Lab Test 1 [25 marks]

Coding Time: 1.5 hours

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 website (except for downloading/uploading files from/to eLearn) during the test. **Anyone caught communicating with another person or accessing any website during the test WILL be considered as cheating and subjected to disciplinary actions.**

3. You are not allowed to use any extensions to IDEs that uses AI to automatically suggest code when you type. Examples include Github Copilot and IntelliCode. Please disable these extensions before you start the test.

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 files 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.2022
```

Instructions on how to submit your solutions:

1. Before the test begins, you will be instructed to download the resource files of the test from eLearn.

2. You will be given a password to unzip the downloaded .zip file. After unzipping the file, **you should rename the folder "your_email_id" to your actual email id.** For example, if your SMU email is lum.ting.wong.2022@scis.smu.edu.sg, you should rename the folder to **lum.ting.wong.2022**. You need to save your solutions directly inside this folder for submission later. You may be penalized for not following our instructions.

3. When the test ends, you will be instructed to submit your solutions as a single zip file to eLearn.

Question 1 (Difficulty Level: *) [7 marks]

Implement a function called **`get_promotion`** in **`q1.py`**, to get recent sale promotions in NTUC FairPrice.

- This function takes a parameter (data type: `str`) called **`food_category`**.
- The function ***returns a string*** that is either a promotion rate or a message of "No Promotion", based on the value of **`food_category`**. The mapping from **`food_category`** to the returned value is pre-defined and given in the following table:

Food category	Return Value
`'vegetable'` (or **`'vegetables'`**)	**`'5%'`**
`'fruit'` (or **`'fruits'`**)	**`'7.5%'`**
`'seasoning'` (or **`'seasonings'`**)	**`'3%'`**
`'flour'` (or **`'flours'`**)	**`'10%'`**
any other values including typos (e.g., **`'seasoningss'`**, **`'frui'`**, **`'seafood'`**, **`'apple'`**, **`'tableware'`**)	**`'No Promotion'`**

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

```
print(get_promotion('vegetable'))
```

the statement generates the following output:

```
5%
```

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

```
print(get_promotion('fruits'))
```

the statement generates the following output:

```
7.5%
```

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

```
print(get_promotion('seasoningss'))
```

the statement generates the following output:

```
No Promotion
```

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

```
print(get_promotion(' '))
```

the statement generates the following output:

```
No Promotion
```

Use **`q1_test.py`** to test your code. You should not modify `q1_test.py`. Your `q1.py` should contain only the function definition of `get_promotion`. We will only grade your `q1.py` and we will use a different test script for grading.

Question 2 (Difficulty Level: **)

Part (a) [5 marks]

Implement a function called **`trim_number`** in **`q2a.py`**. This function takes in two parameters called **`num1`** (type: `int`) and **`num2`** (type: `int`) respectively.

- **`num1`** is a positive integer (i.e., `num1` > 0).
- **`num2`** is a single-digit positive integer (i.e., an integer between 1 and 9, both inclusive).

The function will ***return an integer*** value transformed from **`num1`** whereby the last digit (the right most digit) is replaced by **`num2`**, if the value of the transformed number is *strictly smaller than* the value of the original **`num1`**; otherwise, it will ***return an integer*** with the value of **`num1`**.

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

```
print(trim_number(1000, 5))
```

the statement generates the following output:

```
1000
```

This is because if we replace the last digit of `num1` (which is 1000 here) with `num2` (which is 5 here), we will get a transformed number 1005, which is NOT strictly smaller than 1000. Therefore, the function returns the original value of `num1`, which is 1000.

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

```
print(trim_number(5007, 2))
```

the statement generates the following output:

```
5002
```

This is because if we replace the last digit of `num1` (which is 5007 here) with `num2` (which is 2 here), we will get a transformed number 5002, which is strictly smaller than 5007. Therefore, the function returns the transformed number.

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

