Lab Test 1 [20 marks]

Coding time: 1 hour and 30 mins

General Instructions:

1. You will perform the lab test on your personal laptop.

2. You are not allowed to communicate with anyone or access any network during the test. Disable the following connections on your laptop before the test begins: Wi-Fi, Bluetooth, and any other communication devices (e.g. 3G/4G modems).

4. You may refer to any file on your laptop during the test.

5. Make sure your code can generate exactly the same output as we show in the sample runs. You may be penalized for missing spaces, missing punctuation marks, misspelling, etc. in the output.

6. Do not hardcode. We will use different test cases to test and grade your solutions.

7. Follow standard Python coding conventions (e.g. naming functions and variables).

8. Python script file that cannot be executed will NOT be marked and hence you will be awarded 0 marks. You may wish to comment out the parts in your code which cause execution errors.

9. Include your name as author in the comments of all your submitted source files. For example, include the following block of comments at the beginning of each source file you need to submit.

```
# Name: Lum Ting Wong
# Email ID: lum.ting.wong.2017
```

Instructions on how to submit your solutions:

1. Before the test begins, a thumb-drive will be issued to you. It will contain a folder called `<your-email-id>` with all the resources required for this test. **You should rename the folder to your email id.** For example, if your SMU email is lum.ting.wong.2017@sis.smu.edu.sg, you should rename the folder to lum.ting.wong.2017. You need to save your solutions to the same thumb-drive. You may be penalized for not following our instructions.

2. When the test ends, your thumb-drive will be collected for assessment. Make a copy of your solution code on your desktop before submitting your thumb-drive. Only your solutions on the thumb-drive will be considered for assessment.

3. After all the thumb-drives have been collected, your invigilator will instruct you to enable your laptop's Wi-Fi and submit your solutions as a single zip file to eLearn Assignments. This copy serves as a backup and will not be marked.

Question 1 (Difficulty Level: *) **[4 marks]**

Implement the **get_longest_word** function in **q1.py**. This function takes three parameters: **`first`**, **`second`** and **`third`**, and it will return the longest word. **Note**: You can assume that the three parameters will be of different lengths.

E.g. 1: If the function is invoked like this:

```
print(get_longest_word('', 'a', 'abc'))
```

the statement generates the following output:

```
abc
```

Note: The three words are of length 0, 1 and 3, respectively. 3 is the largest value.

E.g. 2: If the function is invoked like this:

```
print(get_longest_word('abcd', 'a', 'ab'))
```

the statement generates the following output:

```
abcd
```

E.g. 3: If the method is invoked like this:

```
print(get_longest_word('orange', 'strawberry', 'x'))
```

the statement generates the following output:

```
strawberry
```

Question 2 (Difficulty Level: **) **[4 marks]**

On your phone keypad, every letter is mapped to a digit. The mapping is as follows:

Letters	Digit
ABC	2
DEF	3
GHI	4
JKL	5
MNO	6
PQRS	7
TUV	8
WXYZ	9

Implement the **generate_digits** function in **q2.py**. The **generate_digits** function takes in a string (i.e., str data type), **sentence**. It will return a str value of the digits mapped from the letters in **sentence**. For example, if **sentence** is `'PYTHONIC'`, the string `'79846642'` will be returned. You can assume that the parameter **sentence** contains only uppercase letters.

E.g. 1: If the function is invoked like this:

```
print(generate_digits('A'))
```

the statement generates the following output:

```
2
```

E.g. 2: If the function is invoked like this:

```
print(generate_digits('PYTHONIC'))
```

the statement generates the following output:

```
79846642
```

E.g. 3: If the function is invoked like this:

```
print(generate_digits('LEDLIGHT'))
```

the statement generates the following output:

```
53354448
```

Question 3 (Difficulty Level: **) **[4 marks]**

Implement the **reverse_num** function in **q3.py**. The **reverse_num** function takes in a number, **num**, as its parameter. It will return an **int** value transformed from **num** whereby the first digit becomes the last digit, the second digit becomes the second last digit, etc.

