Coding Quiz

Estimated Completion Time: 1.5 hour

This quiz assesses the student's ability to program and solve problems in Python.

Instructions

- There are 3 Python files that you need to complete: `basics.py`, `autograd.py` and `interpreter.py`
- Do not import/use any libraries in your program (this includes the usage of the `eval` function)
- You can use the `test.py` script to test your programs
- Do not edit `test.py`

Basics

The `basics.py` section tests your familiarity with basic Python syntax and programming. It contains 3 sub-problems.

Greatest Common Factor (`greatest_common_factor`)

Write a function `greatest_common_factor` that returns the greatest common factor of the two given integers:

```
def greatest_common_factor(a, b):
    return
```

Sample output:

```
greatest_common_factor(30, 20) # 10
greatest_common_factor(45, 30) # 15
greatest_common_factor(159, 265) # 53
```

Reorder (`reorder`)

Write a function `reorder` that takes a list of numbers and a number, and moves all list elements that are equal to the given number to the end of the list (not in-place; return the moved list as a new list).

```
def reorder(num_list, num):
    return
```

Sample output:

```
reorder([3, 4, 7, 6, 9, 3, 5, 9, 3, 2], 3) # [4, 7, 6, 9, 5, 9, 2, 3, 3, 3]
reorder([-2, -1, 0, 3, -2, -1, 1], 0) # [-2, -1, 3, -2, -1, 1, 0]
reorder([1, 2, 3], 10) # [1, 2, 3]
```

Special Sum (`special_sum`)

Write a function `special_sum` that takes a list of numbers (positive integers), and return the sum of all even numbers minus the sum of all odd numbers.

```
def special_sum(num_list):
    return
```

Sample output:

```
special_sum([1, 2, 3]) # -2
special_sum([3, 2, 6, 9, 0]) # -4
special_sum([1, 2, 3, 4, 5, 6, 7, 8]) # 4
```

Autograd

The `autograd.py` section tests your understanding of mathematical concepts such as derivatives and chain rule, as well as basic programming abstractions such as object-oriented programming. This section has only one problem: you need to implement a very simple "autograd" system.

"Autograd" automatically builds a differentiation graph (or calculation graph) as the program carries out arithemetic operations. Consider this python expression:

```
y = (m * x) + b
```

Its computational graph looks like (`K` is an abstract term representing `m * x`):

```
        y
        |
        +
       / \
      K   b
     /
    *
   / \
  m   x
```

With this in mind, create a `Variable` and `Constant` class such that they support evaluation and partial differentiation as illustrated below:

```
>>> x = Variable(name='x')
>>> m = Variable(name='m')
>>> c = Constant(5)
>>>
>>> y = m * x * c
>>>
>>> y.evaluate(inputs={'m': 10, 'x': 15}) # calculate the value of y
750
>>>
>>> y.grad(respect_to='x', inputs={'m': 2, 'x': 5}) # calculate the derivative of y in respect to x
10
```

Here's another example:

```
>>> a = Variable(name='e')
>>> b = Variable(name='r')
>>> c = Constant(4)
>>> y = (a + b) ** c
>>>
>>> y.grad(respect_to='e', inputs={'e': 3, 'r': 2})
500
```

Chains should also be supported:

```
>>> x = Variable(name='x')
>>> a = Constant(3) * (x + Constant(5))
>>> b = Constant(8) * (x + Constant(20))
>>> c = a + b
>>>
>>> c.evaluate({'x': 30})
505
>>>
>>> c.grad('x', {'x': 10})
11
```

The following operations should be supported: - Addition(+) - Multiplication(*) - Power(**)

Notes

- The power term `c` will always be a constant in any `a ^ c` for simplicity
- If you are not familar with overriding the behavior of operators (`+`, `*`, etc) in Python, check out this tutorial

Interpreter

The `interpreter.py` section tests your ability to manipulate strings and spot recurring patterns. This section has only one problem: you need to write a simple interpreter that evaluates a piece of given text code (fill in the `interpreter` function).

The input will be a string containing the code. The return value is a list of the outputs of the program.

The custom syntax for the code looks like:

```
 a = 5
c = add a 7
output c
```

The above code has a single output value; therefore, its return value should be `[12]`. Lines of code are separated via line breaks `\n`:

```
interpreter('a = 5\nc = add a 7\noutput c') # [12]
```

Consider another example:

```
 a = 6
b = div 5 10
c = mul a b
output c
output 0.5
```

The above code has 2 return values, so the output list must contain both of them in the output order:

```
interpreter('a = 6\nb = div 5 10\nc = mul a b\noutput c\noutputb') # [3, 0.5]
```

Specification

The interpreter must handle 4 prefix arithmetic operators: `add` (adds the 2 parameters), `sub` (subtracts the first parameter by the second one), `mul` (multiplies the 2 parameters) and `div` (divides the first parameter by the second one). Each of them accepts 2 parameters, with each one being a direct number literal (e.g. `4`, `6.75`, `-18`) or a variable name (e.g. `foo`, `a`, `my_var`). Variable names cannot be a keyword (e.g. `add`, `sub`, `output`).

There will be no nested operators (e.g. no `add 5 mul 6 7`).

Assignment of a variable takes a number literal, a variable or an operator expression as the right hand side. Variables can also be reassigned:

```
 my_var = 5.6
a = my_var
b = add my_var 10
a = mul 10 b
```

The `output` function takes a number literal or a variable, and adds the value to the output list (as numbers, not strings!).

There might also be extra space (but there will always be at least one space between an arithmetic operator and its parameters to separate the tokens):

```
   my_var     =      5.6
a   =       my_var
   b     =      add      my_var     10
hello=123
```

All given code are valid; there will be no need for error detection.

If in doubt, take a look at the example test cases in `test.py`.

Good luck!