4.2 SymPy Special Topics

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.

4.2.1 Finite Difference Approximations to Derivatives

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

$$F(x_0 + h) = F(x_0) + \left(\frac{dF}{dx}\right)_{x_0} * h + \frac{1}{2!}\left(\frac{d^2F}{dx^2}\right)_{x_0} * h^2 + \frac{1}{3!}\left(\frac{d^3F}{dx^3}\right)_{x_0} * h^3 + \ldots$$

In order to simplify the notation, we now define a set of coefficients c_n, where

$$c_n := \frac{1}{n!}\left(\frac{d^nF}{dx^n}\right)_{x_0}.$$

So now our series has the form:

$$F(x_0 + h) = F(x_0) + c_1 * h + c_2 * h^2 + c_3 * h^3 + \ldots$$

In the following we will only use a finite grid of values x_i with i running from $1, \ldots, N$ 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.

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:

$$F_{i+1} = F_i + c_1 * h + \frac{1}{2} * c_2 * h^2 + \frac{1}{3!} * c_3 * h^3 + ...$$

$$F_{i+2} = F_i + c_1 * (2h) + \frac{1}{2} * c_2 * (2h)^2 + \frac{1}{3!} * c_3 * (2h)^3 + ...$$

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_i, 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 $P(x, x0, c, n)$).

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=Fi,  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 dth 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])
```

(continues on next page)

(continued from previous page)

```
>>> 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 3×3 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 3×3 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 discritizations of derivatives, but using a much more efficient approach originally due to Bengt Fornberg and now incorporated into SymPy.

Finite differences (page 46)

Finite difference weights (page 240)

4.2.2 Classification of SymPy objects

There are several ways of how SymPy object is classified.

class

Like any other object in Python, SymPy expression is an instance of class. You can get the class of the object with built-in $type()$ function, and check it with $isinstance()$ function.

```
>>> from sympy import Add
>>> from sympy.abc import x,y
>>> type(x + y)
<class 'sympy.core.add.Add'>
>>> isinstance(x + y, Add)
True
```

Classes represent only the programmatic structures of the objects, and does not distinguish the mathematical difference between them. For example, the integral of number and the integral of matrix both have the class $Integral$, although the former is number and the latter is matrix.

```
>>> from sympy import MatrixSymbol, Integral
>>> A = MatrixSymbol('A', 2, 2)
>>> type(Integral(1, x))
<class 'sympy.integrals.integrals.Integral'>
>>> type(Integral(A, x))
<class 'sympy.integrals.integrals.Integral'>
```

kind

Kind indicates what mathematical object does the expression represent. You can retrieve the kind of expression with $.kind$ property.

```
>>> Integral(1, x).kind
NumberKind
>>> Integral(A, x).kind
MatrixKind(NumberKind)
```

This result shows that $Integral(1, x)$ is number, and $Integral(A, x)$ is matrix with number element.

Since the class cannot guarantee to catch this difference, kind of the object is very important. For example, if you are building a function or class that is designed to work only for numbers, you should consider filtering the arguments with $NumberKind$ so that the user does not naively pass unsupported objects such as $Integral(A, x)$.

For the performance, set theory is not implemented in kind system. For example,

$NumberKind$ does not distinguish the real number and complex number.

```
>>> from sympy import pi, I
>>> pi.kind
NumberKind
>>> I.kind
NumberKind
```

SymPy's Set and kind are not compatible.

```
>>> from sympy import S
>>> from sympy.core.kind import NumberKind
>>> S.Reals.is_subset(S.Complexes)
True
>>> S.Reals.is_subset(NumberKind)
Traceback (most recent call last):
...
ValueError: Unknown argument 'NumberKind'
```

sets and assumptions

If you want to classify the object in strictly mathematical way, you may need SymPy's sets and assumptions.

```
>>> from sympy import ask, Q
>>> S.One in S.Reals
True
>>> ask(Q.even(2*x), Q.odd(x))
True
```

See *assumptions* module and *sets* module for more information.

func

func is the head of the object, and it is used to recurse over the expression tree.

```
>>> Add(x + y).func
<class 'sympy.core.add.Add'>
>>> Add(x + x).func
<class 'sympy.core.mul.Mul'>
>>> Q.even(x).func
<class 'sympy.assumptions.assume.AppliedPredicate'>
```

As you can see, resulting head may be a class or another SymPy object. Keep this in mind when you classify the object with this attribute. See *Advanced Expression Manipulation* (page 61) for detailed information.

4.3 List of active deprecations

This pages lists all active deprecations in the SymPy codebase. See the *Deprecation Policy* (page 3017) page for a description of SymPy's deprecation policy, as well as instructions for contributors on how to deprecate things.

In particular, the deprecation policy for SymPy is for deprecations to last at least **1 year** after the first major release that includes the deprecation. After that period, the deprecated functionality may be removed from SymPy, and code will need to be updated to use the replacement feature to continue working.

During the deprecation period, a SymPyDeprecationWarning message will be printed whenever the deprecated functionality is used. It is recommended for users to update their code so that it does not use deprecated functionality, as described below for each given deprecation.

4.3.1 Silencing SymPy Deprecation Warnings

To silence SymPy deprecation warnings, add a filter using the warnings module. For example:

```
import warnings
from sympy.utilities.exceptions import SymPyDeprecationWarning

warnings.filterwarnings(
    # replace "ignore" with "error" to make the warning raise an exception.
    # This useful if you want to test you aren't using deprecated code.
    "ignore",

    # message may be omitted to filter all SymPyDeprecationWarnings
    message=r"(?s).*<regex matching the warning message>",

    category=SymPyDeprecationWarning,
    module=r"<regex matching your module>"
)
```

Here (?s).*<regex matching the warning message> is a regular expression matching the warning message. For example, to filter a warning about sympy.printing, you might use message=r"(?s).*sympy\.printing". The leading (?s).* is there because the warnings module matches message against the start of the warning message, and because typical warning messages span multiple lines.

<regex matching your module> should be a regular expression matching your module that uses the deprecated code. It is recommended to include this so that you don't also silence the same warning for unrelated modules.

This same pattern may be used to instead turn SymPyDeprecationWarning into an error so that you can test that you aren't using deprecated code. To do this, replace "ignore" with "error" in the above example. You may also omit message to make this apply to all SymPyDeprecationWarning warnings.

If you are using pytest, you can use the pytest warnings filtering capabilities to either ignore SymPyDeprecationWarning or turn them into errors.

Note: The Python -W flag and PYTHONWARNINGS environment variable will NOT work to filter SymPy deprecation warnings (see this blog post by Ned Batchelder and this SymPy issue for details on why). You will need to either add a warnings filter as above or use pytest to filter SymPy deprecation warnings.

4.3.2 Version 1.11

New Mathematica code parser

The old mathematica code parser defined in the module `sympy.parsing.mathematica` in the function `mathematica` is deprecated. The function `parse_mathematica` with a new and more comprehensive parser should be used instead.

The `additional_translations` parameter for the Mathematica parser is not available in `parse_mathematica`. Additional translation rules to convert Mathematica expressions into SymPy ones should be specified after the conversion using SymPy's `.replace( )` or `.subs( )` methods on the output expression. If the translator fails to recognize the logical meaning of a Mathematica expression, a form similar to Mathematica's full form will be returned, using SymPy's `Function` object to encode the nodes of the syntax tree.