```
print(trim_number(905, 5))
```

the statement generates the following output:

```
905
```

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

```
print(trim_number(1, 5))
```

the statement generates the following output:

```
1
```

Hint: *One possible solution* is to turn both `num1` and `num2` into strings first. Then we can use *string slicing* to obtain the front part of `num1` without its last digit. This front part can then be combined with `num2` into a new string that represents the transformed number. E.g., if `num1` is `1000` and `num2` is `5`, we can turn them into `'1000'` and `'5'`. Using string slicing, we can get the front part of `'1000'`, which is `'100'`. We can then combine `'100'` with `'5'` to obtain `'1005'`.

Use **`q2a_test.py`** to test your code.

Part (b) **[3 marks]**

Implement a function called **`trim_number_with_list`** in **`q2b.py`**. This function takes in two parameters called **`num`** (type: `int`) and **`num_list`** (type: `list`), respectively.

- **`num`** is always a positive integer (i.e., `num` > 0).
- **`num_list`** is always a list of single-digit positive numbers and contains at least one number.

The function will ***return an integer*** value transformed from **`num`** whereby the last digit (the right most digit) is replaced by the first number in **`num_list`** that makes the value of the transformed number strictly smaller than the value of the original **`num`**; otherwise, it will ***return an integer*** with the value of **`num`**.

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

```
print(trim_number_with_list(1006, [7]))
```

the statement generates the following output:

```
1006
```

This is because the one and only number in `num_list` does not make the transformed number strictly smaller than the original `num`.

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

```
print(trim_number_with_list(1006, [3]))
```

the statement generates the following output:

```
1003
```

This is because the one and only number in `num_list` makes the transformed number strictly smaller than the original `num`.

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

```
print(trim_number_with_list(1004, [7, 9, 2, 3]))
```

the statement generates the following output:

```
1002
```

This is because the first number in `num_list` that makes the transformed number strictly smaller than the original `num` is 2 (i.e., the 3rd element in `num_list`).

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

```
print(trim_number_with_list(1009, [1, 0, 2]))
```

the statement generates the following output:

```
1001
```

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

```
print(trim_number_with_list(1005, [7, 5, 8, 6, 9]))
```

the statement generates the following output:

```
1005
```

Hint: If you wish, you can call the function `trim_number` defined in **`q2a.py`** to help you with this part **by importing `q2a`**. Note that if your `q2b.py` works correctly with the correct implementation of the function `trim_number` in `q2a.py`, then even if your own implementation of `trim_number` is not correct, you may still get marks for part (b). (We will run our grading script with your `q2b.py` together with our correct implementation of `q2a.py`.)

Use **`q2b_test.py`** to test your code.

Question 3:

Part (a) **(Difficulty Level: **)** **[2.5 marks]**

Inside **`q3a.py`**, define a function called **`shift_string`**. The function takes in a single parameter (data type: `str`) called `orig_str`. Assume that `orig_str` has `n` characters. The function returns a list of `n` strings, where the string in the list at index `k` is a string constructed by shifting the characters in `orig_str` to the left by `k`, and characters pushed out on the left-hand side are appended to the right.

For example, if `orig_str` is `'ABCDE'`, then the string at index 0 in the returned list is still `'ABCDE'`, the string at index 1 is `'BCDEA'`, the string at index 2 is `'CDEAB'`, and so on.

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

```
print(shift_string('ABCDE'))
```

the statement generates the following output:

```
['ABCDE', 'BCDEA', 'CDEAB', 'DEABC', 'EABCD']
```

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

```
print(shift_string('1234567'))
```

the statement generates the following output:

```
['1234567', '2345671', '3456712', '4567123', '5671234', '6712345', '7123456']
```

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

```
print(shift_string(''))
```

the statement generates the following output:

```
[]
```

Note that because an empty string has 0 character, the function therefore returns a list with 0 element.

Use **`q3a_test.py`** to test your code.

Part (b) **(Difficulty level: **)** **[2.5 marks]**

Please note that part (b) of Q3 is not related to part (a) of Q3.

Inside **`q3b.py`**, implement a function called `construct_string()`. The function takes in three parameters:

- `orig_str`: A string
- `len_list`: A list of `n` positive integers
- `cnct_list`: A list of `(n-1)` strings

The function will divide `orig_str` into `n` segments, where the length of the `k`-th segment is `len_list[k-1]`. The function then connects these segments using the strings in `cnct_list` sequentially as connectors. The function returns the final string.

You can assume that the sum of the integers in `len_list` is the same as the length of `orig_str`.

Note that `orig_str` can be an empty string, and in this case, both `len_list` and `cnct_list` will be empty, and the function should return an empty string.

Examples:

- When
  ```
        orig_str = 'ABCDEFGHIJKLMN'
        len_list = [2, 5, 3, 1, 2, 1]
        cnct_list = ['-', '$', '', '123', ' ']
  ```
  the following function call
  ```
        construct_string(orig_str, len_list, cnct_list)
  ```
  returns `'AB-CDEFG$HIJK123LM N'`

  To see why the function returns the string above, first, we can see that the original string `'ABCDEFGHIJKLMN'` is divided into 6 segments (where 6 is the length of `len_list`):
  ```
        'AB', 'CDEFG', 'HIJ', 'K', 'LM', 'N'
  ```
  Note that the lengths of these 6 segments are 2, 5, 3, 1, 2 and 1, which are exactly the elements in `len_list`.

  Then, these segments are further combined using the 5 strings in `cnct_list`, where the 1st string in `cnct_list` is used to connect the 1st and the 2nd segments, the 2nd string in `cnct_list` is used to connect the 2nd and the 3rd segments, etc. Note that it is possible for a string in `cnct_list` to be empty, and in this case, there will not be any connector between the two corresponding segments, which is the case for `'HIJ'` and `'K'` in the example above.

-
  ```
  orig_str = '*******'
  len_list = [3, 3, 1]
  cnct_list = ['1', '2']

