Daniel Castro

August 16th, 2023

IT FDN 110 A Su 23: Foundations Of Programming: Python

Assignment 06

https://github.com/DanielCyar/IntroToProg-Python

Lists and Dictionaries

Introduction

In Assignment 6, we will explore the concepts of functions and classes in Python. Through it we'll learn the process of utilizing functions to modularize code, understanding parameters and arguments, and working with the return values. Additionally, we'll learn the distinction between global and local variables, learning how to organize the code by using the "separation of concerns". The practical aspect of the assignment involves modifying a "ToDo" list script, similar to the one done on our previous assignment, but adding more functions to the existing codebase while maintaining clarity and reusability. To conclude the assignment, we include documentation of our knowledge and will share our work on GitHub, along with creating a webpage there for future assignments.

Getting into practice

Building upon the concepts we've explored in the module, assignment 6 involves enhancing a script that manages a "ToDo list" and the primary objective is to apply our understanding of functions to refactor and organize the existing codebase more efficiently. Additionally, we'll learn the process of using GitHub to share our code and create a GitHub webpage.

Modifying the script

Starting with the template provided in the "Assignment06_Starter.py" python file, I first added my information to the comment at the beginning of the script, in the appropriate section of the ChangeLog.

```
# ------------------------------------------------------------------------ #
# Title: Assignment 06
# Description: Working with functions in a class,
#              When the program starts, load each "row" of data
#              in "ToDoToDoList.txt" into a python Dictionary.
#              Add the each dictionary "row" to a python list "table"
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# DanielCastro,8.15.2023,Modified code to complete assignment 06
# ------------------------------------------------------------------------ #
```

Then, I searched for the TODO tasks in the code, to somewhat guide me through the assignment. To do this, I used the TODO section of PyCharm's toolbar in the bottom.

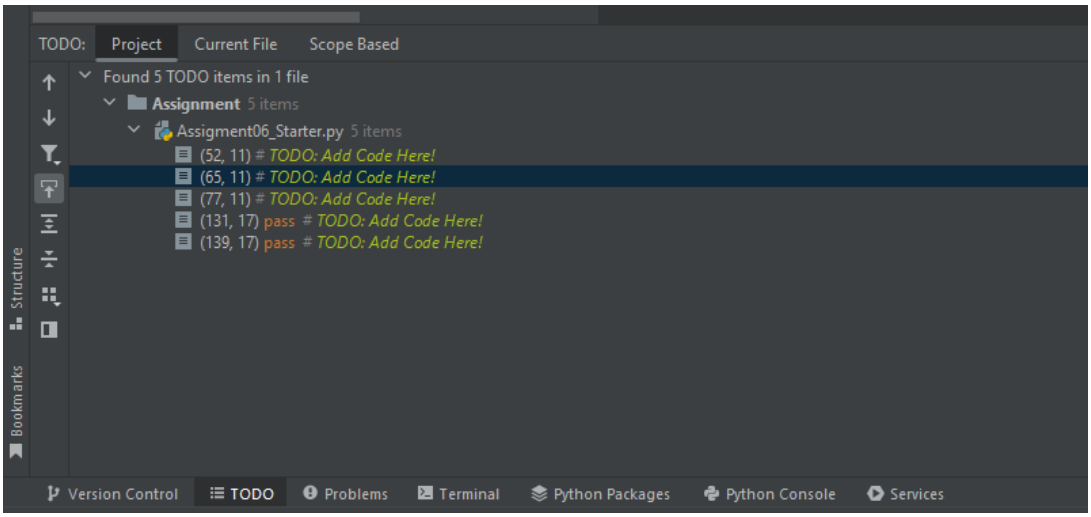

Figure 1. TODO items and toolbar.

