Assignment 3

Course: CS65 - Introduction to Computer Science (Fall 2022)
Instructor: Md Alimoor Reza, Assistant Professor of Computer Science, Drake University
Due: Tuesday, November 8, 2022 by 11:50 PM

Introduction

This assignment will allow you to explore List and Dictionary features in problem-solving. Your programs should be written with multiple user-defined functions in the same file. This helps break down the task into smaller and simpler pieces. Functions also make it easier to test individual components, as parts of the program can be tested separately.

Logistics

Create a python file for each task separately (with a **.py* extension), where you will save your python instructions. By default, Thonny opens an unnamed file in the a *text editor* (top text pane). It is an excellent practice to write a formal header and suppress it as comments. A line of code with a preceding # is considered as a comment in Python. You should type in the necessary parts as shown below, e.g., *your name, your contact email, description, etc.*. Save the file using the format as follows *firstname_lastname_a3_task*.py* (all in lowercase letters). For example, I saved my file as *md_reza_a3_task1.py*. Those who prefer to work alone, please email me. Otherwise, work with your existing group. You should only work together but can submit a separate copy of the assignment.

Task 1 (50 points): List

There are several small sub-tasks, and you should solve each subtask using List.

Subtask 1 (10 points): Finding Divisors

You will be given an integer number **N** as an input. You should write a program that will find all the divisors of that given number **N**. Any number **N** can be divided by at least two numbers, i.e., 1 and **N**. Your program should return all the divisors excluding these two numbers. If the number cannot be divided by any other numbers other than 1 and N, then you should return an empty list [ ]. You should prompt the user to enter one integer number and compute its divisors as described above. For example, if the user enters **12**. Your program should return a list **[2, 3, 4, 6]** as 12 is divisible by 2, 3, 4, and 6. On the other hand, if the user enters **13**. Your program should return an empty list [ ] as 13 cannot be divided by any number other than 1 and 13.

Subtask 2 (10 points): Testing Divisors

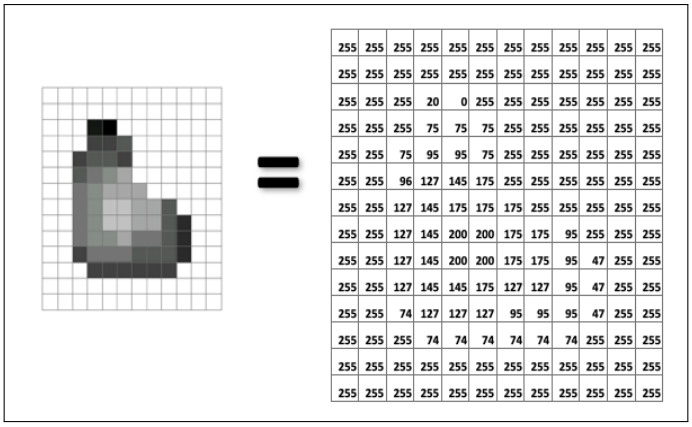
In this task, you should prompt the user to enter integer numbers repetitively until the user enters **-1**. (Hint: In this context, you may consider using a while-loop to receive these numbers from the user. Also, you may consider appending the numbers in an empty list.) Let's assume these list of numbers are saved in list variable *my_list*. Now, you should prompt the user to enter another number **N**. Given these two inputs (one list of numbers stored in variable *my_list* and another integer number N), you should write a function that will eliminate all the numbers from *my_list* that are not divisors of the number **N**. You should return an empty list [ ] if none of them are divisors. For example, if the user enters a list of numbers **12, 4, 15, 27, 9, 40, -1** first, and then enters the other number **60**. Your program should return the list **[12, 4, 15]** as each number is a divisor of 60. Sample inputs and output for this task are shown below:

```
>>> %Run a3_test_divisors.py

  enter a cadidate divisor: 12
  enter a cadidate divisor: 4
  enter a cadidate divisor: 15
  enter a cadidate divisor: 27
  enter a cadidate divisor: 9
  enter a cadidate divisor: 40
  enter a cadidate divisor: -1
  enter another number: 60
  List of candidate divisors are:  [12, 4, 15, 27, 9, 40]
  List of divisors are:  [12, 4, 15]
```

Subtask 3 (30 points): Color Correction

Computer vision researchers (including myself) analyze images to find meaningful content in them. An image is commonly represented by a three-channel grid matrix containing integer numbers. The figure below shows one such channel of an image. Each integer number represents an intensity value. These intensity values ranges between **0** and **255**. A large number signifies a brighter intensity, while a low number corresponds to a darker intensity.



