Assignment 2: due 8AM on Friday, Feb 22, 2019

Summary of Instructions

Note	**Read the instructions carefully and follow them exactly**
Assignment Weight	4% **of your course grade**
Due Date and time	**8AM on Friday, Feb 22, 2019**
Important	**As outlined in the syllabus, late submissions will not be accepted**
	Any files with syntax errors will automatically be excluded from grading. Be sure to test your code before you submit it
	For all functions, both in Part 1 and 2, make sure you've written good docstrings that include type contract, function description and the preconditions if any.

This is an individual assignment. Please review the Plagiarism and Academic Integrity policy presented in the first class,
i.e. read in detail pages 11-19 of course outline (course-outline2019.pdf). You can find that file on Brightspace under Course Info. While at it, also review Course Policies on pages 12- 14.

The goal of this assignment is to learn and practice the concepts covered thus far, in particular: strings (including indexing, slicing and string methods), control structures (if statements and for-loops), use of range function, function design and function calls.

The only collection you can use are strings and lists. You may not use any other collection (such as a set, tuple, or dictionary) in this assignment. Using any of these in a solution to a question constitues changing that question. Consequently, that question will not be graded.

You can make multiple submissions, but only the last submission before the deadline will be graded. What needs to be submitted is explained next.

The assignment has two parts. Each part explains what needs to be submitted. Put all those required documents into a folder called a2_xxxxxx where you changed xxxxxx to your student number, zip that folder and submit it as explained in Lab 1. In particular, the folder should have the following files:
Part 1: a2_part1_xxxxxx.py, a2_part1_xxxxxx.txt
Part 2: a2_part2_xxxxxx.py and a2_part2_xxxxxx.txt

Both of your programs must run without syntax errors. In particular, when grading your assignment, TAs will first open your file a2_part1_xxxxxx.py with IDLE and press Run Module. If pressing Run Module causes any syntax error, the grade for Part 1 becomes zero. The same applies to Part 2, when they open and run file a2_part2_xxxxxx.py.

1 Part 1: Function Library (60 points)

For this part of the assignment, you are required to write and test several functions (as you did in Assignment 1). You need to save all functions in part1_xxxxxx.py where you replace xxxxxx by your student number. You need to test your functions (like you did in Assignment 1) and copy/paste your tests in part2_xxxxxx.txt. Thus, for this part you need to submit two files: a2_part1_xxxxxx.py and a2_part1_xxxxxx.txt

1. Write a function called **print_factors** that takes an integer n as a parameter and prints out all the factors of n, returning True if 2 is a factor of n and returning False otherwise. Recall that a factor is a number between 1 and n that goes evenly into n.

Call	print_factors(24)	print_factors(1)	print_factors(5)	print_factors(25)
output	factors of 24 = 1 2 3 4 6 8 12 24	factors of 1 = 1	factors of 5 = 1 5	factors of 25 = 1 5 25
return	True	False	False	False

You must exactly reproduce the format of these logs. You may assume that the value passed to your function is greater than 0.

2. Write a function called **triangle** that takes an integer size as a parameter and prints a size * 2 – 1 wide by size tall triangle of numbers. See example calls below:

Call	triangle(0)	triangle(1)	triangle(3)	triangle(5)	triangle(6)
Example Output		1	12345 234 3	123456789 2345678 34567 456 5	12345678901 234567890 3456789 45678 567 6

The start number increases by one on each line. The first line contains no leading spaces and one additional leading space is added per line. Notice that after 9, the numbers wrap around back to 0.

3. The constant π is an irrational number with value approximately 3.1415928 . . . The precise value of π is equal to this infinite sum: π = 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + . . .

We can get a good approximation of π by computing the sum of the first few terms. Write a function approxPi() that takes as input a float-value error and approximates constant π within error by computing the preceding sum, term by term, until the difference between the current sum and the previous sum (with one less term) is no greater than error. The function should return the new sum.

```
>>> approxPi(0.01)
3.1465677471829556
>>> approxPi(0.0000001)
3.141592703589814
```

4. Write a function named is_fib_like that takes a list of integers as a parameter and that returns whether or not the sequence matches the pattern of the Fibonacci sequence (True if it does, False if it does not). The Fibonacci sequence begins with the number 1 followed by the number 1 and each successive value is the sum of the two previous values: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, and so on. It is possible to follow this pattern with different starting values. For example, Lucas numbers start with the values 2 and 1 but otherwise follow the Fibonacci pattern. Your function should determine whether each value after the first two is the sum of the previous two values in the sequence, returning True if the sequence has that pattern and returning False if it does not. If the list has two or fewer values, your function should return True. Below are sample lists and the value that should be returned for each:

