Lab Test 1 [25 marks]

Coding time: 1 hour and 30 mins

Instructions on how to download the resource files:

1. Download from http://blue.smu.edu.sg/is111/2019/1920-LT1.zip

General Instructions:

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

2. You are not allowed to communicate with anyone or access any network during the test. After downloading the resource files, 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, misspellings, etc. in the output.

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

7. Follow IS111 code conventions (e.g. naming functions and variables) or risk a 3 mark penalty on your final score.

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    : BAI She Jing
# Email ID: shejing.bai.2019
```

Instructions on how to submit your solutions:

1. When the test ends, zip up all the files required for submission in a zip archive. The name of the zip archive should be your email ID. For example, if your email is `shejing.bai.2019@sis.smu.edu.sg`, you should name the archive as **`shejing.bai.2019.zip`**. You may be penalized for not following our instructions.

2. Once everybody has zipped up his/her files, your invigilator will instruct you to enable your laptop's Wi-Fi and submit your solutions as a single zip file to eLearn Assignments.

Question 1 (Difficulty Level: *) **[6 marks]**

Implement a function called **`repeat`** in **`q1.py`**.

- This function takes 2 parameters: **`word`** (type:**`str`**), **`n`** (type: **`int`**).
- The function returns a string with all the characters in word repeated **`n`** times.

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

```
print(repeat('Hello!', 3))
```

the statement generates the following output:

```
HHHeeellllllooo!!!
```

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

```
print('>' + repeat('apple', 0) + '<')
```

the statement generates the following output:

```
><
```

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

```
print('>' + repeat('', 3) + '<')
```

the statement generates the following output:

```
><
```

Question 2 (Difficulty Level: **) **[6 marks]**

Implement a function called **`conditional_sum`** in **q2.py**.

- This function takes three parameters:
  - **`start`** (type: **`int`**).
  - **`end`**(type: **`int`**). Assume **`end`** is larger than **`start`**.
  - **`multiple`**(type: **`int`**). It is positive.
- The function ***returns a string*** that contains the following:
  - the addition of all the multiples between **`start`** (inclusive) and **`end`** (inclusive) and
  - the sum (of all the multiples)

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

```
print(conditional_sum(1, 10, 3))
```

the statement generates the following output:

```
3 + 6 + 9 = 18
```

Note:

1. There are 10 numbers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) between 1 and 10. Out of the 10 numbers, only 3, 6 and 9 are divisible by 3 (i.e. multiples of 3).
2. The integers 3, 6, and 9 are multiples of parameter 3. Their sum is 18.

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

```
print('>' + conditional_sum(1, 10, 11) + '<')
```

the statement generates the following output:

```
><
```

Note: There are no integers within the range 1 to 10 that is a multiple of 11.

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

Implement a function called **`get_common_chars`** in **q3.py**.

- The **`get_common_chars`** function takes in two parameters:
  - **`first`** (type: `str`)
  - **`second`** (type: `str`)

  You can assume that **`first`** and **`second`** are strings which are non-`None` value.
- This function ***returns*** the characters that are common to **`first`** and **`second`** *preserving the order that* these *characters appear in* **`first`**.

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

```
print(get_common_chars('apple', 'pear'))
```

the statement generates the following output:

```
ape
```

Note:

1. 'p' appears only once in 'pear' and hence even though it appears twice in 'apple', the output has only 1 'p'
2. The characters that appear in the output 'a', 'p' and 'e' appear in the same order from left to right in 'apple' (i.e. the order is preserved).

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

```
print(get_common_chars('bubble', 'rubble'))
```

the statement generates the following output:

```
buble
```

Note: bubble has 3 `'b'`, rubble has 2 `'b'`, hence only 2 `'b'` are common.

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

```
print(get_common_chars('kingkong', 'kiwiking'))
```

the statement generates the following output:

```
kingk
```

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

```
print('>' + get_common_chars('kiwi', 'mango') + '<')
```

the statement generates the following output:

```
><
```

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

Implement a function called **`get_similarity_level`** in **q4.py**.

- This function takes in 2 parameters (type: **`str`**) called **`string1`** and **`string2`** and compares the strings. You can assume that **`string1`** and **`string2`** are strings which are non-`None` value.
- The function ***returns a string*** that is either **`'same`**, **`'close'`**, '**`somewhat`**', **`'weak'`** or **`'different'`**, based on the conditions indicated in the table below.
- The function returns a string that is
  - o same if the strings have the same characters and case in the same position.
  - o close if the strings are exactly the same ignoring the case and are at the same position
  - o somewhat if the strings have exactly the same numeric characters('0' -'9') and alpha characters ignoring the case('a' - 'z', 'A' - 'Z'). They can be at different positions.
  - o weak if the strings have the same count of numeric characters and alpha characters.
  - o different if the above cases do not match

`string1`	**`string2`**	Return Value	Explanation
`'hello'`	**`'applet'`**	**`'different'`**	different number of characters (5 versus 6 characters)
`'bye'`	**`'2be'`**	**`'different'`**	different number of digits and alphabets (0 digit, 3 letters versus 1 digit, 2 letters)
`'ape 123'`	**`'cape123'`**	**`'different'`**	3 digit, 3 letters versus 3 digit, 4 letters
`'be3!'`	**`'####2Me#######'`**	**`'weak'`**	same number of digits and alphabets (1 digit, 2 letters)
`'be3'`	**`'2Me'`**	**`'weak'`**	same number of digits and alphabets (1 digit, 2 letters)
`'123-hello'`	**`'HELLO123!'`**	**`'somewhat'`**	same digits and alphabets of different case with different non-alphanumeric characters.
`'123-hello'`	**`'hello123!'`**	**`'somewhat'`**	same digits and alphabets with different non-alphanumeric characters.
`'hello123'`	**`'Hello123'`**	**`'close'`**	same number of alphabets and digits but the case differs.
`'hello'`	**`'HelLo'`**	**`'close'`**	same number of alphabets but the case differs.
`'hello'`	**`'hello'`**	**`'same'`**	Two strings (all characters) are the same.

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

```
print(get_similarity_level('hello', 'HelLo'))
```

the statement generates the following output:

```
close
```

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

```
print(get_similarity_level('be1', '2Me'))
```

the statement generates the following output:

```
weak
```

Question 5 (Difficulty Level: *)** **[3 marks]**

Implement the **print_number_pattern** function in **q5.py**. The function takes in 2 parameters:

- **`level`** (type:**`int`**). It is a positive value.
- **`start`** (type:**`int`**).

The function will print a triangle using numbers starting from **start**(in the last row) in zig-zag order:

1. 1 character on the 1st line,
2. 2 characters on the 2nd line (right to left)
3. 3 characters on the 3rd line (left to right)
4. 4 characters on the 4th line (right to left)
5. ....

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

```
print_number_pattern(3, 1)
```

the statement generates the following output:

```
6
4|5
3|2|1
```

Note: There is a '|' character separating the 2 numbers.

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

```
print_number_pattern(3, 8)
```

the statement generates the following output:

```
13
11|12
10|*9|*8
```

Note: Each number is formatted to a width of 2 since '10' has 2 characters. Thus '*8' instead of '8'.

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

```
print_number_pattern(4, 8)
```

the statement generates the following output:

```
17
16|15
12|13|14
11|10|*9|*8
```

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

```
print_number_pattern(9, 0)
```

the statement generates the following output:

```
44
42|43
41|40|39
35|36|37|38
34|33|32|31|30
24|25|26|27|28|29
23|22|21|20|19|18|17
*9|10|11|12|13|14|15|16
*8|*7|*6|*5|*4|*3|*2|*1|*0
```

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

```
print_number_pattern(3, -100)
```

the statement generates the following output:

```
*-95
*-97|*-96
*-98|*-99|-100
```

Note: Each number is formatted to a width of 4 since '-100' has 4 characters. Thus '*-95' instead of '-95'.

Question 6: (Difficulty Level: *) [3 marks]**

Implement the **`transform`** function in **q6.py**.

- The function has 2 parameters:
  - **`str1`**(type: **`str`**). The characters in **`str1`** are unique.
  - **`str2`**(type: **`str`**)**.**

- The function **returns** a string which represents a sequence of transformations that can turn **`str1`** to **`str2`** by swapping two *adjacent* characters every time.

Note:

1. You can assume that **`str1`** can always be turned into **`str2`** after a sequence of swaps of adjacent characters, i.e., **`str1`** and **`str2`** contain the same set of characters.
2. You are not allowed to use **`while`** loop (Not taught yet)

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

```
print(transform('ABC', 'CBA'))
```

the statement generates the following output:

```
ABC-BAC-BCA-CBA
```

Note:

1. It shows the series of transformations separated by a dash (`'-'`).

2. Each transformation is done by swapping two adjacent characters.

   ABC (swap A & B) -> B**AC** (swap A & C) -> **BC**A (swap B & C) -> CBA

   `'A'` reaches the correct position first, followed by `'B'`, then `'C'` (in the order of its appearance in **`str2`** from right to left).

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

```
print(transform('ABCDE', 'DBECA'))
```

the statement generates the following output:

```
ABCDE-BACDE-BCADE-BCDAE-BCDEA-BDCEA-BDECA-DBECA
```

END

EMPTY PAGE