CSE 101: Introduction to Computational and Algorithmic Thinking

Lab #3

Fall 2018

Assignment Due: Saturday, September 29, 2018, by 11:59 pm

Assignment Objectives

By the end of this assignment you should be able to design, code, run and test original Python functions that solve programming problems involving `if` statements, strings, lists, and `for` and `while` loops.

Getting Started

The assignments in this course require you to write Python code to solve computational problems. To help you get started on each assignment, we will give you a "bare bones" file with a name like `lab3.py`. These files will contain *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You will fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `lab3.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

Solve the following problems to the best of your ability. This assignment has a total of 4 problems, worth a total of 16 points in all. The automated grading system will execute your solution to each problem several times, using different input values each time. Each test that produces the correct/expected output will earn 1 point. You must earn at least 12 points in order to pass (receive credit for) this lab.

- At the top of the `lab3.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Lab #3)

⚠ Each of your functions must use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

Part I: Updating Version Numbers (4 points)

Everyone loves software updates, except for the people who have to update the version numbers. A software version number is a sequence of one or more integers, separated by periods, like 10.11.6. When a new software update comes out, one of these numbers is incremented by 1; any numbers that follow are reset to 0. For example, updating the second value in 10.11.6 would produce the new version number 10.12.0. Updating the third value in 3.15.2.8.4.1 would result in 3.15.3.0.0.0, and so on.

Complete the `updateVersion()` function, which takes two arguments: a list of integers representing the parts of a version number, and an integer indicating the (0-based) index to increment (you may assume that this value is always a valid index). ***This function does not return anything;*** instead, it directly updates the list parameter to represent the new version number, with the appropriate elements updated according to the description above (for example, the arguments [1, 2, 3, 6] and 2 would lead to the function changing the list to [1, 2, 4, 0]).

Hint: After you increment the specified value in the list, use a `for` loop and the `range()` command to reset any subsequent positions in your list.

Examples:

Function Call	Final Contents of List
`updateVersion([5, 7, 2], 1)`	`[5, 8, 0]`
`updateVersion([11, 4, 10, 1, 7, 4], 2)`	`[11, 4, 11, 0, 0, 0]`
`updateVersion([1, 5, 11, 12, 8], 4)`	`[1, 5, 11, 12, 9]`

Part II: Shifting Bits (4 points)

A common operation within computer hardware involves shifting the bits in a memory location to the right or left by a certain amount. When we perform one specific type of shift (an *arithmetic shift*) to the right, we remove some number of bits (*n*) from the right end of the bit sequence, and replace them by inserting *n* copies of the first (leftmost) bit on the left. The end result is a new sequence of bits with the same length as the original.

In pseudocode, we might describe this process as follows:

Use `pop()` *in a loop, or slicing, to remove the last n bits from the end*

Use `insert()` *in a loop to insert n copies of the first bit at position 0*

For example, suppose that we have the bit sequence `010011` and we with to shift this sequence 2 bits to the right. We remove the two rightmost bits (`11`) and insert two copies of the first (leftmost) bit 0 at the beginning of the sequence (thus preserving the original number of bits). This gives us the new bit sequence `000100`.

Complete the `rightShift()` function, which takes two arguments: a list of 0s and 1s, and a positive integer representing the number of positions to shift the contents of the list. If the shift amount is greater than or equal to the length of the list of bits, the function returns without doing anything. Otherwise, the function should use the process above to directly modify the list parameter; as with the previous problem, *the function does not return any value on its own.*

Examples:

Function Call	Final Contents of List
`rightShift([0, 1, 1, 1, 0, 0, 0, 1], 3)`	`[0, 0, 0, 0, 1, 1, 1, 0]`
`rightShift([1, 1, 0, 1, 1], 1)`	`[1, 1, 1, 0, 1]`
`rightShift([0, 0, 1, 0, 1, 1], 10)`	`[0, 0, 1, 0, 1, 1]` (no change)

Part III: String-Swapping (4 points)

Complete the `swapEnds()` function, which takes three arguments: a string of 3 or more characters, plus two positive integers representing index positions within that string. The function returns a new string according to the following process:

