Lab 7: Symbolic Algebra

Table of Contents

1) Preparation

You may wish to read the whole lab before implementing any code, so that you can make a more complete plan before diving in. For some parts of the lab, it might be better to read all the subsections first as some give useful hints or information regarding the implementation.

> Try making a plan for your code, including the following:
>
> - Which classes are you going to implement? Which classes are subclasses of which classes?
> - What attributes are stored in each class?
> - What methods does each class have?
>
> As you work through, be on the lookout for opportunities to take advantage of class inheritance to avoid repetitious code!
>
> Throughout the lab, you should not use Python's `eval`, `exec`, `isinstance`, or `type` built-ins (or their equivalents), except where we have indicated that it is okay to do so. We enforce this constraint in order to make sure that your solution takes full advantage of object oriented programming.

This lab assumes you have Python 3.7 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab7.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- passing the test cases from `test.py` under the time limit (1.5 points), and
- a brief "checkoff" conversation with a staff member to share your code, including discussion of the bullets below and a review of your code's clarity/style (2.5 points).

Please also review the collaboration policy before continuing.

This lab does not have any concept questions, but you are strongly encouraged to read the specification, and perhaps to get started with trying to implement some of the code, before lecture on 28 Oct. The code submission is due on Friday, 1 Nov.

2) Introduction

In this lab, we will develop a Python framework for symbolic algebra. In such a system, algebraic expressions including variables and numbers are not immediately evaluated, but rather are stored in symbolic form.

We'll start by implementing support for basic arithmetic (`+`, `-`, `*`, and `/`) on variables and integers, and then we'll add support for simplification, differentiation and evaluation of these symbolic expressions. Ultimately, this system will be able to support interactions such as the following:

```
>>> x = Var('x')
>>> y = Var('y')
>>> print(x + y)
x + y
>>> z = x + 2*x*y + x
>>> print(z)
x + 2 * x * y + x
>>> print(z.deriv('x'))
1 + 2 * x * 0 + y * (2 * 1 + x * 0) + 1
>>> print(z.deriv('x').simplify())
1 + y * 2 + 1
>>> print(z.deriv('x'))
1 + 2 * x * 0 + y * (2 * 1 + x * 0) + 1
>>> print(z.deriv('y'))
0 + 2 * x * 1 + y * (2 * 0 + x * 0) + 0
>>> print(z.deriv('y').simplify())
2 * x
>>> z.eval({'x': 3, 'y': 7})
48
```

3) Basic Symbols

In this week's code distribution, we have provided a very minimal skeleton, containing a small definition for three classes:

- `Symbol` will be our base class; all other classes we create will inherit from this class, and any behavior that is common between all expressions (that is, all behavior that is not unique to a particular *kind* of symbolic expression) should be implemented here.
- Instances of `Var` represent variables (such as x or y).
- Instances of `Num` represent numbers within symbolic expressions).

Take a look at the `Var` and `Num` classes. Note that each has `__init__`, `__repr__`, and `__str__` defined for you already.

4) Binary Operations

So far, our system is somewhat uninteresting. It lets us represent variables and numbers, but in order to be able to represent meaningful expressions, we also need ways of combining these primitive symbols together. In particular, we will represent these kinds of combinations as *binary operations*. You should implement a class called `BinOp` to represent a binary operation. Because it is a type of symbolic expression, `BinOp` should be a subclass of `Symbol`.

We will implement four subclasses of `BinOp`:

- `Add`, to represent an addition
- `Sub`, to represent a subtraction
- `Mul`, to represent a multiplication
- `Div`, to represent a division

By virtue of being binary operations, each instance of any of these classes should have two instance variables:

- `left`: a `Symbol` instance representing the left-hand operand
- `right`: a `Symbol` instance representing the right-hand operand

For example, `Add(Add(Var('x'), Num(3)), Num(2))` represents the symbolic expression $x + 3 + 2$. The `left` attribute of this instance should be the `Add` instance variable `Add(Var('x'), Num(3))`, and the `right` attribute should be the `Num` instance `Num(2)`.[1]

> **Check Yourself:**
>
> The structure of each of these classes' `__init__` methods is likely to be almost the same (if not exactly the same). What does that suggest about how/where you should implement `__init__`?

Note that, throughout the lab, you are allowed to add new parameters to the constructor of `BinOp` in order to store extra information you might find useful. Also keep in mind that, while the test cases don't attempt to call the BinOp constructor directly, they do call the `Add`, `Sub`, `Mul`, and `Div` classes. Those constructors should also accept integers or strings as their arguments. `Add(2, 'x')`, for example, should create an instance `Add(Num(2), Var('x'))`. **It is okay to use `isinstance` or `type` in this context, to check if the arguments passed to the constructor are strings or integers.**

Other than the above use case, at this point you may not use other direct ways of checking the types of objects, either through the builtins `isinstance` or `type` or through using explicit information about the kind of each class. For example, while you are allowed to store in the class constructors properties that uniquely identify the class type , you may not conditionally apply different algorithms based on the specific value of these attributes.

