CSE 101: Introduction to Computational and Algorithmic Thinking

Lab #6

Fall 2018

Assignment Due: Saturday, October 27, 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, loops, and recursion.

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 `lab6.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 `lab6.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 `lab6.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 #6)

⚠ 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: Medieval Manuscript Signatures (4 points)

Scribes in Medieval Europe who laboriously hand-copied manuscripts for their patrons were generally not permitted to explicitly take credit for their work, but many of them found a subtle way to do so anyway: they would often include a slightly-obscured version of their names at the end of the documents they had produced. Instead of using a complicated form of encryption, these scribes simply added their first names, with each vowel replaced by the first consonant that came after it in the alphabet. For example, "Aelfred" would be written as "Bflfrfd" ('a' is replaced by 'b', 'e' by 'f', and so forth). The letter 'i' should be replaced by 'k' (remember that most of these scribes were writing in Latin, which does not have a distinct letter 'j').Complete the `signature()` function, which takes a single string argument representing a name. Your function may assume that the name is all lowercase, but it **MAY NOT** assume that it only contains letters. In other words, any letters in the input will be lowercase, but the input may also contain spaces, dashes, digits, and other non-letter characters. If the input string only contains letters, the function should return the enciphered equivalent of this name, using the process described above. If the input string contains at least one non-letter character, your function should return the string "ERROR" instead.

- Your code only needs to replace the five "normal" vowels (a, e, i, o, and u) – don't change or replace any occurrences of 'y'.
- You may assume that the input will never contain the letter 'j' (this technically doesn't affect anything, but we will assume that the input always conforms to this omission from the Latin alphabet)
- Your solution does not need to worry about "collapsing" adjacent vowels into a single consonant. For example, it should translate a name like "aethelred" into "bfthflrfd" without treating the initial "ae" pair as a single vowel (as would have been done in Old English or Middle English).

For example, given the user input "othuil", your program should display the translated name "pthvkl".

Hint: You can test for a valid lowercase letter with the `in` operator and a string that contains all 26 lowercase English letters (like the Python constant `string.ascii_lowercase`).

Examples:

Function Call	Return Value
`signature("uther")`	`vthfr`
`signature("essaul")`	`fssbvl`
`signature("maerl4n")`	`ERROR`

Part II: Odd Runs of Zeroes (4 points)

Complete the `oddRuns()` function, which takes a single string argument that contains only 0s and 1s. The function counts and returns the total number of *odd-length* runs (sequences) of 0s in the string. A run boundary is marked either by a '1' or by the beginning or end of the string. For example, the string "101111111001000" has 2 odd-length runs of 0s: "0" and "000" (there are three runs of 0s in all, but the middle run only contains 2 0s, so it doesn't count).

Hint: In addition to your run-counting variable, use two more variables to track if you are currently in the middle of a run of 0s, and how many 0s you have seen in the current run. If you are **NOT** currently in a run and you encounter a 0, note that you are in a run, and set your count of 0s to 1. When a run of 0s ends (due to encountering a 1 ***OR*** the end of the string), check to see if its length is odd and update your count appropriately, then clear your "currently in a run of 0s" variable.

Examples:

Function Call	Return Value
`oddRuns("0101110000")`	`2`
`oddRuns("001001110010000")`	`0`
`oddRuns("100010110111011100011")`	`5`

Part III: Recursive LOLs (4 points)

Some people like to spam Internet chatrooms with the single word "LOL". This often annoys some of the other users, who will respond by announcing "One more LOL and I'm out". Still other users will parody this response by posting "One more 'One more LOL and I'm out' and I'm out", and so on, nesting more and more layers of "One more {X} and I'm out".

Complete the recursive `lol()` function, which takes a positive (non-zero) integer argument representing the level of nesting ("One more LOL and I'm out" represents one level of nesting). The function returns a string containing the appropriately-nested version of the string above. For example, `lol(3)` would return the string `One more One more One more LOL and I'm out and I'm out and I'm out`. You **DO NOT** need to worry about correct capitalization or printing nested quotes in your answer. **HINT:** Your base case should be level 0, which is just "LOL".

Examples:

Function Call	Return Value
`lol(0)`	`LOL`
`lol(2)`	`One more One more LOL and I'm out and I'm out`
`lol(5)`	`One more One more One more One more One more LOL and I'm out and I'm out and I'm out and I'm out and I'm out`

Part IV: Recursive Number Patterns (4 points)

Complete the `pattern()` function, which takes a single positive (non-zero) integer as its argument. This function returns a new string containing a specific pattern of 1 or more integers, ***each separated by a single space***:

- If the user enters 1, the program should return "1"
- If the user enters 2, the program should return "1 2 1"
- If the user enters 3, the program should return "1 2 1 3 1 2 1"

etc. If the pattern isn't obvious from the examples above, the base case occurs when the argument is 1, which returns "1". The recursive case (when the argument *n* is greater than 1) surrounds the current value of *n* with the results of calling the function with the value (*n* - 1). For example, `pattern(2)` ultimately returns "1 2 1" by calling `pattern(1)`, followed by "2", followed by calling `pattern(1)` again.

Don't worry if your function's return value contains one or more extra leading or trailing spaces, as long as there are no double spaces between any of the integers.

Examples:

Function Call	Return Value
`pattern(1)`	`1`
`pattern(3)`	`1 2 1 3 1 2 1`
`pattern(5)`	`1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1`