Phillip Thompson
5/24/23
IT FDN 110 A
Assignment 6
https://github.com/philthom10/IntroToProg-Python-Mod06

Assignment 6 – To Do List – Functions

1. Introduction

Module 06 this week focused on the use of classes and functions to better organize code. Like assignment 05, the goal was to finish a starter code and have a functioning program that can edit a to-do list, and read / write it to a file. In order to accomplish this, the knowledge of how functions, parameters, and arguments work was necessary. Much of the code used was similar to that in Assignment 05, but the challenge was to make that fit into an existing framework from the starter code and work correctly, which was only possible by understanding how functions use parameters and arguments and return values.

2. Writing the Code

The code for this assignment was written in the PyCharm IDE. I first created a Project called Assignment 06, and then copied the Assignment06_starer.py file into that project folder and renamed it Assignment06.py. I then created a text file in that working directory called "ToDoFile.txt", as that is the name of the file that was assigned to the *file_name_str* constant in the starter code. See below (Figure 1), showing the project structure in PyCharm.

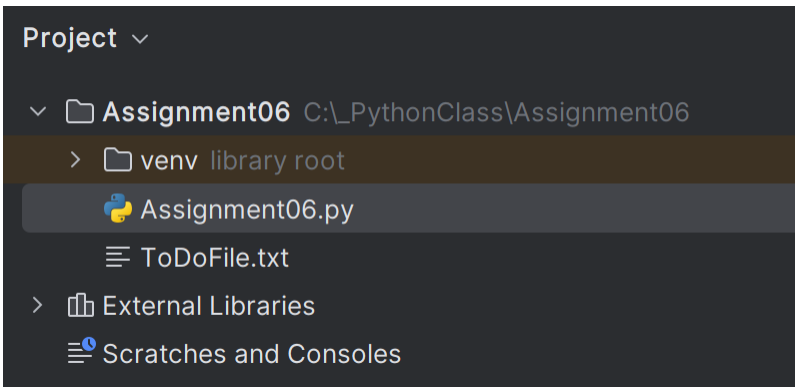

Figure 1: Project structure in PyCharm

Before I started adding any code to the file, I first looked at how the main body of the script was written, as this gave me a good idea of how and where each function was called and how the returned values were being used. The starter code had two existing classes for the functions: Processor (for processing the data), and IO (for presentation, input and output of data).

In order to keep it clear which comments were newly added by me rather than existing from the starter script, I started each comment I added with my initials PT.

2.1. Processor Class

I started the code-writing process with completing the functions in the Processor class.

The Processor class starter code came with the following functions defined for use:

- *read_data_from_file* (Reads data from a file into a list of dictionary rows)
- *add_data_to_list* (Adds data to a list of dictionary rows)
- *remove_data_from_list* (Removes data from a list of dictionary rows)
- *write_data_to_file* (Writes data from a list of dictionary rows to a File)

The code written into the above functions is covered in the following subsections of this paper.

2.1.1. *read_data_from_file* Function

The code from the starter file for the *read_data_from_file* function was already complete. The parameters for this function are *file_name* and *list_of_rows*. This function uses a for loop similar to assignment 05, going through each line in the file, splitting into two values, and then adding those to a dictionary with keys *Task* and *Priority* assigned to local variable *row.* The list with local variable *list_of_rows* is then appended to add any additional dictionary lines created from the file data. The updated *list_of_rows* list is then returned to end the function. The code for this section is shown below (Figure 2).

```
26      @staticmethod
27      def read_data_from_file(file_name, list_of_rows):
28          """ Reads data from a file into a list of dictionary rows
29
30          :param file_name: (string) with name of file:
31          :param list_of_rows: (list) you want filled with file data:
32          :return: (list) of dictionary rows
33          """
34          list_of_rows.clear()  # clear current data
35          file = open(file_name, "r")
36          for line in file:
37              task, priority = line.split(",")
38              row = {"Task": task.strip(), "Priority": priority.strip()}
39              list_of_rows.append(row)
40          file.close()
41          return list_of_rows
```

Figure 2: *read_data_from_file function within the Processor class*

Note that this function, as well as all other functions in this program, is proceeded with the *@staticmethod* decorator, which allows for calling of the function outside of the class.

