CSE 101: Introduction to Computational and Algorithmic Thinking

Lab #4

Fall 2018

Assignment Due: Saturday, October 6, 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 `lab4.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 `lab4.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 `lab4.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 #4)

⚠ 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: Run-Length Encoding (4 points)

Run-length encoding is a relatively simple technique for compressing data; if a particular value (or character sequence) appears multiple times in a row, it is only represented once, followed by an integer indicating the number of times it should be repeated. For example, the string "AAAAA" can be compressed to "A5" (5 occurrences of "A"), saving three characters.

Complete the `expand()` function, which takes a single string argument. You may assume that this string is run-length encoded and contains an even number of characters (each pair of characters consists of a character to be printed, followed by a single (non-zero) digit representing the number of times that character should appear in the expanded string). The function returns a new string that contains the expanded (or decoded) version of its argument. For example, calling `expand("a3b7a2c4")` would return the string "aaabbbbbbbaacccc".

Hint 1: Use a `while` loop to solve this problem. Let your loop variable represent your current position (index) within the string, and increment it by 2 after each iteration of the loop, so that it always holds the index of the current character.

Hint 2: Remember that you can perform multiplication on strings using the * operator. For example, `"abc" * 3` will produce the string `"abcabcabc"`. This will save you from having to use a nested loop to duplicate each character in the encoded string. As you process the string, multiply the character at the current index `i` by the (integer) value of the digit at index `i+1`, then append that string to your result.

Examples:

Function Call	Expanded String
`expand("d3o5z2y1")`	`"dddooooozzy"`
`expand("y2e1Z314p6o1C5")`	`"yyeZZZ1111ppppppoCCCCC"`
`expand("E2r8A5*9e6F4")`	`"EErrrrrrrrAAAAA*********eeeeeeFFFF"`

Part II: Braiding Integers (4 points)

Given a three-element list of integers that is initially filled with 1s, we can "braid" it by selectively swapping or recombining values. For example, an "odd" crossover changes the values [x, y, z] to [y, x+y, z], and an "even" crossover changes the values [x, y, z] to [x, y+z, y]. These crossovers occur in alternating order: odd, then even, then odd, etc.

Complete the `braid()` function, which takes a single integer argument representing the "level" of braiding (level 0 means no change, level 1 means a single "odd" crossover turning [1, 1, 1] into [1, 2, 1], level 2 means an odd crossover followed by an even crossover, etc.). The function returns the final braided list. You may assume that the function argument is always greater than or equal to 0.

Hint: Python's multiple-assignment feature may be useful here. To use it, write the destination variables on the left side, separated by commas; on the right side, write the new values for each variable, again separated by commas:

```
a, b = b, a      # assign b's old value to a, and a's old value to b
```

Examples:

Function Call	Final Braided List
`braid(0)`	`[1, 1, 1]`
`braid(2)`	`[1, 3, 2]`
`braid(5)`	`[6, 9, 4]`

Part III: Do the Faro Shuffle (4 points)

A *Faro shuffle* is often used by stage magicians to rearrange the cards in a deck. The deck is split in half, and the cards from the first half are interleaved (alternated) with the cards from the second half. For example, given the deck [1, 2, 3, 4, 5, 6, 7, 8], we treat the deck as two half-size decks — [1, 2, 3, 4] and [5, 6, 7, 8] — and reassemble them into a single deck, alternating one card from the first half and one card from the second half: [1, 5, 2, 6, 3, 7, 4, 8].

We can repeat this shuffling process multiple times to further rearrange the list. For example, Faro-shuffling the list [4, 2, 8, 1, 7, 3, -3, 9, 6, 2] three times in a row would produce the following series of intermediate rearranged lists (note that continuing to repeat the Faro shuffle will eventually restore the original order of the elements):

[4, 3, 2, -3, 8, 9, 1, 6, 7, 2]

[4, 9, 3, 1, 2, 6, -3, 7, 8, 2]

[4, 6, 9, -3, 3, 7, 1, 8, 2, 2]

Complete the `shuffle()` function, which takes two arguments: a list of integer values and a positive (non-zero) integer representing the number of Faro shuffles to perform. The function performs the Faro shuffle process the specified number of times, and returns ***a new list*** containing the final arrangement of elements. You may assume that the list argument will always contain an even number of values.

As a general solution structure, consider the following:

```
Create a variable new_deck that holds a copy of the original deck

for each round:
    Create a temporary variable with an empty list
    Create a new variable half_1 to hold the first half of new_deck (using slicing)
    Create a new variable half_2 to hold the second half of new_deck (using slicing)
    Do the following (length of new_deck // 2 times:
        Remove the first element of half_1 and append it to your temporary list
        Remove the first element of half_2 and append it to your temporary list
    Delete the contents of new_deck
    Transfer or copy the contents of the temporary list into new_deck

Return new_deck
```

Examples:

Function Call	Final Order of Values
`shuffle([5, 2, 1, 7, 2, 3], 1)`	`[5, 7, 2, 2, 1, 3]`
`shuffle([14, 3, 2, -8, 11, 5, 9, 12], 2)`	`[14, 2, 11, 9, 3, -8, 5, 12]`
`shuffle([14, 3, 2, -8, 11, 5, 9, 12], 3)`	`[14, 3, 2, -8, 11, 5, 9, 12]`

Part IV: The Backspace Boogie (4 points)

On some old Unix terminals, pressing the Backspace (delete) key generates the character sequence ^H (Control-H) to delete the previously-typed character. Some people use this for sarcastic effect when writing, to pretend that they are deleting or replacing some insulting comment:

Be nice to this fool^H^H^H^Hgentleman, he's visiting from corporate HQ.

Complete the `backspace()` function, which takes a single string argument. The function returns a new string where every "^H" is missing and has been applied to remove the character immediately before it. For example, given the input above, the program would produce the output:

Be nice to this gentleman, he's visiting from corporate HQ.

where each "^H" has effectively erased a character from "fool". Note that repeated "^H" sequences operate in "reverse order": the first "^H" deletes the character immediately before it, the next "^H" deletes the character before that one, and so forth. In the example above, the first "^H" deletes the 'l' from "fool", the second "^H" deletes the second 'o' from "fool", the third "^H" deletes the first 'o' from "fool", and the fourth "^H" deletes the 'f' from "fool".

Your program should only perform erasures for an exact match of "^H"; carets not followed by "H" (as in "2^2") and the sequence "^h" (with a lowercase 'h') should be passed through unchanged. You may assume that "^H" will never appear as the first two characters of the input.

The best solution strategy for this problem combines a `while` loop, string slicing, and the `find()` string method (see the lecture slides for more details on this method). Start by using `find()` to search the original string for the sequence "^H"; as long as `find()` returns an index value other than -1 (which means that the search value wasn't found):

- Use slicing to remove every character from the beginning of the current string up to the location of the "^H" sequence and append the result to a new string
- Use slicing to remove the last character from the new string (you can't directly modify a string, so this will involve assigning every character but the last back to the new string)
- Use slicing to reset the current string to every character following the "^H" sequence (basically, *location*+2 onward).
- Call `find()` again to see if "^H" appears anywhere else in the current string

Once the loop ends, append any remaining text from the original string to your new string.

Examples:

Function Call	String to be Returned
`backspace("Hello, w^horld!")`	`"Hello, w^horld!"`
`backspace("A Horse is a Horse, of k^Hcourse")`	`"A Horse is a Horse, of course"`
`backspace("I wish^H^H^Hant some ice cream^H^H^H^H^Hmilk")`	`"I want some ice milk"`