ICE12: Trial Lab Test 2

Resource: https://blue.smu.edu.sg/is111/2020/Trial-LT2-2020.zip

1. **[Difficulty: *]** Implement the `first_unique_char` function. This function takes in 1 parameter: value (all lowercase characters), and returns the first character with exactly 1 occurrence (or None if there is no unique character).

2. **[Difficulty: *]** Implement a function called `get_sum_of_two`. This function takes 1 parameter:
    1. triplets(type:list). Each element in the list is a tuple that contains 3 integers.

   This function returns all the elements whereby one of the numbers is the sum of the other two numbers.

   For example,
   ```
       print(get_triplets([(2, -1, 1), (1, 0, 0), (-1, -1, 0)]))
   ```
   returns
   ```
       [(2, -1, 1), (-1, -1, 0)]
   ```
   Note:
    1. 2 + (-1) = 1
    2. (-1) + 0 = -1

3. **[Difficulty: **]** Implement the function `is_num_times()`. It accepts three parameters:
    1. s1 (type: str): This string does not contain duplicated characters.
    2. s2 (type: str): Another string
    3. n (type: int): A positive number.

   This function returns True if every character in str1 occurs exactly n times in str2. Otherwise it returns False.

4. **[Difficulty: **]** Write a function called add_two_list() that takes in two parameters:
    - values1 (type: list): a list of integers.
    - values2 (type: list): another list of integers

   This function returns a list of integers. The element at a particular index is formed from adding the two elements in the 2 parameter lists at the corresponding indices. For example, if `values1` is [1, 3] and `values2` is [9, 2], then the result will be [10, 5], i.e. the first element is formed by adding the two numbers (1 + 9 = 10, 3 + 2 = 5).

5. **[Difficulty: **]** Implement the create_diagonal_matrix function. It takes in one parameter, row_vector (type:list) and returns a diagonal matrix. if row_vector is [1, 2, 3], the return value would be
   ```
   [ [1, 0, 0],
     [0, 2, 0],
     [0, 0, 3] ]
   ```
   If row_vector is [7, 2, 3, 4, 5], the return value would be
   ```
   [     [7, 0, 0, 0, 0],
         [0, 2, 0, 0, 0],
         [0, 0, 3, 0, 0],
         [0, 0, 0, 4, 0],
         [0, 0, 0, 0, 5]
   ]
   ```

6. **[Difficulty: ***]** Implement the add_two_dict function. It accepts two parameters:
   - d1 (type: dict): key is of type str, value is of type int

- d2 (type: dict): key is of type str, value is of type int

It returns a new dictionary that represents a merged union of the two original dictionaries. For example, if d1 is {'a':1, 'b':2}, and d2 is {'a':3, 'b':5, 'c':7}, then this function should return {'a':4, 'b':7, 'c':7}. The result is a new dictionary that contains every key from d1 and every key from d2. Each value in the result dictionary should be the sum of the corresponding value(s) for the key in d1 and d2. For example, 'a' key will now have the value of 4 (1 + 3).

7. **[Difficulty: ***]** Implement the function `gray_code` that returns a list of Gray codes of a specified length. This function takes in a parameter, `n` which is the length of the gray code. A Gray code is a special type of binary code organized so that any two successive bit patterns differ by only one bit. For example, a 2-bit Gray code is `['00', '01', '11', '10'].`

   Note:
   1. '0**0**' and '0**1**' differ by a value of 1.
   2. '**0**1' and '**1**1' differ by a value of 1.
   3. '1**1**' and '1**0**' differ by a value of 1.

TRIAL PAPER 2

Resource:

Q1: https://tinyurl.com/is111-trial-LT2

For each of the questions below, implement a function inside a given file.

a) **[*]** Implement a function called `count_high_temperatures()` inside q1a.py. This function takes in a list of numbers representing a sequence of body temperature readings. The function returns the number of temperature readings in the input list that are **strictly greater** than 37.5.

