COMSC-205PY FALL 2020 ASSIGNMENT 4

Due November 19th at 9:00am Eastern Time

Submission Checklist

Your final submission should be a zip file that includes the following:

- PostfixEvaluator.py
- Stack.py
- LinkedListNode.py
- UnorderedLinkedList.py

At the top of any file you edit, clearly state:

- Your name and the name of the assignment
- Attribution for any sources used, including people (other than instructors) who you asked for help
- Description of the file

You do not need to add documentation to any data structure files you got from the class materials unless you make changes in those files. All changes you make should be documented.

Read all the way through this assignment twice before you start working on it. The second time through, take notes, plan out how you will work on this.

Background

For this assignment, you will design and implement a program that evaluates postfix arithmetic expressions involving integers and **+, - *, / operations.**

Your program will get a file name from the command line (information below). That file will contain a series of postfix expressions, one complete expression per line of the file. For each expression, the program should evaluate the expression, display the original expression, and display the result of the specified computations.

What is postfix?

Typically, we use infix notation for arithmetic expressions, with a binary operator placed between the two operands (or before the single operand if it's a unary operator). For example, the following are all legal infix expressions

5 + 6

0 - 7

3 * 8

4 / 2

As you know from your experience learning arithmetic and learning to program, infix expressions are evaluated by using rules of precedence and associativity. For example

4 + 5 * 2 → 14

9 - 5 + 2 → 6

The first is an example where precedence rules are used; the multiplication is executed before the addition. The second is an example where associativity rules have to be used, since - and + have the same precedence. They are both left associative operators, so we evaluate the expression from left to right.

How is postfix different?

In postfix, the operator comes after its two operands. Below are a few expressions with infix on the left and the postfix equivalent on the right (extra spacing added for readability):

6 - 9 → 6 9 -

4 + 5 * 2 → 4 5 2 * +

4 * 5 - 2 → 4 5 * 2 -

Postfix expressions are simpler to evaluate than are infix expressions because there are no parentheses or precedence rules. Instead, expressions are evaluated in the order the operands and operators appear. It is for this reason that compilers and interpreters generally translate infix expressions into postfix, and then generate assembly code that carries out the postfix computation.

Evaluating postfix

The rule for evaluating postfix is quite simple: *Scan the expression from left to right. Each time you find an operator, apply it to the two immediately preceding operands, replace those two operands with the result, and remove the operator from the expression.* When there are no operators left, the remaining number is the final result. For example, consider

4 5 2 * +

Scanning the expression from left to right, there is an operand, an operand, an operand, and then the * operator. Applying the * to the 5 and the 2 gives 10, and the expression becomes

4 10 +

The next symbol is the + operator, which is applied to the preceding two operands, resulting in 14. Since there are no more operators in the expression, 14 is the answer.

Now let's consider a more difficult expression. It is presented below in a way that gives a hint about how to implement a postfix expression evaluator.

The current state of processing	**What has not been processed**
	3 4 * 2 5 + - 4 * 2 /
3	4 * 2 5 + - 4 * 2 /
3, 4	* 2 5 + - 4 * 2 /
12	2 5 + - 4 * 2 /
12, 2	5 + - 4 * 2 /
12, 2, 5	+ - 4 * 2 /
12, 7	- 4 * 2 /

5	4 * 2 /
5, 4	* 2 /
20	2 /
20, 2	/
10	

The Assignment

Your program will be run with a command line argument that provides a file name. Your program has to be able to get the file name from the command line, open the file, and read it. This code shows you how to do that.

```
import sys # library that coordinates with operating system

def main():
      filename = sys.argv[1] # get the input file name
      fi = open(filename, "r") # open file in read mode
      line = fi.readline() # read one line from the file
      while line != '': # EOF char is an empty string
            # do all the processing of the line that was read
            print(line[ :len(line)-1]) # print everything except \n
            # postfix evaluation goes here
            # get the next line
            line = fi.readline()

      fi.close() # close the file after it's all been read

main()
```

What's in the data file?

Python brings in a line from the file as a character string, so you can easily loop over the relevant characters of the string in order to process it. The data file will be a plain text file of postfix expressions. There will be one space between each character of the expression, and a '\n' after the last character. In other words, the expression

4 2 -

would be read in as a 6 character string. To divide this string into a list based on where the spaces are, look into the .split() method for python strings.

The data file will contain multiple lines, and each line will hold a legal postfix expression involving integers and arithmetic operators. Your program should print each expression followed by its value. For example, output would take the form

4 2 - results in 2

9 5 * results in 45

3 4 * 2 5 + - 4 * 2 / results in 10

Rubric

Each section will be given a letter grade. The average of those letters will generate your final score. Note: Moodle does not allow letters, so they will be translated into a percentage there.

	F	D	C	B	A
Postfix evaluation	No postfix present	Postfix does not evaluate correctly, but does something	Postfix evaluates at least one operation correctly, but not others	Postfix is correct, but does not use a data structure; or postfix is mostly correct, but has a bug or problem with one operation	Postfix is evaluated correctly with a data structure learned in class
File reading and printing	File is not read in	Nothing is printed	Only one expression is read in, or the answers are missing from the printing	One or more expressions are skipped, or the expressions are not printed	All expressions are read in from the file and printed out with correct answers.
Comments, naming, style	No comments present, no name in file, method names incorrect	Name in file, some methods named correctly, no comments	Name in file, description of file in comment, methods named correctly (typos ok), more than one error that prevents running	Name in file, description of file and most methods, methods named correctly, some variables named meaningfully, significant over- or under-use of comments or one error that prevents running	Name in file, description of file and all methods, methods named correctly, all variables named meaningfully, other comments as needed, no errors that prevent running