CMPSC-132: Programming and Computation II
Fall 2019

Homework 3

Due Date: 11/03/2019, 11:59PM
100 pts + Extra Credit

Read the instructions carefully before starting the assignment. Make sure your code follows the stated guidelines to ensure full credit for your work.

Instructions:

- This is a teamwork assignment. Each team should be working on their own. Only one student (the team leader) must upload the submission, however, you must add the second member of the team into the Gradescope submission. Watch how to submit group submissions on Gradescope here: https://youtu.be/rue7p_kATLA (failing to do this will result in your teammate receiving a 0 score in the assignment)
- **Download the starter code file from the HW3 Assignment on Canvas. Do not change the function names or given started code on your script.**
- You are responsible for debugging and testing your code with enough data, you can share testing code on Piazza.

Goal:

✓ Part 1: In the Module 5 video lectures, we discussed the abstract data structure Stack. A stack is a collection of items where items are added to and removed from the top (LIFO). Use the Node class (an object with a data field and a pointer to the next element) to implement the stack data structure with the following operations:

- push(item) adds a new Node with value=item to the top of the stack. It needs the item and returns nothing.
- pop() removes the top Node from the stack. It needs no parameters and returns the **value** of the Node removed from the stack. Modifies the stack.
- peek() returns the **value** of the top Node from the stack but does not remove it. It needs no parameters. The stack is not modified.
- isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- len(stack_object) returns the number of items on the stack. It needs no parameters and returns an integer.

You are not allowed to use any other data structures to copy the elements of the Stack for manipulating purposes.

EXAMPLE

```
>>> x=Stack()
>>> x.pop()
>>> x.push(2)
>>> x.push(4)
>>> x.push(6)
>>> x
Top:Node(6)
Stack:
6
4
2
>>> x.pop()
6
>>> x
Top:Node(4)
Stack:
4
2
>>> len(x)
2
>>> x.isEmpty()
False
>>> x.push(15)
>>> x
Top:Node(15)
Stack:
15
4
2
>>> x.peek()
15
>>> x
Top:Node(15)
Stack:
15
4
2
```

Tips:

- Make sure you update the top pointer according to the operation performed
- Starter code contains the special methods __str__ and __repr__, use them to ensure the stack operations are updating the elements in the stack correctly
- When a method is asking to return the value of a node, make sure you are returning node.value and not a Node object

✓ Part 2: As discussed in the Module 6 video lectures, an arithmetic expression can be written in three different but equivalent notations: infix, prefix and postfix. Postfix is a notation for writing arithmetic expressions in which the operands appear before their operators (a+b is ab+). In postfix expressions, there are no precedence rules to learn, and parentheses are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of the multiple parentheses required in nontrivial infix expressions. In the starter code, there is a class called Calculator. This class takes no arguments and initializes Calculator object with an attribute called *expr* set to None. This attribute is then set to a non-empty string that represents an arithmetic expression in infix notation.

```
class Calculator:
    def __init__(self):
        self.expr = None
```

This an arithmetic expression **might** include numeric values, five arithmetic operators (+, -, /, * and ^), parentheses and extra spaces. Examples of such expression are `' -4.75    *     5 - 2.01  /  (3  *  7) +      2     ^     5'` and `'4.75+5-2.01*5'`

This class includes 3 methods that you must implement:

- **isNumber**(txt), where *txt* is a non-empty string. It returns a boolean value. True if *txt* is a string convertible to float, otherwise False. Note that `'   -25.22222 '` is a string convertible to float but `'  -22    33 '` and `'122 ; 45'` are not. An easy way to check if str to float is possible is with a try-except block

```
isNumber('Hello') returns False
isNumber('      2658.5     ') returns True
isNumber('      2            8     ') returns False
```

