combsimp

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

```
>>> n, k = symbols('n k', integer = True)
>>> combsimp(factorial(n)/factorial(n - 3))
n·(n - 2)·(n - 1)
>>> combsimp(binomial(n+1, k+1)/binomial(n, k))
n + 1
─────
k + 1
```

gammasimp

To simplify expressions with gamma functions or combinatorial functions with non-integer argument, use `gammasimp()`.

```
>>> gammasimp(gamma(x)*gamma(1 - x))
   π
─────────
sin(π·x)
```

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 $a_0, \ldots, a_n$ are integers, and $a_1, \ldots, a_n$ 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 $[a_0; a_1, \ldots, a_n]$. 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])
```

(continues on next page)

(continued from previous page)

```
      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`, ..., `a4`.

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

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

```
>>> frac = cancel(frac)
>>> frac
a₀·a₁·a₂·a₃·a₄ + a₀·a₁·a₂ + a₀·a₁·a₄ + a₀·a₃·a₄ + a₀ + a₂·a₃·a₄ + a₂ + a₄
──────────────────────────────────────────────────────────────────────────
                  a₁·a₂·a₃·a₄ + a₁·a₂ + a₁·a₄ + a₃·a₄ + 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
                a₂·a₃·a₄ + a₂ + a₄
a₀ + ──────────────────────────────────────────
     a₁·a₂·a₃·a₄ + a₁·a₂ + a₁·a₄ + a₃·a₄ + 1
>>> l.append(a0)
>>> frac = 1/(frac - a0)
>>> frac
a₁·a₂·a₃·a₄ + a₁·a₂ + a₁·a₄ + a₃·a₄ + 1
──────────────────────────────────────────
           a₂·a₃·a₄ + a₂ + a₄
```

Now we repeat this process

```
>>> frac = apart(frac, a1)
>>> frac
          a₃·a₄ + 1
a₁ + ──────────────────
     a₂·a₃·a₄ + a₂ + a₄
>>> l.append(a1)
>>> frac = 1/(frac - a1)
>>> frac = apart(frac, a2)
>>> frac
          a₄
a₂ + ──────────
     a₃·a₄ + 1
>>> l.append(a2)
>>> frac = 1/(frac - a2)
>>> frac = apart(frac, a3)
>>> frac
     1
a₃ + ──
     a₄
>>> l.append(a3)
>>> frac = 1/(frac - a3)
>>> frac = apart(frac, a4)
>>> frac
a₄
>>> l.append(a4)
>>> list_to_frac(l)
              1
a₀ + ──────────────────
                 1
     a₁ + ────────────
                   1
          a₂ + ──────
                    1
               a₃ + ──
                    a₄
```

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
```

In SymPy, on the above example, try to reproduce `l` from `frac`. I have deleted `l` at the end to remove the temptation for peeking (you can check your answer at the end by calling `cancel(list_to_frac(l))` on the list that you generate at the end, and comparing it to `orig_frac`.

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 $a_0 + \frac{1}{a_1+\cdots}$ when it is canceled).

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)
```

Derivatives

To take derivatives, use the `diff` function.

```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
     ⎛ 2⎞
     ⎝x ⎠
2⋅x⋅e
```

`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)
24⋅x
>>> diff(x**4, x, 3)
24⋅x
```

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·z
x ·y ·⎝x ·y ·z  + 14·x ·y ·z  + 52·x·y·z + 48⎠·e
>>> diff(expr, x, y, 2, z, 4)
 3  2 ⎛ 3  3  3       2  2  2                ⎞  x·y·z
x ·y ·⎝x ·y ·z  + 14·x ·y ·z  + 52·x·y·z + 48⎠·e
>>> diff(expr, x, y, y, z, 4)
 3  2 ⎛ 3  3  3       2  2  2                ⎞  x·y·z
x ·y ·⎝x ·y ·z  + 14·x ·y ·z  + 52·x·y·z + 48⎠·e
```

`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·z
x ·y ·⎝x ·y ·z  + 14·x ·y ·z  + 52·x·y·z + 48⎠·e
```

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
    ∂     ⎛ x·y·z⎞
──────────⎝e     ⎠
  4   2
∂z  ∂y  ∂x
```

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

```
>>> deriv.doit()
 3  2 ⎛ 3  3  3       2  2  2                ⎞  x·y·z
x ·y ·⎝x ·y ·z  + 14·x ·y ·z  + 52·x·y·z + 48⎠·e
```

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 50) section).

Derivatives of unspecified order can be created using tuple `(x, n)` where `n` is the order of the derivative with respect to `x`.

```
>>> m, n, a, b = symbols('m n a b')
>>> expr = (a*x + b)**m
>>> expr.diff((x, n))
  n
 ∂ ⎛          m⎞
──⎝(a·x + b) ⎠
  n
∂x
```

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 50)).

Quick Tip

∞ in SymPy is `oo` (that's the lowercase letter "oh" twice). This is because `oo` looks like ∞, 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))
π
```

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) - 2·x·log(x) + 2·x
```

`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
⌠
⎮ ⎛ 4    2  x    2        x          x⎞  x
⎮ ⎝x  + x ·e  - x  - 2·x·e  - 2·x - e ⎠·e
⎮ ──────────────────────────────────────── dx
⎮                2        2 ⎛ x    ⎞
⎮        (x - 1) ·(x + 1) ·⎝e  + 1⎠
⌡
>>> integ.doit()
                  x
   ⎛ x    ⎞     e
log⎝e  + 1⎠ + ──────
                 2
              x  - 1
```

```
>>> integ = Integral(sin(x**2), x)
>>> integ
⌠
⎮    ⎛ 2⎞
⎮ sin⎝x ⎠ dx
⌡
>>> integ.doit()
          ⎛√2·x⎞
3·√2·√π·S⎜────⎟·Γ(3/4)
          ⎝ √π ⎠
──────────────────────
       8·Γ(7/4)
```

```
>>> integ = Integral(x**y*exp(-x), (x, 0, oo))
>>> integ
∞
⌠
```

(continues on next page)

(continued from previous page)

```
⎮  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 $\Re(y) > 1$.

