478  
 sympy.geometry.ellipse (module), 482  
 sympy.geometry.entity (module), 432  
 sympy.geometry.line (module), 448  
 sympy.geometry.line3d (module), 464  
 sympy.geometry.plane (module), 516  
 sympy.geometry.point (module), 437  
 sympy.geometry.point3d (module), 443  
 sympy.geometry.polygon (module), 495  
 sympy.geometry.util (module), 434  
 sympy.integrals (module), 522  
 sympy.integrals.meijerint\_doc (module), 540  
 sympy.integrals.transforms (module), 522  
 sympy.logic (module), 563  
 sympy.matrices (module), 573

sympy.matrices.expressions (module), 647  
sympy.matrices.expressions.blockmatrix (module),  
    650  
sympy.matrices.immutable (module), 645  
sympy.matrices.matrices (module), 573  
sympy.matrices.sparse (module), 632  
sympy.ntheory.continued\_fraction (module), 267  
sympy.ntheory.egyptian\_fraction (module), 270  
sympy.ntheory.factor\_ (module), 250  
sympy.ntheory.generate (module), 244  
sympy.ntheory.modular (module), 259  
sympy.ntheory.multinomial (module), 262  
sympy.ntheory.partitions\_ (module), 263  
sympy.ntheory.primeTest (module), 263  
sympy.ntheory.residue\_nttheory (module), 264  
sympy.plotting.plot (module), 872  
sympy.polys (module), 652  
sympy.printing.ccode (module), 854  
sympy.printing.codeprinter (module), 867  
sympy.printing.conventions (module), 867  
sympy.printing.fcode (module), 856  
sympy.printing.gtk (module), 860  
sympy.printing.lambdarepr (module), 861  
sympy.printing.latex (module), 861  
sympy.printing.mathematica (module), 860  
sympy.printing.mathml (module), 863  
sympy.printing.precedence (module), 868  
sympy.printing.pretty.pretty (module), 853  
sympy.printing.pretty.pretty\_symbology (module),  
    868  
sympy.printing.pretty.stringpict (module), 869  
sympy.printing.preview (module), 866  
sympy.printing.printer (module), 850  
sympy.printing.python (module), 864  
sympy.printing.repr (module), 864  
sympy.printing.str (module), 864  
sympy.printing.tree (module), 864  
sympy.series.gruntz (module), 900  
sympy.series.limits (module), 897  
sympy.series.order (module), 898  
sympy.series.residues (module), 900  
sympy.series.series (module), 898  
sympy.sets.fancysets (module), 914  
sympy.sets.sets (module), 903  
sympy.simplify.cse\_main (module), 896, 933  
sympy.simplify.epathtools (module), 936  
sympy.simplify.hyperexpand (module), 935  
sympy.simplify.hyperexpand\_doc (module), 946  
sympy.simplify.sqrtdenest (module), 933  
sympy.simplify.traversaltools (module), 936  
sympy.solvers (module), 1045  
sympy.solvers.inequalities (module), 1078  
sympy.solvers.ode (module), 1033  
sympy.solvers.pde (module), 1044  
sympy.solvers.recurr (module), 1056  
sympy.solvers.solveset (module), 1081  
sympy.stats (module), 948  
sympy.stats.crv (module), 981  
sympy.stats.crv\_types (module), 982  
sympy.stats.Die() (in module sympy.stats.crv\_types),  
    982  
sympy.stats.frv (module), 981  
sympy.stats.frv\_types (module), 982  
sympy.stats.Normal() (in module sympy.stats.crv\_types),  
    982  
sympy.stats.rv (module), 981  
sympy.tensor (module), 1086  
sympy.tensor.index\_methods (module), 1093  
sympy.tensor.indexed (module), 1087  
sympy.tensor.tensor (module), 1096  
sympy.utilities (module), 1110  
sympy.utilities.autowrap (module), 1111  
sympy.utilitiescodegen (module), 1115  
sympy.utilities.decorator (module), 1124  
sympy.utilities.enumerative (module), 1125  
sympy.utilities.iterables (module), 1131  
sympy.utilities.lambdify (module), 1148  
sympy.utilities.memoization (module), 1150  
sympy.utilities.misc (module), 1150  
sympy.utilities.pkgdata (module), 1152  
sympy.utilities.pytest (module), 1152  
sympy.utilities.randtest (module), 1153  
sympy.utilities.runtests (module), 1154  
sympy.utilities.source (module), 1160  
sympy.utilities.timeutils (module), 1161  
SymPyDocTestFinder (class) in  
    sympy.utilities.runtests), 1154  
SymPyDocTestRunner (class) in  
    sympy.utilities.runtests), 1154  
SymPyOutputChecker (class) in  
    sympy.utilities.runtests), 1155  
sympytestfile() (in module sympy.utilities.runtests),  
    1157  
syzygy\_module() (sympy.polys.agca.modules.SubModule  
    method), 740