For example, suppose you want `F` to be a function that returns the maximum value multiplied by the minimum value, the previous way to specify this conversion was:

```
>>> from sympy.parsing.mathematica import mathematica
>>> mathematica('F[7,5,3]', {'F[*x]': 'Max(*x)*Min(*x)'})
21
```

Now you can do the same with

```
>>> from sympy.parsing.mathematica import parse_mathematica
>>> from sympy import Function, Max, Min
>>> parse_mathematica("F[7,5,3]").replace(Function("F"), lambda *x:␣
↪Max(*x)*Min(*x))
21
```

Redundant static methods in `carmichael`

A number of static methods in `~.carmichael` are just wrappers around other functions. Instead of `carmichael.is_perfect_square` use `sympy.ntheory.primetest.is_square` and instead of `carmichael.is_prime` use `~.isprime`. Finally, `carmichael.divides` can be replaced by instead checking

```
n % p == 0
```

The `check` argument to `HadamardProduct`, `MatAdd` and `MatMul`

This argument can be used to pass incorrect values to `~.HadamardProduct`, `~.MatAdd`, and `~.MatMul` leading to later problems. The `check` argument will be removed and the arguments will always be checked for correctness, i.e., the arguments are matrices or matrix symbols.

4.3.3 Version 1.10

Some traversal functions have been moved

Some traversal functions have moved. Specifically, the functions

- `bottom_up`
- `interactive_traversal`
- `postorder_traversal`
- `preorder_traversal`
- `use`

have moved to different SymPy submodules.

These functions should be used from the top-level `sympy` namespace, like

```
sympy.preorder_traversal
```

or

```
from sympy import preorder_traversal
```

In general, end-users should use the top-level `sympy` namespace for any functions present there. If a name is in the top-level namespace, its specific SymPy submodule should not be relied on, as functions may move around due to internal refactorings.

`sympy.core.trace`

The trace object `sympy.core.trace.Tr()` was moved to `sympy.physics.quantum.trace.Tr()`. This was because it was only used in the `sympy.physics.quantum` submodule, so it was better to have it there than in the core.

The `sympy.core.compatibility` submodule

The `sympy.core.compatibility` submodule is deprecated.

This submodule was only ever intended for internal use. Now that SymPy no longer supports Python 2, this module is no longer necessary, and the remaining helper functions have been moved to more convenient places in the SymPy codebase.

Some of the functions that were in this module are available from the top-level SymPy namespace, i.e.,

```
sympy.ordered
sympy.default_sort_key
```

or

```
from sympy import ordered, default_sort_key
```

In general, end-users should use the top-level `sympy` namespace for any functions present there. If a name is in the top-level namespace, its specific SymPy submodule should not be relied on, as functions may move around due to internal refactorings.

The remaining functions in `sympy.core.compatibility` were only intended for internal SymPy use and should not be used by user code.

Additionally, these two functions, `ordered` and `default_sort_key`, also used to be in `sympy.utilities.iterables` but have been moved from there as well.

4.3.4 Version 1.9

expr_free_symbols

The `expr_free_symbols` attribute of various SymPy objects is deprecated.

`expr_free_symbols` was meant to represent indexed objects such as `MatrixElement` and *Indexed* (page 1402) as free symbols. This was intended to make derivatives of free symbols work. However, this now works without making use of the method:

```
>>> from sympy import Indexed, MatrixSymbol, diff
>>> a = Indexed("A", 0)
>>> diff(a**2, a)
2*A[0]
>>> X = MatrixSymbol("X", 3, 3)
>>> diff(X[0, 0]**2, X[0, 0])
2*X[0, 0]
```

This was a general property that was added to solve a very specific problem but it added a layer of abstraction that is not necessary in general.

1. objects that have structural "non-expression" nodes already allow one to focus on the expression node if desired, e.g.

   ```
   >>> from sympy import Derivative, symbols, Function
   >>> x = symbols('x')
   >>> f = Function('f')
   >>> Derivative(f(x), x).expr
   f(x)
   ```

   introduction of this property encourages imprecise thinking when requesting free_symbols since it allows one to get symbols from a specific node of an object without specifying the node

2. the property was incorrectly added to `AtomicExpr` so numbers are returned as `expr_free_symbols`:

   ```
   >>> S(2).expr_free_symbols
   2
   ```

3. the application of the concept was misapplied to define `Subs.expr_free_symbols`: it added in `expr_free_symbols` of the point but the point is a `Tuple` so nothing was added

4. it was not used anywhere else in the codebase except in the context of differentiating a `Subs` object, which suggested that it was not something of general use, this is also confirmed by the fact that,

5. it was added without specific tests except for test of the derivatives of the Subs object for which it was introduced

See issue #21494 for more discussion.

sympy.stats.sample(numsamples=n)

The `numsamples` parameter to *sympy.stats.sample()* (page 2971) is deprecated.

`numsamples` makes `sample()` return a list of size `numsamples`, like

```
>>> from sympy.stats import Die, sample
>>> X = Die('X', 6)
>>> sample(X, numsamples=3)
[3, 2, 3]
```

However, this functionality can be easily implemented by the user with a list comprehension

```
>>> [sample(X) for i in range(3)]
[5, 4, 3]
```

Additionally, it is redundant with the `size` parameter, which makes `sample` return a NumPy array with the given shape.

```
>>> sample(X, size=(3,))
array([6, 6, 1])
```

Historically, `sample` was changed in SymPy 1.7 so it returned an iterator instead of sample value. Since an iterator was returned, a numsamples parameter was added to specify the length of the iterator.

However, this new behavior was considered confusing, as discussed in issue #21563, so it was reverted. Now, `sample_iter` should be used if a iterator is needed. Consequently, the `numsamples` parameter is no longer needed for `sample()`.

sympy.polys.solvers.RawMatrix

The `RawMatrix` class is deprecated. The `RawMatrix` class was a subclass of `Matrix` that used domain elements instead of `Expr` as the elements of the matrix. This breaks a key internal invariant of `Matrix` and this kind of subclassing limits improvements to the `Matrix` class.

The only part of SymPy that documented the use of the `RawMatrix` class was the Smith normal form code, and that has now been changed to use `DomainMatrix` instead. It is recommended that anyone using `RawMatrix` with the previous Smith Normal Form code should switch to using `DomainMatrix` as shown in issue #21402. A better API for the Smith normal form will be added later.

Non-Expr objects in a Matrix

In SymPy 1.8 and earlier versions it was possible to put non-*Expr* (page 947) elements in a *Matrix* (page 1361) and the matrix elements could be any arbitrary Python object:

```
>>> M = Matrix([[(1, 2), {}]])
```

This is not useful and does not really work, e.g.:

```
>>> M + M
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Dict' and 'Dict'
```

The main reason for making this possible was that there were a number of `Matrix` subclasses in the SymPy codebase that wanted to work with objects from the polys module, e.g.