If at any point throughout the lab you are unsure whether your code is violating these constraints, ask us, as points will be deducted during check-off for illegal type checks.

5) Display

As of right now, attempting to print an instance of `BinOp` (or a subclass thereof) will not really tell us much useful information. In this section, we'll improve on Python's ability to display these objects.

Python has two different ways to get a representation of an object as a string. First, `repr(obj)` (which calls `obj.__repr__()` under the hood) should produce a string containing a representation that, when passed back into Python, would evaluate to an

equivalent object. Second, `str(obj)` (which calls `obj.__str__()` under the hood) should produce a human-readable string representation.

> **Check Yourself:**
>
> Take a look at the `__repr__` and `__str__` methods in `Var` and `Num`. What is the difference between them?

Implement `__repr__` and `__str__` in appropriate places (avoiding repeating code where possible), such that, for any symbolic expression `sym`, `repr(sym)` will produce a string containing a Python expression that, when evaluated, will result in a symbolic expression equivalent to `sym`. Similarly, `str(sym)` should produce a human-readable representation, given by `left_string + ' ' + operand + ' ' + right_string`, where `left_string` and `right_string` are the string representations of the `left` and `right` attributes, respectively, and `operand` is a string representation of the operand associated with the specific class in question. For example:

```
>>> z = Add(Var('x'), Sub(Var('y'), Num(2)))
>>> repr(z)  # notice that this result, if fed back into Python, produces an equivalent object.
"Add(Var('x'), Sub(Var('y'), Num(2)))"
>>> str(z)  # this result cannot necessarily be fed back into Python, but it looks nicer.
'x + y - 2'
```

As always, try to avoid repetitious code when implementing! You may wish to store additional class and/or instance variables in the subclasses of `BinOp` in order to accomplish this. For example, we need a way to look up a string representing the associated operator for each of the supported operations.

6) Parenthesization

Note that, while a `__repr__` implementation that follows the rules described above works well for complicated expressions (as seen in the expressions in the last example above), a `__str__` implementation that follows those rules results in some possible ambiguities or erroneous interpretations. In particular, consider the expression `Mul(Var('x'), Add(Var('y'), Var('z')))`. According to the rules above for `__str__`, that expression's string representation would be `"x * y + z"`, but the internal structure of the expression suggests something different! It would be nice for the string representation instead to be `"x * (y + z)"`, to better align with the actual structure represented by the expression.

To address this, we add rules for parenthesization as follows, where `B` is the `BinOp` instance whose string representation we are finding[2]:

- If `B.left` and/or `B.right` themselves represent expressions with lower precedence than `B`, wrap their string representations in parentheses (here, precedence is defined using the standard "PEMDAS" ordering).
- 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. To understand why this is the case, think of how the following example expression should be evaluated: `3 - 1 - 1`. Do we group the first two or the last two terms?

Individual numbers or variables should never be wrapped in parentheses.

Check Yourself:

Think about the rules for parenthesization described above in terms of algebraic expressions, and work through parenthesizing some example expressions by hand to get a feel for how these rules work. Do these rules seem to work in a general sense -- will they always work across different operations, and across different levels of expression complexity? Why do they work? Why are subtraction and division treated differently from addition and multiplication?

Note that information about concrete syntax (including parenthesization) should not be explicitly stored in any of your instances; grouping should be reflected in the structure of your objects and the connections between them.

By this point in the lab, you can expect to pass all the test cases in `Test_0_Display`.

7) Python Integration

Entering expressions of the form `Add(Var('x'), Add(Num(3), Num(2)))` can get a little bit tedious. It would be much nicer, for example, to be able to enter that expression as `Var('x') + Num(3) + Num(2)`.

Add methods to appropriate class(es) in your file such that, for any arbitrary symbolic expressions `E1` and `E2`:

- `E1 + E2` results in an instance `Add(E1, E2)`
  (note: you can override the behavior of `+` with the `__add__` and `__radd__` "dunder" methods)
- `E1 - E2` results in an instance `Sub(E1, E2)`
  (note: you can override the behavior of `-` with the `__sub__` and `__rsub__` "dunder" methods)
- `E1 * E2` results in an instance `Mul(E1, E2)`
  (note: you can override the behavior of `*` with the `__mul__` and `__rmul__` "dunder" methods)
- `E1 / E2` results in an instance `Div(E1, E2)`
  (note: you can override the behavior of `/` with the `__truediv__` and `__rtruediv__` "dunder" methods)

Check Yourself:

Try to avoid duplicating code when implementing these behaviors! In what class(es) should you implement `__add__`?