Limits

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

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

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 ∞, 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 fourth argument to `limit`. For example, to compute

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

do

```
>>> limit(1/x, x, 0, '+')
∞
```

As opposed to

```
>>> limit(1/x, x, 0, '-')
-∞
```

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 + ── + O⎝x ⎠
        2
```

The $O\left(x^4\right)$ 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 where $x \to \infty$). 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  + O⎝x ⎠
>>> 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
```

(continues on next page)

(continued from previous page)

```
     (x - 6)    (x - 6)    (x - 6)    (x - 6)          ⎛       6       ⎞
-5 + ─────── + ─────── + ─────── + ─────── + x + O⎝(x - 6) ; x → 6⎠
       2          6          24         120
```

Finite differences

So far we have looked at expressions with analytic 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.

The simplest way the differentiate using finite differences is to use the `differentiate_finite` function:

```
>>> f, g = symbols('f g', cls=Function)
>>> differentiate_finite(f(x)*g(x))
-f(x - 1/2)·g(x - 1/2) + f(x + 1/2)·g(x + 1/2)
```

If you already have a `Derivative` instance, you can use the `as_finite_difference` method to generate approximations of the derivative to arbitrary order:

```
>>> f = Function('f')
>>> dfdx = f(x).diff(x)
>>> dfdx.as_finite_difference()
-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')
>>> d2fdx2.as_finite_difference([-3*h,-h,2*h])
f(-3·h)   f(-h)   2·f(2·h)
─────── - ──── + ────────
    2       2        2
 5·h     3·h     15·h
```

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 the function 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_difference` method of `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)
  3·a   b   2·c
- ─── - ─ + ───
   20   4    5
```

Solvers

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

A Note about Equations

Recall from the *gotchas* (page 12) 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 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

```
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
>>> solveset(Eq(x**2 - 1, 0), x)
{-1, 1}
>>> solveset(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 `solveset(Eq(expr, 0), x)`, you can just use `solveset(expr, x)`.

Solving Equations Algebraically

The main function for solving algebraic equations is `solveset`. The syntax for `solveset` is `solveset(equation, variable=None, domain=S.Complexes)` Where `equations` may be in the form of Eq instances or expressions that are assumed to be equal to zero.

Please note that there is another function called `solve` which can also be used to solve equations. The syntax is `solve(equations, variables)` However, it is recommended to use `solveset` instead.

When solving a single equation, the output of `solveset` is a `FiniteSet` or an `Interval` or `ImageSet` of the solutions.

```
>>> solveset(x**2 - x, x)
{0, 1}
>>> solveset(x - x, x, domain=S.Reals)
ℝ
>>> solveset(sin(x) - 1, x, domain=S.Reals)
⎧        π │      ⎫
⎨2⋅n⋅π + ─ │ n ∈ ℤ⎬
⎩        2 │      ⎭
```

If there are no solutions, an `EmptySet` is returned and if it is not able to find solutions then a `ConditionSet` is returned.

```
>>> solveset(exp(x), x)     # No solution exists
∅
>>> solveset(cos(x) - x, x)  # Not able to find solution
{x │ x ∈ ℂ ∧ (-x + cos(x) = 0)}
```

In the `solveset` module, the linear system of equations is solved using `linsolve`. In future we would be able to use linsolve directly from `solveset`. Following is an example of the syntax of `linsolve`.

- List of Equations Form:

```
>>> linsolve([x + y + z - 1, x + y + 2*z - 3 ], (x, y, z))
{(-y - 1, y, 2)}
```

- Augmented Matrix Form:

```
>>> linsolve(Matrix(([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
{(-y - 1, y, 2)}
```

- A*x = b Form

```
>>> M = Matrix(((1, 1, 1, 1), (1, 1, 2, 3)))
>>> system = A, b = M[:, :-1], M[:, -1]
>>> linsolve(system, x, y, z)
{(-y - 1, y, 2)}
```

Note: The order of solution corresponds the order of given symbols.

In the `solveset` module, the non linear system of equations is solved using `nonlinsolve`. Following are examples of `nonlinsolve`.

1. When only real solution is present:

```
>>> a, b, c, d = symbols('a, b, c, d', real=True)
>>> nonlinsolve([a**2 + a, a - b], [a, b])
{(-1, -1), (0, 0)}
>>> nonlinsolve([x*y - 1, x - 2], x, y)
{(2, 1/2)}
```

2. When only complex solution is present:

```
>>> nonlinsolve([x**2 + 1, y**2 + 1], [x, y])
{(-i, -i), (-i, i), (i, -i), (i, i)}
```

3. When both real and complex solution are present:

```
>>> from sympy import sqrt
>>> system = [x**2 - 2*y**2 -2, x*y - 2]
>>> vars = [x, y]
>>> nonlinsolve(system, vars)
{(-2, -1), (2, 1), (-√2·i, √2·i), (√2·i, -√2·i)}
```

```
>>> system = [exp(x) - sin(y), 1/y - 3]
>>> nonlinsolve(system, vars)
{({2·n·i·π + log(sin(1/3)) | n ∈ ℤ}, 1/3)}
```

4. When the system is positive-dimensional system (has infinitely many solutions):

```
>>> nonlinsolve([x*y, x*y - x], [x, y])
{(0, y)}
```

```
>>> system = [a**2 + a*c, a - b]
>>> nonlinsolve(system, [a, b])
{(0, 0), (-c, -c)}
```

Note:

1. The order of solution corresponds the order of given symbols.

2. Currently `nonlinsolve` doesn't return solution in form of `LambertW` (if there is solution present in the form of `LambertW`).