1. `RawMatrix` (see *above* (page 168)) was used in `solve_lin_sys` which was part of `heurisch` and was also used by `smith_normal_form`. The `NewMatrix` class used domain elements as the elements of the Matrix rather than `Expr`.
2. `NewMatrix` was used in the `holonomic` module and also used domain elements as matrix elements
3. `PolyMatrix` used a mix of `Poly` and `Expr` as the matrix elements and was used by `risch`.

All of these matrix subclasses were broken in different ways and the introduction of *DomainMatrix* (page 2671) (#20780, #20759, #20621, #19882, #18844) provides a better solution for all cases. Previous PRs have removed the dependence of these other use cases on Matrix (#21441, #21427, #21402) and now #21496 has deprecated having non-`Expr` in a `Matrix`.

This change makes it possible to improve the internals of the Matrix class but it potentially impacts on some downstream use cases that might be similar to the uses of `Matrix` with non-`Expr` elements that were in the SymPy codebase. A potential replacement for code that used `Matrix` with non-`Expr` elements is *DomainMatrix* (page 2671) if the elements are something like domain elements and a domain object can be provided for them. Alternatively if the goal is just printing support then perhaps `TableForm` can be used.

It isn't clear what to advise as a replacement here without knowing more about the usecase. If you are unclear how to update your code, please open an issue or write to our mailing list so we can discuss it.

The `get_segments` attribute of plotting objects

The `get_segments` method implemented in *Line2DBaseSeries* (page 2865) is used to convert two list of coordinates, `x` and `y`, into a list of segments used by Matplotlib's `LineCollection` to plot a line.

Since the list of segments is only required by Matplotlib (for example, Bokeh, Plotly, Mayavi, K3D only require lists of coordinates), this has been moved inside the `MatplotlibBackend` class.

Note that previously, the method *get_points()* (page 2866) always returned uniformly sampled points, which meant that some functions were not plotted correctly when using `get_points()` to plot with Matplotlib.

To avoid this problem, the method `get_segments()` could be used, which used adaptive sampling and which could be used with Matplotlib's `LineCollection`. However, this has been changed, and now `get_points()` can also use adaptive sampling. The *get_data()* (page 2866) method can also be used.

The `mdft` function in `sympy.physics.matrices`

The `sympy.physics.matrices.mdft()` function is deprecated. It can be replaced with the `DFT` class in `sympy.matrices.expressions.fourier`.

In particular, replace `mdft(n)` with `DFT(n).as_explicit()`. For example:

```
>>> from sympy.physics.matrices import mdft
>>> mdft(3) # DEPRECATED
Matrix([
[sqrt(3)/3,                 sqrt(3)/3,                 sqrt(3)/3],
[sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3,  sqrt(3)*exp(2*I*pi/3)/3],
[sqrt(3)/3,  sqrt(3)*exp(2*I*pi/3)/3, sqrt(3)*exp(-2*I*pi/3)/3]])
```

```
>>> from sympy.matrices.expressions.fourier import DFT
>>> DFT(3)
DFT(3)
>>> DFT(3).as_explicit()
Matrix([
[sqrt(3)/3,                 sqrt(3)/3,                 sqrt(3)/3],
[sqrt(3)/3, sqrt(3)*exp(-2*I*pi/3)/3,  sqrt(3)*exp(2*I*pi/3)/3],
[sqrt(3)/3,  sqrt(3)*exp(2*I*pi/3)/3, sqrt(3)*exp(-2*I*pi/3)/3]])
```

This was changed because the `sympy.physics` submodule is supposed to only contain things that are specific to physics, but the discrete Fourier transform matrix is a more general mathematical concept, so it is better located in the `sympy.matrices` module. Furthermore, the `DFT` class is a *matrix expression* (page 1369), meaning it can be unevaluated and support symbolic shape.

The private `SparseMatrix._smat` and `DenseMatrix._mat` attributes

The `._mat` attribute of *Matrix* (page 1361) and the `._smat` attribute of *SparseMatrix* (page 1364) are deprecated.

The internal representation of `Matrix` and `SparseMatrix` was changed to be a *DomainMatrix* (page 2671) in #21626 so that it is no longer possible to expose a mutable list/dict as a way of mutating a `Matrix`. Instead of `._mat` the new `.flat()` method can be used, which returns a new list that cannot be used to mutate the `Matrix` itself. Instead of `._smat` the `.todok()` method can be used which returns a new dict.

Note that these attributes are already changed in SymPy 1.9 to return read-only copies, so that any code that relied on mutating them will be broken. Also these attributes were technically always private (they started with an underscore), so user code should not really have been using them in the first place.

laplace_transform of a Matrix with noconds=False

Prior to version 1.9, calling *laplace_transform()* (page 577) on a *Matrix* (page 1361) with `noconds=False` (which is the default), resulted in a Matrix of tuples:

```
>>> from sympy import laplace_transform, symbols, eye
>>> t, z = symbols('t z')
>>> laplace_transform(eye(2), t, z)
Matrix([
[(1/z, 0, True),   (0, 0, True)],
[  (0, 0, True), (1/z, 0, True)]])
```

However, `Matrix` is only designed to work with `Expr` objects (see *Non-Expr objects in a Matrix* (page 169) above).

To avoid this, either use `noconds=True` to remove the convergence conditions

```
>>> laplace_transform(eye(2), t, z, noconds=True)
Matrix([
[1/z,   0],
[  0, 1/z]])
```

or use `legacy_matrix=False` to return the new behavior, which will be to return a single tuple with the Matrix in the first argument and the convergence conditions combined into a single condition for the whole matrix.

```
>>> laplace_transform(eye(2), t, z, legacy_matrix=False)
(Matrix([
[1/z,   0],
[  0, 1/z]]), 0, True)
```

When this deprecation is removed the `legacy_matrix=False` behavior will become the default, but the flag will be left intact for compatibility.

4.3.5 Version 1.8

sympy.printing.theanocode

Theano has been discontinued, and forked into a new project called Aesara. The `sympy.printing.theanocode` module has been renamed to *sympy.printing.aesaracode* (page 2166), and all the corresponding functions have been renamed (e.g., `theano_code` has been renamed to *aesara_code()* (page 2167), `TheanoPrinter` has been renamed to *AesaraPrinter* (page 2166), and so on).

sympy.assumptions.handlers.AskHandler and related methods

`Predicate` has experienced a big design change. Previously, its handler was a list of `AskHandler` classes and registration was done by `add_handler()` and `remove_handler()` functions. Now, its handler is a multipledispatch instance and registration is done by `register()` or `register_many()` methods. Users must define a predicate class to introduce a new one.

Previously, handlers were defined and registered this way:

```
class AskPrimeHandler(AskHandler):
    @staticmethod
    def Integer(expr, assumptions):
        return expr.is_prime

register_handler('prime', AskPrimeHandler)
```

It should be changed to this:

```
# Predicate definition.
# Not needed if you are registering the handler to existing predicate.
class PrimePredicate(Predicate):
    name = 'prime'
Q.prime = PrimePredicate()

# Handler registration
@Q.prime.register(Integer)
def _(expr, assumptions):
    return expr.is_prime
```


See GitHub issue #20209.