In the Data Section from the separation of concerns, I verified which variables were being initialized to use them throughout the code. I also placed a TODO comment for this as step 0.

```
# Data ---------------------------------------------------------------------- #
# Declare variables and constants
file_name_str = "ToDoFile.txt"  # The name of the data file
file_obj = None  # An object that represents a file
row_dic = {}  # A row of data separated into elements of a dictionary {Task,Priority}
table_lst = []  # A list that acts as a 'table' of rows
choice_str = ""  # Captures the user option selection
```

Then, moving on to the Processing Section, the script has a class named Processor to group the corresponding functions that will be processing the data.

```
# Processing  --------------------------------------------------------------- #
class Processor:
    """  Performs Processing tasks """
```

In this case, the functions that are being grouped in this class are:

- read_data_from_file
- add_data_to_list
- remove_data_from_list
- write_data_to_file

Read data from file

In this one, there was nothing to be done originally, but I wanted to apply some coding I already tried in assignment 5. For example, what if there's no text file when running the code. To prevent this, I decided to add error handling with try/except. It's pretty much the same as with assignment 5, where it will try to open the text file in read mode and append each row as dictionaries with key:value pairs in a list of rows.

```
@staticmethod
def read_data_from_file(file_name, list_of_rows):
    """ Reads data from a file into a list of dictionary rows

    :param file_name: (string) with name of file:
```

```
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    list_of_rows.clear()  # clear current data
    # TODO: [Personal] What if there's no text file?
    try:                  # Added error handling in case there's no text file
        file = open(file_name, "r")
        for line in file:
            task, priority = line.split(",")
            row = {"Task": task.strip(), "Priority": priority.strip()}
            list_of_rows.append(row)
        file.close()
    except IOError:   # Using I/O exception for error handling, similar to Assign. 5
        print(f"\n[File '{file_name}' not found. Starting with a blank entry]")
    return list_of_rows
```

Add data to list

For this one, we had the parameters for the function already defined and had to place the code that would add the data to the list of dictionary rows. We already had the row defined as a dictionary with their keys and values (strings). We were only missing the append method to add that row of data to the list and returns the list of rows to the main body of the script.

```
@staticmethod
def add_data_to_list(task, priority, list_of_rows):
    """ Adds data to a list of dictionary rows

    :param task: (string) with name of task:
    :param priority: (string) with name of priority:
    :param list_of_rows: (list) you want to add more data to:
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
    # TODO: Add Code Here!
    list_of_rows.append(row)  # Add a new row of data to the list

    return list_of_rows
```

Remove data from list

Same as the prior function, we had the parameters for the function already defined. But this time we had to place the code that would remove the data from the list of dictionary rows. In this one, I created a for loop to find the selected task in the list of dictionary rows. The if condition then verifies if the task does match with an item from the list (in here I implemented the lower method to avoid issues due to case sensitivity). If it does find the task, it will remove the item row from the list and break the loop. Then returns the updated list of rows.

```
@staticmethod
def remove_data_from_list(task, list_of_rows):
    """ Removes data from a list of dictionary rows

    :param task: (string) with name of task:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    # TODO: Add Code Here!
    for row in list_of_rows: # Look for the matching row using the dictionary key "Task"
        if row["Task"].lower() == task.lower():  # Use the lower method to handle caps when
validating the row
            list_of_rows.remove(row)     # Remove the matching row from the list
            break                        # Break the loop since the row was removed
    return list_of_rows
```

Write data to file

This final function from the Processing Section was created to write data from the list of dictionary rows to the text file. In this case, It only opens the text file in write mode and for every row of data

that was inserted in the list of rows, it writes down the task and its priority. Then closes the text file, prints where it was saved and returns the list of rows to the main body of the script.

```
@staticmethod
def write_data_to_file(file_name, list_of_rows):
    """ Writes data from a list of dictionary rows to a File

    :param file_name: (string) with name of file:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    # TODO: Add Code Here!
    file = open(file_name, "w")
    for row in list_of_rows:
        file.write(row["Task"] + "," + row["Priority"] + "\n")
    file.close()
    print("Data saved to ", file_name)
    return list_of_rows
```

Now, moving on to the Presentation (Input/Output) Section, the script has a class named IO to group the corresponding functions that will be presenting data to the user.

```
# Presentation (Input/Output)  ---------------------------------------- #


class IO:
    """ Performs Input and Output tasks """
```

In this case, the functions that are being grouped in this class are:

- output_menu_tasks
- input_menu_choice
- output_current_tasks_in_list
- input_new_task_and_priority
- input_task_to_remove

