Symbolic Algebra

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

Note

You may wish to read the whole lab before implementing any code, so that you can make a more complete plan before diving in. 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 inheritance to avoid repetitious code!

Throughout the lab, you should not use Python's `eval`, `exec`, `isinstance`, or `type` built-ins, nor should you be explicitly checking for the type of any particular object, except where we have indicated that it is okay to do so. Instead of explicitly checking the types of our objects and

> determining our behavior based on those results, our goal is to use Python's own mechanism of method/attribute lookup to implement these different behaviors without the need to do any type checking of our own.
>
> Things that are considered "explicit type checking" include not only checks using `type` or `isinstance`, but also using unique class attributes as a way to ask the same question one would ask of `isinstance` or `type` (what class is this object an instance of?).

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

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

Note that passing all of the tests on the server will require that your code runs reasonably efficiently.

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- passing the style check (1 point)
- passing the tests in `test.py` (4 points)

> **Reminder: Academic Integrity**
>
> Please also review the academic-integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

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.

These kinds of systems abound in the "real world," and they can be incredibly useful. Examples of similar systems include Wolfram Alpha and Sympy (a really impressive open-source Python module for symbolic algebra). We won't quite approach the sophistication of those kinds of packages in this lab, but we'll get a pretty good start in that direction!

We'll start by implementing support for basic arithmetic (`+`, `-`, `*`, and `/`) on variables and numbers, and then we'll add support for simplification and differentiation 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'))  # derivative of z with respect to x
1 + 2 * x * 0 + y * (2 * 1 + x * 0) + 1
>>> print(z.deriv('x').simplify())
1 + y * 2 + 1
>>> print(z.deriv('y'))  # derivative of z with respect to y
0 + 2 * x * 1 + y * (2 * 0 + x * 0) + 0
>>> print(z.deriv('y').simplify())
2 * x
>>> z.eval({'x': 3, 'y': 7})  # evaluate an expression with particular
values for the variables
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. Importantly, while you are welcome to add additional machinery to these objects if you want to, it will be important that the existing instance variables continue to work as they do in the provided skeletons (our test suite relies on this behavior).

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 start with 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 `Add(Var('x'), Num(3))`, and the `right` attribute should be the `Num` instance `Num(2)`.

Importantly, instances of `BinOp` (or subclasses thereof) **should *only* have these two instance attributes**. You are welcome to store additional information in class attributes, but each instance should only store `left` and `right`.

> **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__`?

These constructors should also accept integers, floats, 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 numbers.**

Note: it is good style to include docstrings for classes. According to PEP257 style guidelines, "The docstring for a class should summarize its behavior and list the public methods and instance variables." For our purposes, a one-line sentence summary should suffice.

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 + ' ' + operator + ' ' + right_string`, where `left_string` and `right_string` are the string representations of the `left` and `right` attributes, respectively, and `operator` is a string representation of the operator 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'
```

Note: The test cases are checking that your code is avoiding unnecessary repetition. You will need to create an additional class attribute in the subclasses of `BinOp` that stores the operation associated with the class ("+", "-", etc.). **This class attribute string should not contain spaces!**

Note: you may access a string containing the class name using the following syntax:

```
>>> z = Num(2)
>>> z.__class__.__name__
Num
```

5.1) 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[1]:

- 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.

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?

> **Check Yourself:**
>
> Importantly, you should implement the behavior for **str** and **repr** without explicitly checking the type of `self`, `self.left`, or `self.right`.
>
> In order to pass the test cases, your code will need to do this by storing a couple of additional class attributes:
>
> - All symbols should have a class attribute called `precedence`, which should be a number representing precedence. Greater numbers should represent greater precedence. What classes should have the highest precedence? What classes should have the same precedence?
> - All binary operations should have a class attribute called `wrap_right_at_same_precedence`, which should be a Boolean that indicates whether to add parentheses around the right side of the expression in the special case described above.

6) Using Python Operators with Symbolic Expressions

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:**
>
> To pass the test cases you will need 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, a number, or a string.

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'))
```

> **Note**
>
> We have seen `__add__`, `__sub__`, `__mul__`, and `__truediv__` before, but what about the versions with the `r` on the front of them? They represent the same operations but with the operands swapped.
>
> It turns out that Python has a small set of rules for deciding which of these methods to call in various situations. For example, consider the small expression `x + y`. In order to decide what to do here, Python will:
>
> 1. If `x` is not a built-in type, and it has an `__add__` method, invoke `x.__add__(y)`.
> 2. Otherwise, if `x` and `y` are of different types, and if `y` has an `__radd__` method, invoke `y.__radd__(x)`.

> In the context of our program, implementing both `__add__` and `__radd__` will allow both of the following expressions to work:
>
> ```
> >>> x = Var('x')
> >>> x + 2  # here, __add__ will be called
> Add(Var('x'), Num(2))
> >>> 2 + x  # here, __radd__ will be called
> Add(Num(2), Var('x'))
> ```
>
> Feel free to ask us if you have questions about this! The page of the Python documentation about the Python Data Model has more information about these methods as well.

Note that you should be able to implement this behavior without additional explicit type checking.

