COSC 101 Homework 8 Fall 2014

Due date: **Wednesday, October 29, 11:55pm**

This homework is designed to give you more practice writing functions and help you study for the upcoming exam. You are asked to implement a series of functions. Each function has some subtlety to it and should help you further hone your coding skills. **You are not allowed to use string built-in functions that have been introduced in Unit 5.**

If you finish this homework "early," you are strongly encouraged to spend your "free time" doing additional practice problems for the exam (available on moodle).

Please write your answers in `hw8.py`. You are required for each function to include **the docstring that explains what the function does**. When complete, please fill in the header at the top of the file, including hours spent, collaborators, and any feedback you have.

Finally, **please note that some functions impose certain constraints on your solution** – such as no for loops, no break statement, etc. Be sure to abide by these constraints!

1. `eliminate_duplicates` (no for loops allowed)

   Write a function named `eliminate_duplicates` that accepts a list of strings representing grocery items as a parameter, and constructs and returns a new list that contains only the unique items from the grocery list. You should *not* modify the list passed as a parameter. The strings in the list returned from the function can appear in any order.

   Important: You can only use `while` loops in your solution — no `for` loops are allowed.

   For example:

   ```
   grocery_list = ['yogurt', 'milk', 'eggs', 'milk', 'bananas', 'yogurt', 'yogurt']
   shorter_list = eliminate_duplicates(grocery_list)
   print shorter_list
       # shorter_list should contain 'yogurt',' milk, 'eggs', 'bananas',
       # in any order
   ```

2. `merge_lists` (no for loops allowed)

   Say that you and your roommate wrote separate grocery lists. To make shopping easier, it would be better to merge the lists into a single list and eliminate duplicate items. Write a function called `merge_lists` that accepts two lists of strings, and constructs and returns a new list that contains only the unique items from the combined lists. The items in the list returned from the function can appear in any order.

   For example:

   ```
   my_grocery_list = ['milk', 'eggs', 'milk', 'bananas', 'tortillas', 'beans']
   your_grocery_list = ['salsa', 'tortillas', 'avocados', 'beans', 'milk', 'salsa']
   merged_list = merge_lists(my_grocery_list, your_grocery_list)
   print merged_list
       # merged_list should contain 'milk', 'eggs', 'bananas', 'tortillas',
       # 'beans', 'salsa', and 'avocados' in any order
   ```

   Important: You can only use `while` loops in your solution — no `for` loops are allowed. (For this problem, it is also possible to solve it with a function body that does not have any loops whatsoever.)

3. `compute_running_sum` (no for loops allowed)

   Write a function called `compute_running_sum` that accepts a list of integers named `input_list`, and constructs a new list such that item *i* in the output list contains the sum of items in `input_list` from indexes 0 through *i*, inclusive. You should not modify the list passed as a parameter.

   For example, if the `input_list` is `[1,2,3]` your function should return the list `[1,3,6]`. Notice that:

   - index 0 in the result list just contains the integer at index 0 in the input list (1),
   - index 1 in the result list contains the sum of integers at indexes 0 and 1 in the input list (1+2),
   - index 2 in the result list contains the sum of integers at indexes 0, 1, and 2 in the input list (1+2+3).

   As another example, if the `input_list` is `[2,4,6,8]`, the function should return `[2,6,12,20]`.

   Important: You can only use `while` loops in your solution — no `for` loops are allowed.

4. `double_reverse` (for loops permitted)

   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']`
   - `double_reverse(['abcd', 'efg', 'hijkl'])` returns `['lkjih', 'gfe', 'dcba']`

5. `slice67` (for loops permitted)

   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]

6. `gamblers_ruin` (for loops permitted)

   Imagine the following scenario. The casino offers some game where the probability of winning is exactly $\frac{1}{2}$. Each time you play, you bet $1: if you win, you get $1; if you lose, you pay $1. You arrive at the casino with a certain amount of cash in hand – let's call this the stake – and your goal is to win a certain amount of money – let's call it the goal amount. You will keep playing this game until either (a) you win the goal, or (b) you go broke. Let's call this process of playing the game repeatedly (until one of these outcomes is reached) playing until "win or bust."

   The question is: what is the likelihood that you win? Clearly it depends on the stake and the goal. If the stake is $99 and the goal is $100, the likelihood is pretty good – it's at least 50% (why?), and actually quite a bit higher. On other hand, if your stake is $1 and the goal is $100, the likelihood is pretty small – it's at most 50%, and actually quite a bit lower.

   Write a function `gamblers_ruin` that takes no inputs and prints out an *estimated* probability of winning a goal of $200 for varying stakes. Here's an example output (with ?!! masking the estimated answer):

```
With stake = 20 and goal = 200 the probability of win = !?!
With stake = 40 and goal = 200 the probability of win = !?!
With stake = 60 and goal = 200 the probability of win = !?!
With stake = 80 and goal = 200 the probability of win = !?!
With stake = 100 and goal = 200 the probability of win = !?!
With stake = 120 and goal = 200 the probability of win = !?!
With stake = 140 and goal = 200 the probability of win = !?!
With stake = 160 and goal = 200 the probability of win = !?!
```

```
With stake = 180 and goal = 200 the probability of win = !?!
```

You should estimate the probability through simulation, much as you did the game of craps in lab. While this problem only asks you to write one function, `gamblers_ruin`, you will definitely want to break the task down into multiple functions. A **major goal of this problem** is to have you think about what the right break down should be.

At a high level, your code must carry out the following tasks, written in pseudocode (a hybrid between Python and English):

```
for each stake (from 20 to 180)
    repeat many times (at least 100 trials):
        play the game until 'win or bust'
        if reach goal, count as a win, else count as a bust
    estimate the probability of winning as no. of wins over no. of trials
    print probability of win given this stake
```

What's written above is a pretty terse description – e.g., to simply play the game until "win or bust", you will need to write more than one line of Python!

Some of these problems were taken or adapted from Nick Parlante's codingbat.