A toy image and computer's internal representation as a grid of intensity values.

Let us assume you are given these intensity values as list of triplets (red, green, blue). Unfortunately, some of these triples got corrupted and values outside the valid range (between 0 and 255). Your program will receive a list of lists. Each inner list is a triplet representing three color values: one for red intensity, one for green intensity, and finally one for the blue intensity. For example, [[100, 0, 0], [40, -156, 0], [0, 156, 0], [40, 156, 500], [0, 0, 250]] represents a list of five color-triplets. **You should write three separate functions as instructed below.** Each function will perform a separate operation and accordingly will return a different list.

- *get_invalid_colors()*: If any intensity value inside a triplet has either a negative value (less than zero) or a positive value greater than 255, you should return the indices of those triplets inside the list. For example, given an input list of [[100, 0, 0], [40, -156, 0], [0, 156, 0], [40, 156, 500], [0, 0, 250]], your program should return list **[1, 3]** since indices at 1 and 3 contain two color-triplets with either a negative intensity or intensity value greater than 255.

- *correct_invalid_colors()*: If any intensity value inside a color-triplet has a negative value, you should replace that value with **0**. On the other hand, if any intensity value inside a color-triplet has a positive value greater than 255, you should replace that value with **255**. For example, given an input list of [[100, 0, 0], [40, -156, 0], [0, 156, 0], [40, 156, 500], [0, 0, 250]], your program should return [[100, 0, 0], [40, **0**, 0], [0, 156, 0], [40, 156, **255**], [0, 0, 250]].

- *discard_invalid_colors()*: If any intensity value inside a triplet has either a negative value (less than zero) or a positive value greater than 255, you should remove that color-triplet. For example, given an input list of [[100, 0, 0], [40, -156, 0], [0, 156, 0], [40, 156, 500], [0, 0, 250]], your program should return [[100, 0, 0], [0, 156, 0], [0, 0, 250]].

```
# this fucntion returns the list of indices with invalid color-triplets
def get_invalid_colors(my_list):


    return

# this fucntion returns the corrected list of color-triplets
def correct_invalid_colors(my_list):


    return

# this fucntion removes the invalid color-triplets
def discard_invalid_colors(my_list):


    return
```

Figure: empty function definitions for the color correction task.

While writing your python program, you should consider using various list operations or methods (eg, *.append(), .remove(), .pop(), del*) as discussed during class. Empty python files have been uploaded, which you may find useful.

Task 2 (50 points): Dictionary

You should use Dictionary for this task. In this task, you will write several functions that can enable us to encrypt a message with an encoding mechanism called **Caeser Cipher**. It is named after Caeser, who used this encoding form to send secret messages for personal correspondences. Secret messages have been used historically by various entities, e.g., military missions, secret societies, other personal usages, etc.

Subtask 1: Manipulate Dictionary

- *concat_dictionary(my_dict1, my_dict2)*: Write a function that will concatenate two dictionaries. It will create a new dictionary by appending the (key, val) pairs from the second dictionary *my_dict*2 to the first one *my_dict*1. Finally, it will return the newly created dictionary containing all the (key, value) pairs. If the original dictionaries are {'a':'d', 'b':'e', 'c':'f'} and {'A':'D', 'B':'E', 'C':'F'}, the new dictionary will become {'a':'d', 'b':'e', 'c':'f', 'A':'D', 'B':'E', 'C':'F'}.