2.1.2. *add_data_to_list* Function

The *add_data_to_list* function was mostly complete in the starter code, with just one missing line. The existing code showed the function with the parameters *task*, *priority*, and *list_of_rows*. The local variable row had a dictionary made of the *task* and *priority* parameters. Finally, the function showed that it would return *list_of_rows*. Since I knew from the main code and the docstring of this function that this would need to add a new task and priority item to the overall list of all tasks, I knew that I needed to append the *list_of_rows* parameter with the *row* dictionary. I did this by simply adding the line:

```
list_of_rows.append(row)
```

The final code for the function is shown below (Figure 3)

```
43      @staticmethod
44      def add_data_to_list(task, priority, list_of_rows):
45          """ Adds data to a list of dictionary rows
46
47          :param task: (string) with name of task:
48          :param priority: (string) with name of priority:
49          :param list_of_rows: (list) you want to add more data to:
50          :return: (list) of dictionary rows
51          """
52          row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
53          list_of_rows.append(row)  # PT: appending table with new entries as new dictionary row
54          return list_of_rows
```

Figure 3: *add_data_to_list function within the Processor class*

2.1.3. *remove_data_from_list* Function

The starter code defined the *remove_data_from_list* function and showed that the return would be *list_of_rows*, but the code to perform the process was blank, for me to add to finish the function. The code for this function is shown below (Figure 4).

```
56      @staticmethod
57      def remove_data_from_list(task, list_of_rows):
58          """ Removes data from a list of dictionary rows
59
60          :param task: (string) with name of task:
61          :param list_of_rows: (list) you want filled with file data:
62          :return: (list) of dictionary rows
63          """
64          for row in list_of_rows:
65              if row["Task"].lower() == task.lower():  # PT: searches each row in table for entered value
66                  list_of_rows.remove(row)  # PT: removes row from table if it has selected value
67                  #removal_status = True  # PT: flag to show removed
68          return list_of_rows
```

Figure 4: *remove_data_from_list function within the Processor class*

The existing code showed that the function would receive a task value through the parameter *task*. In order to remove the desired task, a For loop is needed. It is already established in the other portions of the starter code, as well as the docstring of the function, that the *list_of_rows* parameter is a list made up of "rows" (elements) of dictionaries. The For loop goes through each "row" and compares the element with key "Task" in the dictionary to the value of parameter *task*. If there is a row that has an element matching *task*, then that row (dictionary element of list) is removed using the *remove* method on the list. The updated list is then returned as *list_of_rows*.

This code was almost identical to the one I used in Assignment 05 for the same task, but in this program, the user input of selecting the task to be removed is separated from this processing code. Instead of taking the input and assigning to a global variable that is compared to values in the dictionary, this function receives the value to compare as a parameter and then compares that to the dictionary values.

2.1.4. *write_data_to_file* Function

Similar to the *read_data_from_file* function, the *write_data_to_file* function has the parameters of *file_name* and *list_of_rows*, and returns *list_of_rows*. The code used in this function is very similar to what I used in Assignment 05 for saving to a file. I used the simple *file* local variable rather than more complex global variable in Assignment 05, and assigned the value received as parameter *file_name* opened in write-mode to that *file* variable. The For loop is the same that I used in Assignment 05, just with the file variable changed to match this program. The code for this function is shown below (Figure 5).

```
70      @staticmethod
71      def write_data_to_file(file_name, list_of_rows):
72          """ Writes data from a list of dictionary rows to a File
73
74          :param file_name: (string) with name of file:
75          :param list_of_rows: (list) you want filled with file data:
76          :return: (list) of dictionary rows
77          """
78          file = open(file_name, "w")  # PT opens file with passed in file_name parameter
79          for row in list_of_rows:
80              file.write(str(row["Task"]) + "," + str(row["Priority"]) + "\n")  # PT: saves each row to file
81          file.close()
82          # saveStatus = True  # PT: toggles save status to true once data has been saved to file
83          return list_of_rows
```

Figure 5: *write_data_to_file function within the Processor class*

2.2. IO (Input / Output) Class

The next step in writing the code was completing the functions in the IO (Input / Output) class.