The first three are defined to present some options for the end user. `output_menu_tasks` will print the menu of options to the user; `input_menu_choice` will ask the user to select one of the options from the menu given by the first function; and `output_current_tasks_in_list` will only display the current tasks that are appended in the list of dictionary rows.

```
@staticmethod
def output_menu_tasks():
    """  Display a menu of choices to the user

    :return: nothing
    """
    print('''
    Menu of Options
    1) Add a new Task
    2) Remove an existing Task
    3) Save Data to File
    4) Exit Program
    ''')
    print()  # Add an extra line for looks
```

```
@staticmethod
def input_menu_choice():
    """ Gets the menu choice from a user

    :return: string
    """

    choice = str(input("Which option would you like to perform? [1 to 4] - ")).strip()
    print()  # Add an extra line for looks
    return choice




@staticmethod
def output_current_tasks_in_list(list_of_rows):
    """ Shows the current Tasks in the list of dictionaries rows

    :param list_of_rows: (list) of rows you want to display
    :return: nothing
    """
    print("\n\n******* The current tasks ToDo are: *******")
    for row in list_of_rows:
        print(row["Task"] + " (" + row["Priority"] + ")")
    print("*********************************************")
    print()  # Add an extra line for looks
```

The last two functions from this IO class are the ones that need to be completed.

Input new task and priority

The purpose of this function is to prompt the end user to input a task and priority to be added to the list of rows. For this one, we just needed to get the input while guiding the user on the information that was needed and then, only return the task and its priority back to the function caller.

```
@staticmethod
def input_new_task_and_priority():
    """  Gets task and priority values to be added to the list

    :return: (string, string) with task and priority
    """
    # TODO: Add Code Here!
    task = input("Enter the task: ")            # Gets the user input for task
    priority = input("Enter the priority: ")    # Gets the user input for the task's
priority
    return task, priority        # Returns those values to be used by the function caller
```

Input task to remove

Similar to the previous one, this function asks the end user to input a task (that will then be used as a key finder for the row of data in the list of dictionaries) that will be removed from the list later on. For this one, we just needed to get the input while guiding the user on the information that was needed and then, only return the task that is going to be removed.

```
@staticmethod
def input_task_to_remove():
    """  Gets the task name to be removed from the list

    :return: (string) with task
    """
    # TODO: Add Code Here!
```

```
    task_to_remove = input("Enter the task you want to remove: ")
    return task_to_remove
```

Finally, for the main body of the script. We will run the functions, remembering to call them from their respective classes, to first read the data from the file (if any), and then display the menu of options to the end user inside of a while loop. Then, the user will be prompted to select one of the options to run and it will run the respective functions according to the choice he made with if/elif/else conditional statements.

```
# Main Body of Script  ------------------------------------------------------ #


# Step 1 - When the program starts, Load data from ToDoFile.txt.
Processor.read_data_from_file( file_name=file_name_str, list_of_rows=table_lst)  # read file
data

# Step 2 - Display a menu of choices to the user
while True:
    # Step 3 Show current data
    IO.output_current_tasks_in_list(list_of_rows=table_lst)  # Show current data in the
list/table
    IO.output_menu_tasks()  # Shows menu
    choice_str = IO.input_menu_choice()  # Get menu option

    # Step 4 - Process user's menu choice
    if choice_str.strip() == '1':  # Add a new Task
        task, priority = IO.input_new_task_and_priority()
        table_lst = Processor.add_data_to_list(task=task, priority=priority,
list_of_rows=table_lst)
        continue  # to show the menu

    elif choice_str == '2':  # Remove an existing Task
        task = IO.input_task_to_remove()
        table_lst = Processor.remove_data_from_list(task=task, list_of_rows=table_lst)
        continue  # to show the menu

    elif choice_str == '3':  # Save Data to File
        table_lst = Processor .write_data_to_file(file_name=file_name_str,
list_of_rows=table_lst)
        continue  # to show the menu

    elif choice_str == '4':  # Exit Program
        print("Goodbye!")
        input("[Press Enter to Exit the program ...]")
        break  # by exiting loop
```

NOTE: I edited the 4th choice to place an input so that the program doesn't close when we run it within the Command Line. Also, I placed an else conditional statement to make sure that the user selects an option from 1 to 4 or else it will just return them to the Menu of Options and make a new choice.

```
# TODO: [Personal] What if the input is not in the options?
else:
    print("\n[Invalid option. Select an option from 1 to 4]")
```

Running the script in PyCharm

Figure 2 shows the program running on PyCharm.

