CS 112 – Fall 2020 – Programming Assignment 5

Loops & Lists

Due Date: Sunday, October 4th, 11:59pm

The purpose of this assignment is to practice using loops and basic lists effectively.

See the **"assignment basics"** file for more detailed information about getting assistance, running the test file, grading, commenting, and many other extremely important things. Each assignment is governed by the rules in that document.

Background

Loop statements allow us to run the same block of code repeatedly, with the chance to use different values for variables each time as the accumulated effects pile up. Lists are sequences that can be inspected value-by-value and modified at will. In this assignment we will use loops to perform calculations over **one-dimensional lists** (in future assignments we will work with multi-dimensional lists too). However, working with one-dimensional lists doesn't necessarily mean that a single **`while/for`** loop is always enough, sometimes a repeated iteration over a list might require a (double or even triple) **nested loop**.

Restrictions

Any function that violates any of the following will receive **zero** points even if it passes all tests.

- You are **not** allowed to **`import`** anything
- You are **not** allowed to use slicing
- You are **not** allowed to use **`sets`** and **`dictionaries`**
- You are **not** allowed to use anything that hasn't been covered in class
- No built-in function except **`range()`** and **`len()`** is allowed
- You are **not** allowed to use any string method
- From list methods, you are allowed to use only **`.append()`**. Please **do not ask on piazza** whether you can use `.sort()`, `sorted()`, `.index()`, `.count()` etc.
- You are **not** allowed to use the **`del`** operator
- You are **not** allowed to use the **`in`** operator anywhere outside the syntax of the **`for`** statement

Testing

From this assignment on, testing will be done a bit different than before. The grader we use to grade your work will be different from the tester we provide. This means that there will be cases that the tester doesn't check and, therefore, your code might have logic errors that the tester doesn't detect. You must do additional checks, on your own, to make sure that you haven't missed anything and your code is correct. You do ***not*** need to modify the tester we provide, just test on your own any way you like. The goal is to help you put more focus on writing logically correct programs instead of trying to pass certain tests only.

Thus, the grader (a.k.a. autograder) will be made available on Gradescope after the deadline.

Grading Rubric

Submitted correctly:	2	# see assignment basics file for file requirements!
Code is well commented:	8	# see assignment basics file for how to comment!
Autograder:	90	# see assignment basics file for how to test!
TOTAL:	100	

Note: If your code does not run and crashes due to errors, it will receive **zero** points. Turning in running code is essential.

Functions

The signature of each function is provided below, do ***not*** make any changes to them otherwise the tester will not work properly. The following are the functions you must implement:

`replace_contraction(text)` **[15pts]**

Description: Creates a new string where the contractions **I'm** and **you're** have been replaced with the respective full form, **I am** and **you are**. The function is case sensitive, i.e. it will ignore any variations in case. Be reminded that you are **not** allowed to use any string method.

Parameters: **`text`** is a `string` of variable size

Return value: the new `string` that is generated from the replacements

Examples:

```
replace_contraction("hello")              →   "hello"
replace_contraction("hello, I'm Jack")    →   "hello, I am Jack"
replace_contraction("I'M and you're...") →   "I'M and you are..."
```

`swap_adjacent(numbers)` **[15pts]**

Description: It swaps the position of adjacent items of **`numbers`**. Assume that the length of the list is even.

Parameters: **`numbers`** (`list` of `int`)

Return value: Nothing is returned. The swapping occurs ***in-place***, i.e. you modify **`numbers`** itself and no list is created.

Examples:

```
ls = [1,2,3,4,5,6]
swap_adjacent(ls)       # it doesn't return anything
print(ls)               # it prints [2,1,4,3,6,5]

ls = [5,7,-3,2,1,1,-6,6]
swap_adjacent(ls)       # it doesn't return anything
print(ls)               # it prints [7,5,2,-3,1,1,6,-6]
```

`within_1_sd(numbers)` [15pts]

Description: Given a list of **`numbers`** it returns a new list that includes, in the same order, all those numbers that are less than **one standard deviation** away from the mean. Check the following website for an intuitive description of standard deviation (in your calculations assume that **`numbers`** is a *population*, not a *sample*):
https://www.mathsisfun.com/data/standard-deviation.html

Parameters: **`numbers`** is a `list` of mixed `int` and `float` that are **not ordered**

Return value: a **new** `list` of mixed `int` and `float`

Examples:

```
within_1_sd([-5,1,8,7,2])            →   [1, 7, 2]          # μ=2.6   σ=4.67
within_1_sd([1.5,4,-4,6,7,3.9,9.6])  →   [1.5,4,6,7,3.9]    # μ=4     σ=4.04
```

`equal_sum(numbers)` [20pts]

Description: Given a list of **`numbers`**, it checks whether the value of its first item equals the sum of the next two items, which equals to the sum of the next three items, which equals to the sum of the next four items, and so forth. Assume that the length of the list matches exactly the length that this pattern requires.

Parameters: **`numbers`** (`list` of `int`) that are **not ordered**

Return value: **`True`** or **`False`**

Examples:

```
equal_sum([6, 2, 4, 2, 2, 2, 1, 5, 0, 0])                    →    True
equal_sum([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, -2, -1])   →    True
equal_sum([2, 1, 1, -3, 5, 1])                               →    False
```

`is_sequence(numbers)` [25pts]

Description: Checks whether a list of integers form a sequence of consecutive numbers or not. Be reminded that you can't use the **`del`** operator and you're not allowed to use the **`in`** operator anywhere outside the syntax of the **`for`** statement.

Parameters: **`numbers`** (`list` of `int`) that are **not ordered**

Return value: **`True`** or **`False`**

Examples:

```
is_sequence([5, 7, 6, 4, 3])            →   True
is_sequence([1, 5, 7, 6, 4, 3, 8])      →   False
is_sequence([1, -2, -1, 2, 0, -3])      →   True
```