The IO class starter code came with the following functions defined for use:

- *output_menu_tasks* (Display a menu of choices to the user)

- *input_menu_choice* (Gets the menu choice from a user)
- *output_current_tasks_in_list* (Shows the current Tasks in the list of dictionaries rows)
- *input_new_task_and_priority* (Gets task and priority values to be added to the list)
- *input_task_to_remove (Gets the task name to be removed from the list)*

As a way to demonstrate my knowledge of functions and maintain some of the functionality I added in my Assignment 05 code, I also added the following function to the IO class:

- *input_yes_or_no* (gets input yes or no)

The code written into the above functions is covered in the following subsections.

2.2.1. *output_menu_tasks* , *input_menu_choice* , and *output_current_tasks_in_list* Functions

The first three functions in the IO class: *output_menu_tasks*, *input_menu_choice*, and *output_current_tasks_in_*list were already complete in the starter code.

The code for the *output_menu_tasks* function, shown below (Figure 6), simply prints out the menu options to the user. It does not receive any values as parameters and does not return anything. The purpose of this function is to keep the IO print statements out of the main code, keeping the code organized and more concise.

```
92      @staticmethod
93      def output_menu_tasks():
94          """  Display a menu of choices to the user
95
96          :return: nothing
97          """
98          print('''
99          Menu of Options
100         1) Add a new Task
101         2) Remove an existing Task
102         3) Save Data to File
103         4) Exit Program
104         ''')
105         print()  # Add an extra line for looks
```

Figure 6: *output_menu_tasks function within the IO class*

The code for the *input_menu_choice* function is shown below (Figure 7). This function simply prompts the user to type in a number that mentions the menu options presented in the previous function, assigns that to local variable *choice* and returns that string value to end the function.

```
    1 usage
    @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

```

Figure 7: *input_menu_choice function within the IO class*

The *output_current_tasks_in_list* function receives a list value as parameter *list_of_rows*, and prints the two elements of each dictionary row by calling out their keys "Task" and "Priority" in a print function within a for loop. The code itself is similar to what was done in Assignment 05 for printing back current data to the user when that option was selected, but doing it in a function like this is useful in reducing the number of lines in the main code for having to type out the print for loops within the main body of the code. The code for this function is shown below (Figure 8).

```
    @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******* The current tasks ToDo are: *******")
        for row in list_of_rows:
            print(row["Task"] + " (" + row["Priority"] + ")")
        print("*******************************************")
        print()  # Add an extra line for looks
```

Figure 8: *output_current_tasks_in_list* function within the IO class

2.2.2. *input_new_task_and_priority* and *input_task_to_remove* Functions

The *input_new_task_and_priority* and the *input_task_to_remove* functions were the two functions started in the starter code that needed to be completed. The code for these two functions is shown below (Figure 9 and Figure 10, respectively).

```
130     @staticmethod
131     def input_new_task_and_priority():
132         """  Gets task and priority values to be added to the list
133 
134         :return: (string, string) with task and priority
135         """
136         new_task = input("Task name: ").title().strip()  # PT:  captures user input and uses strip method
137         new_priority = input("Task priority (High, Medium, Low): ").capitalize().strip()  # PT
138         print()  # PT: Add an extra line for looks
139         return new_task, new_priority
```

Figure 9: *input_new_task_and_priority* function within the IO class

```
142     def input_task_to_remove():
143         """  Gets the task name to be removed from the list
144 
145         :return: (string) with task
146         """
147         removal = input("Which task would you like to delete? ").strip().title()  # PT: captures user input to return
148         print()  # PT: Add an extra line for looks
149         return removal
```

Figure 10: input_task_to_remove *function within the IO class*

Both functions take user input, assigns them to local variables that I defined within those functions, then returns the values. Both of the function's code is similar to the *input_menu_choice* function. The main difference is that the *input_new_task_and_priority* function uses two local variables and two input statements, and it returns the two input values, rather than just one.

