# CSE 101: Introduction to Computational and Algorithmic Thinking

## Lab #2

## Fall 2018

**Assignment Due: September 21, 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` 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 `lab2.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 `lab2.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 (technically 5) 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 `lab2.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 #2)

⚠ 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.

⚠ 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: Drop Dead Dice (4 points)

*Drop Dead* is a game played with dice. Players take turns rolling up to five dice and earn points based on the numbers that they roll. If a player rolls a 2 or 5 on any of the dice, those dice are declared "dead" and may not be re-rolled that turn; in addition, the player scores 0 for that roll. Otherwise, if no 2s or 5s are rolled, the player earns points equal to the sum of the values shown on all of the dice. For example, a roll of { 6, 1, 3, 6, 3 } is worth 19 points; a roll of { 4, 1, 6, 3, 2 } earns 0 points (due to the 2).

Complete the `score()` function, which takes a list of five integer values (representing the values rolled on five ordinary six-sided dice) as its sole argument. If the list contains any 2s or 5s, the function returns 0. Otherwise, the function returns the score for that "hand" of dice.

**Examples:**

| Function Call | Return Value |
|---|---|
| `score([ 6, 3, 3, 1, 5 ])` | `0` |
| `score([ 6, 1, 3, 1, 4 ])` | `15` |
| `score([ 4, 1, 2, 4, 4 ])` | `0` |

## Part II: Descending Factors (4 points)

Complete the `factors()` function, which takes a single positive integer as its argument. This function returns a list that contains **all** of the integer factors of the argument (including 1 and the number itself), and *only* those values, in *descending* order. For example, `factors(24)` would return the list [ 24, 12, 8, 6, 4, 3, 2, 1 ].

**Hint 1:** Use a `for` loop that counts *down* from the original number through 1 (you can do this by supplying three arguments to Python's `range()` command).

**Hint 2:** Use the modulo (%) operator to test whether one number is evenly divisible by another (meaning that the remainder is 0).

**Examples:**

| Function Call | Return Value |
|---|---|
| `factors(38)` | `[38, 19, 2, 1]` |
| `factors(128)` | `[128, 64, 32, 16, 8, 4, 2, 1]` |
| `factors(17)` | `[17, 1]` |

## Part III: Climbing The Ladder (4 points)

You have been given a stepladder and a small robot that can climb up and down that ladder, one step at a time. The robot can be programmed using a language that contains exactly three commands:

- C (climb) causes the robot to move one step up the ladder. If the robot attempts to climb up when it is already on the top step of the ladder, it falls off the ladder and is destroyed.
- D (descend) causes the robot to move one step down the ladder. If the robot is already on the first step of the ladder, nothing happens (it stays where it is on step 1).
- R (reset) causes the robot to immediately return to step 1 of the ladder (this counts as a single operation).

The `ladder()` function takes two arguments: a positive (non-zero) integer representing the ladder's size (total number of steps) and a string representing a sequence of commands. This function returns the total number of instructions that will be executed before the robot falls to its doom from the top of the ladder (this count includes the final, fatal instruction). You may assume that every instruction sequence eventually leads to the destruction of the unfortunate robot. You may also assume that every instruction sequence only contains the command characters listed above.

The robot always begins on step 1 of the ladder.

For example, suppose that you have a 4-step ladder and the instruction sequence "CDDCCRCCCCCCCDDDD". In this case:

1. The robot first moves up one step to step 2.
2. The robot moves down one step to step 1.
3. The robot attempts to move down another step. It is already on the first step, so nothing happens.
4. The robot moves up one step to step 2.
5. The robot moves up one step to step 3.
6. The robot resets and moves back to step 1.
7. The robot moves up one step to step 2.
8. The robot moves up one step to step 3.
9. The robot moves up one step to step 4. Note that this is the top step of the ladder.
10. The robot attempts to move up one step and falls off the ladder.

Thus, the robot self-destructs after performing 10 instructions. Any remaining commands are ignored.

