6.009: Fundamentals of Programming

Tutorial 10: Symbolic Algebra Engine

With special guests:

- Inheritance
- Recursive-Descent Parsing

22 April 2020

Symbolic Algebra

Today we'll implement a **symbolic algebra engine**, designed to represent and manipulate symbolic expressions in Python.

Example: SymPy (`https://sympy.org`)

```
>>> from sympy import *
>>> x, y = var('x'), var('y')
>>> x*2 + x*y + 3*y
x*y + 2*x + 3*y
>>> (x*2 + x*y + 3*y).diff('x')
y + 2
>>> (x*2 + x*y + 3*y).evalf(subs={'x': 2, 'y': 7})
39.0000000000000
```

SymPy: ~800 contributors, 13 years

Symbolic Algebra

Today we'll implement a **symbolic algebra engine**, designed to represent and manipulate symbolic expressions in Python.

Example: SymPy (`https://sympy.org`)

```
>>> from sympy import *
>>> x, y = var('x'), var('y')
>>> x*2 + x*y + 3*y
x*y + 2*x + 3*y
>>> (x*2 + x*y + 3*y).diff('x')
y + 2
>>> (x*2 + x*y + 3*y).evalf(subs={'x': 2, 'y': 7})
39.0000000000000
```

SymPy: ~800 contributors, 13 years

We'll see what we can do in an hour or so :)

Symbolic Algebra

Depending on time, we'll see what we can get to, maybe including:

- representing variables and numbers
- arithmetic operations (+, -, *, /)
- symbolic differentiation
- simplification of expressions
- substituting numerical values
- parsing expressions

Atomic Expressions

We'll start small, with a way to represent variables and numbers (included in the `symbalg.py` file for this tutorial).

Note that `__repr__` provides a representation that would produce an equivalent Python object if evaluated, whereas `__str__` provides a more human-readable representation.

Operations

Next, we'll add ways to represent *operations*. These aren't going to look like much to start with, but we'll add some power as we go along.

Python Integration

As we saw last week, Python's "dunder" methods give us a nice way to integrate things very tightly with the language, including allowing us to use the built-in operators on expressions.

To make our little library more usable, we'd like the following kinds of operations to work, each just creating a new instance of one of our `BinOp` classes:

```
>>> Var('a') * Var('b')
Mul(Var('a'), Var('b'))
>>> 2 + Var('x')
Add(Num(2), Var('x'))
>>> Num(3) / 2
Div(Num(3), Num(2))
>>> Num(3) + 'x'
Add(Num(3), Var('x'))
```

Parenthesization

We have a problem, unfortunately. What happens, for example, if we do the following?

```
>>> z = 2 * (Var('x') + 3)
>>> print(z)
```

What happens? How does this compare against what we expect?

Parenthesization

We have a problem, unfortunately. What happens, for example, if we do the following?

```
>>> z = 2 * (Var('x') + 3)
>>> print(z)
```

What happens? How does this compare against what we expect?

Unfortunately, this shows something that is simply incorrect. We can fix this by implementing parenthesization inside of `__str__`, following the "PEMDAS" rule.

Parenthesization

In general, we can fix this issue by adjusting `BinOp`'s `__str__` method so that, for an instance `B`:

- If `B.left` and/or `B.right` themselves represent expressions with lower precedence than `B`, wrap their string representations in parentheses.
- As a special case, if `B` represents a subtraction or a division and `B.right` represents an expression with *the same precedence* as `B`, wrap `B.right`'s string representation in parentheses.

How can we best implement this in our Python file?