4.3.6 Version 1.7.1

Calling sympy.stats.StochasticProcess.distribution with RandomIndexedSymbol

The `distribution` method of `sympy.stats` *stochastic processes* (page 2944) used to accept a `RandomIndexedSymbol` (that is, a stochastic process indexed with a timestamp), but should now only be called with the timestamp.

For example, if you have

```
>>> from sympy import symbols
>>> from sympy.stats import WienerProcess
>>> W = WienerProcess('W')
>>> t = symbols('t', positive=True)
```

Previously this would work

```
W.distribution(W(t)) # DEPRECATED
```

It should now be called like

```
>>> W.distribution(t)
NormalDistribution(0, sqrt(t))
```

This was change was made as part of a change to store only `Basic` objects in `sympy.stats.args`. See issue #20078 for details.

4.3.7 Version 1.7

`sympy.stats.DiscreteMarkovChain.absorbing_probabilites()`

The `absorbing_probabilites` method name was misspelled. The correct spelling *absorbing_probabilities()* (page 2946) ("absorbing probabilities") should be used instead.

`sympy.utilities.misc.find_executable()`

The function `sympy.utilities.misc.find_executable()` is deprecated. Instead use the standard library `shutil.which()` function, which has been in the standard library since Python 3.3 and is more powerful.

Mutable attributes in `sympy.diffgeom`

Several parts of *sympy.diffgeom* (page 2799) have been updated to no longer be mutable, which better matches the immutable design used in the rest of SymPy.

- Passing strings for symbol names in *CoordSystem* (page 2801) is deprecated. Instead you should be explicit and pass symbols with the appropiate assumptions, for instance, instead of

  ```
  CoordSystem(name, patch, ['x', 'y']) # DEPRECATED
  ```

  use

  ```
  CoordSystem(name, patch, symbols('x y', real=True))
  ```

- Similarly, the `names` keyword argument has been renamed to `symbols`, which should be a list of symbols.
- The `Manifold.patches` attribute is deprecated. Patches should be tracked separately.
- The `Patch.coord_systems` attribute is deprecated. Coordinate systems should be tracked separately.
- The `CoordSystem.transforms` attribute, `CoordSystem.connect_to()` method, and `CoordSystem.coord_tuple_transform_to()` method are deprecated. Instead, use the `relations` keyword to the `CoordSystem` class constructor and the *CoordSystem.transformation()* (page 2805) and *CoordSystem.transform()* (page 2805) methods (see the docstring of *CoordSystem* (page 2801) for examples).

The `unicode` argument and attribute to `sympy.printing.pretty.stringpict.prettyForm` and the `sympy.printing.pretty.pretty_symbology.xstr` function

The `sympy.printing.pretty.pretty_symbology.xstr` function, and the `unicode` argument and attribute to *sympy.printing.pretty.stringpict.prettyForm* (page 2188) were both present to support the Unicode behavior of Python 2. Since Unicode strings are the default in Python 3, these are not needed any more. `xstr()` should be replaced with just `str()`, the `unicode` argument to `prettyForm` should be omitted, and the `prettyForm.unicode` attribute should be replaced with the `prettyForm.s` attribute.

Passing the arguments to `lambdify` as a `set`

Passing the function arguments to lambdify as a set is deprecated. Instead pass them as a list or tuple. For example, instead of

```
lambdify({x, y}, x + 2*y) # WRONG
```

use

```
lambdify((x, y), x + 2*y) # RIGHT
```

This is because sets are unordered. For instance, in the above example it would be impossible for `lambidfy` to know if it was called with `{x, y}` or `{y, x}`. Thus, when passed the arguments as a set `lambdify` would have to guess their order, which would lead to an incorrect function if it guessed incorrectly.

Core operators no longer accept non-Expr args

The core operator classes *Add* (page 1013), *Mul* (page 1009), and *Pow* (page 1005) can no longer be constructed directly with objects that are not subclasses of *Expr* (page 947).

Expr (page 947) is the superclass of all SymPy classes that represent scalar numeric quantities. For example, *sin* (page 389), *Symbol* (page 976), and *Add* (page 1013) are all subclasses of *Expr* (page 947). However, may objects in SymPy are not *Expr* (page 947) because they represent some other type of mathematical object. For example, *Set* (page 1185), *Poly* (page 2378), and *Boolean* (page 1163) are all non-`Expr`. These do not make mathematical sense inside of `Add`, `Mul`, and `Pow`, which are designed specifically to represent the addition, multiplication, and exponentiation of scalar complex numbers.

Manually constructing one of these classes with such an object is possible, but it will generally create something that will then break. For example

```
Mul(1, Tuple(2)) # This is deprecated
```

works and creates `Tuple(2)`, but only because `Mul` is "tricked" by always treating $1 \cdot x = x$. If instead you try

```
Mul(2, Tuple(2)) # This is deprecated
```

it fails with an exception

```
AttributeError: 'Tuple' object has no attribute 'as_coeff_Mul'
```

because it tries to call a method of `Expr` on the `Tuple` object, which does not have all the `Expr` methods (because it is not a subclass of `Expr`).

If you want to use the `+`, `*`, or `**` operation on a non-`Expr` object, use the operator directly rather than using `Mul`, `Add` or `Pow`. If functional versions of these are desired, you can use a `lambda` or the `operator` module.

4.3.8 Version 1.6

Various `sympy.utilities` submodules have moved

The following submodules have been renamed.

- `sympy.utilities.benchmarking` → `sympy.testing.benchmarking`
- `sympy.utilities.pytest` → `sympy.testing.pytest`
- `sympy.utilities.randtests` → `sympy.core.random`
- `sympy.utilities.runtests` → `sympy.testing.runtests`
- `sympy.utilities.tmpfiles` → `sympy.testing.tmpfiles`

`sympy.testing.randtest`

`sympy.testing.randtest` is deprecated. The functions in it have been moved to `sympy.core.random`. The following functions have been moved.

- `sympy.testing.randtest.random_complex_number` → `sympy.core.random.random_complex_number`
- `sympy.testing.randtest.verify_numerically` → `sympy.core.random.verify_numerically`
- `sympy.testing.randtest.test_derivative_numerically` → `sympy.core.random.test_derivative_numerically`
- `sympy.testing.randtest._randrange` → `sympy.core.random._randrange`
- `sympy.testing.randtest._randint` → `sympy.core.random._randint`

Mixing `Poly` and non-polynomial expressions in binary operations

In previous versions of SymPy, *Poly* (page 2378) was a subclass of *Expr* (page 947), but it has been changed to only be a subclass of *Basic* (page 927). This means that some things that used to work with `Poly` are now deprecated because they are only designed to work with *Expr* (page 947) objects.

This includes combining `Poly` with `Expr` objects using binary operations, for example

```
Poly(x)*sin(x) # DEPRECATED
```

To do this, either explicitly convert the non-`Poly` operand to a `Poly` using *Expr.as_poly()* (page 956) or convert the `Poly` operand to an *Expr* (page 947) using *Poly.as_expr()* (page 2383), depending on which type you want the result to be.

The print_cyclic flag of sympy.combinatorics.Permutation

The print_cyclic attribute of *sympy.combintorics.Permutation* (page 257) controls whether permutations print as cycles or arrays. This would be done by setting Permutation.print_cyclic = True or Permutation.print_cyclic = False. However, this method of controlling printing is bad because it is a global flag, but printing should not depend on global behavior.