- **postfix**(expr): where *expr* is a non-empty string. This method takes an arithmetic expression in infix notation (string) and returns a string that contains *expr* in postfix notation (see doctest in starter code for string formatting).
  - All numbers in the string must be represented using its float format. The method isNumber could be extremely useful to determine whether the string between two operators is a valid number or not.
  - You must use the Stack code from Part 1 to convert *expr* into a postfix expression, otherwise your code will not get credit. See the Notes at the end of the instructions for examples of how the stack could be used for this process
  - If postfix receives an invalid expression, it should return a string containing the word 'error' or the None keyword. Examples of invalid expressions are `'  4 *  /  5'`, `' 4 + '`, `'  ^4 5'`, `'(   3.5  +   5  '`, `'3.      5 + 4'`

- **calculate**(): This property method takes the non-empty string self.*expr*, an arithmetic expression in infix notation, calls postfix(*expr*) to obtain the postfix expression of self.*expr* and then uses a Stack as described in the Stack video lecture to evaluate the obtained postfix expression. All values returned by calculator should be float, unless the expression is invalid.

- The function must **return** the computed value as float if *expr* is a correct formula, otherwise it must return None or an error message.
- **Do not use *exec* or *eval* function**. You will not receive credit if your program uses any of the two functions anywhere

Grading Notes:

- Supported operations are addition (+), subtraction (-), multiplication(*), division (/) and exponentiation (^), note that exponentiation is ** in Python
- You can **write other utility methods** to generalize your code and assist you with string processing and input validation, but don't forget to document them.
- The grading script will feed 10 randomly chosen test inputs for each function. Three of them will be inputs that should cause an error such as `'4.2 * 5/ 2 + ^2'` or `'4 * (5/ 2) + ^)2('`, whose expected returned value and error message or the None keyword.
- Grading Percentage: The correctness of your code represents a fraction of your score, but other items are also part of your final assignment score:

  Correctness-Functionality - 70%
  Format and Effort – 20%   (comments, clarity, etc)
  Submit Peer evaluation survey – 10% (release after due date)

- **If *calculator* checks if the expression is valid before passing it to postfix, there is no need to check it again in postfix, so you can ignore/remove the error message cases from the postfix doctest and add them in calculator and viceversa**
- You can define precedence of operators using a dictionary

Invalid expression for this assignment:
An error message (or None) should be returned when the expression:

- Contains non supported operators, for example `'   4 *   $   5'`
- Contains two consecutive operators `'   4 *   +   5'`
- Has unbalanced parentheses `'   )4 *     5('`, `'   (4 * 5)+('`
- `'3(5)'` is an invalid expression, valid multiplication will be denoted as `'3*5'` or `'3*(5)'`
- `'   2      5'` is an invalid expression, so be careful if you are removing spaces using replace, you could be transforming an invalid expression into a valid one. Useful tip: Remember that strip() removes spaces at the beginning and at the end of a string, so calling `'   2      5   '.strip()` returns a new string `'2      5'`

Deliverables:

- Submit it to the HW3 Gradescope assignment before the due date

Notes

Infix to Postfix Review

3+4*5/6 to postfix?

Step 1: If not done, parenthesize the entirely infix expression according to the order of priority you want. Since the given expression is not parenthesized, two examples are shown below:

(((3+4)*5)/6) and (3+((4*5)/6))

Step 2: Move each operator to its corresponding right parenthesis

(((3+4)*5)/6) ===> (((3 4) + 5) * 6)/

(3+((4*5)/6)) ===> (3 ((4 5) * 6) /) +

Step 3: Remove all parentheses

(((3+4)*5)/6) ==> (((3 4) + 5) * 6)/ ==> 3 4 + 5 * 6 /

(3+((4*5)/6)) ==> (3 ((4 5) * 6) /) + ==> 3 4 5 * 6 / +

(300+23)*(43-21)/(84+7) to postfix?

(((300+23)*(43-21))/(84+7)) ==> (((300 23) + (43 21) -)* (84 7)+)/

300 23 + 43 21 - * 84 7 + /

Tips for using the Stack to convert to postfix

Let's consider an expression in infix notation:

expr= '2 *   5   +   3   ^ 2-1 +4'

pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
expr	2		*			5			+			3			^	2	+	1		+	4

postfix_expression ?