- Example #1: `count_high_temperatures([36.9, 37.6, 37.2, 37.1, 38.1])` returns 2
- Example #2: `count_high_temperatures([])` returns 0
- Example #3: `count_high_temperatures( [37.9, 38.2, 38.8, 37.5, 37.3, 37.0])` returns 3

Use **q1a-test.py** to check your implementation.

b) **[**]** Implement a function called `create_email_dict()` inside q1b.py. This function takes in a list of strings representing people's names and their email addresses. In this list, a person's name is always followed by his/her email address. For example, the list may look like the following:

```
["Jack", "jack@gmail.com", "Mary", "mary.loh@smu.edu.sg", "John",
"john.smith@microsoft.com"]
```

You can assume that the input list is always properly constructed in the above format. You can also assume that people's names are unique in the list.

The function will re-organize the information stored in the input list so that each person's name is mapped to his/her email address. The function returns a dictionary that stores all the mappings.

For example, `create_email_dict(["Jack", "jack@gmail.com", "Mary", "mary.loh@smu.edu.sg", "John", "john.smith@microsoft.com"])` returns the following dictionary:

```
{'Jack': 'jack@gmail.com', 'Mary': 'mary.loh@smu.edu.sg', 'John':
'john.smith@microsoft.com'}
```

Use **q1b-test.py** to check your implementation.

Q2 [**]:

In the file q2.py, write a program that prompts the user for a valid smu email address. You can assume that the user enters a single word (a string without any space). Keep prompting the user until the input string contains the substring "@smu.edu.sg" at the end and no other '@' anywhere else. Also make sure that there is a username portion prefixing the substring @smu.edu.sg

A sample run of the program looks as follows. Text in bold font is user input.

```
Please enter your SMU email address:abc@gmail.com
Invalid!
Please enter a valid SMU email address:www@sis.smu.edu.sg
Invalid!
Please enter a valid SMU email address:ma@jack@smu.edu.sg
Invalid!
Please enter a valid SMU email address:@smu.edu.sg
Invalid!
Please enter a valid SMU email address:jack@smu.edu.sg
Thanks!
```

Q3:

a) **[*]** Implement a function called `retrieve_email()` in q3a.py. The function takes in **two parameters**:
   a. a **dictionary** that maps names to emails
   b. a person's name

   The function returns the email address of that person or `None` if the person cannot be found in the dictionary.

   Run **q3a-test.py** to test your implemented functions.

b) **[*]** Implement a function called `add_new_email()` q3b.py. The function takes in **two parameters**:
   a. a dictionary that maps names to emails
   b. a tuple of two elements, representing a person's name and his/her email address

   The function tries to add the tuple as a new (key, value) pair to the dictionary.
   - If the person already exists in the dictionary, the new value is used to replace the old value, and the old value is returned by the function.
   - If the person does not exist in the dictionary originally, the function adds the new (key, value) pair and returns `None`.

   Run **q3b-test.py** to test your implemented functions.

Q4 [***]:

You are given a file that contains multiple lines. Each line shows several groups of numbers. The numbers are separated by single spaces, and the groups are separated by the symbol '#'. For example, the file may look like the following:

```
10 20 30#100 300 200 400#-10 -20 0#1.0 3.0
1000 3000#15 45 60#5.0
```

Define a function called `process_numbers()` that takes in two parameters:

- The name of an input file as described above.
- The name of an output file.

The function should process the input file and write the following information to the output file:

- For each line of the input file, write the number of groups in that line to the output file.
- For each group of numbers in each line of the input file, calculate the average of the numbers in the group and write the average value to the output file. Use '#' to separate the average values of different groups.
- Each line of the input file has a corresponding line in the output file showing the information above.
- In the end of the output file, write the total number of groups found in the input file and the maximum average value among all the groups.