Instead, users should use the perm_cyclic flag of the corresponding printer. The easiest way to configure this is to set the flag when calling *init_printing()* (page 2119), like

```
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False) # Makes Permutation print in array form
>>> from sympy.combinatorics import Permutation
>>> Permutation(1, 2)(3, 4)
⎛0 1 2 3 4⎞
⎝0 2 1 4 3⎠
```

The *Permutation* (page 257) docstring contains more details on the perm_cyclic flag.

Using integrate with Poly

In previous versions of SymPy, *Poly* (page 2378) was a subclass of *Expr* (page 947), but it has been changed to only be a subclass of *Basic* (page 927). This means that some things that used to work with Poly are now deprecated because they are only designed to work with *Expr* (page 947) objects.

This includes calling *integrate()* (page 598) or *Integral* (page 601) with Poly.

To integrate a Poly, use the *Poly.integrate()* (page 2398) method. To compute the integral as an *Expr* (page 947) object, call the *Poly.as_expr()* (page 2383) method first.

See also *Mixing Poly and non-polynomial expressions in binary operations* (page 175) above.

The string fallback in sympify()

The current behavior of *sympify()* (page 918) is that sympify(expr) tries various methods to try to convert expr into a SymPy objects. If all these methods fail, it takes str(expr) and tries to parse it using *parse_expr()* (page 2122). This string fallback feature is deprecated. It is problematic for a few reasons:

- It can affect performance in major ways. See for instance issues #18056 and #15416 where it caused up to 100x slowdowns. The issue is that SymPy functions automatically call sympify on their arguments. Whenever a function is passed something that sympify doesn't know how to convert to a SymPy object, for instance, a Python function type, it passes the string to *parse_expr()* (page 2122). This is significantly slower than the direct conversions that happen by default. This occurs specifically whenever sympify() is used in library code instead of _sympify() (or equivalently sympify(strict=True)), but presently this is done a lot. Using strict=True will at some point be the default for all library code, but this is a harder change to make.
- It can cause security issues, since strings are evaled, and objects can return whatever string they want in their __repr__. See also https://github.com/sympy/sympy/pull/12524.

- It really isn't very useful to begin with. Just because an object's string form can be parsed into a SymPy expression doesn't mean it should be parsed that way. This is usually correct for custom numeric types, but an object's repr could be anything. For instance, if the string form of an object looks like a valid Python identifier, it will parse as a `Symbol`.

There are plenty of ways to make custom objects work inside of *sympify()* (page 918).

- Firstly, if an object is intended to work alongside other SymPy expressions, it should subclass from *Basic* (page 927) (or *Expr* (page 947)). If it does, *sympify()* (page 918) will just return it unchanged because it will already be a valid SymPy object.
- For objects that you control, you can add the `_sympy_` method. The *sympify docstring* (page 918) has an example of this.
- For objects that you don't control, you can add a custom converter to the `sympy.core.sympify.converter` dictionary. The *sympify()* (page 918) docstring also has an example of this.

To silence this deprecation warning in all cases, you can pass `strict=True` to `sympify()`. However, note that this will also disable some other conversions such as conversion of strings (for converting strings to SymPy types, you can explicitly use *parse_expr()* (page 2122)).

Creating an indefinite `Integral` with an `Eq` argument

Passing an *Eq()* (page 1023) object to *integrate()* (page 598) is deprecated in the case where the integral is indefinite. This is because if $f(x) = g(x)$, then $\int f(x)\,dx = \int g(x)\,dx$ is not true in general, due to the arbitrary constants (which `integrate` does not include).

If you want to make an equality of indefinite integrals, use `Eq(integrate(f(x), x), integrate(g(x), x))` explicitly.

If you already have an equality object `eq`, you can use `Eq(integrate(eq.lhs, x), integrate(eq.rhs, x))`.

4.3.9 Version 1.5

`Tensor.fun_eval` and `Tensor.__call__`

`TensExpr.fun_eval` and `Tensor.__call__` (i.e., calling a tensor to evaluate it) are deprecated. The `Tensor.substitute_indices()` method should be used. This was changed because `fun_eval` was considered a confusing name and using function evaluation was considered both confusing and dangerous.

`TensorType`

The `TensorType` class is deprecated. Use *tensor_heads()* (page 1413) instead. The `TensorType` class had no purpose except shorter creation of *TensorHead* (page 1411) objects.

See also *The tensorhead() function* (page 178) below.

The dummy_fmt argument to TensorIndexType

The dummy_fmt keyword argument to *TensorIndexType* (page 1409) is deprecated. Setting dummy_fmt='L' leads to _dummy_fmt='L_%d', which is confusing and uses obsolete string formatting. dummy_name should be used instead. This change was made because dummy_name is a clearer name.

The metric argument to TensorIndexType

The metric keyword argument to *TensorIndexType* (page 1409) is deprecated. The name "metric" was ambiguous because it meant "metric symmetry" in some places and "metric tensor" in others.

Either the metric_symmetry keyword or the TensorIndexType.set_metric() method should be used instead.

The get_kronecker_delta() and get_epsilon() methods of TensorIndexType

The get_kronecker_delta() and get_epsilon() methods of *TensorIndexType* (page 1409) are deprecated. Use the TensorIndexType.delta and TensorIndexType.epsilon properties instead, respectively.

The tensorsymmetry() function

The tensorsymmetry() function in sympy.tensor is deprecated. Use the *TensorSymmetry* (page 1420) class constructor instead.

TensorSymmetry is preferred over tensorsymmetry() because the latter

1. Does not have any extra functionality
2. Involves obscure Young tableau
3. Is not a member of the TensorSymmetry class

The tensorhead() function

The tensorhead() function is deprecated in favor of *tensor_heads()* (page 1413). tensor_heads() is more consistent with other SymPy names (i.e., Symbol and symbols() or TensorIndex and tensor_indices()). It also does not use Young tableau to denote symmetries.

Methods to sympy.physics.units.Quantity

The following methods of *sympy.physics.units.quantities.Quantity* (page 1587) are deprecated.

- `Quantity.set_dimension()`. This should be replaced with `unit_system.set_quantity_dimension` or `Quantity.set_global_dimension()`.
- `Quantity.set_scale_factor()`. This should be replaced with `unit_system.set_quantity_scale_factor` or *Quantity.set_global_relative_scale_factor()* (page 1588)
- `Quantity.get_dimensional_expr()`. This is now associated with *UnitSystem* (page 1587) objects. The dimensional relations depend on the unit system used. Use `unit_system.get_dimensional_expr()` instead.
- `Quantity._collect_factor_and_dimension`. This has been moved to the *UnitSystem* (page 1587) class. Use `unit_system._collect_factor_and_dimension(expr)` instead.

See *The dimension and scale_factor arguments to sympy.physics.units.Quanitity* (page 182) below for the motivation for this change.

The is_EmptySet attribute of sets

The `is_EmptySet` attribute of *Set* (page 1185) objects is deprecated. Instead either use

```
from sympy import S
s is S.EmptySet
```

or

```
s.is_empty
```

The difference is that `s.is_empty` may return `None` if it is unknown if the set is empty.