Contents of list passed to is_fib_like	Value returned by is_fib_like
[]	True
[42]	True
[18, 42]	True
[1, 1, 1]	False
[1, 2, 3]	True
[0, 0, 0, 0, 0]	True
[1, 1, 2, 3, 5, 8, 13, 21]	True
[2, 1, 3, 4, 7, 11, 18, 29]	True
[1, 1, 2, 3, 5, 12, 17]	False

5. Write a function named longest_name that reads names typed by the user and prints the longest name (the name that contains the most characters) in the format shown below. Your method should accept an integer n as a parameter and should then prompt for n names. The longest name should be printed with its first letter capitalized and all subsequent letters in lowercase, regardless of the capitalization the user used when typing in the name. If there is a tie for longest between two or more names, use the tied name that was typed earliest. Also print a message saying that there was a tie, as in the right log below. It's possible that some shorter names will tie in length, such as ryan and TITO in the left log below; but don't print a message unless the tie is between the longest names.You may assume that n is at least 1, that each name is at least 1 character long, and that the user will type single-word names consisting of only letters. The following table shows two sample calls and their output.

Call	longest_name(5)	longest_name(7)
Output	name #1? **ryan** name #2? **TITO** name #3? **John** name #4? **lAuRaLyN** name #5? **SujaN** Lauralyn's name is longest	name #1? **PeTer** name #2? **eric** name #3? **RAFAEL** name #4? **brian** name #5? **sarina** name #6? **LIOR** name #7? **EmIlIo** Rafael's name is longest (There was a tie!)

6. Some substitution cipher for the digits 0, 1, 2, 3, . . . , 9 substitutes each digit in 0, 1, 2, 3, . . . , 9 with another

digit in 0, 1, 2, 3, . . . , 9. It can be represented as a 10-digit string specifying how each digit in 0, 1, 2, 3, . . . , 9 is substituted. For example, the 10-digit string '3941068257' specifies a substitution cipher in which digit 0 is substituted with digit 3, 1 with 9, 2 with 4, and so on. To encrypt a nonnegative integer, substitute each of its digits with the digit specified by the encryption key. Implement function encrypt( ) that takes as input a 10-digit string key and a digit string (i.e., the clear text to be encrypted) and returns the encryption of the clear text.

```
>>> encrypt('3941068257 ', '132')
'914'
>>> encrypt('3941068257 ', '111')
999'
```

1 Part 2: Gradanator

This *interactive program* focuses on if/else statements, input, and returning values. Turn in a file named a2_part2_xxxxxx.py. The program reads as input a student's grades on homework and three exams and uses them to compute the student's course grade.

Below is one example log of execution from the program. This program behaves differently depending on the user input; user input is bold and underlined below. Your output should match our examples exactly given the same input. (Be mindful of spacing, such as after input prompts and between output sections.)

```
This program reads exam/homework scores
and reports your overall course grade.

Midterm 1:
Weight (0-100)? 10
Score earned? 78
Were scores shifted (1=yes, 2=no)? 2
Total points = 78 / 100
Weighted score = 7.8 / 10

Midterm 2:
Weight (0-100)? 10
Score earned? 84
Were scores shifted (1=yes, 2=no)? 2
Total points = 84 / 100
Weighted score = 8.4 / 10

Final:
Weight (0-100)? 30
Score earned? 95
Were scores shifted (1=yes, 2=no)? 1
Shift amount? 10
Total points = 100 / 100
Weighted score = 30.0 / 30

Homework:
Weight (0-100)? 50
Number of assignments? 3
Assignment 1 score? 14
Assignment 1 max? 15
Assignment 2 score? 17
Assignment 2 max? 20
Assignment 3 score? 19
Assignment 3 max? 25
How many sections did you attend? 5
Section points = 15 / 34
Total points = 65 / 94
Weighted score = 34.6 / 50

Overall percentage = 80.8
Your grade will be at least: B
<< your custom grade message here >>
```

The program begins with an introduction message that briefly explains the program. The program then reads scores in four categories: midterm 1, midterm 2, homework and final. Each category is weighted: its points are scaled up to a fraction of the 100 percent grade for the course. As the program begins reading each category, it first prompts for the category's weight.

