Article from

Predictive Analytics and Futurism

June 2017
Issue 15

Using Python to Solve, Simplify, Differentiate and Integrate Mathematical Expressions

By Jeff Heaton

This article introduces SymPy[1], a computer algebra system (CAS) for the Python programming language. All software presented in this article is free and open source software (FOSS). When SymPy and Numpy (another FOSS package for Python) are combined with Python Jupyter notebooks, your computer becomes a sophisticated CAS. To make use of the examples presented in this article you should have Python 3.6 (or higher) installed. Additionally, the Python packages Numpy and SymPy should also be installed. Anaconda Python is the suggested Python platform for this article because of its inclusion of many packages needed for numerical computation.

At first glance, programming languages such as Python might seem very algebraic. Consider the following expression:

$$\frac{2x+3x}{2}$$

In Python, this would be written as:

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

The grouping parentheses are necessary in Python because the grouping implied by the algebraic ratio operator is not as obvious as when represented in source code. To Python (and most programming languages) this expression is simply a set of instructions that specify something to be done with x. The programming language is not concerned with simplifying the expression to $1.5x$ or other mathematical processes such as root finding, solving, differentiation or integration.

It is also important to note that because computer programs lack some of the grouping capabilities of written algebra it is always a good idea to use parentheses if you are unsure of how the programming language handles precedence. Though most programming languages follow the same rules of precedence as defined by algebra, there are exceptions. Excel is one such exception. The expression –2^2 evaluates to –4 in any programming language that I've worked with (except Microsoft Excel). The negative operator is evaluated after the power operator. However, Excel treats the negative in –2 not as an operator, but as an intrinsic part of the constant being squared. Thus, in Microsoft Excel, this expression evaluates to 4.

MATHEMATICAL NOTATION IN JUPYTER, WORD AND LaTEX

Mathematical formulas in Wikipedia are always expressed as LaTex. For example, the familiar quadratic equation in LaTex is written as follows:

```
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
```

In a Python Jupyter notebook, LaTex can be rendered by enclosing it in dollar signs ($):

```
$ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} $
```

Designate this as a markdown cell (via escape m), and Jupyter renders this equation as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

You can also right-click a Jupyter notebook LaTex rendering and export to MathML, which can be inserted into MS Word. Simply right-click an equation rendering in Jupyter and choose "Show Math As," and then "MathML Code." This will pop open a window showing an XML rendering of your equation. Copy this text into the clipboard and paste into Windows Notepad. Then recopy the text from Notepad and paste into Word. Unfortunately, the extra step of copying into Notepad overcomes some weaknesses in Words' import capabilities. I often find that I must copy/paste text through Notepad to simplify that text for Word consumption.

I've found this to be a very valuable feature of Jupyter notebooks. I often need to reference an equation from Wikipedia in Word. Trying to transcribe the equation from Wikipedia to Word's equation editor is a tedious and error-prone process. All the equations in this article were produced either by Jupyter or LaTex and imported into Word by the process just described.

LaTex is very common in the scientific community, as well as Wikipedia. Because of this, SymPy uses LaTex to display the mathematical expressions being processed.

The code presented in this article makes use of SymPy 1.0. An older version of SymPy might exist on your machine and prevent all the code in this article from working. To check the

version of SymPy installed on your machine, execute the following code from a Jupyter notebook.

```
import sympy
print(sympy.__version__)
```

This should respond with 1.0 (or later). If it does not, use the following command (from DOS/command line) to update SymPy:

```
pip install sympy --upgrade
```

ALGEBRAIC CAPABILITIES OF SYMPY

To begin using SymPy, open a Jupyter notebook and add the following lines of code as a cell:

```
from sympy import *
from IPython.display import display
from sympy.printing.mathml import mathml
from IPython.display import display, Math, Latex

x, y, z = symbols('x y z')
init_printing(use_unicode=True)
```

The **from** commands import the necessary libraries to make use of SymPy. The **symbols** definition lists the variables that will be used in algebraic expressions. For the examples provided in this article, the expressions will use the variables *x*, *y* and *z*. The **init_printing** command will allow mathematical expressions to be nicely formatted. To print mathematical equations we also define an **mprint** function, which is used to graphically render an expression:

```
def mprint(e):
  display(Math(latex(e)))
```

To demonstrate some of SymPy's capabilities, consider the following ratio of polynomials (note that ** means exponent in Python; 2**4 is 2 to the power of 4):

```
expr = (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
```