For example, given the file shown above, the output file should look like the following:

```
4: 20.0#250.0#-10.0#2.0
3: 2000.0#40.0#5.0
Total number of groups: 7
Maximum average: 2000.0
```

Q5: [*]**

In this question, you need to process a file that contains many documents. Each line of the file contains two columns separated by tabs. The first column is a string, which is the ID of a document. The second column is the content of the corresponding document, shown as a sequence of words separated by one or more whitespaces.

You can open "documents.txt" to see an example file of this format.

In q5.py, define a function called `get_document_pair()`. The function takes in the name of a file as its input. The function tries to find out which pair of two documents in the given file shares the largest number of common words.

For example,

```
D1      w1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W12 W13 W14   W15
D2      W10 W10 W1 W2 W3 W4 W5 W6 W7 W8 W9
D3      W2 W1 W3 W4 W5 W6 W7 W8 W9 W10 W10 W11 W13 W12 W14
W15
```

"D1" and "D2", "D2" and "D3" share 10 words in common, and "D1" and "D3" share 15 words in common (case-insensitive so 'w1' is equivalent to 'W1'), then the pair ("D1", "D3") has more common words than the pair ("D1", "D2").

When counting common words, if a word appears more than once in a document, it should be counted only once. For example, if the word "W10" appears 2 times in "D2" and 2 times in "D3", it should still be counted as 1 common word shared by "D2" and "D3".

The function should return a list of tuples . Each tuple consists of three elements: the IDs of the two documents and the number of common words they share. Therefore, it should return [('D1', 'D3', 15)]. If there are more than one set of documents with the largest number of common words, all pairs are returned.

OPTIONAL

Note: No resource is given for the optional questions

8. **[Difficulty: **]** In mathematics, the sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit. The algorithm is as follows:

   1. Create a list of consecutive integers from 2 through *n*: (2, 3, 4, ..., *n*).
   2. Initially, let *p* equal 2, the smallest prime number.
   3. Enumerate the multiples of *p* by counting to *n* from 2*p* in increments of *p*, and mark them in the list (these will be 2*p*, 3*p*, 4*p*, ...; the *p* itself should not be marked).
   4. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this new number (which is the next prime), and repeat from step 3.
   5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below *n*.

   Note: It is sufficient to mark the numbers in step 3 starting from p^2, as all the smaller multiples of p will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4 when p^2 is greater than n.

   For example, to find all the prime numbers less than or equal to 20:

   a. First, generate a list of integers from 2 to 20:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

   b. The first number in the list is 2; cross out all the multiples of 2 in the list.

2 3 **~~4~~** 5 **~~6~~** 7 **~~8~~** 9 **~~10~~** 11 **~~12~~** 13 **~~14~~** 15 **~~16~~** 17 **~~18~~** 19 **~~20~~**

   c. The next number in the list after 2 is 3; cross out all the multiples of 3 in the list (that have yet to be crossed out).

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ **~~9~~** ~~10~~ 11 ~~12~~ 13 ~~14~~ **~~15~~** ~~16~~ 17 ~~18~~ 19 ~~20~~

   d. The next number not yet crossed out in the list after 3 is 5. However, we can stop as 5^2 is greater than 20. The remaining numbers left are prime numbers (2,3,,5,7,11,13,17,19).

   Implement the function `sieve()` that takes in a parameter `n` and returns all the prime numbers smaller or equal to `n`.

   Reference: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

OUT OF SCOPE

Warning

We discourage you from learning this if you are having issues with tackling 2-star or 3-star questions. No partial marks will be awarded for any questions attempted using the concepts covered below. Use it wisely and carefully!