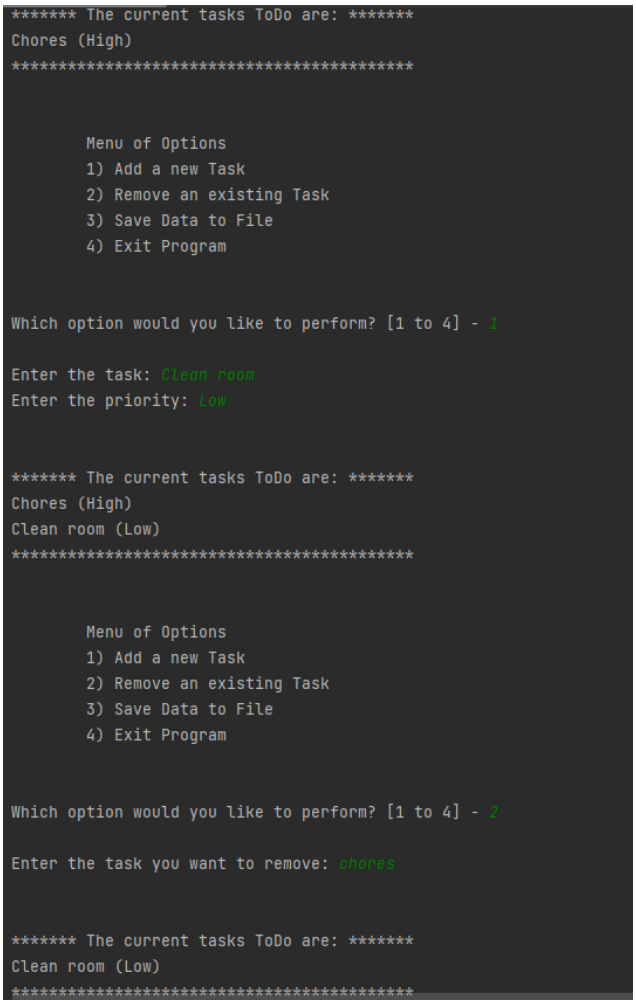

Figure 2. Assignment06.py running with PyCharm.

Running the script in Windows Command Line

To run the script from the Windows Command Line, it's just accessing the directory where the 'Assignment06.py' file is saved and type the name of the Python file. The script will start running (Figure 3).

