COSC 101 Homework 4 Spring 2016

Due date: **Monday, March 7, 10:00 a.m.**

Submit what you have **BEFORE** class: no late submissions are accepted for this homework, submit what you have. If your files are not on Moodle before class you will receive **zero** for hw4.

This homework is designed to give you practice writing functions and help you study for the upcoming exam. First you are asked to implement a series of functions that do rather silly and arbitrary things. Pay attention as each function has some subtlety to it and should help you further hone your coding skills.
Then you write two functions to determine if two sentences are anagrams of each other. Make sure you understand the question and you have worked out a solution on paper before starting coding. If you don't have a strategy, typing some code is not productive. Don't hesitate to come and see me or to go to open labs.

You are **not allowed** to use **string built-in functions** until we cover them in Unit 5.

Instructions

This homework is due before next Monday class so we can go over the correction then so you have immediate feedback and can study additional practice problems for the exam (available on moodle).

Please write your answers in `hw4.py` and `hw4_anagram.py`.

For each function you are required

- to include **the docstring** and
- to write sufficient tests to assure you the code is correct. Call the functions with different parameters, write as comments the console output, handling some general cases and special ones as we have done in class.

Don't forget to fill in the header at the top of the files, including hours spent, collaborators, and any feedback you have. Cheating is cheating yourself first.

Part I: Small functions

`sum13`

Return the sum of the numbers in the list, returning 0 for an empty list. Except the number 13 is so special that it does not count and the number that comes immediately after a 13 also do not count.

- `sum13([1, 2, 2, 1])` returns 6
- `sum13([1, 1])` returns 2
- `sum13([13, 1, 4])` returns 4
- `sum13([1, 2, 2, 1, 13])` returns 6
- `sum13([1, 2, 13, 2, 1, 13])` returns 4

`slice67`

Return a list of numbers similar to the parameter list, except that the numbers between a `6` and the next `7` are not included (every `6` will be followed by at least one `7`). Return an empty list when appropriate.

- `slice67([1, 2, 2])` returns `[1, 2, 2]`
- `slice67([1, 2, 2, 6, 99, 99, 7])` returns `[1, 2, 2]`
- `slice67([1, 1, 6, 7, 2])` returns `[1, 1, 2]`

`no_teen_sum` and `fix_teen`

Given 3 integer values, `a`, `b` and `c`, return their sum. However, if any of the values is a teen–in the range 13..19 inclusive–then that value counts as `0`, except `15` and `16` which do not count as a teens. Write a separate helper function named `fix_teen` that takes in an integer value and returns that value fixed for the teen rule. In this way, you avoid repeating the teen code three times.

- `no_teen_sum(1, 2, 3)` returns `6`
- `no_teen_sum(2, 13, 1)` returns `3`
- `no_teen_sum(2, 1, 14)` returns `3`
- `no_teen_sum(15, 15, 19)` returns `30`

`double_reverse`

This function takes a list as a parameter. You can assume that every item in the list is a string. The function returns a new list that contains the strings in reverse order and with each string reversed. You can assume the list is non-empty. Think about writing a helper function.

- `double reverse(['moo', 'cow', 'mooo'])` returns `['ooom', 'woc', 'oom']`

`xyz_there`

Return `True` if the given string contains an appearance of `'xyz'` where the `'xyz'` is not directly preceded by a period (`.`). So `'xxyz'` returns `True` while `'x.xyz'` returns `False`. You are not allowed to use string functions.

- `xyz_there('abcxyz')` returns `True`
- `xyz_there('abc.xyz')` returns `False`
- `xyz_there('xyz.abc')` returns `True`

Last small function

Write, comment and test either of the following two functions `make_tag_list` or `make_bricks`.

`make_tag_list(tag, lst)`

The web is built with HTML strings like `'<i>Yay</i>'`, which formats `Yay` as italic text, *Yay*, on the web-page. In this example, the string `'i'` makes the opening tag `'<i>'` and the closing tag `'</i>'` to surround the string. Given the tag string and a list of strings, `lst`, write the function to return a new list of HTML strings with tags around each string in `lst`.

- `make_tag_list('i', ['Yay', 'Hello'])` returns `['<i>Yay</i>', '<i>Hello</i>']`
- `make_tag_list('cite', ['Yay', 'cs'])` returns `['<cite>Yay</cite>', '<cite>cs</cite>']`

`make_bricks(small, big, goal)`

We want to make a row of bricks that is `goal` inches long. We have a number of `small` bricks (1 inch each) and `big` bricks (5 inches each).

Return `True` if it is possible to make the `goal` by choosing from the given bricks. Note that not all the bricks need to be used. This is a little harder than it looks and can be done without any loops.

- `make_bricks(3, 1, 8)` returns `True`
- `make_    bricks(3, 1, 9)` returns `False`
- `make_bricks(3, 2, 10)` returns `True`

Anagram

Your task is to write a program `hw4_anagram.py` that checks if two strings are anagrams). An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once.

This program should have a similar structure to the first program: you will define a few functions including one called `main`. At the end of the program, you should have a call to `main`.

First write a `remove_single` method that takes a string and a character and returns the parameter string with the character removed if it exists. If the character appears multiple times in the string, only a single occurrence is removed.

For example

```
>>> remove_single('anagram', 'a')
'nagram'
>>> remove_single(remove_single(remove_single('anagram', 'a'), 'a'), 'a')
'ngrm'
```

Next write a function `is_anagram` that takes two strings and uses `remove_single` to determine if the two strings are anagrams: `is_anagram` returns `True` if the two strings have exactly the same letters and may only differ by white spaces and returns `False` if there are any difference in letters.

For example

```
>>> is_anagram("orchestra", "carthorse")
True
>>> is_anagram("orchestra", "courthouse")
False
>>> is_anagram("a decimal point", "im a dot in place")
True
>>> is_anagram("debit card", "bad credit")
True
>>> is_anagram("snooze alarms", "alas no more zzs")
False
```

Finally, write the `main` function. It should ask the user for two inputs and tells the user whether or not they are anagrams. Here are two example executions.

```
Enter the first phrase: dormitory
Enter the second phrase: dirty room
Yes, these are anagrams!

Enter the first phrase: dormitory
Enter the second phrase: clean room
Sorry, these are not anagrams!
```

Extra Credit: Challenge Problem

Revisit the problem in `hw2` in which you were asked to draw something with turtle following a drawing of your own. This time, do the same thinking how functions can help your draw something more complex in a file called `hw4_challenge.py`. You should at least define and call two drawing functions and use a diagram to guide your code. You are required to bring to class your diagram that served you as a plan to receive extra credit. (Chapter 6 of the textbook might be helpful.)