## T

T (sympy.matrices.expressions.MatrixExpr attribute), 647  
T (sympy.matrices.matrices.MatrixBase attribute),  
    585  
table() (sympy.matrices.matrices.MatrixBase  
    method), 613  
tail\_degree() (sympy.polys.rings.PolyElement  
    method), 819  
tail\_degrees() (sympy.polys.rings.PolyElement  
    method), 819  
take() (in module sympy.utilities.iterables), 1146

tan (class in `sympy.functions.elementary.trigonometric`), `to_dict()` (`sympy.polys.polyclasses.ANP` method), **339**  
 tangent\_lines() (`sympy.geometry.ellipse.Ellipse` `to_dict()` (`sympy.polys.polyclasses.DMP` method),  
     method), **492**  
 tanh (class in `sympy.functions.elementary.hyperbolic`), `to_dnf()` (in module `sympy.logic.boolalg`), **570**  
     **340**  
 taylor\_term() (`sympy.core.expr.Expr` method), **94**  
     **767**  
 taylor\_term() (`sympy.functions.elementary.exponential`.`expexact()` (`sympy.polys.polytools.Poly` method), **713**  
     static method), **325**  
 taylor\_term() (`sympy.functions.elementary.exponential`.`log` `to_field()` (`sympy.polys.polyclasses.DMP` method),  
     static method), **329**  
     **767**  
 taylor\_term() (`sympy.functions.elementary.hyperbolic`.`sinhlist()` (`sympy.polys.polyclasses.ANP` method), **769**  
     static method), **337**  
 TC() (`sympy.polys.polyclasses.ANP` method), **768**  
 TC() (`sympy.polys.polyclasses.DMP` method), **762**  
 TC() (`sympy.polys.polytools.Poly` method), **680**  
 TensAdd (class in `sympy.tensor.tensor`), **1105**  
 TensExpr (class in `sympy.tensor.tensor`), **1104**  
 TensMul (class in `sympy.tensor.tensor`), **1107**  
 tensor\_indices() (in module `sympy.tensor.tensor`),  
     **1099**  
 tensor\_mul() (in module `sympy.tensor.tensor`), **1109**  
 TensorHead (class in `sympy.tensor.tensor`), **1101**  
 TensorIndex (class in `sympy.tensor.tensor`), **1098**  
 TensorIndexType (class in `sympy.tensor.tensor`),  
     **1097**  
 TensorProduct (class in `sympy.diffgeom`), **1194**  
 TensorSymmetry (class in `sympy.tensor.tensor`), **1099**  
 tensorsymmetry() (in module `sympy.tensor.tensor`),  
     **1100**  
 TensorType (class in `sympy.tensor.tensor`), **1100**  
 terminal\_width() (`sympy.printing.pretty.stringpict.stringPict`.`method`), **870**  
 terms() (`sympy.polys.polyclasses.DMP` method), **767**  
 terms() (`sympy.polys.polytools.Poly` method), **712**  
 terms() (`sympy.polys.rings.PolyElement` method),  
     **819**  
 terms\_gcd() (in module `sympy.polys.polytools`), **670**  
 terms\_gcd() (`sympy.polys.polyclasses.DMP` method),  
     **767**  
 terms\_gcd() (`sympy.polys.polytools.Poly` method),  
     **713**  
 termwise() (`sympy.polys.polytools.Poly` method), **713**  
 test() (in module `sympy.utilities.runtests`), **1158**  
 test\_derivative\_numerically() (in module  
     `sympy.utilities.randtest`), **1153**  
 threaded() (in module `sympy.utilities.decorator`),  
     **1125**  
 threaded\_factory() (in module  
     `sympy.utilities.decorator`), **1125**  
 timed() (in module `sympy.utilities.timeutils`), **1161**  
 to\_algebraic\_integer() (`sympy.polys.numberfields.AlgebraicInteger`.`method`), **720**  
 to\_cnf() (in module `sympy.logic.boolalg`), **570**  
 to\_DNF() (`sympy.polys.polytools.Poly` method), **767**  
 to\_DNF() (`sympy.polys.polyclasses.DMP` method),  
     **767**  
 to\_exact() (`sympy.polys.polyclasses.DMP` method),  
     **767**  
 to\_field() (`sympy.polys.polytools.Poly` method), **713**  
     **767**  
 to\_field() (`sympy.polys.polytools.Poly` method), **713**  
     **767**  
 to\_number\_field() (in module  
     `sympy.polys.numberfields`), **720**  
 to\_prufer() (`sympy.combinatorics.prufer.Prufer` static  
     method), **218**  
 to\_rational() (`sympy.polys.domains.RealField`  
     method), **761**  
 to\_ring() (`sympy.polys.polyclasses.DMP` method),  
     **767**  
 to\_ring() (`sympy.polys.polytools.Poly` method), **714**  
 to\_sympy() (`sympy.polys.domains.AlgebraicField`  
     method), **759**  
 to\_sympy() (`sympy.polys.domains.domain.Domain`  
     method), **754**  
 to\_sympy() (`sympy.polys.domains.ExpressionDomain`  
     method), **762**  
 to\_sympy() (`sympy.polys.domains.FiniteField`  
     method), **757**  
 to\_sympy() (`sympy.polys.domains.FractionField`  
     method), **760**  
 to\_sympy() (`sympy.polys.domains.PolynomialRing`  
     method), **758**  
 to\_sympy() (in module `sympy.polys.polyclasses.ANP` method), **761**  
 to\_sympy\_dict() (in module `sympy.polys.polyclasses.ANP` method), **769**  
 to\_sympy\_dict() (in module `sympy.polys.polyclasses.DMP` method), **767**  
 to\_sympy\_list() (in module `sympy.polys.polyclasses.ANP` method), **769**  
 to\_tree() (`sympy.combinatorics.prufer.Prufer` static  
     method), **218**  
 to\_tuple() (`sympy.polys.polyclasses.ANP` method),  
     **769**  
 to\_tuple() (`sympy.polys.polyclasses.DMP` method),  
     **767**  
 together() (in module `sympy.polys.rationaltools`), **724**  
 together() (`sympy.core.expr.Expr` method), **95**  
 TokenError (class in `sympy.parsing.sympy_tokenize`),  
     **1163**  
 takeNize() (in module  
     `sympy.parsing.sympy_tokenize`), **1162**

