CMPSC-132: Programming and Computation II
Spring 2020

Homework 3

Due Date: 03/27/2020, 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:

- The work in this assignment must be completed **alone and must be your own**. If you have questions about the assignment, contact the instructor or Teaching/Learning Assistants instead of asking for a solution online. If this assignment is recently found in sites like Chegg, Stackoverflow, etc., all similar submissions to those sites will be graded with a 10% penalty and they will be reported for academic integrity.
- **Download the starter code file from the HW3 Assignment on Canvas. Do not change the function names or given started code on your script.**
- A doctest is provided as an example of code functionality. Getting the same result as the doctest does not guarantee full credit. You are responsible for debugging and testing your code with enough data, you can share ideas and testing code during your recitation class. As a reminder**, Gradescope should not be used to debug and test code**!
- Each function must return the output (Do not use print in your final submission, otherwise your submissions will receive a -10 point deduction)
- Do not include test code outside any function in the upload. Printing unwanted or ill-formatted data to output will cause the test cases to fail. Remove all your testing code before uploading your file (You can also remove the doctest). Do not include the input() function in your submission.
- **Examples of functionality are given in the starter code. Your code should return None when an expression was not validated.**
- You are responsible for debugging and testing your code with enough data, you can share testing code on Piazza.

Goal:

✓ Part 1 [20 pts]: 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. You are not allowed to use the built-in stack tool from Python, or to use a Python list to simulate a stack. Your stack must be implemented with a Linked List, in other words, create nodes and use the *next* attribute to link the items in the stack (the procedure for this is given in the video lectures)

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. You are not allowed to modify them
- 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 [80 pts]: An arithmetic expression can be written in three different but equivalent notations: infix, prefix and postfix (Check the Module 6 video lectures for more on postfix notation). 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 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'`. Methods in this class will scan the expression from left to right, following PEMDAS rules. This means, we will first perform exponents, division and multiplication in the order they occur from left to right, and finally addition and subtraction

This class includes 3 methods that you must implement:

- **[15 pts] 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. *Hint*: 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
```

- **[45 pts] _getPostfix**(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 _getPostfix receives an invalid expression, it should return the None keyword. Examples of invalid expressions are `' 4 * / 5', ' 4 + ', ' ^4 5' , '( 3.5 + 5 ', '3. 5 + 4'`

- **[20 pts] calculate**: This property method takes the non-empty string self.__*expr*, an arithmetic expression in infix notation, calls _getPostfix(*expr*) to obtain the postfix expression of

self.__*expr* and then uses a Stack (as described in the Stack video lecture from Module 5) to evaluate the obtained postfix expression. All values returned by calculator should be float if *expr* is a correct expression, otherwise it must return None.

- **Do not use *exec* or *eval* function**. You will not receive credit if your program uses any of the two functions anywhere
- Note that you can write the code for this method without having an implementation for _getPostfix. The process for calculate is explained during the stack video lectures.

Grading Notes:

- This assignment requires multiple methods interactioning with each other. **Debugging** should be the first step when encountering semantic errors, and runtime errors such as infinite loops or Python exceptions. Use the tools described in Module 2 for debugging, we will not do it for you!
- Supported operations are addition (+), subtraction (-), multiplication(*), division (/) and exponentiation (^), note that exponentiation is ** in Python
- 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.
- The correctness of your code represents a big fraction of your score, however, proper encapsulation of code is expected in this assignment. If you don't use proper variable names, your assignment contains repeated/unnecessary code or you are putting all the work in one function, point deductions will be made, **regardless of the functionality of the code.**
    - To encapsulate your code, you can (and should) write other methods to generalize your code and to assist you with processes like string processing and input validation, but don't forget to document them.
    - Don't forget to document your code. We are not expecting comments in every line, but we do expect to see important block of code describing its purpose.
- **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:

The None keyword should be returned when the expression:

- Contains non supported operators, for example `'  4 *  $  5'`
- Contains two consecutive operators `'  4 *  +  5', '  4 *  -5'`. In other words, negation (`'  4 *  -  5'`) is not supported
- 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 + /

Infix to Postfix Online converter:

For creating extra test cases:
https://www.mathblog.dk/tools/infix-postfix-converter/
https://scanftree.com/Data_Structure/prefix-postfix-infix-online-converter

DISCLAIMER: I didn't test out these two sites with a lot of cases, so I recommend you to verify by hand the given result

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 None

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

Note that exponentiation is right-associative, so in order to support consecutive exponentiation (i.e `' 4 ^ 2 ^3'`) you need to keep equal operators in the stack.

More Infix to Postfix Conversion Examples

Example 1: 3 + 4 * 5 / 6

3 + 4 * 5 / 6
- Stack:
- Output: 3

3 + 4 * 5 / 6
- Stack: +
- Output: 3

3 + 4 * 5 / 6
- Stack: +
- Output: 3 4

3 + 4 * 5 / 6
- Stack: + *
- Output: 3 4

3 + 4 * 5 / 6
- Stack: + *
- Output: 3 4 5

3 + 4 * 5 / 6
- Stack: +
- Output: 3 4 5 *
- Stack: + /

3 + 4 * 5 / 6
- Stack: + /
- Output: 3 4 5 * 6

3 + 4 * 5 / 6
- Stack: +
- Output: 3 4 5 * 6 /
- Stack:
- Output: **3 4 5 * 6 / +**

Done!

Example 2: (4+8)*(6-5)/((3-2)*(2+2))

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output:

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output: 4

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: (+
- Output: 4

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: (+
- Output: 4 8

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: (
- Output: 4 8 +
- Stack:
- Output: 4 8 +

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: *
- Output: 4 8 +

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 +

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 + 6

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (-
- Output: 4 8 + 6

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (-
- Output: 4 8 + 6 5

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: * (
- Output: 4 8 + 6 5 -

(cont.)
- Stack: *
- Output: 4 8 + 6 5 -

(4+8)*(6-5)/((3-2)*(2+2))
- Stack:
- Output: 4 8 + 6 5 - *
- Stack: /
- Output: 4 8 + 6 5 - *

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: / (
- Output: 4 8 + 6 5 - *

(4+8)*(6-5)/((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - *

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - * 3

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / ((-
- Output: 4 8 + 6 5 - * 3

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / ((-
- Output: 4 8 + 6 5 - * 3 2

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / ((
- Output: 4 8 + 6 5 - * 3 2 -
- Stack: / (
- Output: 4 8 + 6 5 - * 3 2 -

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / (*
- Output: 4 8 + 6 5 - * 3 2 -

(4+8)*(6-5)/(((3-2)*(2+2))
- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 -

(4+8)*(6-5)/((3-2)*(2+2))

- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 - 2

(4+8)*(6-5)/((3-2)*(2+2))

- Stack: / (* (+
- Output: 4 8 + 6 5 - * 3 2 - 2

(4+8)*(6-5)/((3-2)*(2+2))

- Stack: / (* (+
- Output: 4 8 + 6 5 - * 3 2 – 2 2

(4+8)*(6-5)/((3-2)*(2+2))

- Stack: / (* (
- Output: 4 8 + 6 5 - * 3 2 – 2 2 +
- Stack: / (*
- Output: 4 8 + 6 5 - * 3 2 – 2 2 +

(4+8)*(6-5)/((3-2)*(2+2))

- Stack: / (
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + *
- Stack: /
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + *

(4+8)*(6-5)/((3-2)*(2+2))

- Stack:
- Output: 4 8 + 6 5 - * 3 2 – 2 2 + * /

Done!

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

EXTRA CREDIT #1: 10 pts, added regardless of the maximum 100

In the regular assignment, two consecutive operators return None

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

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

Modify your code to support negation

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

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

>>> _getPostfix('-2 *    -5.3   +   3    ^ 2+1  +     4')
'-2.0 -5.3 * 3.0 2.0 ^ + 1.0 + 4.0 +'
```

```
>>> _getPostfix('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: 10 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 self.__*expr* before passing it to _getPostfix. 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 ) ')
52.5

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

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