1. The `map` applies a function to all the items in a list, and returns a sequence.

   The syntax is

   ```
   return_value = map(fn, list)
   ```

   This is equivalent to a function as follows:

   ```
   def map_equivalent(fn, list):
       result = []
       for value in list:
           result.append(fn(value))
       return result
   ```

   For example, the following code:

```
>>> def map_equivalent(fn, list):
...     result = []
...     for value in list:
...         result.append(fn(value))
...     return result
...
>>> values = ['a','abc','bc']
>>> values = map_equivalent(len, values)
>>> values
[1, 3, 2]
```

can be replaced by the map function:

```
>>> values = ['a','abc','bc']
>>> values = map(len, values)
>>> list(values)
[1, 3, 2]
>>> values = ['a','abc','bc']
>>> values = map(len, values)
>>> for i in values: # since it returns a sequence, u can loop through it
...     print(i)
...
1
3
2
```

There is no exercise for Q1.

2. A `lambda` function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression. *Use it when you need a small piece of function that will be run just once.*

   The syntax is:

   ```
   lambda arg1, arg2, ...argN : expression(using arguments)
   ```

   This is equivalent to a function as follows:

   ```
   def nameless_function(arg1, arg2 … , argN):
       return expression(using arguments)
   ```

   Here is an example:

```
>>> def get_temperature_in_celsius(fahrenheit):
...     return 5/9 * (fahrenheit - 32)
...
>>>
>>> values_in_f = [100.2, 105.3, 90, 80, 110]
>>> # the usual way
>>> list(map(get_temperature_in_celsius, values_in_f))
[37.88888888888889, 40.72222222222222, 32.22222222222222,
26.666666666666668, 43.333333333333336]
>>> # get_temperature_in_celsius defined as a lambda function
>>> list(map(lambda x: (float(5)/9)*(x-32), values_in_f))
[37.88888888888889, 40.72222222222222, 32.22222222222222,
26.666666666666668, 43.333333333333336]
```

   For each of the following code, write a single statement using map to produce the desired output:

```
# example
values = ['hello', 'world']
result = map(len, values)
print(list(result)) # [5, 5]

# Part 1
values = ['10', '100', '1000']
result =
print(list(result)) # [10, 100, 1000]

# Part 2
values = [2, 3, 4]
result =
print(list(result)) # [4, 9, 16]

# Part 3
values = ['apple orange pear', 'papaya durian', 'chiku guava']
result =
# [['apple', 'orange', 'pear'], ['papaya', 'durian'], ['chiku', 'guava']]
print(list(result))
```

3. A `filter` function constructs an iterator from elements of an iterable for which a function returns true. The syntax is:

   ```
   result = filter(func, sequence)
   ```

   It offers an elegant way to filter out all the elements of a list, for which the function `func` returns `True`. The needs a function `func` as its first argument. `func` returns a Boolean value, i.e. either `True` or `False`. This function will be applied to every element of the `iterable`. If `func` returns `True`, the element of the iterable will be included in the result iterable.

   This is equivalent to a function as follows:

   ```
   def filter_equivalent(func, sequence):
       result = []
       for element in sequence:
           if func(element):
               result.append(element)
       return result
   ```

```
>>> result = filter(str.isalpha, ['x', 'y', '2', '3', 'a'])
>>>
>>> type(result)
<class 'filter'>
>>> list(result)
['x', 'y', 'a']
>>> result = filter(str.isalpha, ['x', 'y', '2', '3', 'a'])
>>> for item in result:
...     print(item)
...
x
y
a
>>> for item in result:
...     print(item)
...
>>> # no result here. u can't iterate through items the second time!!
```

For each of the following code on the next page, write a single statement using map to produce the desired output:

```
# example
values = ['apple', 'orange', 'pear']
result = filter(lambda value: len(value) > 4, values)
print(list(result))
```

```
# Part 1
values = [1, -5, 3, 5, 7, -11]
result =                       # removes negative values
print(list(result))            # [1, 3, 5, 7]


# Part 2
values = [2, 3, 4, 6, 7, 10, 11]
result =                       # removes even values
print(list(result))            # [3, 7, 11]


# Part 3
values = ['radar', 'ape', 'peep', 'what']
result =
# returns all palindrome in values
print(list(result))
```