These behaviors should work so long as *at least one* of `E1` and `E2` is a symbolic expression, and the other is either a symbolic expression, an integer, or a string.

See this section of the python "Documentation" for details on why reverse dunders like `__radd__`, etc. are needed.

Check Yourself:

Which of the dunder methods, `__sub__` or `__rsub__`, will be called when evaluating the following expression: Var("x") - 3? What about 3 - Var("x") ?

For example:

```
>>> 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'))
```

After implementing support for arithmetic operations, your code should pass the `Test_1_Combine` test class.

8) Derivatives

Next, we'll make the computer do our 18.01 homework for us. Well, not quite. But we'll implement support for symbolic differentiation. In particular, we would like to implement the following rules for partial derivatives (where x is an arbitrary variable, c is a constant or a variable other than x, and u and v are arbitrary expressions):

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

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

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

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

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

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

Even though it may not be obvious from looking at first glance, these mathematical definitions are recursive! That is to say, partial derivatives of compound expressions are defined in terms of partial derivatives of component parts.

Implement differentiation by adding a method called `deriv` to your classes. This method should take a single argument (a string containing the name of the variable with respect to which we are differentiating), and it should return a symbolic expression representing the result of the differentiation. For example:

```
>>> x = Var('x')
>>> y = Var('y')
>>> z = 2*x - x*y + 3*y
>>> print(z.deriv('x'))  # unsimplified, but the following gives us (2 - y)
2 * 1 + x * 0 - (x * 0 + y * 1) + 3 * 0 + y * 0
>>> print(z.deriv('y'))  # unsimplified, but the following gives us (-x + 3)
2 * 0 + x * 0 - (x * 1 + y * 0) + 3 * 1 + y * 0
```

The tests in `Test_2_Derivative` check the correctness of this section, so after implementing support for taking derivatives all the tests in this suite should now pass.

9) Simplification

The above code works, but it leads to output that is not very readable (it is very difficult to see, for example, that the above examples correspond to $2 - y$ and $-x + 3$, respectively). To help with this, implement a method called `simplify` to your class(es). `simplify` should take no arguments, and it should return a simplified form of the expression, according to the following 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.

Note that there are other simplifications we might apply on an expression besides these. You are encouraged to implement those in your free time, but you should not include them in your solution to this lab. Adding additional simplifications might cause some test cases to fail.

> **Check Yourself:**
>
> Think about the simplification rules described above in terms of algebraic expressions, and work through simplifying some example expressions by hand to get a feel for how these rules work. Do these rules seem to work in a general sense -- will they always work across our different operations, and across different levels of expression complexity, producing sensible simplifications? Why?

Within `simplify`, you are welcome to use `isinstance` or `type` to check the type of subexpressions.

Your simplification method does not need to handle cases like $3 + (x + 2)$, where the terms that could be combined are separated from each other in the tree.

For example, continuing from above:

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

After completing this section, make sure that you pass all test cases up to (and including) the `Test_3_Simplify` suite.

10) Evaluation

Next, we'll add support for evaluating expressions for particular values of variables. Add method(s) to your class(es) such that, for any symbolic expression `sym`, `sym.eval(mapping)` will find a numerical value for the given expression. `mapping` should be a dictionary mapping variable names to values. 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
```

In the test cases we'll run, the given dictionary will always contain all of the bindings needed to fully evaluate the expression. What should your code do in the case where the expression contains a variable not present in the given `mapping` dictionary?

After implementing evaluation, run the tests in `Test_4_Eval` to make sure they pass.

11) Parsing Symbolic Expressions

We would now like to support parsing strings into symbolic expressions (to provide yet another means of input). For example, we would like to do something like:

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

Define a stand-alone function called `sym`, which takes a single string as input. This string should contain either:

- a single variable name,
- a single number, 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 `/`).

You may assume that the string is always well-formed and fully parenthesized (you do not need to handle erroneous input), but it should work for arbitrarily deep nesting of expressions.

This process is often broken down into two pieces: *tokenizing* (to break the input string into meaningful units) and *parsing* (to build our internal representation from those units).

11.1) Tokenizing

A good helper function to write is `tokenize`, which should take a string as described above as input and should output a list of meaningful *tokens* (parentheses, variable names, numbers, or operands).

For our purposes, you may assume that variables are always single-character alphabetic characters, that all numbers are integers, and you may also assume that there are spaces separating operands and operators. Additionally, negative numbers won't contain any spaces between the negative sign and the first digit. For example, you won't encounter a number formatted as `- 4`, but you might see `-4`.

As an example:

```
>>> tokenize("(x * (2 + 3))")
['(', 'x', '*', '(', '2', '+', '3', ')', ')']
```

Note that your code should also be able to handle numbers with more than one digit, and negative numbers! A number like -200, for example, should be represented by a single token `'-200'`.

Tip: While there are multiple ways to go about identifying strings that represent digits in your tokenize function, you might find the `isdigit()` method on strings useful too.

11.2) Parsing

Another helper function, `parse`, could take the output of `tokenize` and convert it into an appropriate instance of `Symbol` (or some subclass thereof). For example:

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

Our "little language" for representing symbolic expressions can be parsed using a *recursive descent* parser. One way to structure `parse` is as follows:

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

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

- the expression found starting at the location given by `index` (an instance of one of the `Symbol` subclasses), and
- the index beyond where this expression ends (i.e., if the expression ends at the token with index `6` in the `tokens` list, then the returned value should be `7`.

In the definition of this procedure, we make sure that we call it with the value `index` corresponding to the start of an expression. So, we need to handle three cases. Let `token` be the token at location `index`; the cases are:

- **Number**: If `token` represents an integer, then make a corresponding `Num` instance and return that, paired with `index + 1` (since a number is represented by a single token).
- **Variable**: If `token` represents a variable name (a single alphabetic character), then make a corresponding `Var` instance and return that, paired with `index + 1` (since a variable is represented by a single token).
- **Operation**: Otherwise, the sequence of tokens starting at `index` must be of the form: `(E1 op E2)`. Therefore, `token` must be `(`. In this case, we need to recursively parse the two subexpressions, combine them into an appropriate instance of a subclass of `BinOp` (determined by `op`), and return that instance, along with the index of the token beyond the final right parenthesis.

Implement the `sym` function in your code (possibly using the helper functions described above). **Your implementation of the function `sym` should not use Python's built-in `eval`, `exec`, `type`, or `isinstance` functions.**

You can find test cases for parsing in the `Test_5_Parse` class. At this point, your code should pass all tests in the `test.py` file.

12) Testing Your Code: REPL

Sometimes we are interested in doing some manual testing on our parser. This means running the parser on a lot of example strings that are small enough so that we can confirm their correctness just by looking at the output. (Probably this is what you have been using the space under `if __name__ == "__main__"` for.)

This means having to print the result of a `sym()` call as a line of Python code for each such test case.

As a last part of this lab, we'll make manual testing of the parser slightly easier by writing a simple REPL! (a "**R**ead, **E**valuate, **P**rint **L**oop") for arithmetic expressions. Your REPL should continually prompt the user for input until they type `QUIT`. Until then, it:

- accepts input from the user,
- breaks down the input string into tokens,
- parses the tokens into a `BinOp`,
- prints the `repr()` of the obtained result.

If an error occurs during any of these steps (maybe you typed an invalid input to the console, in which case the `sym` function may or may not throw an exception), an error message of your choice should be displayed. The invalid expression may be ignored, but the program should not exit.

To implement the REPL, we can make use of Python's built-in `input` function. `input` takes an argument representing a prompt to be displayed to the user and returns the string that they type (it is returned when they hit enter).

The following example shows how your REPL is expected to work:

```
in> ((x + 3) * 5)
  out> Mul(Add(Var('x'), Num(3)), Num(5))

in> (9 - -2)
  out> Sub(Num(9), Num(-2))

in> QUIT
```

Note that your REPL only needs to handle one-line inputs. You do not need to handle the case of a multi-line arithmetic expression being entered through the REPL.

The functionality of your REPL will not be tested automatically, but **we will ask you to demonstrate how your parser works by running the REPL during the checkoff**. The REPL should only start when the lab is run directly, not when `lab.py` is imported from another script.

13) Code Submission

Select File No file selected

Submit

You have submitted this assignment 0 times.

This question is due on Friday November 01, 2019 at 04:00:00 PM.

14) Checkoff

Once you are finished with the code, please come to a lab session or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular, be prepared to discuss:

- Your implementation of parenthesization, including using inheritance to avoid repetitious code.
- Your implementation of `deriv`, including using inheritance to avoid repetitious code.
- Your implementation of `simplify`, including using inheritance to avoid repetitious code.
- Your implementation of `eval`, including using inheritance to avoid repetitious code.
- Your implementation of `sym`.
- A demonstration of using the REPL to simulate some symbolic algebra problems of your own choosing.
- A brief explanation of the rules for parenthesization and simplification, and why those rules work.

14.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

Footnotes

[1] Notice how these 4 classes can be defined in terms of two other Symbols, which themselves might be another one of `Add`, `Sub`, `Mul`, and `Div`, or a `Var`/`Num`. We can therefore think of the `Var` and `Num` as the units/building blocks/base cases of arithmetic expressions, while the other operation classes can be viewed as recursive cases. Remember the lecture on Recursive Data Structures!

[2] Note that these rules should be sufficient for the four operations we have implemented. If we wanted to add additional operations, we may need to reconsider these rules