ProductSet(iterable)

Passing a single iterable as the first argument to *ProductSet* (page 1199) is deprecated. Creating a product set from an iterable should be done using `ProductSet(*iterable)`, or as each individual argument. For example

```
>>> from sympy import ProductSet
>>> sets = [{i} for i in range(3)]
>>> ProductSet(*sets)
ProductSet({0}, {1}, {2})
>>> ProductSet({1, 2}, {1})
ProductSet({1, 2}, {1})
```

This is done because sets themselves can be iterables, and sets of sets are allowed. But the product set of a single set should mathematically be that set itself (or more exactly, the set of 1-tuples of elements of that set). Automatically denesting a single iterable makes it impossible to represent this object and makes `ProductSet` not generalize correctly when passed 1 argument. On the other hand, treating the first argument differently if it is a set than if it is another type of iterable (which is what is currently done in the deprecated code path) is confusing behavior.

The set_potential_energy method in sympy.physics.mechanics

The set_potential_energy() methods of *sympy.physics.mechanics.particle.Particle* (page 1743) and *sympy.physics.mechanics.rigidbody.RigidBody* (page 1746) are deprecated.

Instead one should set the *Particle.potential_energy* (page 1746) and *RigidBody.potential_energy* (page 1749) attributes to set the potential energy, like

```
P.potential_energy = scalar
```

This change was made to be more Pythonic, by using setters and getters of a @property method rather than an explicit set_ method.

Using a set for the condition in ConditionSet

Using a set for the condition in ConditionSet is deprecated. A boolean should be used instead. This is because the condition is mathematically a boolean, and it is ambiguous what a set should mean in this context.

To fix this deprecation, replace

```
ConditionSet(symbol, set_condition)
```

with

```
ConditionSet(symbol, And(*[Eq(lhs, 0) for lhs in set_condition]))
```

For example,

```
ConditionSet((x, y), {x + 1, x + y}, S.Reals) # DEPRECATED
```

would become

```
ConditionSet((x, y), Eq(x + 1, 0) & Eq(x + y, 0), S.Reals)
```

The max_degree and get_upper_degree properties of sympy.polys.multivariate_resultants.DixonResultant

The max_degree property and get_upper_degree() methods of DixonResultant are deprecated. See issue #17749 for details.

Eq(expr) with the rhs defaulting to 0

Calling *Eq* (page 1023) with a single argument is deprecated. This caused the right-hand side to default to 0, but this behavior was confusing. You should explicitly use Eq(expr, 0) instead.

Non-tuple iterable for the first argument to Lambda

Using a non-tuple as the first argument to *Lambda* (page 1039) is deprecated. If you have a non-tuple, convert it to a tuple first, like `Lambda(tuple(args), expr)`.

This was done so that `Lambda` could support general tuple unpacking, like

```
>>> from sympy import Lambda, symbols
>>> x, y, z = symbols('x y z')
>>> f = Lambda((x, (y, z)), x + y + z)
>>> f(1, (2, 3))
6
```

The evaluate flag to differentiate_finite

The `evaluate` flag to *differentiate_finite()* (page 241) is deprecated.

`differentiate_finite(expr, x, evaluate=True)` expands the intermediate derivatives before computing differences. But this usually not what you want, as it does not satisfy the product rule.

If you really do want this behavior, you can emulate it with

```
diff(expr, x).replace(
    lambda arg: arg.is_Derivative,
    lambda arg: arg.as_finite_difference())
```

See the discussion on issue #17881.

4.3.10 Version 1.4

TensorIndexType.data and related methods

The `TensorIndexType.data` property is deprecated, as well as several methods which made use of it including the `get_matrix()`, the `__getitem__()` (indexing), `__iter__()` (iteration), `_components_data_full_destroy()`, and `__pow__()` (**) methods. Storing data on tensor objects was a design flaw and not consistent with how the rest of SymPy works.

Instead, the *TensExpr.replace_with_arrays()* (page 1414) method should be used.

The clear_cache and clear_subproducts keywords to Matrix.is_diagonalizable

The `clear_cache` and `clear_subproducts` keywords to *Matrix.is_diagonalizable()* (page 1241) are deprecated. These used to clear cached entries, but this cache was removed because it was not actually safe given that `Matrix` is mutable. The keywords now do nothing.

The rows and cols keyword arguments to Matrix.jordan_block

The rows and cols keywords to *Matrix.jordan_block* (page 1345) are deprecated. The size parameter should be used to specify the (square) number of rows and columns.

The non-square matrices created by setting rows and cols are not mathematically Jordan block matrices, which only make sense as square matrices.

To emulate the deprecated jordan_block(rows=n, cols=m) behavior, use a general banded matrix constructor, like

```
>>> from sympy import Matrix, symbols
>>> eigenvalue = symbols('x')
>>> def entry(i, j):
...     if i == j:
...         return eigenvalue
...     elif i + 1 == j: # use j + 1 == i for band='lower'
...         return 1
...     return 0
>>> # the same as the deprecated Matrix.jordan_block(rows=3, cols=5,␣
↪eigenvalue=x)
>>> Matrix(3, 5, entry)
Matrix([
[x, 1, 0, 0, 0],
[0, x, 1, 0, 0],
[0, 0, x, 1, 0]])
```

4.3.11 Version 1.3

The source() function

The *source()* (page 2116) function is deprecated. Use inspect.getsource(obj) instead, or if you are in IPython or Jupyter, use obj??.

The dimension and scale_factor arguments to sympy.physics.units.Quanitity

The dimension and scale_factor arguments to *sympy.physics.units.quantities.Quantity* (page 1587) are deprecated.

The problem with these arguments is that **dimensions** are **not** an **absolute** association to a quantity. For example:

- in natural units length and time are the same dimension (so you can sum meters and seconds).
- SI and cgs units have different dimensions for the same quantities.

At this point a problem arises for scale factor as well: while it is always true that kilometer / meter == 1000, some other quantities may have a relative scale factor or not depending on which unit system is currently being used.

Instead, things should be managed on the *DimensionSystem* (page 1584) class. The DimensionSystem.set_quantity_dimension() method should be used instead of the

`dimension` argument, and the `DimensionSystem.set_quantity_scale_factor()` method should be used instead of the `scale_factor` argument.

See issue #14318 for more details. See also *Methods to sympy.physics.units.Quantity* (page 179) above.

Importing `classof` and `a2idx` from `sympy.matrices.matrices`

The functions `sympy.matrices.matrices.classof` and `sympy.matrices.matrices.a2idx` were duplicates of the same functions in `sympy.matrices.common`. The two functions should be used from the `sympy.matrices.common` module instead.

4.3.12 Version 1.2

Dot product of non-row/column vectors

The *Matrix.dot()* (page 1296) method has confusing behavior where `A.dot(B)` returns a list corresponding to `flatten(A.T*B.T)` when `A` and `B` are matrices that are not vectors (i.e., neither dimension is size 1). This is confusing. The purpose of `Matrix.dot()` is to perform a mathematical dot product, which should only be defined for vectors (i.e., either a $n \times 1$ or $1 \times n$ matrix), but in a way that works regardless of whether each argument is a row or column vector. Furthermore, returning a list here was much less useful than a matrix would be, and resulted in a polymorphic return type depending on the shapes of the inputs.