tolist() (sympy.matrices.dense.DenseMatrix method), 626  
tolist() (sympy.matrices.sparse.SparseMatrix method), 639  
topological\_sort() (in module sympy.utilities.iterables), 1146  
total\_degree() (sympy.polys.polyclasses.DMP method), 767  
total\_degree() (sympy.polys.polytools.Poly method), 714  
totient() (in module sympy.nttheory.factor\_), 258  
Trace (class in sympy.matrices.expressions), 649  
trailing() (in module sympy.nttheory.factor\_), 251  
transcendental, 61  
transform() (sympy.geometry.point.Point method), 442  
transform() (sympy.geometry.point3d.Point3D method), 447  
transform() (sympy.integrals.integrals.Integral method), 553  
transform\_variable (sympy.integrals.transforms.Integral attribute), 555  
transformation\_to\_DN() (in module sympy.solvers.diophantine), 1069  
transformation\_to\_normal() (in module sympy.solvers.diophantine), 1078  
transitivity\_degree (sympy.combinatorics.perm\_groups attribute), 212  
translate() (sympy.geometry.curve.Curve method), 482  
translate() (sympy.geometry.entity.GeometryEntity method), 434  
translate() (sympy.geometry.point.Point method), 442  
translate() (sympy.geometry.point3d.Point3D method), 447  
Transpose (class in sympy.matrices.expressions), 649  
transpose() (sympy.matrices.expressions.blockmatrix method), 651  
transpose() (sympy.matrices.matrices.MatrixBase method), 614  
transpositions() (sympy.combinatorics.permutations Permutation class method), 180  
tree() (in module sympy.printing.tree), 865  
tree.cse() (in module sympy.simplify.cse\_main), 935  
tree\_repr (sympy.combinatorics.prufer.Prufer attribute), 219  
Triangle (class in sympy.geometry.polygon), 509  
Triangular() (in module sympy.stats), 972  
trigamma() (in module sympy.functions.special.gamma\_functions), 365  
trigintegrate() (in module sympy.integrals.trigonometry), 549  
trigsimp() (in module sympy.simplify.simplify), 924  
trigsimp() (sympy.core.expr.Expr method), 95  
trunc() (in module sympy.polys.polytools), 672  
trunc() (sympy.polys.polyclasses.DMP method), 767  
trunc() (sympy.polys.polytools.Poly method), 714  
Tuple (class in sympy.core.containers), 150  
tuple\_count() (sympy.core.containers.Tuple method), 151  
twoform.to\_matrix() (in module sympy.diffgeom), 1199