- *toggle_key_value_pair(my_dict)*: This function swaps the (key, value) pairs from a dictionary to create a new dictionary. Then it returns the new dictionary. If the original dictionary is {'a':'d', 'b':'e', 'c':'f'} the new dictionary will become {'d':'a', 'e':'b', 'f':'c'}

```
def concat_dictionary(my_dict1, my_dict2):
    # YOUR CODE HERE

    return # return a dictionary


def toggle_key_value_pair(my_dict):
    # YOUR CODE HERE


    return # return a dictionary
```

Subtask 2: Create Caesar Cipher Mapping

You will need to shift each letter in the alphabet by a fixed position to the right, given an alphabet. For instance, if the alphabets are [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’], and if you are given the shift amount of 3. Then each character will be mapped to 3 letters to its right. More specifically, a→ d, b→ e, ..., f→ c. Notice when there is more character to the right, it should wrap around. Therefore, by right shifting ‘f’ 3 positions to its right, it gets mapped to ‘c’.

Key	A	B	C	D	E	F
Mapping	D	E	F	A	B	C

Now write a function (with the signature *create_caesar_cipher_mapping(my_vocab, shift_amount)*) that will create a dictionary using Caesar cipher mapping. It will receive two inputs. The first one, *my_vocab*, is a list of characters representing the alphabet. The second one, *shift_amount*, is a positive number denoting the amount by which the letters will be shifted to the right. Using these two inputs, the function will create a Caesar cipher mapping using the rule discussed above. You can assume that they are letters in the alphabet are sorted in ascending order.

```
# This function will create a dictionary with (key, value) mapping
def create_caesar_cipher_mapping(my_vocab, shift_amount):


    # create a dictionary that will contain {'a':mapped_char_a,
    #                                        'b':mapped_char_b, ...,
    #                                        'z':mapped_char_z}
    # YOUR CODE HERE


    return # return a dictionary
```

If this function is called with list *my_vocab* = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’] and *shift_amount* = 3. The mapping would be as follows:

```
Alphabet in my_vocab1 =  ['a', 'b', 'c', 'd', 'e', 'f']
Caesar cipher mapping:
--------------------
 {'a': 'd', 'b': 'e', 'c': 'f', 'd': 'a', 'e': 'b', 'f': 'c'}
```

Subtask 3: Encrypt Messages

Finally, you should write a function that will encrypt a message. This function will use the Caesar cipher mapping to encrypt a given message. Your function will receive two inputs: i) a dictionary (containing the mapping) and ii) a string (the message to be encrypted). More specifically, *my_dict* is a dictionary containing the (key, value) pairs of the encryption eg, {‘a’:‘d’, ‘b’:‘e’, ‘c’:‘f’}, and *my_sting* the message we want to encrypt eg, ‘hello world! HELLO WORLD!’.

```
def encrypt_message(my_dict, my_string):

    encrypted_message = ""
    # YOUR CODE HERE


    return encrypted_message
```

The following figure shows the output of your program when three different dictionaries of Caesar mappings are used to encrypt the messages.

```
{'a': 'd', 'b': 'e', 'c': 'f', 'd': 'a', 'e': 'b', 'f': 'c'}
Original  message: hello world! HELLO WORLD!
---------------

Encrypted message1: hbllo worla! HELLO WORLD!
---------------

{'A': 'D', 'B': 'E', 'C': 'F', 'D': 'A', 'E': 'B', 'F': 'C'}
Original  message: hello world! HELLO WORLD!
---------------

Encrypted message2: hello world! HBLLO WORLA!
---------------

{'a': 'd', 'b': 'e', 'c': 'f', 'd': 'g', 'e': 'h', 'f': 'i', 'g': 'j', 'h': 'k', 'i': 'l', 'j': 'm', 'k
': 'n', 'l': 'o', 'm': 'p', 'n': 'q', 'o': 'r', 'p': 's', 'q': 't', 'r': 'u', 's': 'v', 't': 'w', 'u':
'x', 'v': 'y', 'w': 'z', 'x': 'a', 'y': 'b', 'z': 'c'}
Original  message: hello world! HELLO WORLD!
---------------

Encrypted message3: khoor zruog! HELLO WORLD!
---------------
```