The user begins by entering scores earned on midterm 1. The program asks whether exam scores were shifted, interpreting an answer of 1 to mean "yes" and 2 to mean "no." If there is a shift, the program prompts for the shift amount, and the shift is added to the user's midterm 1 score. Exam scores are capped at a max of 100; for example, if the user got 95 and there was a shift of 10, the score to use would be 100. The midterm's "weighted score" is printed, which is equal to the user's score multiplied by the exam's weight.

Next, the program prompts for data about midterm 2 and then about the final. The behavior for each is the same as the behavior for midterm 1.

Next, the user enters information about his/her homework, including the weight and how many assignments were given. For each assignment, the user enters a score and points possible. Use a **cumulative sum** as described in lecture 7.

Part of the homework score comes from sections attended. We will simplify the formula to assume that each section attended is worth 3 points, up to a maximum of 34 points.

Once the program has read the user information for both exams and homework, it prints the student's overall percentage earned in the course, which is the sum of the weighted scores from the four categories, as shown below:

$$Grade = WeightedMidterm1Score + WeightedMidterm2Score + WeightedFinalExamScore + WeightedHomeworkScore$$

$$Grade = \left(\frac{78}{100}\times 10\right) + \left(\frac{84}{100}\times 10\right) + \left(\frac{100}{100}\times 30\right) + \left(\frac{14+17+19+(10\times 3)}{15+20+25+34}\times 50\right)$$

$$Grade = 7.8 + 8.4 + 30.0 + 42.6$$

$$Grade = 88.8$$

The program prints a loose guarantee about a minimum grade the student will get in the course, based on the following scale. See the logs of execution logs.txt to see the expected output for each grade range.

90% and above: A; **89.99% - 80%:** B; **79.99% - 70%:** C; **69.99% - 60%:** D; **under 60%:** F

After printing the guaranteed minimum grade, print a custom message of your choice about the grade. This message should be different for each grade range shown above. It should be at least 1 line of any non-offensive text you like.

This program processes user input using input. You should handle the following two special cases of input:

- A student can receive extra credit on an *individual assignment*, but **the total points for homework are capped at the maximum possible**. For example, a student can receive a score of 22/20 on one homework assignment, but if their total homework score for all assignments is 63/60, this score should be capped at 60/60. Section points are capped at 34.
- **Cap exam scores at 100.** If the raw or shifted exam score exceeds 100, a score of 100 is used.

Otherwise, you may assume the user enters **valid input**. When prompted for a value, the user will enter an integer in the proper range. The user will enter a number of homework assignments ≥ 1, and the sum of the four weights will be exactly 100. The weight of each category will be a non-negative number. Exam shifts will be ≥ 0.

Development Strategy and Hints:

- Tackle parts of the program (midterm 1, midterm 2, homework, final exam) one at a time, rather than writing the entire program at once. Write a bit of code, get it to run, and test what you have so far. If you try to

write large amounts of code without attempting to run it, you may encounter a large list of errors and/or bugs.

- To compute homework scores, you will need to cumulatively sum not only the total points the student has earned, but also the total points possible on all homework assignments.
- Many students get errors because they forget to pass / return a needed value, forget to store a returned value into a variable, and refer to a variable by the wrong name.
- All weighted scores and grades are printed with no more than 1 digit after the decimal point. Achieve this with a custom function or round. The following code prints variable x rounded to the nearest tenth:

```
x = 1.2345
print("x rounded to the nearest tenth is " + round(x, 1))  # 1.2
```

- Use max and min to constrain numbers to within a particular bound.

Style Guidelines:

A major part of this assignment is demonstrating that you understand parameters and return values. Use functions, parameters, and returns for structure and to eliminate redundancy. For full credit, use **at least 4 non-trivial functions** other than main.

You should not have print statements in your main function. Also, main should be a concise summary of the overall program; main should make calls to several of your other functions that implement the majority of the program's behavior. Your functions will need to make appropriate use of parameters and return values. Each function should perform a coherent task and should not do too large a share of the overall work. Avoid lengthy "chaining" of function calls, where each function calls the next, no values are returned, and control does not come back to main.

This document describes several numbers that are important to the overall program. For full credit, you should make at least one of such numbers into a constant so that the constant could be changed and your program would adapt.

Some of your code will use conditional execution with if and if/else statements. Part of your grade will come from using these statements properly. Review the portion of lecture 6 and 7 about nested if/else statements and factoring them.

Give meaningful names to functions and variables, and use proper indentation and whitespace. Follow Python's naming standards as specified in lecture. Localize variables when possible; declare them in the smallest scope needed. Include meaningful comment headers at the top of your program and at the start of each function. Limit line lengths to 100 chars.