Usually a programming language would attempt to calculate the expression, using the current value of *x*. Python would normally assign this value to the variable expr. However, since we defined *x*, *y* and *z* as SymPy symbols, something different happens. We can ask Python what type of variable *expr* is with the following command:

```
print(type(expr))
```

Python tells us that this expression is of type Add, which just happens to be the root of the expression tree. However, the point is that Python did not attempt to calculate the expression. Rather, Python stored the expression itself. We can easily turn this expression into a displayable equation with the following command:

```
mprint(expr)
```

This results in the following expression being displayed:

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

This expression almost screams "simplify me," which we can easily accommodate with the following commands:

```
expr = simplify(expr)
mprint(expr)
```

This results in the following:

$$x - 1$$

Of course, this is true only if *x* is not equal to –1, or the original expression would result in a division by zero. SymPy does not check for such assumptions.

To evaluate the expression with a specific value of x, use the following code:

```
print(expr.subs(x,5))
```

This code substitutes 5 for x and results in 4.

SymPy can also solve equations. There is considerable documentation provided by SymPy to discuss equation solving. SymPy is able to solve systems of equations, differential equations, equations involving complex numbers and other options. For this section we will see how to solve a simple algebraic equation. The next section will discuss derivatives and integrals. For more details on equation solving for advanced situations, refer to Sympy's documentation on equation solving.[2]

An equation is an expression that is equal to something. In math, an expression that does not contain an equality sign is typically assumed to equal zero. In computer programming, an expression that does not contain an equality sign is assumed to evaluate to a numeric quantity that will be printed or assigned to another variable. In SymPy, equations are written using the function **Eq**. It is not possible to write the following in SymPy:

```
3*x +5 = 10
```

Though this equation is mathematically sound, it does not make sense in computer programming. In computer programming the above literally says "create an expression of 3*x+5 and assign

that expression to the constant value of 10." That is a type mismatch: an integer cannot be assigned into a expression. To create a true equation in SymPy, use the following:

```
eql = Eq(3*x+5,10)
```

This expresses the equality (and stores it in *eql*). Now that we have an equation, we can solve it:

```
z = solveset(eql,x)
display(Math(latex(z)))
```

This results in 5/3. Notice that Sympy keeps this value as a ratio, rather than creating a repeating decimal. By evaluating expressions algebraically, rather than converting everything to floating point numbers, equations can be calculated more precisely than most programming languages allow.

CALCULUS CAPABILITIES OF SYMPY

The following code demonstrates how to take the derivative of a simple formula. To test this functionality I used a question from my undergraduate calculus textbook. The derivative of sin(x) divided by x squared can be obtained by:

```
from sympy import *
x, y, z = symbols('x y z')
init_printing(use_unicode=True)
expr = diff(sin(x)/x**2, x)
mprint(expr)
```

This results in:

$$\frac{1}{x^2}\cos(x)-\frac{2}{x^3}\sin(x)$$

My textbook gave an equivalent answer, though it combined the difference into a single ratio. To test integration, we can calculate the antiderivative of the expression we just obtained:

```
expr_i = integrate(expr,x)
mprint(expr_i)
```

This takes us right back to where we started:

$$\frac{1}{x^2}\sin(x)$$

Definite integrals can be calculated as well.

OTHER APPLICATIONS

SimPy can be a very useful component of a data scientist's toolbox. At the most basic level SymPy can be used to transform a Jupyter notebook into an advanced CAS. More advanced uses allow Python code to be created to perform automated tasks that require differentiation and integration of arbitrary expressions.

I often make use of genetic programming, which can fit an actual expression to a set of training data. Genetic programming works very similarly to linear regression and neural networks, except the final model is a readable expression—the ultimate in transparency. However, genetic programs are often very unwieldy and can benefit greatly from algebraic simplification. Additionally, gradient descent can be used to optimize the coefficients of the genetic programs. By using SymPy to differentiate genetic programming–generated expressions, gradient descent can be used to optimize their coefficients.

A Jupyter notebook containing the source code presented in this article can be found at the author's github account.[3] ■

Jeff Heaton , Ph.D., is the author of the *AI for Humans* series of books and lead data scientist at Reinsurance Group of America (RGA) in Chesterfield, Mo. He can be reached at *jHeaton@rgare.com.*

ENDNOTES

1 SymPy can be obtained from *http://www.sympy.org/en/index.html*.

2 Solving SymPy equations: *http://docs.sympy.org/latest/tutorial/solvers.html*.

3 Source code can be found at *https://github.com/jeffheaton/present/blob/master/SOA/paf-sympy/sympy-soa.ipynb*.