7) 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 (meaning a `float` or an `int`, not an instance of `Num`) 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 not always contain all of the bindings needed to evaluate the expression fully. The test cases will expect you to raise a `NameError`, rather than just letting Python's normal exceptions happen, if a provided dictionary does not contain all the necessary bindings.

Hint: in order to avoid repetition, you might consider defining helper methods in your `BinOp` subclasses that performs the desired Python operation on the left and right sub-expressions.

Note that representing mathematically undefined expressions like `0 / 0` or `x / 0` using our `Symbol` classes should not cause errors. The tests do not attempt to evaluate undefined expressions, but you may add code to handle these cases if you want.

8) Equality

Next, we'll add support for checking if two expressions are equal.

Currently, if you try running the following example at the bottom of your `lab.py` file you should see the commented Boolean values output.

```
a = Num(4)
b = Num(4)
print(a == b)         # False
print(a == Num(4.0)) # False
print(a == a)         # True
print(4 == 4.0)       # True
```

Even though Python can normally compare floats and ints for equality, currently instances of our `Symbol` class ignore the inner attributes of `a` and `b`. Instead, by default classes in Python check for equality by determining whether the two objects share the same id using `is`. The `is` keyword essentially tests if two references are pointing to the same object in memory (which is why `a == a` evaluates to True!)

In order to specify what we want equality to mean for `Num` (and other Symbol objects), we need to implement the `__eq__` dunder method to overide Python's default behavior. For the purposes of this lab, we will ignore simplifying the expression and the associative property and determine equality by testing whether the expression contains equivalent objects in the same order.

For example:

```
>>> Sub(Num(4), Var('y')) == Sub(Num(4), Var('y'))
True
>>> Add(Num(4), Num(1)) == Num(5)
False
>>> Mul(Num(4), Var('y')) == Mul(Var('y'), Num(4))
False
>>> Div(Num(1), Add(Num(4.0), Var('z'))) == Div(Num(1.0), Add(Num(4),
Var('z')))
True
>>> Num(4) != Var('z')
True # implementing equality correctly also handles inequality checks as
well!
```

In order to pass the test cases, your code will need to both correctly check for equality (and inequality!) as well as avoid unnecessary repetition / implementations of the `__eq__` method. You may however, use `type` or `isinstance` as part of this method.

9) 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):

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

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

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

$$\frac{\partial}{\partial x}(u-v) = \frac{\partial}{\partial x}u - \frac{\partial}{\partial x}v$$

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

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

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, which may suggest strategies for implementing these rules in your program!

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
>>> w = Div(x, y)
>>> print(w.deriv('x'))
```

```
(y * 1 - x * 0) / (y * y)
>>> print(repr(w.deriv('x')))  # deriv always returns a new Symbol object
Div(Sub(Mul(Var('y'), Num(1)), Mul(Var('x'), Num(0))), Mul(Var('y'),
Var('y')))
```

Importantly, we would like computing any derivative to result in a symbolic expression. Make sure that your code has this property!

10) 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, add 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.

> **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?

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. In fact, our test cases are expecting **only** the above rules to be implemented. Implementing extra rules here can be a great opportunity for extra practice, but it will cause some test cases to fail (so maybe best to implement such things only after submitting!).

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')
```

Notes:

- Your `simplify` method should not mutate self.
- Simplification should always result in a symbolic expression. Make sure that your code has this property!
- You may use type checking to determine whether a symbol represents a `Num` as part of implementing this behavior.

11) Parsing Symbolic Expressions

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

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

Define a stand-alone function called `expression`, 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 and that all numbers are positive or negative integers or floats. You may also assume that there are spaces separating operands and operators.

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 -200.5, for example, should be represented by a single token `'-200.5'`.

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 an argument an integer indexing 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 (e.g., 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 or a float, 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 `expression` function in your code (possibly using the helper functions described above). **Your implementation of the function `expression` should not use Python's built-in `eval`, `exec`, `type`, or `isinstance` functions.**

However, you can *try* to cast a string to a float:

```
>>> float("1")
1.0
>>> float("-6.5")
-6.5
>>> float("x")
...
ValueError: could not convert string to float: 'x'
```

12) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a symbolic_algebra /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.

13) Additional (Optional) Extensions

We think the program we've built here is pretty impressive! But there are many opportunities to improve and/or extend its behavior to do even more impressive things. If you have the time and interest, there are a number of ways that you could expand on this framework. Note that some extensions may break the behavior the test cases are expecting, so it is a good idea to finish and submit the lab before working on the optional extensions. Some ideas are listed below:

- adding additional operations, including support for derivatives, simplification, etc.
- adding additional rules for simplification similar to those we've seen above (for example, for any expression u, u/u could simplify to 1)
- adding support for solving systems of linear equations
- modifying your code to be able to handle situations like $3 + (x + 2)$, where simplification is possible in theory, but where the methods described above won't fully simplify the expression
- adding more simplification rules relating addition to multiplication, or multiplication to exponentiation. For example, could we simplify $x + x + x$ to $3x$, or $(3y)(4y)$ to $12y^2$?
- adding representations for polynomial expressions, which may provide additional opportunities for simplification
- modifying your linear equation solver so that it outputs partial results even for systems that can't be solved (i.e., for systems that are underconstrained)

Feel free to implement your own ideas as well!

Footnotes

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