## U

U() (in module sympy.printing.pretty.pretty\_symbology), 868  
ufuncify() (in module sympy.utilities.autowrap), 1113  
UfuncifyCodeWrapper (class in sympy.utilities.autowrap), 1112  
Unequality (class in sympy.core.relational), 127  
unflatten() (in module sympy.utilities.iterables), 1147  
UnificationFailed (class in sympy.polys.polyerrors), 839  
Uniform() (in module sympy.stats), 973  
UniformSum() (in module sympy.stats), 974  
unify() (sympy.polys.domains.domain.Domain method), 754  
unify() (sympy.polys.polyclasses.ANP method), 769  
PermutationGroup (sympy.polys.polyclasses.DMP method), 767  
unify() (sympy.polys.polytools.Poly method), 714  
Union (class in sympy.sets.sets), 910  
union() (sympy.polys.agca.ideals.Ideal method), 743  
union() (sympy.polys.agca.modules.SubModule method), 740  
union() (sympy.series.gruntz.SubsSet method), 902  
union() (sympy.sets.sets.Set method), 907  
uniq() (in module sympy.utilities.iterables), 1147  
unit (sympy.polys.polytools.Poly attribute), 715  
UnivariatePolynomialError (class in sympy.polys.polyerrors), 839  
UniversalSet (class in sympy.sets.sets), 913  
unrad() (in module sympy.solvers.solvers), 1055  
unrank() (sympy.combinatorics.graycode.GrayCode class method), 228  
unrank() (sympy.combinatorics.prufer.Prufer class method), 219  
unrank.binary() (sympy.combinatorics.subsets.Subset class method), 225  
unrank.gray() (sympy.combinatorics.subsets.Subset class method), 225  
unrank.lex() (sympy.combinatorics.permutations Permutation class method), 180  
unrank\_nonlex() (sympy.combinatorics.permutations Permutation class method), 181  
unrank\_trotterjohnson() (sympy.combinatorics.permutations Permutation

class method), 181  
`untokenize()` (in module `sympy.parsing.sympy_tokenize`), 1162  
`upper` (`sympy.tensor.indexed.Idx` attribute), 1089  
`upper_triangular_solve()`  
    (`sympy.matrices.matrices.MatrixBase`  
    method), 614  
`uppergamma` (class in `sympy.functions.special.gamma_functions`), 366  
`use()` (in module `sympy.simplify.traversaltools`), 936

**V**

`values()` (`sympy.core.containers.Dict` method), 151  
`values()` (`sympy.matrices.matrices.MatrixBase`  
    method), 614  
`var()` (in module `sympy.core.symbol`), 98  
`variables` (`sympy.concrete.expr_with_limits.ExprWithLimits`  
    attribute), 297  
`variables` (`sympy.core.function.Lambda` attribute), 134  
`variables` (`sympy.core.function.Subs` attribute), 142  
`variables` (`sympy.utilities.codegen.Routine` attribute), 1117  
`variance()` (in module `sympy.stats`), 979  
`variations()` (in module `sympy.utilities.iterables`), 1147  
`vec()` (`sympy.matrices.matrices.MatrixBase` method), 614  
`vech()` (class in `sympy.matrices.matrices.MatrixBase`  
    method), 614  
`vectorize` (class in `sympy.core.multidimensional`), 133  
`vectors_in_basis()` (in module `sympy.diffgeom`), 1199  
`verify_numerically()` (in module `sympy.utilities.randtest`), 1153  
`vertices` (`sympy.combinatorics.polyhedron.Polyhedron`  
    attribute), 215  
`vertices` (`sympy.geometry.polygon.Polygon` attribute), 501  
`vertices` (`sympy.geometry.polygon.RegularPolygon`  
    attribute), 508  
`vertices` (`sympy.geometry.polygon.Triangle` attribute), 515  
`VF()` (in module `sympy.printing.pretty.pretty_symbology`), 869  
`viete()` (in module `sympy.polys.polyfuncs`), 718  
`vobj()` (in module `sympy.printing.pretty.pretty_symbology`), 869  
`VonMises()` (in module `sympy.stats`), 974  
`vradius` (`sympy.geometry.ellipse.Circle` attribute), 495  
`vradius` (`sympy.geometry.ellipse.Ellipse` attribute), 492  
`vring()` (in module `sympy.polys.rings`), 813

