**Week 6 In-Class Exercises (More on Lists)**

**Note:** You can use either Jupyter Notebook or Anaconda Prompt for the questions below.

**Q1: Combine Content in Brackets [ \*\* ]**

**Part (a)**

Define a function called combine\_content\_in\_brackets. The function takes in a list of strings (called str\_list) as its only parameter. Each string in str\_list contains *exactly* one pair of *round brackets*, where the opening bracket is always before the closing bracket. The function extracts the content inside the brackets and concatenate them into a single string, which is returned by the function.

Here are some examples:

* combine\_content\_in\_brackets(['12(abc)34', '&($#%)xy', '(-)']) returns 'abc$#%-'.
* combine\_content\_in\_brackets(['(A)[B]', '{#}(\*\*)']) returns 'A\*\*'. (Note that square brackets, curly brackets, etc. are not considered.)
* combine\_content\_in\_brackets(['<<()>>', ' (1) ']) returns '1'. (Note that sometimes the brackets can be empty inside.)
* combine\_content\_in\_brackets([]) returns ''. (If the list is empty, the function returns an empty string.)

**Part (b)**

Define a function called add\_numbers\_in\_brackets. The function takes in a parameter that is the same as described in Part (a). In addition, each pair of round brackets contains a positive integer number inside rather than an arbitrary string. The function extracts these numbers and return their sum.

Here are some examples:

* add\_numbers\_in\_brackets(['1(15)3', '&(22)xy', '(9)']) returns 46. (Note that 15 + 22 + 9 = 46.)
* add\_numbers\_in\_brackets(['(8)[10]', '{#}(3)']) returns 11.
* add\_numbers\_in\_brackets([]) returns 0. (If the list is empty, the function returns 0.)

**Q2: List of Numbers [ \*\* ]**

**Part (a)**

Define a function called get\_larger\_values(). This function takes in a list of float values as its parameter. The function **returns** a new list of float values, which are those values from the original list which are above the average of the values in the original list. The original list should remain unchanged.

You can assume that the original list contains at least one value.

For example, get\_larger\_values([2.5, 3.5, 5.5, 1.0]) should return the list [3.5, 5.5], because the average of the values in the original list is 3.125 ( i.e., (2.5 + 3.5 + 5.5 + 1.0) / 4 = 3.125), and 3.5 and 5.5 are the two values in the original list that are above 3.125.

**Part (b)**

Define a function called merge\_list(). The function takes in two lists of numbers as its two parameters. It merges the two lists by alternating between the two lists to take their values one by one and inserting these values into a new list. It then returns the new list. The two original lists should remain unchanged.

The two lists may not be of the same length. The extra elements from the longer list are added to the returned list without merging with any element from the shorter list.

For example, merge\_list([1, 3, 10, 15, 4, 7, 12], [9, 5, 2]) returns the list [1, 9, 3, 5, 10, 2, 15, 4, 7, 12]. Note that 15, 4, 7 and 12 (the extra elements from the longer list) are included in the returned list at the end.

**Part (c)**

Define a function called get\_divisible\_pairs(). The function takes in two lists of positive integers (called int\_list\_1 and int\_list\_2) as its two parameters. You can assume that both lists contain only positive integers, and in each list, there are no duplicate numbers. The function returns a list of all tuples(m, n), where m is from int\_list\_1, n is from int\_list\_2, and m is divisible by n.

If either int\_list\_1 or int\_list\_2 is empty, the function returns an empty list.

Here are some examples:

* If int\_list\_1 is [9, 15, 12] and int\_list\_2 is [2, 3, 5], then get\_divisible\_pairs(int\_list\_1, int\_list\_2) returns

[(9, 3), (15, 3), (15, 5), (12, 2), (12, 3)].

* If int\_list\_1 is [10, 100] and int\_list\_2 is [10, 20, 40, 50], then get\_divisible\_pairs(int\_list\_1, int\_list\_2) returns

[(10, 10), (100, 10), (100, 20), (100, 50)].

**Q3: Movies [ \*\* ]**

*(Note that this question is the same as Q2 in Week 5’s extra in-class exercises.)*

Take a look at the test script movies\_test.py that’s given to you. Inside the file, you are given a list of tuples, where each tuple represents the title, genre and duration (in minutes) of a movie.

Inside another file called movies\_utility.py, implement the following functions:

* get\_title\_of\_longest\_movie(): This function takes in a list of tuples as described above and returns the title of the movie whose duration is the longest among all the movies in the list. If the list is empty, the function returns an empty string.
* get\_movies\_with\_keyword(): This function takes in a list of tuples as described above together with a string that represents a keyword. The function returns a new list of tuples where each tuple still represents a movie. The returned list contains those movies in the original list whose titles have the specified keyword as a substring. If the original list is empty, the function returns an empty list.
* get\_average\_duration(): This function takes in a list of tuples as described above. The function returns the average duration of all the movies in the list. If the list is empty, the function returns 0.0.
* get\_num\_movies\_of\_genre(): This function takes in a list of tuples as described above together with a string that indicates a genre. The function returns the number of movies belonging to that genre in the give list. If the list is empty, the function returns 0.