Current portion of *expr*	**Stack**	**postfix_expression**	
2		2	
*	*	2	
5	*	2 5	
+	+	2 5 *	+ has lower precedence than *
3	+	2 5 * 3	
^	+ ^	2 5 * 3	
2	+ ^	2 5 * 3 2	
−	-	2 5 * 3 2 ^ +	- has lower precedence that ^ and equal precedence than +.
1	-	2 5 * 3 2 ^ + 1	
+	+	2 5 * 3 2 ^ + 1 -	+ has equal precedence than -.
4	+	2 5 * 3 2 ^ + 1 - 4	End of expr reached
		2 5 * 3 2 ^ + 1 – 4 +	

Now, let's consider an expression in infix notation with parentheses:

expr= '2* ((5 + 3)) /9'

pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
expr	2	*			(	(	5		+			3	)	)	/	9

Before starting any computations, does the expression has balanced parenthesis? No, return error

postfix_expression ?

Current portion of *expr*	**Stack**	**postfix_expression**	
2		2	
*	*	2	
(	* (	2	
(	* ((	2	
5	* ((	2 5	
+	* ((+	2 5	
3	* ((+	2 5 3	
)	* (	2 5 3 +	inner expression
)	*	2 5 3 +	inner expression
/	/	2 5 3 + *	/ has equal precedence than *.
9	/	2 5 3 + * 9	End of expr reached
		2 5 3 + * 9 /	

In general:

- If the item is a left parenthesis, push it on the stack
- If the item is a right parenthesis: discard it and pop from the stack until you encounter a left parenthesis
- If the item is an operator and has higher precedence than the operator on the top of the stack, push it on the stack
- If the item is an operator and has lower or equal precedence than the operator on the top of the stack, pop from the stack until this is not true. Then, push the incoming operator on the stack

---

**** If you are attempting any of the extra credit opportunities, you must submit your code in the HW3 – Extra Credit Gradescope assignment before the due date**

The given examples just represent the expected output, not the syntax for using the function

EXTRA CREDIT #1: 30 pts, added regardless of the maximum 100

In the regular assignment, two consecutive operators return an error message

```
>>> postfix('     2 *    5.4   +   3    ^ -2+1  +4     ')
'error'

>>> calculate('     2 *    5   +   3    ^ -2+1  +4     ')
'error'
```

Modify your code to support negation

```
>>> postfix('     2 *    5   +   3    ^ -2+1  +4     ')
'2.0 5.0 * 3.0 -2.0 ^ + 1.0 + 4.0 +'

>>> postfix('-2 *    5   +   3    ^ 2+1   +     4')
'-2.0 5.0 * 3.0 2.0 ^ + 1.0 + 4.0 +'

>>> postfix('-2 *    -5   +   3    ^ 2+1   +     4')
'-2.0 -5.0 * 3.0 2.0 ^ + 1.0 + 4.0 +'

>>> postfix('2 *  +  5   +   3    ^ 2       +1  +4')
'error'

>>> calculate('     2 *    5   +   3    ^ -2+1  +4     ')
15.11111111111111

>>> calculate('-2 *    5   +   3    ^ 2+1   +     4')
4.0

>>> calculate(' -2 / (-4) * (3 - 2*( 4- 2^3)) + 3')
8.5
```

```
>>> calculate('2 + 3 * ( -2 +(-3) *(5^2 - 2*3^(-2) ) *(-4) ) * ( 2 /8 + 2*( 3 -
1/ 3) ) - 2/ 3^2')
4948.611111111111
```

EXTRA CREDIT #2: 50 pts, added regardless of the maximum 100

In our regular assignment, `'3(5)'` is an invalid expression, since valid multiplication is denoted as `'3*5'` or `'3*(5)'`. Modify your code to support the * omission. For example, `'3(5)'` is treated as `'3*(5)'`. You can create a method to add the missing stars in in self.*expr* before passing it to postfix. You need to figure out the necessary and sufficient rules to add the missing *, for example, if there is an operand followed by a “(“, then insert a *.

```
>>> calculate(' 3 ( 5 )- 15 + 85 (12) ')
1020.0

>>> calculate(' (-2/6)+      (5(9.4)) ')
46.6666666667
```