`vstack()` (`sympy.matrices.matrices.MatrixBase` class  
    method), 615

**W**

`WedgeProduct` (class in `sympy.diffgeom`), 1195  
`Weibull()` (in module `sympy.stats`), 975  
`where()` (in module `sympy.stats`), 979  
`width` (`sympy.categories.diagram_drawing.DiagramGrid`  
    attribute), 1182  
`width()` (`sympy.printing.pretty.stringpict.stringPict`  
    method), 870  
`WignerSemicircle()` (in module `sympy.stats`), 976  
`Wild` (class in `sympy.core.symbol`), 95  
`WildFunction` (class in `sympy.core.function`), 135  
`with_metaclass()` (in module `sympy.core.compatibility`), 155  
`write()` (`sympy.utilitiescodegen.CodeGen` method), 1118  
`write()` (`sympy.utilities.runtests.PyTestReporter`  
    method), 1154  
`wronskian()` (in module `sympy.matrices.dense`), 619

**X**

`x` (`sympy.geometry.point.Point` attribute), 442  
`x` (`sympy.geometry.point3d.Point3D` attribute), 448  
`xdirection` (`sympy.geometry.line.Ray` attribute), 461  
`xdirection` (`sympy.geometry.line3d.Ray3D` attribute), 475  
`xobj()` (in module `sympy.printing.pretty.pretty_symbology`), 868  
`Xor` (class in `sympy.logic.boolalg`), 567  
`xreplace()` (`sympy.core.basic.Basic` method), 73  
`xring()` (in module `sympy.polys.rings`), 812  
`xstr()` (in module `sympy.printing.pretty.pretty_symbology`), 868  
`xsym()` (in module `sympy.printing.pretty.pretty_symbology`), 869  
`xthreaded()` (in module `sympy.utilities.decorator`), 1125  
`xypic_draw_diagram()` (in module `sympy.categories.diagram_drawing`), 1186  
`XypicDiagramDrawer` (class in `sympy.categories.diagram_drawing`), 1184

**Y**

`y` (`sympy.geometry.point.Point` attribute), 442  
`y` (`sympy.geometry.point3d.Point3D` attribute), 448  
`ydirection` (`sympy.geometry.line.Ray` attribute), 461  
`ydirection` (`sympy.geometry.line3d.Ray3D` attribute), 475  
`yn` (class in `sympy.functions.special.bessel`), 393  
`Ynm` (class in `sympy.functions.special.spherical_harmonics`), 421

Ynm\_c() (in module  
sympy.functions.special.spherical\_harmonics),  
423

## Z

z (sympy.geometry.point3d.Point3D attribute), 448  
zdirection (sympy.geometry.line3d.Ray3D attribute),

476

zero, 61

Zero (class in sympy.core.numbers), 104

zero (sympy.polys.polytools.Poly attribute), 715

ZeroMatrix (class in sympy.matrices.expressions),  
650

zeros() (in module sympy.matrices.dense), 616

zeros() (sympy.matrices.dense.DenseMatrix class  
method), 626

zeros() (sympy.matrices.sparse.SparseMatrix class  
method), 640

zeta (class in sympy.functions.special.zeta\_functions),  
401

zip\_row\_op() (sympy.matrices.dense.MutableDenseMatrix  
method), 630

zip\_row\_op() (sympy.matrices.sparse.MutableSparseMatrix  
method), 643

Znm (class in sympy.functions.special.spherical\_harmonics),  
423