CS 172 – Lab 6: Stack Math

In this lab, we will experiment with the Stack data structure and see how some problems lend themselves well to this data structure.

The Stack data structure

As mentioned in class, Python 3 provides the `LifoQueue` class as part of the `Queue` library. To see how this is just a wrapper around the basic list class below is our own **`Stack`** class that uses a Python list.

This code has been provided to you below in a module called **stackclass.py**.

You **MAY NOT** change the structure of the provided `Stack` class in **ANY** way. Your post-fix logic should go in your main script. If you find yourself wanting to add more methods to the class, it means you are writing redundant code and should stop.

Milestone 1: The `Stack` class.

The `Stack` class has the following public interface:

- `__init__()`: creates an empty stack.
- `__str__()`: returns a string representation of the stack.
- `push(data)`: adds a new element (data) to the top of the stack.
- `pop()`: removes and returns the top element from the stack.
- `top()`: returns what on top of the stack but doesn't remove it.
- `isEmpty()`: returns `True` if the stack is empty and `False` otherwise.

Create a file named **main.py** and in the main script (under `if __name__ == "__main__":`) write code to confirm that all the provided methods in the `Stack` class work as described. In other words, create `Stack` object, populate it with a few values, print the Stack remove a couple of values from the Stack and print it again. Next check which element is at the top of the Stack and finally check if the Stack is empty.

This is a good point to submit your code and earn 10 points! You will get partial lab credit for completing this part, even if you do not complete the rest of the lab.

Postfix Math

You are probably most familiar with math written in **infix** notation. For example, **4 + 3 - 7**. In **infix** notation, operators are placed between their operands. This is a very nice way to read math, but it is nontrivial for a computer to parse.

An alternative representation is called **postfix** notation. In **postfix** we write the two operands followed by the operator. The previous example would be **4 3 + 7 -**. This is much easier to parse because we have both operands before the operator. We also don't have to worry about parenthesis.

Note: You may assume all symbols will have spaces between them. This makes it easy to **split** the expression. You will never be given **2 3+** instead of **2 3 +**. You may also assume that the only operators supported are **+, -, *, /**.

Examples:

Postfix	Infix	Result
2 1 +	2 + 1	3
5 4 -	5 - 4	1
7 2 *	7 * 2	14
9 3 /	9 / 3	3
2 5 + 7 *	(2 + 5) * 7	49

Postfix notation works well with a Stack. Here's how you can do it:

1. Push Numbers onto Stack as read from input.
2. When an operator is seen:

a. Pop top 2 items from the Stack. Since the second item popped off the top came before the top item in the postfix statement, it is the left operand.
b. Complete Operation.
c. Push result onto Stack.

3. Repeat until end of input.
4. Final result will be on top of Stack.

Milestone 2: Postfix Math - Subtraction

Within your **main.py** file, implement a function **`postfix(expression)`** that takes a `string` **`expression`** containing a postfix math expression as a parameter and returns the result as a floating-point number. For this milestone you will implement only subtraction. You **MUST** use the provided `Stack` class to implement this function.

Test that your **`postfix()`** function is working properly by adding the following code in the main script and verifying that the results are accurate:

```
print(postfix("4 5 -"))          # should print -1.0
print(postfix("4 9 7 - -"))      # should print 2.0
print(postfix("1 2 3 4 - - -")) # should print -2.0
print(postfix("1 2 - 3 - 4 -")) # should print -8.0
```

If you got your code to produce the correct results, then this is a good point to submit your code and earn 10 more points! You will get partial lab credit for completing this part, even if you do not complete the rest of the lab.

Milestone 3: Postfix Math – Addition, Multiplication, Division

Continue working on the **`postfix()`** function and add the addition, multiplication, and division operations. Test you code:

```
print(postfix("15 20 12 + +"))       # should print 47.0
print(postfix("11 2  3 * *"))        # should print 66.0
print(postfix("100 4 2 4 / / /"))    # should print 12.5
print(postfix("4 -1 9 5 2 3 + * - * /")) # should print 0.25
print(postfix("2 5 + 7 *"))             # should print 49.0
print(postfix("1 2 * 7 + 9 * 11 +"))    # should print 92.0
print(postfix("-1 2 * 7 + -9 * 11 +")) # should print -34.0
```

If you got your code to produce the correct results, then this is a good point to submit your code and earn 35 more points! You will get partial lab credit for completing this part, even if you do not complete the rest of the lab.

Milestone 4: The Main Application

In your **main.py** script (which should now also include your `postfix(expression)` function) you can now comment out all the tests you wrote before this point.

Add code to ask the user to enter a postfix expression and print the result. Repeat this process until the user enters "**exit**".

Here is an example, which should demonstrate the expected I/O:

```
Welcome to Postfix Calculator
Enter exit to quit
Enter Expression
1 2 +
Result: 3.0
Enter Expression
2 3 +
Result: 5.0
Enter Expression
4 5 +
Result: 9.0
Enter Expression
6 7 +
Result: 13.0
Enter Expression
10 12 +
Result: 22.0
Enter Expression
99 1 +
Result: 100.0
Enter Expression
-9 -8 +
Result: -17.0
Enter Expression
1 2 3 4 5 6 + + + + +
Result: 21.0
Enter Expression
3 4 *
```

```
Result: 12.0
Enter Expression
6 5 *
Result: 30.0
Enter Expression
9 7 *
Result: 63.0
Enter Expression
10 9 *
Result: 90.0
Enter Expression
100 -9 *
Result: -900.0
Enter Expression
90 3 *
Result: 270.0
Enter Expression
5 5 5 5 * * *
Result: 625.0
Enter Expression
4 3 -
Result: 1.0
Enter Expression
3 4 -
Result: -1.0
Enter Expression
9 8 -
Result: 1.0
Enter Expression
12 3 -
Result: 9.0
Enter Expression
3 12 -
Result: -9.0
Enter Expression
10 9 8 7 6 - - - -
Result: 8.0
Enter Expression
1 2 /
Result: 0.5
Enter Expression
4 2 /
```

```
Result: 2.0
Enter Expression
9 3 /
Result: 3.0
Enter Expression
25 5 /
Result: 5.0
Enter Expression
125 5 /
Result: 25.0
Enter Expression
1 7 /
Result: 0.14285714285714285
Enter Expression
100 2 2 2 / / /
Result: 50.0
Enter Expression
100 2 2 2 * * /
Result: 12.5
Enter Expression
4 -1 9 5 2 3 + * - * /
Result: 0.25
Enter Expression
100 2 / 2 / 2 /
Result: 12.5
Enter Expression
1 2 * 7 + 9 * 11 +
Result: 92.0
Enter Expression
exit
```

If you have this part working, you can now make your final submission. Do not forget to include a header comment and document your function with a docstring.

Scoring

The score for the assignment is determined as follows:

- 15 points: Correct use of the given Stack class (as given, with no modifications).
- 45 points - Correctly implemented `postfix(expression)`
    - 10 points per each correctly implemented operator: + - * /

    - 5 points for functions correctly evaluating expressions with mixed operators.
- 10pts: main script in main.py is implemented correctly.
- 15pts: Program runs without problems and performs postfix correctly.
- 15 points - Program style: header comment, function docstring, good variable names.

Note: Please make sure you put write both lab partners' names and userIDs (in the form abc123) in the header comment of the file you submit for this assignment.