`solve` can be used for such cases:

```
>>> solve([x**2 - y**2/exp(x)], [x, y], dict=True)
⎡⎧          ____⎫  ⎧          ____⎫⎤
⎢⎨         ╱  x ⎬  ⎨         ╱  x ⎬⎥
⎣⎩y: -x⋅╲╱  e  ⎭, ⎩y: x⋅╲╱  e  ⎭⎦
>>> solve(x**2 - y**2/exp(x), x, dict=True)
⎡⎧      ⎛-y ⎞⎫  ⎧      ⎛y⎞⎫⎤
⎢⎨x: 2⋅W⎜───⎟⎬, ⎨x: 2⋅W⎜─⎟⎬⎥
⎣⎩      ⎝ 2 ⎠⎭  ⎩      ⎝2⎠⎭⎦
```

3. Currently `nonlinsolve` is not properly capable of solving the system of equations having trigonometric functions.

`solve` can be used for such cases (but does not give all solution):

```
>>> solve([sin(x + y), cos(x - y)], [x, y])
⎡⎛-3⋅π   3⋅π⎞  ⎛-π   π⎞  ⎛π  3⋅π⎞  ⎛3⋅π  π⎞⎤
⎢⎜─────, ───⎟, ⎜───, ─⎟, ⎜─, ───⎟, ⎜───, ─⎟⎥
⎣⎝  4     4 ⎠  ⎝ 4   4⎠  ⎝4   4 ⎠  ⎝ 4   4⎠⎦
```

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

```
>>> solveset(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.

Note: Currently `solveset` is not capable of solving the following types of equations:

- Equations solvable by LambertW (Transcendental equation solver).

`solve` can be used for such cases:

```
>>> solve(x*exp(x) - 1, x )
[W(1)]
```

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 40) 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, x) - 2*f(x).diff(x) + f(x), sin(x))
>>> diffeq
                    2
          d        d
f(x) - 2⋅──(f(x)) + ──(f(x)) = sin(x)
          dx         2
                   dx
```

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

```
>>> dsolve(diffeq, f(x))
                       x   cos(x)
f(x) = (C₁ + C₂·x)·e  + ──────
                          2
```

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))) - 1, f(x))
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.

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`.

Basic Operations

Here are some basic operations on `Matrix`.

Shape

To get the shape of a matrix, use *shape()* (page 1155) function.

```
>>> from sympy import shape
>>> M = Matrix([[1, 2, 3], [-2, 0, 4]])
>>> M
⎡1   2  3⎤
⎢        ⎥
⎣-2  0  4⎦
>>> shape(M)
(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`.

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):
...
NonInvertibleMatrixError: 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⎦
```

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 1×1 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⎦
```

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 tuple 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 ⎤
```

(continues on next page)

(continued from previous page)

```
⎢                ⎥
⎢2   3   4   7 ⎥
⎢                ⎥
⎣-1  -3  -3  -4⎦
>>> M.rref()
⎛⎡1  0   1    3 ⎤        ⎞
⎜⎢                ⎥        ⎟
⎜⎢0  1  2/3  1/3⎥, (0, 1)⎟
⎜⎢                ⎥        ⎟
⎝⎣0  0   0    0 ⎦        ⎠
```

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

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  ⎦⎦
```

Columnspace

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

```
>>> M = Matrix([[1, 1, 2], [2 ,1 , 3], [3 , 1, 4]])
>>> M
⎡1  1  2⎤
⎢       ⎥
⎢2  1  3⎥
```

(continues on next page)

(continued from previous page)

```
⎢       ⎥
⎣3  1  4⎦
>>> M.columnspace()
⎡⎡1⎤  ⎡1⎤⎤
⎢⎢ ⎥  ⎢ ⎥⎥
⎢⎢2⎥, ⎢1⎥⎥
⎢⎢ ⎥  ⎢ ⎥⎥
⎣⎣3⎦  ⎣1⎦⎦
```

Eigenvalues, Eigenvectors, and Diagonalization

To find the eigenvalues of a matrix, use `eigenvals`. `eigenvals` returns a dictionary of `eigenvalue: algebraic_multiplicity` pairs (similar to the output of *roots* (page 49)).