E.g. 1: If the function is invoked like this:

```
print(reverse_num(7102))
```

the statement generates the following output:

```
2017
```

E.g. 2: If the function is invoked like this:

```
print(reverse_num(230))
```

the statement generates the following output:

```
32
```

E.g. 3: If the function is invoked like this:

```
print(reverse_num(0))
```

the statement generates the following output:

```
0
```

E.g. 4: If the function is invoked like this:

```
print(reverse_num(-12))
```

the statement generates the following output:

```
-21
```

Note: The position of sign doesn't change for negative inputs.

Question 4 (Difficulty Level: *)** **[4 marks]**

Implement the **find_tags** function in **q4.py**. This function takes one parameter:

- **sentence**, of str type

This function will return a string that contains all the substrings enclosed within square brackets in **sentence** , i.e., between '[' and ']'. Each substring is prefixed by a sequence number followed by the '-' symbol. (See examples below.)

You can assume that inside **sentence** the square brackets are always properly paired, i.e., there would not be any missing **'['** or missing **']'**. You can also assume that there are no nested square brackets, i.e., you would not encounter cases such as **"[A[BC]]DEF"**.

E.g. 1: If the function is invoked like this:

```
print(find_tags('Coding[Rocks]'))
```

the statement generates the following output:

```
1-Rocks
```

Note:

- '1-' is prefixed before 'Rocks' because it is the 1st substring enclosed within square brackets encountered.

E.g. 2: If the function is invoked like this:

```
print(find_tags('[apple]and[orange]and[apple]again!'))
```

the statement generates the following output:

```
1-apple,2-orange,3-apple
```

Note:

- '1-' is prefixed before 'apple' because it is the 1st substring enclosed within square brackets encountered.
- '2-' is prefixed before 'orange' because it is the 2nd substring enclosed within square brackets encountered.
- '3-' is prefixed before apple because it is the 3rd substring enclosed within square brackets encountered.

E.g. 3: If the function is invoked like this:

```
print(find_tags('IAmTaking[mrt]To[SMU]ButThe[nel]IsDelayedAgain.'))
```

the statement generates the following output:

```
1-mrt,2-SMU,3-nel
```

The function should also handle empty strings and strings that do not contain any square brackets properly, as shown below:

E.g. 4: If the function is invoked like this:

```
print('>' + find_tags('') + '<')
```

the statement generates the following output:

```
><
```

E.g. 5: If the function is invoked like this:

```
print('>' + find_tags('apple') + '<')
```

the statement generates the following output:

```
><
```

Question 5 (Difficulty Level: **) [4 marks]**

Implement the **print_triangle** method in **q5.py**. The **print_triangle** function takes in one parameter, **sentence** of **str** type. The function **prints** a triangle on the screen and also **returns** a value. It returns **True** if the triangle is printed successfully. Otherwise, this function does not print anything and returns **False**.

E.g. 1: If the function is invoked like this:

```
result = print_triangle('abcdefghijkl')
print ('Return value:' + result)
```

the statement generates the following output:

```
   a
  b l
 c   k
defghij
Return value:True
```

E.g. 2: If the function is invoked like this:

```
print(print_triangle('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'))
```

the statement generates the following output:

```
          A
         B 9
        C   8
       D     7
      E       6
     F         5
    G           4
   H             3
  I               2
JKLMNOPQRSTUVWXYZ01
True
```

E.g. 3: If the function is invoked like this:

```
print(print_triangle('abcdefghij'))
```

the statement generates the following output:

```
False
```

Note:

- The string **'abcdefghij'** cannot be used to properly construct a triangle as shown above because of its length. Therefore **False** is returned.

E.g. 4: If the function is invoked like this:

```
print(print_triangle('abc'))
```

the statement generates the following output:

```
False
```

Note:

- Again, it is not possible to construct a triangle using **'abc'** because of the length of this string.

E.g. 5: If the function is invoked like this:

```
print(print_triangle ('abcd'))
```

the statement generates the following output:

```
 a
bcd
True
```