**Week 7 In-Class Exercises (Extra)**Part I: Strings and Loops

**Q1: Encode Strings [ \*\*\* ]**

Write a function called encode\_message() that takes in a string as its only parameter. Call this parameter text. You can assume that text doesn’t contain any space. The function ***returns*** a string that encodes text in the following way: for each segment of consecutive characters that are the same in text, the segment will be encoded as the character followed by the number of times it appears in the segment and a space. For example, 'aaa' is encoded as 'a3 ', 't' is encoded as 't1 '. The entire string text is encoded by encoding all its segments.

For example,

* encode\_message('aaabbcccccde') returns 'a3 b2 c5 d1 e1'.
* encode\_message('112333&&$9999999999') returns '12 21 33 &2 $1 910'.
* encode\_message('') returns ''.

**Q2: Print Patterns [ \*\*\* ]**

1. Define a function called print\_diamond\_using\_str(). The function takes in a string text as its parameter. The function prints a diamond shape using characters from text sequentially, starting from the top of the diamond.

You can assume that

* text doesn’t contain any space,
* text has at least 4 characters,
* the length of text is a multiple of 4.

For example,

* print\_diamond('12345678') prints

1  
 2 8  
 3 7  
 4 6  
 5

* print\_diamond('opqrstuvwxyz') prints

o  
 p z  
 q y  
r x  
 s w  
 t v  
 u

1. Define a function called print\_squares(). The function takes in an integer n as its parameter. You can assume that n is at least 3 and (n-3) is always a multiple of 4. The function prints out the following kind of pattern, which always consists of squares.

For example,

* print\_squares(3) prints

\*\*\*  
\* \*  
\*\*\*

* print\_squares(7) prints

\*\*\*\*\*\*\*  
\* \*  
\* \*\*\* \*  
\* \* \* \*  
\* \*\*\* \*  
\* \*  
\*\*\*\*\*\*\*

* print\_squares(11) prints

\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* \*\*\*\*\*\*\* \*  
\* \* \* \*  
\* \* \*\*\* \* \*  
\* \* \* \* \* \*  
\* \* \*\*\* \* \*  
\* \* \* \*  
\* \*\*\*\*\*\*\* \*  
\* \*  
\*\*\*\*\*\*\*\*\*\*\*

Part II: Lists

**Q1: Matrix Transpose and Matrix Multiplication [ \*\*\* ]**

A matrix can be represented as a list of lists of numbers.

For example, given the following matrix:

We can use the following list to represent it:

[[1.0, 2.0, 1.5], [2.5, 3.0, 2.0], [4.5, 1.5, 2.5]]

You can see that each element of the list above corresponds to a single row of the matrix.

**Part (a)**

The transpose of a matrix changes its row vectors into column vectors. For example, the transpose of the matrix above is the following:

Define a function call get\_matrix\_transpose() that takes in a list that represents a matrix as its parameter. The function returns a new list that represents the transpose of the original matrix.

Note that the input matrix can be of any dimension , where may be different from .

For example:

* get\_matrix\_transpose([[1.0, 2.0, 1.5], [2.5, 3.0, 2.0], [4.5, 1.5, 2.5]]) should return [ [1.0, 2.5, 4.5], [2.0, 3.0, 1.5], [1.5, 2.0, 2.5] ].
* get\_matrix\_transpose([[1.0, 2.0], [2.5, 3.0], [4.5, 1.5]]) should return [ [1.0, 2.5, 4.5], [2.0, 3.0, 1.5] ].

**Part (b)**

An matrix can be multiplied by an matrix. See the definition of matrix multiplication below:

<https://en.wikipedia.org/wiki/Matrix_multiplication#Definition>

Define a function called multiply\_matrices(). The function takes in two lists, each representing a matrix. You can assume that the number of columns of the first matrix is always equal to the number of rows of the second matrix. (Otherwise it’s not possible to multiply the two matrices.) The function returns the result of matrix multiplication

For example,

So multiply\_matrices( [[1.0, 2.0, -1.0], [2.5, 1.5, 0.5]], [[2.0, 2.0], [1.5, 1.0], [0.5, 2.0]] ) should return [[4.5, 2.0], [7.5, 7.5]].

**Hint:**

* You might want to take the transpose of the second matrix first.
* You might want to introduce a function that calculates the dot product (<https://en.wikipedia.org/wiki/Dot_product>) of two vectors.

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

Implement a function called flatten(). The function takes in a list called my\_list as its parameter, which contains numbers and possibly lists as its elements. These lists also contain numbers and possibly lists as their elements. This can go on recursively.

For example, my\_list may be

[3, [4, 8, 0], 5]

or

[4, 39, [35, 12, [45], 32], 4, [35, [4, [6]]]]

The function returns a list of numbers that is flattened from the original list.

For example, flatten([4, 39, [35, 12, [45], 32], 4, [35, [4, [6]]]]) returns [4, 39, 35, 12, 45, 32, 4, 35, 4, 6].

**Hint:**

* To check if a variable if of a certain date type, you can use the isinstance() function. See <https://docs.python.org/3.6/library/functions.html#isinstance> .

For example, to check if the variable x is a list, you can use isinstance(x, list), which will return either True or False.

* You can write a ***recursive function*** to solve the problem.