```
>>> 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 `eigenvects`. `eigenvects` returns a list of tuples of the form `(eigenvalue, algebraic_multiplicity, [eigenvectors])`.

```
>>> M.eigenvects()
⎡⎛       ⎡⎡0⎤⎤  ⎛      ⎡⎡1⎤⎤  ⎛      ⎡⎡1⎤  ⎡0 ⎤⎤⎞⎤
⎢⎜       ⎢⎢ ⎥⎥  ⎜      ⎢⎢ ⎥⎥  ⎜      ⎢⎢ ⎥  ⎢  ⎥⎥⎟⎥
⎢⎜       ⎢⎢1⎥⎥  ⎜      ⎢⎢1⎥⎥  ⎜      ⎢⎢1⎥  ⎢-1⎥⎥⎟⎥
⎢⎜-2, 1, ⎢⎢ ⎥⎥, ⎜3, 1, ⎢⎢ ⎥⎥, ⎜5, 2, ⎢⎢ ⎥, ⎢  ⎥⎥⎟⎥
⎢⎜       ⎢⎢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*P**-1 == M
True
```


Quick Tip

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

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.as_expr())
       2
(λ - 5) ·(λ - 3)·(λ + 2)
```

Possible Issues

Zero Testing

If your matrix operations are failing or returning wrong answers, the common reasons would likely be from zero testing. If there is an expression not properly zero-tested, it can possibly bring issues in finding pivots for gaussian elimination, or deciding whether the matrix is inversible, or any high level functions which relies on the prior procedures.

Currently, the SymPy's default method of zero testing `_iszero` is only guaranteed to be accurate in some limited domain of numerics and symbols, and any complicated expressions beyond its decidability are treated as `None`, which behaves similarly to logical `False`.

The list of methods using zero testing procedures are as follows:

`echelon_form` , `is_echelon` , `rank` , `rref` , `nullspace` , `eigenvects` , `inverse_ADJ` , `inverse_GE` , `inverse_LU` , `LUdecomposition` , `LUdecomposition_Simple` , `LUsolve`

They have property `iszerofunc` opened up for user to specify zero testing method, which can accept any function with single input and boolean output, while being defaulted with `_iszero`.

Here is an example of solving an issue caused by undertested zero. While the output for this particular matrix has since been improved, the technique below is still of interest.[1][2][3]

```
>>> from sympy import *
>>> q = Symbol("q", positive = True)
>>> m = Matrix([
... [-2*cosh(q/3),      exp(-q),           1],
... [      exp(q), -2*cosh(q/3),           1],
... [           1,            1, -2*cosh(q/3)]])
>>> m.nullspace()
[]
```

You can trace down which expression is being underevaluated, by injecting a custom zero test with warnings enabled.

```
>>> import warnings
>>>
>>> def my_iszero(x):
...     try:
...         result = x.is_zero
...     except AttributeError:
...         result = None
...
...     # Warnings if evaluated into None
...     if result is None:
...         warnings.warn("Zero testing of {} evaluated into None".format(x))
...     return result
...
>>> m.nullspace(iszerofunc=my_iszero)
__main__:9: UserWarning: Zero testing of 4*cosh(q/3)**2 - 1 evaluated into␣
↪None
```

(continues on next page)

[1] Inspired by https://gitter.im/sympy/sympy?at=5b7c3e8ee5b40332abdb206c
[2] Discovered from https://github.com/sympy/sympy/issues/15141
[3] Improved by https://github.com/sympy/sympy/pull/19548

(continued from previous page)

```
__main__:9: UserWarning: Zero testing of (-exp(q) - 2*cosh(q/3))*(-2*cosh(q/
↪3) - exp(-q)) - (4*cosh(q/3)**2 - 1)**2 evaluated into None
__main__:9: UserWarning: Zero testing of 2*exp(q)*cosh(q/3) - 16*cosh(q/3)**4␣
↪+ 12*cosh(q/3)**2 + 2*exp(-q)*cosh(q/3) evaluated into None
__main__:9: UserWarning: Zero testing of -(4*cosh(q/3)**2 - 1)*exp(-q) -␣
↪2*cosh(q/3) - exp(-q) evaluated into None
[]
```

In this case, `(-exp(q) - 2*cosh(q/3))*(-2*cosh(q/3) - exp(-q)) - (4*cosh(q/3)**2 - 1)**2` should yield zero, but the zero testing had failed to catch. possibly meaning that a stronger zero test should be introduced. For this specific example, rewriting to exponentials and applying simplify would make zero test stronger for hyperbolics, while being harmless to other polynomials or transcendental functions.