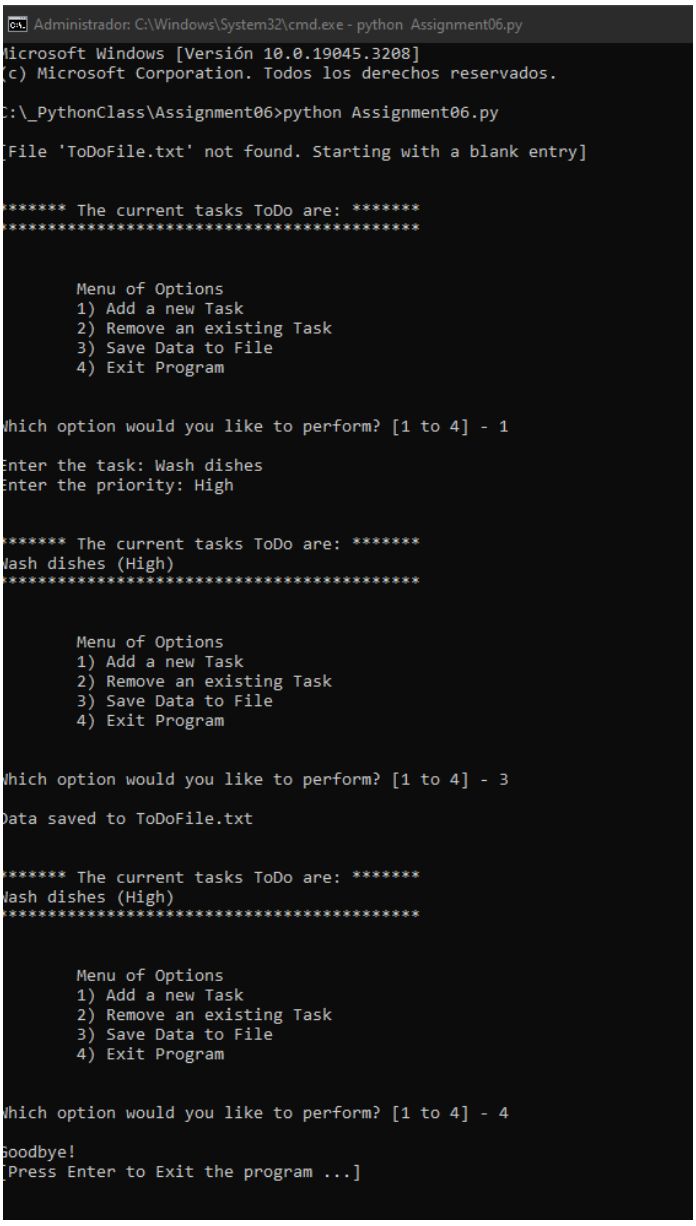

Figure 3 Assignment06.py running in Windows Command Line.

Once we press Option 4 and then press Enter, the code stops running. Figure 4 displays how the data was saved inside the ToDoList.txt file.

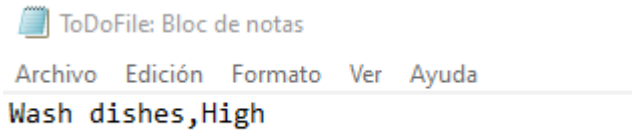

Figure 4. ToDoList.txt file with the saved data.

GitHub Webpage

To finish the activity, I followed the steps given to create a webpage within GitHub. Then, I added it to the README file in the main branch (Figure 5).

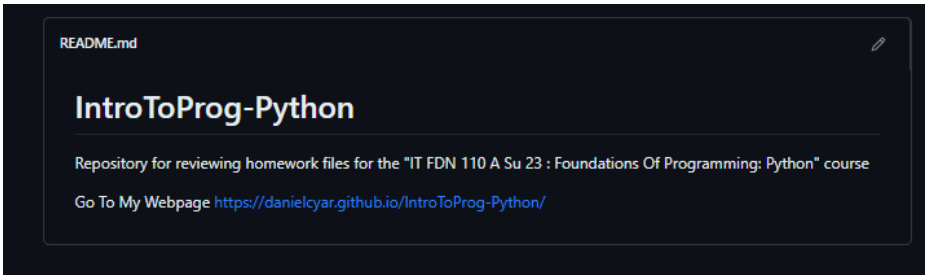

Figure 5. README file with the webpage on it.

Figure 6 shows how it ended up looking for the time being. We will be making some changes to it for the next assignment.

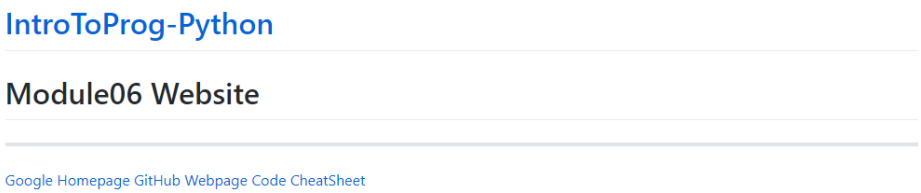

Figure 6. Webpage with the given format.

Summary

In Assignment 6, we deepened our understanding of Python's functions and classes. The practical segment involved refining an existing "ToDo list" script by incorporating new functions. By leveraging functions, we enhanced code organization and modularity within the separation of concerns, promoting readability and reusability. Additionally, we continued working with error-handling techniques to gracefully manage potential issues. This assignment also introduced us to GitHub, enabling code sharing and webpage creation.