This behavior is deprecated. `Matrix.dot` should only be used to do a mathematical dot product, which operates on row or column vectors. Use the `*` or `@` operators to do matrix multiplication.

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[2, 3], [1, 2]])
>>> A*B
Matrix([
[ 4,  7],
[10, 17]])
>>> A@B
Matrix([
[ 4,  7],
[10, 17]])
```

`sympy.geometry.Line3D.equation` no longer needs the `k` argument

The `k` argument to *sympy.geometry.line.Line3D.equation()* (page 2244) method is deprecated.

Previously, the function `Line3D.equation` returned `(X, Y, Z, k)` which was changed to `(Y-X, Z-X)` (here `X`, `Y` and `Z` are expressions of `x`, `y` and `z` respectively). As in 2D an equation is returned relating `x` and `y` just like that in 3D two equations will be returned relating `x`, `y` and `z`.

So in the new `Line3D.equation` the `k` argument is not needed anymore. Now the `k` argument is effectively ignored. A `k` variable is temporarily formed inside `equation()` and then gets substituted using `subs()` in terms of `x` and then `(Y-X, Z-X)` is returned.

Previously:

```
>>> from sympy import Point3D,Line3D
>>> p1,p2 = Point3D(1, 2, 3), Point3D(5, 6, 7)
>>> l = Line3D(p1, p2)
>>> l.equation()
(x/4 - 1/4, y/4 - 1/2, z/4 - 3/4, k)
```

Now:

```
>>> from sympy import Point3D, Line3D, solve
>>> p1,p2 = Point3D(1, 2, 3), Point3D(5, 6, 7)
>>> l = Line3D(p1,p2)
>>> l.equation()
(-x + y - 1, -x + z - 2)
```

CHAPTER

FIVE

API REFERENCE

This section contains a summary of SymPy modules, functions, classes, and methods. All functions and objects implemented in the `sympy` core subpackage are documented below.

5.1 Basics

Contains a description of operations for the basic modules. Subcategories include: *absolute basics*, *manipulation*, *assumptions*, *functions*, *simplification*, *calculus*, *solvers*, and some other subcategories.

5.2 Code Generation

Contains a description of methods for the generation of compilable and executable code.

5.3 Logic

Contains method details for the *logic* and *sets* modules.

5.4 Matrices

Discusses methods for the matrices, tensor and vector modules.

5.5 Number Theory

Documents methods for the Number theory module.

5.6 Physics

Contains documentation for Physics methods.

5.7 Utilities

Contains docstrings for methods of several utility modules. Subcategories include: *Interactive*, *Parsing*, *Printing*, *Testing*, *Utilities*.

5.8 Topics

Contains method docstrings for several modules. Subcategories include : *Plotting*, *Polynomials*, *Geometry*, *Category Theory*, *Cryptography*, *Differential*, *Holonomic*, *Lie Algebra*, and *Stats*.

5.8.1 Basics

Contents

Assumptions

A module to implement logical predicates and assumption system.

Predicate

class sympy.assumptions.assume.**Predicate**(**args*, ***kwargs*)

Base class for mathematical predicates. It also serves as a constructor for undefined predicate objects.

Explanation

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, `AppliedPredicate` instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function `ask`.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of `Q`. For example, `Q.even` returns the predicate which checks if the argument is even number.

To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to `Q`. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get `UndefinedPredicate` which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

Examples

Applying and evaluating to boolean value:

```
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
>>> Q.sexyprime = SexyPrimePredicate()
>>> @Q.sexyprime.register(Integer, Integer)
... def _(int1, int2, assumptions):
...     args = sorted([int1, int2])
...     if not all(ask(Q.prime(a), assumptions) for a in args):
...         return False
...     return args[1] - args[0] == 6
>>> ask(Q.sexyprime(5, 11))
True
```

Direct constructing returns `UndefinedPredicate`, which can be applied but cannot be dispatched.

```
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P(1)
Q.P(1)
```

(continues on next page)

(continued from previous page)

```
>>> Q.P.register(Integer)(lambda expr, assump: True)
Traceback (most recent call last):
  ...
TypeError: <class 'sympy.assumptions.assume.UndefinedPredicate'> cannot␣
↪be dispatched.
```

References

[R5], [R6]

eval(*args*, *assumptions=True*)

Evaluate `self(*args)` under the given assumptions.

This uses only direct resolution methods, not logical inference.

handler = <dispatched AskPredicateHandler>

classmethod register(**types*, ***kwargs*)

Register the signature to the handler.

classmethod register_many(**types*, ***kwargs*)

Register multiple signatures to same handler.

class sympy.assumptions.assume.**AppliedPredicate**(*predicate*, **args*)

The class of expressions resulting from applying `Predicate` to the arguments. `AppliedPredicate` merely wraps its argument and remain unevaluated. To evaluate it, use the `ask()` function.

Examples

```
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The `function` attribute returns the predicate, and the `arguments` attribute returns the tuple of arguments.

```
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```

Applied predicates can be evaluated to a boolean value with `ask`:

```
>>> ask(Q.integer(1))
True
```

property 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
```

property arguments

Return the arguments which are applied to the predicate.

property function

Return the predicate.

Querying

Queries are used to ask information about expressions. Main method for this is `ask()`:

`sympy.assumptions.ask.`**ask**(*proposition*, *assumptions=True*, *context={}*)

Function to evaluate the proposition with assumptions.

Parameters

proposition : Any boolean expression.

Proposition which will be evaluated to boolean value. If this is not `AppliedPredicate`, it will be wrapped by `Q.is_true`.

assumptions : Any boolean expression, optional.

Local assumptions to evaluate the *proposition*.

context : AssumptionsContext, optional.

Default assumptions to evaluate the *proposition*. By default, this is `sympy.assumptions.global_assumptions` variable.

Returns

`True`, `False`, or `None`

Raises

TypeError : *proposition* or *assumptions* is not valid logical expression.

ValueError : assumptions are inconsistent.

Explanation

This function evaluates the proposition to `True` or `False` if the truth value can be determined. If not, it returns `None`.

It should be discerned from *refine()* (page 197) which, when applied to a proposition, simplifies the argument to symbolic `Boolean` instead of Python built-in `True`, `False` or `None`.

Syntax

- **ask(proposition)**
  Evaluate the *proposition* in global assumption context.

- **ask(proposition, assumptions)**
  Evaluate the *proposition* with respect to *assumptions* in global assumption context.

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(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, `None` will be returned.

```
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

`ValueError` is raised if assumptions are inconsistent.

```
>>> ask(Q.integer(x), Q.even(x) & Q.odd(x))
Traceback (most recent call last):
  ...
ValueError: inconsistent assumptions Q.even(x) & Q.odd(x)
```

Notes

Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

sympy.assumptions.refine.refine **(page 197)**
Simplification using assumptions. Proposition is not reduced to `None` if the truth value cannot be determined.

`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 an object 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
```

Contents

Ask

Module for querying SymPy objects about assumptions.

class sympy.assumptions.ask.**AssumptionKeys**

This class contains all the supported keys by ask. It should be accessed via the instance sympy.Q.

sympy.assumptions.ask.**ask**(*proposition*, *assumptions=True*, *context={}*)

Function to evaluate the proposition with assumptions.