```
>>> def my_iszero(x):
...     try:
...         result = x.rewrite(exp).simplify().is_zero
...     except AttributeError:
...         result = None
...
...     # Warnings if evaluated into None
...     if result is None:
...         warnings.warn("Zero testing of {} evaluated into None".format(x))
...     return result
...
>>> m.nullspace(iszerofunc=my_iszero)
__main__:9: UserWarning: Zero testing of -2*cosh(q/3) - exp(-q) evaluated␣
↪into None
⎡⎡  ⎛   q          ⎛q⎞⎞  -q          2⎛q⎞     ⎤⎤
⎢⎢- ⎜- e  - 2⋅cosh⎜─⎟⎟⋅e   + 4⋅cosh ⎜─⎟ - 1⎥⎥
⎢⎢  ⎝              ⎝3⎠⎠              ⎝3⎠     ⎥⎥
⎢⎢──────────────────────────────────────────⎥⎥
⎢⎢          ⎛      2⎛q⎞    ⎞     ⎛q⎞       ⎥⎥
⎢⎢        2⋅⎜4⋅cosh ⎜─⎟ - 1⎟⋅cosh⎜─⎟       ⎥⎥
⎢⎢          ⎝       ⎝3⎠    ⎠     ⎝3⎠       ⎥⎥
⎢⎢                                          ⎥⎥
⎢⎢           ⎛   q          ⎛q⎞⎞            ⎥⎥
⎢⎢          -⎜- e  - 2⋅cosh⎜─⎟⎟            ⎥⎥
⎢⎢           ⎝              ⎝3⎠⎠            ⎥⎥
⎢⎢          ──────────────────────          ⎥⎥
⎢⎢                    2⎛q⎞                  ⎥⎥
⎢⎢             4⋅cosh ⎜─⎟ - 1               ⎥⎥
⎢⎢                    ⎝3⎠                   ⎥⎥
⎢⎢                                          ⎥⎥
⎣⎣                    1                     ⎦⎦
```

You can clearly see `nullspace` returning proper result, after injecting an alternative zero test.

Note that this approach is only valid for some limited cases of matrices containing only numerics, hyperbolics, and exponentials. For other matrices, you should use different method opted for their domains.

Possible suggestions would be either taking advantage of rewriting and simplifying, with tradeoff of speed[4] , or using random numeric testing, with tradeoff of accuracy[5] .

If you wonder why there is no generic algorithm for zero testing that can work with any symbolic entities, it's because of the constant problem stating that zero testing is undecidable[6] , and not only the SymPy, but also other computer algebra systems[7][8] would face the same fundamental issue.

However, discovery of any zero test failings can provide some good examples to improve SymPy, so if you have encountered one, you can report the issue to SymPy issue tracker[9] to get detailed help from the community.

Advanced Expression Manipulation

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

Understanding Expression Trees

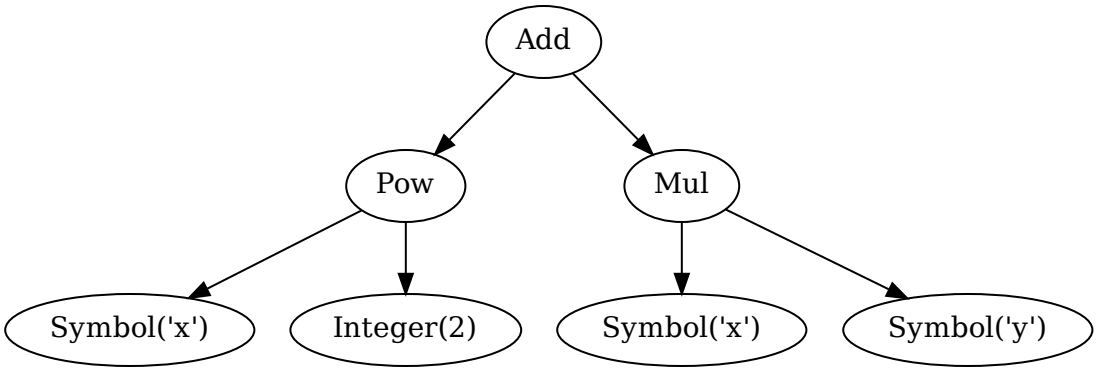
Before we can do this, we need to understand how expressions are represented in SymPy. A mathematical expression is represented as a tree. Let us take the expression $x^2 + xy$, i.e., `x**2 + x*y`. We can see what this expression looks like internally by using `srepr`

```
>>> 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:

[4] Suggested from https://github.com/sympy/sympy/issues/10120
[5] Suggested from https://github.com/sympy/sympy/issues/10279
[6] https://en.wikipedia.org/wiki/Constant_problem
[7] How mathematica tests zero https://reference.wolfram.com/language/ref/PossibleZeroQ.html
[8] How matlab tests zero https://www.mathworks.com/help/symbolic/mupad_ref/iszero.html
[9] https://github.com/sympy/sympy/issues



Note: The above diagram was made using Graphviz and the *dotprint* (page 2188) 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"[1]. 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`[2].

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

(continues on next page)

[1] We have been using `symbols` instead of `Symbol` because it automatically splits apart strings into multiple `Symbols`. `symbols('x y z')` returns a tuple of three `Symbols`. `Symbol('x y z')` returns a single `Symbol` called `x y z`.

[2] Technically, it is an internal function called `_sympify`, which differs from `sympify` in that it does not convert strings. `x + '2'` is not allowed.

(continued from previous page)

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

We have seen that `x**2` is represented as `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 `Mul`.

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

Thus, we could have created the same object by writing `Mul(x, y)`.