**Hints:**

- Your general solution structure should be a `for` loop containing one or more conditional (`if`, `elif`, etc.) statements.
- Use extra variables to track the robot's current step on the ladder and the total number of instructions performed so far.
- If (technically *when*) the robot is destroyed, use a `return` statement *inside* your loop to break out of the function immediately and send back the final instruction count.

**Examples:**

| Function Call | Return Value |
|---|---|
| `ladder(1, "C")` | `1` |
| `ladder(1, "DDRCDCC")` | `4` |
| `ladder(4, "CDDCCCCCRCCCCDDDD")` | `7` |

## Part IV: Babylonian Numbers (4 or 8 points)

The ancient Babylonians represented numbers by writing their digits, in order, as sequences of cuneiform symbols (pressed into clay tablets with a stylus). They used a base-60 number system, so a single "digit" could range in value from 0 through 59; compare this to the base-10 Indo-Arabic number system that we use, where a single digit can only represent a value in the range 0–9. We can't easily reproduce these cuneiform symbols in text, so we will have to improvise:

- < represents the quantity 10
- `T` represents the quantity 1
- \ represents the quantity 0 (technically, the Babylonians didn't have the concept of "zero", but they used this symbol, by itself, as a placeholder between the other digits in a number, as in 101)

A single digit is represented by a sequence of one or more of the symbols above. To calculate the digit's value, add up the values associated with each symbol in the sequence (you may find Python's `count()` string method useful here). For example, "< <T" represents the quantity 21 (two 10's, plus 1). Within a sequence, all of the '<' symbols (if there are any) will come before any 'T' symbols, so you'll never see something like "<TT<". Simimlarly, a single sequence will never contain more than five '<' symbols, or more than nine 'T' symbols.

**NOTE:** The backslash character has special meaning within a Python string. To represent a (single) backslash character in a string (for example, to test for equality), use a double-backslash, as in "\\".

Complete the `babylonian()` function, which takes one argument: a list of strings, each of which contains a sequence of the pseudo-cuneiform symbols listed above. This function translates each string into the corresponding Indo-Arabic number (a value between 0 and 59, inclusive) and returns a new list containing those integers *in the same order as they appear in the original list*. For example, the input ["<TT", "\", "< <"] would produce the translated list [12, 0, 20].

**Optional Problem Extension:**

The program above translates a Babylonian number into digits, but it does not compute the final number. Each column in a Babylonian number is equal to the digit in that column, multiplied by a power of 60. The last (far right) column is the "ones" column; its digit value is multiplied by $60^0$ or 1. As we move to the left, each column increments the previous power of 60 by 1 (i.e., the digit in the next-to-last column is multiplied by $60^1$ or 60, the column immediately to its left is multiplied by $60^2$ or 3600, and so on). The final number equals the sum of these products. For example, the 4-digit sequence [22, 0, 5, 48] represents the numerical quantity

$$22\times60^3+0\times60^2+5\times60^1+48\times60^0 = 22\times216,000+0\times3,600+5\times60+48\times1 = 4,752,000+0+300+48 = 4,752,348$$

Modify your function so that, after translating the list of digits, it returns the combined final numerical value (instead of the list of digits). The process is (roughly) as follows:

1. Create variables to hold the final answer (initialized to 0) and the current exponent (if your loop runs from index 0 through the end, initialize this value to 1 less than the number of digits)
2. For each digit in your list, multiply it by 60 raised to the current exponent value. Add the product to your running total, then decrement the value of the exponent by 1

This modification is worth 2 points per correct answer, rather than 1.

**Examples:**

| Function Call | Return Value (1 point) | Return Value (2 points) |
|---|---|---|
| `babylonian(["TTT", "<"])` | `[3, 10]` | `190` |
| `babylonian(["<T", "\\", "TTTTTT"])` | `[11, 0, 6]` | `39606` |
| `babylonian(["<<<TTTTTTT", "<", "\\", "TTTT", "<<<<<"])` | `[37, 10, 0, 4, 50]` | `481680290` |