Parameters

proposition : Any boolean expression.

Proposition which will be evaluated to boolean value. If this is not AppliedPredicate, it will be wrapped by Q.is_true.

assumptions : Any boolean expression, optional.

Local assumptions to evaluate the *proposition*.

context : AssumptionsContext, optional.

Default assumptions to evaluate the *proposition*. By default, this is sympy.assumptions.global_assumptions variable.

Returns

True, False, or None

Raises

TypeError : *proposition* or *assumptions* is not valid logical expression.

ValueError : assumptions are inconsistent.

Explanation

This function evaluates the proposition to `True` or `False` if the truth value can be determined. If not, it returns `None`.

It should be discerned from *refine()* (page 197) which, when applied to a proposition, simplifies the argument to symbolic `Boolean` instead of Python built-in `True`, `False` or `None`.

Syntax

- **ask(proposition)**
  Evaluate the *proposition* in global assumption context.
- **ask(proposition, assumptions)**
  Evaluate the *proposition* with respect to *assumptions* in global assumption context.

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(4*x), Q.integer(x))
False
```

If the truth value cannot be determined, `None` will be returned.

```
>>> print(ask(Q.odd(3*x))) # cannot determine unless we know x
None
```

`ValueError` is raised if assumptions are inconsistent.

```
>>> ask(Q.integer(x), Q.even(x) & Q.odd(x))
Traceback (most recent call last):
  ...
ValueError: inconsistent assumptions Q.even(x) & Q.odd(x)
```

Notes

Relations in assumptions are not implemented (yet), so the following will not give a meaningful result.

```
>>> ask(Q.positive(x), x > 0)
```

It is however a work in progress.

See also:

sympy.assumptions.refine.refine **(page 197)**
Simplification using assumptions. Proposition is not reduced to `None` if the truth value cannot be determined.

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.

Deprecated since version 1.8.: Use multipledispatch handler instead. See *Predicate* (page 195).

sympy.assumptions.ask.**remove_handler**(*key*, *handler*)
Removes a handler from the ask system. Same syntax as register_handler

Deprecated since version 1.8.: Use multipledispatch handler instead. See *Predicate* (page 195).

Assume

A module which implements predicates and assumption context.

class sympy.assumptions.assume.**AppliedPredicate**(*predicate*, **args*)
The class of expressions resulting from applying `Predicate` to the arguments. `AppliedPredicate` merely wraps its argument and remain unevaluated. To evaluate it, use the `ask()` function.

Examples

```
>>> from sympy import Q, ask
>>> Q.integer(1)
Q.integer(1)
```

The `function` attribute returns the predicate, and the `arguments` attribute returns the tuple of arguments.

```
>>> type(Q.integer(1))
<class 'sympy.assumptions.assume.AppliedPredicate'>
>>> Q.integer(1).function
Q.integer
>>> Q.integer(1).arguments
(1,)
```

Applied predicates can be evaluated to a boolean value with `ask`:

```
>>> ask(Q.integer(1))
True
```

property 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
```

property arguments

Return the arguments which are applied to the predicate.

property function

Return the predicate.

class sympy.assumptions.assume.**AssumptionsContext**

Set containing default assumptions which are applied to the `ask()` function.

Explanation

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

The default assumption context is `global_assumptions`, which is initially empty:

```
>>> from sympy import ask, Q
>>> from sympy.assumptions import global_assumptions
>>> global_assumptions
AssumptionsContext()
```

You can add default assumptions:

```
>>> from sympy.abc import x
>>> global_assumptions.add(Q.real(x))
>>> global_assumptions
AssumptionsContext({Q.real(x)})
>>> ask(Q.real(x))
True
```

And remove them:

```
>>> global_assumptions.remove(Q.real(x))
>>> print(ask(Q.real(x)))
None
```

The `clear()` method removes every assumption:

```
>>> global_assumptions.add(Q.positive(x))
>>> global_assumptions
AssumptionsContext({Q.positive(x)})
```

(continues on next page)

(continued from previous page)

```
>>> global_assumptions.clear()
>>> global_assumptions
AssumptionsContext()
```

See also:

assuming (page 197)

add(**assumptions*)

Add assumptions.

class sympy.assumptions.assume.**Predicate**(**args*, ***kwargs*)

Base class for mathematical predicates. It also serves as a constructor for undefined predicate objects.

Explanation

Predicate is a function that returns a boolean value [1].

Predicate function is object, and it is instance of predicate class. When a predicate is applied to arguments, `AppliedPredicate` instance is returned. This merely wraps the argument and remain unevaluated. To obtain the truth value of applied predicate, use the function `ask`.

Evaluation of predicate is done by multiple dispatching. You can register new handler to the predicate to support new types.

Every predicate in SymPy can be accessed via the property of `Q`. For example, `Q.even` returns the predicate which checks if the argument is even number.

To define a predicate which can be evaluated, you must subclass this class, make an instance of it, and register it to `Q`. After then, dispatch the handler by argument types.

If you directly construct predicate using this class, you will get `UndefinedPredicate` which cannot be dispatched. This is useful when you are building boolean expressions which do not need to be evaluated.

Examples

Applying and evaluating to boolean value:

```
>>> from sympy import Q, ask
>>> ask(Q.prime(7))
True
```

You can define a new predicate by subclassing and dispatching. Here, we define a predicate for sexy primes [2] as an example.

```
>>> from sympy import Predicate, Integer
>>> class SexyPrimePredicate(Predicate):
...     name = "sexyprime"
>>> Q.sexyprime = SexyPrimePredicate()
>>> @Q.sexyprime.register(Integer, Integer)
... def _(int1, int2, assumptions):
```

(continues on next page)

(continued from previous page)

```
...     args = sorted([int1, int2])
...     if not all(ask(Q.prime(a), assumptions) for a in args):
...         return False
...     return args[1] - args[0] == 6
>>> ask(Q.sexyprime(5, 11))
True
```

Direct constructing returns `UndefinedPredicate`, which can be applied but cannot be dispatched.

```
>>> from sympy import Predicate, Integer
>>> Q.P = Predicate("P")
>>> type(Q.P)
<class 'sympy.assumptions.assume.UndefinedPredicate'>
>>> Q.P(1)
Q.P(1)
>>> Q.P.register(Integer)(lambda expr, assump: True)
Traceback (most recent call last):
  ...
TypeError: <class 'sympy.assumptions.assume.UndefinedPredicate'> cannot␣
↪be dispatched.
```

References

[R3], [R4]

eval(*args*, *assumptions=True*)

Evaluate `self(*args)` under the given assumptions.

This uses only direct resolution methods, not logical inference.

handler = <dispatched AskPredicateHandler>

classmethod register(**types*, ***kwargs*)

Register the signature to the handler.

classmethod register_many(**types*, ***kwargs*)

Register multiple signatures to same handler.

class sympy.assumptions.assume.**UndefinedPredicate**(*name*, *handlers=None*)

Predicate without handler.

Explanation

This predicate is generated by using `Predicate` directly for construction. It does not have a handler, and evaluating this with arguments is done by SAT solver.