CHAPTER

III

Week 1 - Assignment 1A

Warming up exercises

In this chapter:

Use the **exercises** below to practice before going to the mini-project. They were thought to help you to break the concepts needed to solve the mini-project into smaller problems. You don't need to submit them in your assignment.

Warming up exercises help you to consolidate concepts separately, by working them in smaller, simpler problems.

Exercise 1. The code below creates a dictionary that contains integers as key, and their double as its values.

```
double = {}

for i in range(100):
    double[i] = 2*i
```

Write a code to ask for a key to the user, check if that key exists, and print its value if so.

Exercise 2. Write a function that receive a dictionary as input and invert the its keys and values. In other words, your function should creates another dictionary in which the keys of the input dictionary are now the values, and the values of the input dictionary are now the keys. Use the dictionary defined in exercise 1 to test your function.

Exercise 3. The program below ask the user for the metadata of multiple songs and accumulate them in a list of tuples.

```
add_song = True
song_list = []
while add_song:
        song_name = input("Song name:\n")
        song_dur = input("Song duration (min):\n")
        song_composer = input("Song composer:\n")

        song = (song_name, song_composer, int(song_dur))
        song_list.append(song)

        option = input("Add another song? [yes/no]")
        if option != "yes":
            add_song = False
```

Modify that program to print the composer of all the songs in the list after the user finishes adding the songs.

Exercise 4. Using the song list defined in the previous exercise, write a program that asks for the name of a song and prints all the information about it, if the song is present in the list. If the song is not in the list your program should not crash. Try to handle this error without using a try/except statement.

Exercise 5. Transform the list of tuple from exercise 3 into a dictionary in which the keys are the song names and the values are tuples with the rest of the information. Then write a code to ask for a song name and print the duration of that given song.

Mini-Project 1 *Pair Programming*

Imagine you want to create a program to handle the contact information of your phone. In each entry, you want to store a name, a mobile phone and the birthday of your contact. This information will be stored in a dictionary, with the name of the contact being the key, and the phone and birthday stored in a list L that is the value of that key.

For a given key, the first element of the list L should be a string, containing only 9 digits. The second element of L should be a tuple whose first value is an int for the birth day, and the second an int for the birth month.

Example. For example, a contact dictionary with 2 keys should look like:

```
contacts = {"Alice": ["611234456", (24,03)],
            "Bob": ["665544332", (2,12)]}
print(contacts)
```

```
contacts =
"Alice":  ["611234456", (24,03)],
"Bob":  ["665544332", (2,12)]
```

Use the template `mini_proj_1.py` in Brightspace to implement the following functions:

- `add_entry(contacts: dict, name: str, phone: str, birthdate: tuple)`: Add a new entry defined by name, birthdate (a tuple with day and month) to the contact dictionary in *contacts*. The output of the function is the updated contact dictionary. If a name is already in the contact dictionary, the function does not replace it, and just return the original dictionary. If an invalid birthdate (the day should be in the range of 1-31 and the month in the range of 1-12) or invalid phone (should have 9 numbers) is given, the function does not add a new entry, and just return the original contact dictionary.

- `change_entry(contacts: dict, name: str, phone: str)`: Change the phone number of an entry that already exists (specified by name) in the contact dictionary *contacts*. The output of the function is the updated contact dictionary. If an invalid phone (9 numbers) or name is given, the function does not change the entry, and just return the original contact dictionary.

- `delete_entry(contacts: dict, name: str)`: Delete the specified entry if it exists in the contact dictionary. If not, return the original contact dictionary, without stopping the program.

- `find_phone(contacts: dict, name: str)`: Return the phone of the contact specified by name, if it exists. Return None if the contact does not exist.

- `month_birthdays(contacts: dict, date: tuple)`: Return a list of tuples, each of them containing the name (in the first element) and **day** (in the second element) of the contacts with birthdays in the current month. The list should include past birthdays. The list should be empty if there are no birthdays.

- `list_all_names(contacts)`: Return a list of strings containing all the names in the contact dictionary. It returns an empty list if the contact dictionary is empty. The main function should print the names in the list after calling this function.
- `main()`: a main function that will first initialize an empty contact dictionary and repeatedly ask the user to input a number from 1-7 to choose among the functionalities above (until the user chooses option 7 - exit). The program should ask for the needed information accordingly. Ex.: If the user chooses '2 - change entry', the program should ask for a name and a phone to pass to the change_entry() function.

Remark. The input and the output of the functions should be precisely what is described above. In other words, you may use different parameter names in your functions, but do not change the order of the parameters.

Testing your solutions

Because most of the exercises now involve implementing very well-defined functions, it's easy to test whether a your scripts are correct. Your functions will be (mostly) automatically tested, similar to Project Lovelace exercises and this time you can also test it yourself. Together with the assignment you will sometimes receive a testing script to test your codes.

The way that works is that the **self-testing script** imports your functions (as a module) and uses them in a predefined way, testing if the output is exactly what was expected.

Use the self-test to look for errors yourself and fix them before submitting.

Self-testing:

1. Download the self-testing script in the Assignment files.
2. Adjust the import statement (if needed) at the test script to import your solutions.
3. Run the test-script from the terminal or from Spyder and check which tests you passed.

The **self-testing scripts** will test if the main purpose of a function is being made as well as some edge cases.
For example:

```
c = add_entry({}, "Alice"
    , "611234456",
    (24,03))
```

If the value returned by your function is correct, that the following expression should be True:

```
c == {"Alice": ["
    611234456", (24,03)]}
```

Your code will be tested for all the other specified cases. (Ex.: adding a key that already exists, adding a phone with more or less than 9 digits, etc..). So make sure to deal with them inside your function.

Submission instructions

Extra documents:
If you have worked together with another student, create a .txt file called `programming_partner.txt` containing the S-number of both of you.

> **Example.** `programming_partner.txt`

```
Problem 1:
s012345
s123456

```

Assignment 1A is part of Assignment 1, which has its deadline after Week 2. This means that you will still be able to submit corrections to your scripts later on. However, submit what you have so far in Brightspace.

Both students should submit their work individually on Brightspace.

General submission:
Create a zip file called `Assignment1` containing the files:

- `mini_proj_1.py`
- `programming_partner.txt`

Submit your file to Assignment 1A, in Brightspace.