- If either index value is invalid (greater than or equal to the length of the string), or if the index values are equal to one another, return the original string
- If the first index is greater than the second index, swap their values (see below for a tip on how to do this easily)
- Finally, return a new string that consists of the following sections of the original string, in this order:
    1. All of the characters from the original string, from the character *after the second index* through the end
    2. All of the characters from the first index *through* the second index
    3. All of the characters from the beginning of the string up to (but not including) the first index

For example, consider the string "sesquipedalian", with a first index value of 10 and a second index value of 5. The first index is greater than the second index, so we exchange their values to get 5 and 10. We now have three substrings:

- "ian" (the characters from index 11 through the end of the string)
- "ipedal" (the characters from index 5 through index 10)
- "sesqu" (the characters from the beginning of the string up to, but not including, index 5)

Combining these strings using the + operator, we get a final result of "ianipedalsesqu".

Hint: Python provides a simple way to exchange the values of two variables using something called *multiple assignment*:

```
a, b = b, a
```

Examples:

Function Call	Return Value
`swapEnds("space:  the final frontier", 5, 14)`	`l frontier:  the finaspace`
`swapEnds("these are the voyages", 12, 7)`	` voyagesre thethese a` (with a leading space)
`swapEnds("where no one has gone before", 8, 42)`	`where no one has gone before` (no change)

Part IV: Reverse Polish Notation (4 points)

Reverse Polish Notation (RPN), also known as postfix notation, is an alternate way to represent a sequence of arithmetic operations. In RPN, we write the operands/values first, followed by the applicable arithmetic operator (for example, 5 + 2 would be written as 52+). To evaluate a postfix/RPN expression, we store each operand that we see, in order, until we reach an operator. Once we see an operator, we apply it to the last two operands we saw (in the appropriate order) and then store the result back to our list while we continue scanning for operands and operators. When we are done, we should have exactly one value in the list; this is our final answer.

For example, consider the expression 43*5- (assume that each operand is initially one digit in length). We begin by storing 4, then 3. When we encounter the multiplication operator, we multiply our last two stored values (4 and 3) and store the result (12). We store the next operand (5). Finally, we encounter the subtraction operator, so we calculate 12-5 to get 7.

In Python, we can perform this calculation using a list. Each time we read an operand from the input, we use `append()` to add it to the end of our (initially empty) list. Each time we read an operator from the input, we use `pop()` to remove and store the last two elements of the list in temporary variables. We apply the operator and append the result to the end of the list. When we are done, if the list contains exactly one value, that value is our final answer. If it does not, then there must have been an error in the original input value.

Complete the `rpn()` function, which takes a string representing an RPN expression (with single-digit operands) as its only argument. The function returns either the answer to the expression, or it returns the string "Error" (note the capitalization) if the final internal list is empty or contains more than 1 value. You may assume that you will never receive an input string with more operators than operands, or with any characters other than digits and the four operators +-*/.

Note: Since we are working exclusively with integers for now, every division operator performs floor division. Thus 83/ evaluates to 2, not 2.67.

Your basic solution approach will probably resemble the following pseudocode:

```
For each character in the input:
  If the current character is a digit:
    Convert it to an integer and append it to your list
  Otherwise (meaning the current character is an operator):
    Pop the last number from the list and save it to a variable op2
    Pop the (new) last number from the list and save it to a variable op1
    Using an if-elif chain with four parts (one for each operator):
      Compute the result of op1 operator op2 (e.g., op1 - op2 for subtraction)
      Append the result to your list

If the list has exactly one element, return that value.
Otherwise, return the string "Error"
```

Hint: To identify an integer, you can either use the string method `isdigit()`, or use the `in` operator to check whether the character appears in the string "1234567890" or the constant `string.digits` from the `string` module:

`my_character.isdigit()` **OR** `my_character in string.digits`

Examples:

Function Call	Return Value
`rpn("532*+")`	`11`
`rpn("273-")`	`Error`
`rpn("94/12+-")`	`-1`