  construct_string(orig_str, len_list, cnct_list)
  ```
  returns `'***1***2*'`

Use **`q3b_test.py`** to test your code.

Question 4: Rectangles [5 marks]

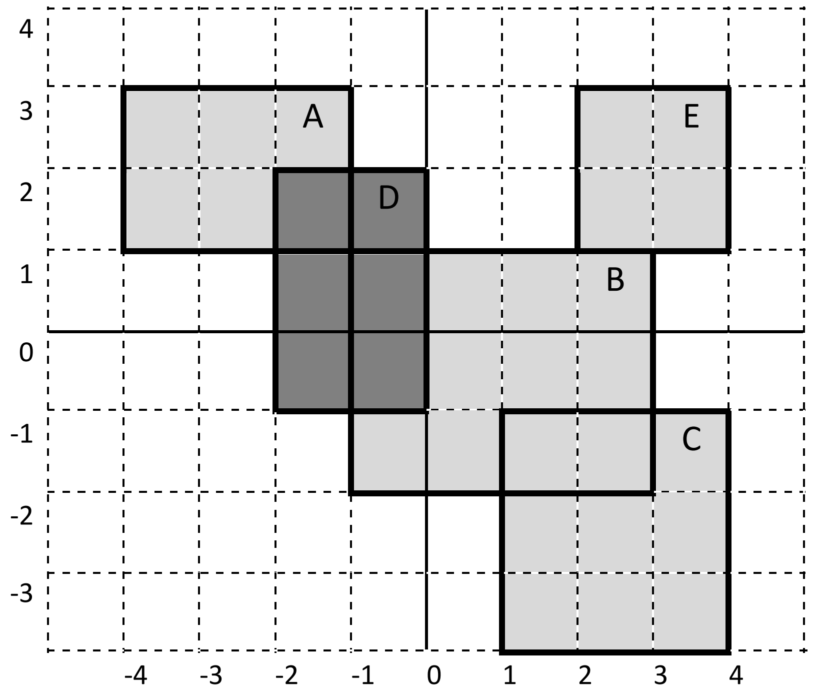
Part (a) **(Difficulty Level: **)**

In **`q4a.py`**, define a function called `is_overlapping()`. The function takes in two parameters called `rect_1` and `rect_2`. Each parameter represents a rectangle whose four sides are parallel to either the x-axis or the y-axis. To represent such a rectangle, we use a tuple consisting of 5 elements:

- a `str` representing the name of the rectangle.
- an `int` representing the x-coordinate of the *bottom-left* corner of the rectangle. Note that there is no lower or upper limit of this number, i.e., this number can be anything between -∞ (negative infinity) and ∞ (infinity).
- an `int` representing the y-coordinate of the *bottom-left* corner of the rectangle. Note that there is no lower or upper limit of this number, i.e., this number can be anything between -∞ (negative infinity) and ∞ (infinity).
- an `int` representing the width of the rectangle. Note that this is always a positive integer.
- an `int` representing the height of the rectangle. Note that this is always a positive integer.

For example,

- to represent the rectangle A in the diagram below, we will use `("A", -4, 1, 3, 2)`
- to represent the rectangle B, we will use `("B", -1, -2, 4, 3)`
- to represent the rectangle C, we will use `("C", 1, -4, 3, 3)`
- to represent the rectangle D, we will use `("D", -2, -1, 2, 3)`
- to represent the rectangle E, we will use `("E", 2, 1, 2, 2)`



The function `is_overlapping(rect_1, rect_2)` returns `True` if the two rectangles overlap and returns `False` if the two rectangles do not overlap.

For example,

- `is_overlapping( ("A", -4, 1, 3, 2), ("B", -1, -2, 4, 3) )` returns `False`
- `is_overlapping( ("B", -1, -2, 4, 3), ("C", 1, -4, 3, 3) )` returns `True`
- `is_overlapping( ("E", 2, 1, 2, 2), ("B", -1, -2, 4, 3) )` returns `False`

Use **`q4a_test.py`** to test your code.

Part (b) **(Difficulty Level: **)**

Inside **`q4b.py`**, define a function called `find_overlapping_pairs()`. The function takes in a list of rectangles called `rect_list`. Each element in the list is a tuple representing a rectangle as described above. You can assume that all the rectangles in the list have different names.

The function returns a list that contains *all* pairs of rectangles in `rect_list` that overlap with each other. The order of these pairs in the list does not matter. For the same pair of overlapping rectangles, only one tuple should be included in the list. E.g., if B and C overlap, then either `('B', 'C')` or `('C', 'B')` should be included in the returned list but not both.

For example, given the five rectangles above, the function should return `[('A', 'D'), ('B', 'D'), ('B', 'C')]` (or these three pairs of rectangles in different orders, such as `[('D', 'A'), ('B', 'D'), ('B', 'C')]` or `[('C', 'B'), ('D', 'B'), ('A', 'D')]`).

Note: If you wish, you can use the function `is_overlapping` defined in part (a) for part (b) **by importing `q4a`**. If your code in part (b) works with the correct implementation of `is_overlapping`, then even if your part (a) is incorrect, you may still get marks for part (b).

Use **`q4b_test.py`** to test your code.

Part (c) **(Difficulty Level: ***)**

Inside **`q4c.py`**, define a function called `find_overlap_size`. The function takes in two rectangles represented as described above. It returns an integer indicating how many squares of size 1 are in the overlapping area of the two rectangles, or 0 if the two rectangles do not overlap.

For example,

- given rectangle D and rectangle B above, the function should return 2
- given rectangle A and rectangle D above, the function should return 1
- given rectangle A and rectangle C above, the function should return 0

Use **`q4c_test.py`** to test your code.