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

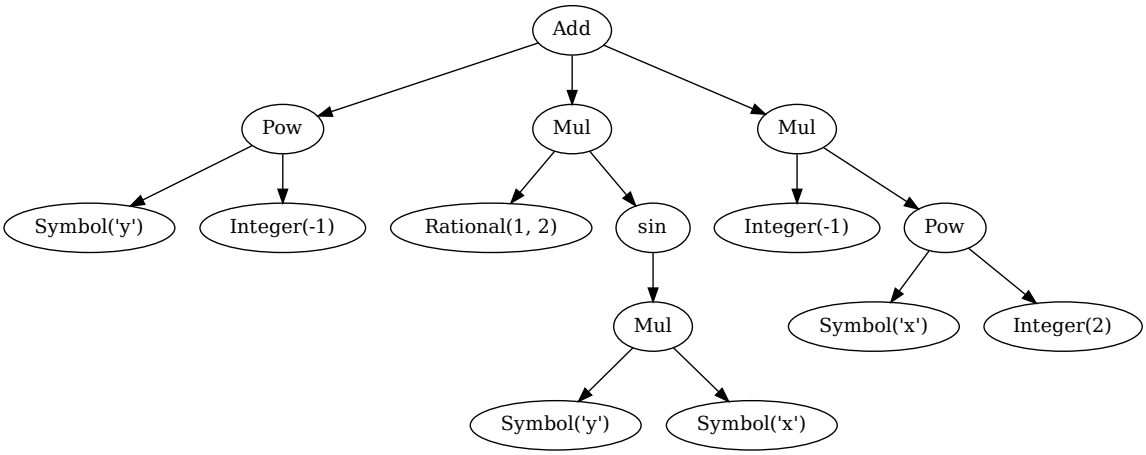
Now we get to our final expression, `x**2 + x*y`. This is the addition of our last two objects, `Pow(x, 2)`, and `Mul(x, y)`. The SymPy class for addition is `Add`, so, as you might expect, to create this object, we use `Add(Pow(x, 2), Mul(x, y))`.

```
>>> 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

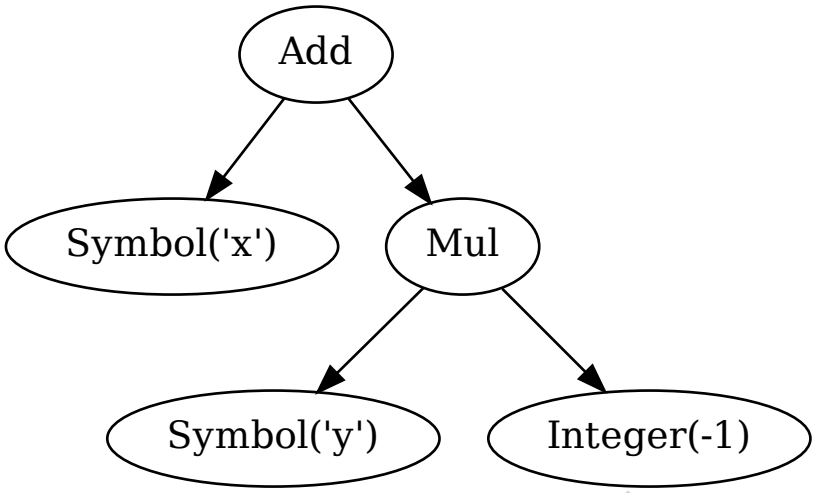


This expression reveals some interesting things about SymPy expression trees. Let's go through them one by one.

Let's first look at the term `x**2`. As we expected, we see `Pow(x, 2)`. One level up, we see we have `Mul(-1, Pow(x, 2))`. There is no subtraction class in SymPy. `x - y` is represented as

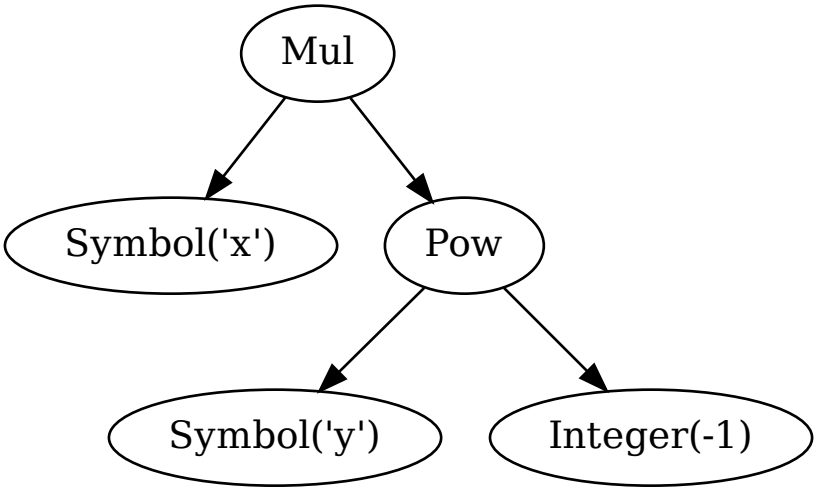
`x + -y`, or, more completely, `x + -1*y`, i.e., `Add(x, Mul(-1, y))`.

```
>>> 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.

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[3].

```
>>> 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.

[3] 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.

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`.

```
>>> 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
```

Prevent expression evaluation

There are generally two ways to prevent the evaluation, either pass an `evaluate=False` parameter while constructing the expression, or create an evaluation stopper by wrapping the expression with `UnevaluatedExpr`.

For example:

```
>>> from sympy import Add
>>> from sympy.abc import x, y, z
>>> x + x
2*x
>>> Add(x, x)
2*x
>>> Add(x, x, evaluate=False)
x + x
```

If you don't remember the class corresponding to the expression you want to build (operator overloading usually assumes `evaluate=True`), just use `sympify` and pass a string:

```
>>> from sympy import sympify
>>> sympify("x + x", evaluate=False)
x + x
```

Note that `evaluate=False` won't prevent future evaluation in later usages of the expression:

```
>>> expr = Add(x, x, evaluate=False)
>>> expr
x + x
>>> expr + x
3*x
```

That's why the class `UnevaluatedExpr` comes handy. `UnevaluatedExpr` is a method provided by SymPy which lets the user keep an expression unevaluated. By *unevaluated* it is meant that the value inside of it will not interact with the expressions outside of it to give simplified outputs. For example:

```
>>> from sympy import UnevaluatedExpr
>>> expr = x + UnevaluatedExpr(x)
>>> expr
x + x
>>> x + expr
2*x + x
```

The x remaining alone is the x wrapped by `UnevaluatedExpr`. To release it:

```
>>> (x + expr).doit()
3*x
```

Other examples:

```
>>> from sympy import *
>>> from sympy.abc import x, y, z
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
```

(continues on next page)

(continued from previous page)

```
>>> uexpr
(5/7)*(3/4)
>>> x*UnevaluatedExpr(1/x)
x*1/x
```

A point to be noted is that `UnevaluatedExpr` cannot prevent the evaluation of an expression which is given as argument. For example:

```
>>> expr1 = UnevaluatedExpr(x + x)
>>> expr1
2*x
>>> expr2 = sympify('x + x', evaluate=False)
>>> expr2
x + x
```

Remember that `expr2` will be evaluated if included into another expression. Combine both of the methods to prevent both inside and outside evaluations:

```
>>> UnevaluatedExpr(sympify("x + x", evaluate=False)) + y
y + (x + x)
```

`UnevalutedExpr` is supported by SymPy printers and can be used to print the result in different output forms. For example

```
>>> from sympy import latex
>>> uexpr = UnevaluatedExpr(S.One*5/7)*UnevaluatedExpr(S.One*3/4)
>>> print(latex(uexpr))
\frac{5}{7} \cdot \frac{3}{4}
```

In order to release the expression and get the evaluated LaTeX form, just use `.doit()`:

```
>>> print(latex(uexpr.doit()))
\frac{15}{28}
```

What's Next

Congratulations on finishing the SymPy tutorial!

If you are a developer interested in using SymPy in your code, please visit the *How-to Guides* (page 71) which discuss key developer tasks.

Intermediate SymPy users and developers might want to visit the *Explanations* (page 141) section for common pitfalls and advanced topics.

The *SymPy API Reference* (page 185) has a detailed description of the SymPy API.

If you are interested in contributing to SymPy, visit the *contribution guides* (page 2983) and the full Development Workflow guide on the SymPy wiki.

CHAPTER

THREE

HOW-TO GUIDES

How-to guides are step-by-step instructions on how to do specific tasks.

For a deeper and elaborate exploration of other SymPy topics, see the *Explanations* (page 141) and *API reference* (page 185) sections.

3.1 Assumptions

This page outlines the core assumptions system in SymPy. It explains what the core assumptions system is, how the assumptions system is used and what the different assumptions predicates mean.

Note: This page describes the core assumptions system also often referred to as the "old assumptions" system. There is also a "new assumptions" system which is described elsewhere. Note that the system described here is actually the system that is widely used in SymPy. The "new assumptions" system is not really used anywhere in SymPy yet and the "old assumptions" system will not be removed. At the time of writing (SymPy 1.7) it is still recommended for users to use the old assumption system.

Firstly we consider what happens when taking the square root of the square of a concrete integer such as 2 or -2:

```
>>> from sympy import sqrt
>>> sqrt(2**2)
2
>>> sqrt((-2)**2)
2
>>> x = 2
>>> sqrt(x**2)
2
>>> sqrt(x**2) == x
True
>>> y = -2
>>> sqrt(y**2) == y
False
>>> sqrt(y**2) == -y
True
```

What these examples demonstrate is that for a positive number x we have $\sqrt{x^2} = x$ whereas for a negative number we would instead have $\sqrt{x^2} = -x$. That may seem obvious but the

situation can be more surprising when working with a symbol rather then an explicit number. For example

```
>>> from sympy import Symbol, simplify
>>> x = Symbol('x')
>>> sqrt(x**2)
sqrt(x**2)
```

It might look as if that should simplify to x but it does not even if *simplify()* (page 661) is used:

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

This is because SymPy will refuse to simplify this expression if the simplification is not valid for *every* possible value of x. By default the symbol x is considered only to represent something roughly like an arbitrary complex number and the obvious simplification here is only valid for positive real numbers. Since x is not known to be positive or even real no simplification of this expression is possible.

We can tell SymPy that a symbol represents a positive real number when creating the symbol and then the simplification will happen automatically:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

This is what is meant by "assumptions" in SymPy. If the symbol y is created with positive=True then SymPy will *assume* that it represents a positive real number rather than an arbitrary complex or possibly infinite number. That *assumption* can make it possible to simplify expressions or might allow other manipulations to work. It is usually a good idea to be as precise as possible about the assumptions on a symbol when creating it.

3.1.1 The (old) assumptions system

There are two sides to the assumptions system. The first side is that we can declare assumptions on a symbol when creating the symbol. The other side is that we can query the assumptions on any expression using the corresponding is_* attribute. For example:

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

We can query assumptions on any expression not just a symbol:

```
>>> x = Symbol('x', positive=True)
>>> expr = 1 + x**2
>>> expr
x**2 + 1
>>> expr.is_positive
True
>>> expr.is_negative
False
```

The values given in an assumptions query use three-valued "fuzzy" logic. Any query can return `True`, `False`, or `None` where `None` should be interpreted as meaning that the result is *unknown*.

```
>>> x = Symbol('x')
>>> y = Symbol('y', positive=True)
>>> z = Symbol('z', negative=True)
>>> print(x.is_positive)
None
>>> print(y.is_positive)
True
>>> print(z.is_positive)
False
```

Note: We need to use `print` in the above examples because the special value `None` does not display by default in the Python interpretter.

There are several reasons why an assumptions query might give `None`. It is possible that the query is *unknowable* as in the case of `x` above. Since `x` does not have any assumptions declared it roughly represents an arbitrary complex number. An arbitrary complex number *might* be a positive real number but it also might *not* be. Without further information there is no way to resolve the query `x.is_positive`.

Another reason why an assumptions query might give `None` is that there does in many cases the problem of determining whether an expression is e.g. positive is *undecidable*. That means that there does not exist an algorithm for answering the query in general. For some cases an algorithm or at least a simple check would be possible but has not yet been implemented although it could be added to SymPy.

The final reason that an assumptions query might give `None` is just that the assumptions system does not try very hard to answer complicated queries. The system is intended to be fast and uses simple heuristic methods to conclude a `True` or `False` answer in common cases. For example any sum of positive terms is positive so:

```
>>> from sympy import symbols
>>> x, y = symbols('x, y', positive=True)
>>> expr = x + y
>>> expr
x + y
>>> expr.is_positive
True
```

The last example is particularly simple so the assumptions system is able to give a definite answer. If the sum involved a mix of positive or negative terms it would be a harder query:

```
>>> x = Symbol('x', real=True)
>>> expr = 1 + (x - 2)**2
>>> expr
(x - 2)**2 + 1
>>> expr.is_positive
True
>>> expr2 = expr.expand()
>>> expr2
```

(continues on next page)

(continued from previous page)

```
x**2 - 4*x + 5
>>> print(expr2.is_positive)
None
```

Ideally that last example would give `True` rather than `None` because the expression is always positive for any real value of `x` (and `x` has been assumed real). The assumptions system is intended to be efficient though: it is expected many more complex queries will not be fully resolved. This is because assumptions queries are primarily used internally by SymPy as part of low-level calculations. Making the system more comprehensive would slow SymPy down.

Note that in fuzzy logic giving an indeterminate result `None` is never a contradiction. If it is possible to infer a definite `True` or `False` result when resolving a query then that is better than returning `None`. However a result of `None` is not a *bug*. Any code that uses the assumptions system needs to be prepared to handle all three cases for any query and should not presume that a definite answer will always be given.

The assumptions system is not just for symbols or for complex expressions. It can also be used for plain SymPy integers and other objects. The assumptions predicates are available on any instance of *Basic* (page 927) which is the superclass for most classes of SymPy objects. A plain Python `int` is not a *Basic* (page 927) instance and can not be used to query assumptions predicates. We can "sympify" regular Python objects to become SymPy objects with *sympify()* (page 918) or `S` (*SingletonRegistry* (page 945)) and then the assumptions system can be used:

```
>>> from sympy import S
>>> x = 2
>>> x.is_positive
Traceback (most recent call last):
...
AttributeError: 'int' object has no attribute 'is_positive'
>>> x = S(2)
>>> type(x)
<class 'sympy.core.numbers.Integer'>
>>> x.is_positive
True
```

3.1.2 Gotcha: symbols with different assumptions

In SymPy it is possible to declare two symbols with different names and they will implicitly be considered equal under *structural equality*:

```
>>> x1 = Symbol('x')
>>> x2 = Symbol('x')
>>> x1
x
>>> x2
x
>>> x1 == x2
True
```

However if the symbols have different assumptions then they will be considered to represent distinct symbols:

```
>>> x1 = Symbol('x', positive=True)
>>> x2 = Symbol('x')
>>> x1
x
>>> x2
x
>>> x1 == x2
False
```

One way to simplify an expression is to use the *posify()* (page 668) function which will replace all symbols in an expression with symbols that have the assumption `positive=True` (unless that contradicts any existing assumptions for the symbol):

```
>>> from sympy import posify, exp
>>> x = Symbol('x')
>>> expr = exp(sqrt(x**2))
>>> expr
exp(sqrt(x**2))
>>> posify(expr)
(exp(_x), {_x: x})
>>> expr2, rep = posify(expr)
>>> expr2
exp(_x)
```

The *posify()* (page 668) function returns the expression with all symbols replaced (which might lead to simplifications) and also a dict which maps the new symbols to the old that can be used with *subs()* (page 941). This is useful because otherwise the new expression with the new symbols having the `positive=True` assumption will not compare equal to the old:

```
>>> expr2
exp(_x)
>>> expr2 == exp(x)
False
>>> expr2.subs(rep)
exp(x)
>>> expr2.subs(rep) == exp(x)
True
```

3.1.3 Applying assumptions to string inputs

We have seen how to set assumptions when *Symbol* (page 976) or *symbols()* (page 978) explicitly. A natural question to ask is in what other situations can we assign assumptions to an object?

It is common for users to use strings as input to SymPy functions (although the general feeling among SymPy developers is that this should be discouraged) e.g.:

```
>>> from sympy import solve
>>> solve('x**2 - 1')
[-1, 1]
```

When creating symbols explicitly it would be possible to assign assumptions that would affect the behaviour of *solve()* (page 836):

```
>>> x = Symbol('x', positive=True)
>>> solve(x**2 - 1)
[1]
```

When using string input SymPy will create the expression and create all of the symbolc implicitly so the question arises how can the assumptions be specified? The answer is that rather than depending on implicit string conversion it is better to use the *parse_expr()* (page 2122) function explicitly and then it is possible to provide assumptions for the symbols e.g.:

```
>>> from sympy import parse_expr
>>> parse_expr('x**2 - 1')
x**2 - 1
>>> eq = parse_expr('x**2 - 1', {'x':Symbol('x', positive=True)})
>>> solve(eq)
[1]
```

Note: The *solve()* (page 836) function is unusual as a high level API in that it actually checks the assumptions on any input symbols (the unknowns) and uses that to tailor its output. The assumptions system otherwise affects low-level evaluation but is not necessarily handled explicitly by high-level APIs.

3.1.4 Predicates

There are many different predicates that can be assumed for a symbol or can be queried for an expression. It is possible to combine multiple predicates when creating a symbol. Predicates are logically combined using *and* so if a symbol is declared with `positive=True` and also with `integer=True` then it is both positive *and* integer:

```
>>> x = Symbol('x', positive=True, integer=True)
>>> x.is_positive
True
>>> x.is_integer
True
```

The full set of known predicates for a symbol can be accessed using the *assumptions0* (page 929) attribute:

```
>>> x.assumptions0
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
```

(continues on next page)