Derivatives

Next, we'll make our little system do our 18.01 homework for us by adding support for partial derivatives. In particular, we'll implement the following rules:

$$\frac{\partial}{\partial x} c = 0$$

$$\frac{\partial}{\partial x} x = 1$$

$$\frac{\partial}{\partial x}\left(u + v\right) = \frac{\partial}{\partial x} u + \frac{\partial}{\partial x} v$$

$$\frac{\partial}{\partial x}\left(u \cdot v\right) = u\left(\frac{\partial}{\partial x} v\right) + v\left(\frac{\partial}{\partial x} u\right)$$

$$\frac{\partial}{\partial x}\left(\frac{u}{v}\right) = \frac{v\left(\frac{\partial}{\partial x} u\right) - u\left(\frac{\partial}{\partial x} v\right)}{v^2}$$

Simplification

Consider the following example using our `deriv` code:

```
>>> x = Var('x')
>>> y = Var('y')
>>> z = 2*x - x*y + 3*y
>>> print(z.deriv('x'))  # (2 - y)
2 * 1 + x * 0 - (x * 0 + y * 1) + 3 * 0 + y * 0
>>> print(z.deriv('y'))  # (-x + 3)
2 * 0 + x * 0 - (x * 1 + y * 0) + 3 * 1 + y * 0
```

While this works, it is not very readable. It would be *way* nicer if the above printed `2 - y` and `-x + 3`, respectively.

Simplification

We can implement a rudimentary simplification algorithm by implementing a few rules:

- Any binary operation on two numbers should simplify to a single number containing the result.
- Adding 0 to (or subtracting 0 from) any expression E should simplify to E.
- Multiplying or dividing any expression E by 1 should simplify to E.
- Multiplying any expression E by 0 should simplify to 0.
- Dividing 0 by any expression E should simplify to 0.
- A single number or variable always simplifies to itself.

Evaluation

One common use of a symbolic calculator is to compute the value of an expression with various values substituted in for the variables in the expression, for example:

```
>>> z = Add(Var('x'), Sub(Var('y'), Mul(Var('z'), Num(2))))
>>> z.eval({'x': 7, 'y': 3, 'z': 9})
-8
>>> z.eval({'x': 3, 'y': 10, 'z': 2})
9
```

Parsing Symbolic Expressions

Finally, we would like to be able to accept symbolic expressions in a more human-readable format (as a single string containing a complicated expression):

```
>>> sym('(x * (2 + 3))')
Mul(Var('x'), Add(Num(2), Num(3)))
```

We will set things up so that `sym` can handle three kinds of strings:

- a single variable name,
- a single integer, or
- a fully-parenthesized expression of the form `(E1 op E2)`, representing a binary operation (where `E1` and `E2` are themselves strings representing expressions, and `op` is one of `+`, `-`, `*`, or `/`)

To do this, we will need to look at this string to figure out the structure it represents, and to create an equivalent symbolic expression. We'll break this down into two steps: **tokenizing** and **parsing**.

Tokenizing

We'll start by defining a helper function `tokenize`, which will take a string as input, and which will output a list of meaningful "tokens" (parentheses, variable names, numbers, and operators). For example:

```
>>> tokenize("(foo * (20 + 3))")
['(', 'foo', '*', '(', '20', '+', '3', ')', ')']
```

Tokens can be multiple characters long, but they are always separated by spaces or by parentheses.

Parsing

We will also implement a separate helper function called `parse`, which will take as input a list of tokens (i.e., the output from `tokenize`) and return an equivalent symbolic expression, for example:

```
>>> tokens = tokenize("(x * (2 + 3))")
>>> parse(tokens)
Mul(Var('x'), Add(Num(2), Num(3)))
```

We'll implement this using a technique called *recursive-descent parsing*.

A Recursive Descent Parser

One way that we can structure our `parse` function is as follows:

```
def parse(tokens):
   def parse_expression(index):
       pass  # your code here
   parsed_expression, next_index = parse_expression(0)
   return parsed_expression
```

Note that the helper function `parse_expression` is a recursive function that takes as argument an integer into the tokens list and returns a pair of values:

- the expression found starting at the location given by index, and
- the index beyond where this expression ends

How can we adapt this style to parse the kinds of expressions we'll be looking at?

Woo-hoo!

We're "done" in the sense that that's where we'll leave things for this tutorial. However, if you're interested, there are a number of ways to improve on the system we've built here, for example:

- implement any pieces that we didn't get to today
- introduce additional kinds of simplifications
- support more operations, trig functions, etc
- ...