You should also create a new dictionary by calling the function *concat_dictionary()* with two different dictionaries *my_dict*3 and *my_dict*4. Your output should be as follows:

```
my_dict3 =  {'a': 'd', 'b': 'e', 'c': 'f', 'd': 'g', 'e': 'h', 'f': 'i', 'g': 'j', 'h': 'k', 'i': 'l', 'j': 'm
', 'k': 'n', 'l': 'o', 'm': 'p', 'n': 'q', 'o': 'r', 'p': 's', 'q': 't', 'r': 'u', 's': 'v', 't': 'w', 'u': 'x
', 'v': 'y', 'w': 'z', 'x': 'a', 'y': 'b', 'z': 'c'}

my_dict4 =  {'A': 'D', 'B': 'E', 'C': 'F', 'D': 'G', 'E': 'H', 'F': 'I', 'G': 'J', 'H': 'K', 'I': 'L', 'J': 'M
', 'K': 'N', 'L': 'O', 'M': 'P', 'N': 'Q', 'O': 'R', 'P': 'S', 'Q': 'T', 'R': 'U', 'S': 'V', 'T': 'W', 'U': 'X
', 'V': 'Y', 'W': 'Z', 'X': 'A', 'Y': 'B', 'Z': 'C'}

Result of dictionary concatenation is my_dict5 =  {'a': 'd', 'b': 'e', 'c': 'f', 'd': 'g', 'e': 'h', 'f': 'i',
'g': 'j', 'h': 'k', 'i': 'l', 'j': 'm', 'k': 'n', 'l': 'o', 'm': 'p', 'n': 'q', 'o': 'r', 'p': 's', 'q': 't',
'r': 'u', 's': 'v', 't': 'w', 'u': 'x', 'v': 'y', 'w': 'z', 'x': 'a', 'y': 'b', 'z': 'c', 'A': 'D', 'B': 'E',
'C': 'F', 'D': 'G', 'E': 'H', 'F': 'I', 'G': 'J', 'H': 'K', 'I': 'L', 'J': 'M', 'K': 'N', 'L': 'O', 'M': 'P',
'N': 'Q', 'O': 'R', 'P': 'S', 'Q': 'T', 'R': 'U', 'S': 'V', 'T': 'W', 'U': 'X', 'V': 'Y', 'W': 'Z', 'X': 'A',
'Y': 'B', 'Z': 'C'}
```

You should apply this new dictionary *my_dict*5 to encrypt the message. The following figure shows the output of your program of the encrypted message with *my_dict*5.

```
{'a': 'd', 'b': 'e', 'c': 'f', 'd': 'g', 'e': 'h', 'f': 'i', 'g': 'j', 'h': 'k', 'i': 'l', 'j': 'm', 'k': 'n', 'l': 'o', 'm': 'p', 'n': 'q', 'o': 'r',
'p': 's', 'q': 't', 'r': 'u', 's': 'v', 't': 'w', 'u': 'x', 'v': 'y', 'w': 'z', 'x': 'a', 'y': 'b', 'z': 'c', 'A': 'D', 'B': 'E', 'C': 'F', 'D': 'G', '
E': 'H', 'F': 'I', 'G': 'J', 'H': 'K', 'I': 'L', 'J': 'M', 'K': 'N', 'L': 'O', 'M': 'P', 'N': 'Q', 'O': 'R', 'P': 'S', 'Q': 'T', 'R': 'U', 'S': 'V', 'T
': 'W', 'U': 'X', 'V': 'Y', 'W': 'Z', 'X': 'A', 'Y': 'B', 'Z': 'C'}
Original  message: hello world! HELLO WORLD!
---------------

Encrypted message5: khoor zruog! KHOOR ZRUOG!
```

What to turn in

You should submit your source code (python files). You **do not need** to submit screenshots of your outputs.

Grading Rubric:

Submitted correctly	05
Code is well commented	05
List and dictionary usage	45
Code produces correct outputs	45
Total	100