2.2.3. *input_yes_or_no* Function

I wanted to include the feature that my program in Assignment 05 had that recognized if the user had unsaved data or not when they select the Exit Program option and prompt them to choose if they would like to save it or not. I took this as an opportunity to add a new function to the IO class, as shown below (Figure 11). This *input_yes_or_no* function is intended to be one that can be reused in other programs, as it is a common input request in programs. In order to keep this function more generic, I decided to enter the string text for the input question as an argument that is received by the function as parameter *question*. This allows the specific wording of the input to be tailored to the program usage without needing to modify the code within the function, provided the input is looking to capture a Yes or No response.

```
151     @staticmethod
152     def input_yes_or_no(question):
153         """  Gets choice of yes or no
154
155         :param: question: (string) with text for input prompt:
156         :return: (string) with save choice
157         """
158         response = None
159         while response not in ("y", "n"):
160             response = input(question).lower().strip()
161         return response
```

Figure 11: *input_task_to_remove* function within the IO class

The function then returns the lowercase string "y" or "n" value to end the function. This code was based off of the code in paragraph "Receiving and Returning Values in the Same Function" in Chapter 6 of the class textbook **(Dawson, 165)**.

Writing my own function from the start was good practice in filling out and formatting the docstring as well. I originally forgot to add the information about the *question* parameter in the docstring, and I did not notice it until the time of creating this knowledge document. It took a couple of tries to get the colon on the "param" line in the docstring correct. I verified whether or not the docstring was displaying correctly by hovering over the function name in PyCharm, as shown below (Figure 12).

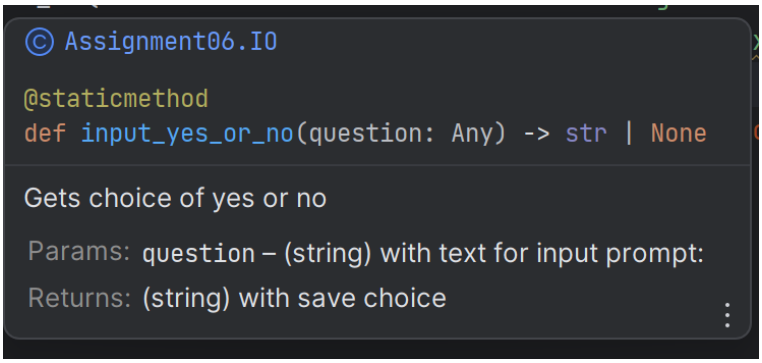

Figure 12: Verifying the docstring for the new function

2.3. Main Body of Code

The code for the main body of the program was complete and working in the starter code. Because I added the new *input_yes_or_no* function, I had to modify the code in the main body to call that upon the selection of option 4 with unsaved data. First, the program would need to know when unsaved data existed. I did this using a Boolean flag, similar to Assignment 05. I declared the new global variable *save_status* along with the existing variables and constants, as shown below (Figure 13).

```
12  # Data ---------------------------------------------------------------------- #
13  # Declare variables and constants
14  file_name_str = "ToDoFile.txt"  # The name of the data file
15  file_obj = None  # An object that represents a file
16  row_dic = {}  # A row of data separated into elements of a dictionary {Task,Priority}
17  table_lst = []  # A list that acts as a 'table' of rows
18  choice_str = ""  # Captures the user option selection
19  save_status = True  # PT: flag for save status
```

Figure 13: Variables and constants

That initial value of *save_status* is set to True, as there wouldn't be any changes to the data until the user selected either removing or adding new tasks. The status value is then assigned as False within the main body after calling either the *add_data_to_list* or *remove_data_from_list* functions, as that would indicate a change to the data. Once the *write_data_to_file* function is called in the main body and run, the *save_status* is assigned the value of True, as that means all changes have been saved to the file.

The code for the main body of the script is shown below (Figure 14). The *if* and *elif* statements for the save status are similar to that of Assignment 05, with changes made to use functions. As mentioned in section of this paper, the text for the input is added as an argument in lines 199 and 200, to be passed to the function *input_yes_or_no*.

```
str_save_pick = IO.input_yes_or_no(" ATTENTION: You have unsaved changes."
                        " Would you like to save before exiting? (Y or N) ")
```

(space intentionally left blank)

```python
# 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)
        save_status = False  # PT: sets save status flag to false
        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)
        save_status = False # PT: sets save status flag to false
        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)
        print("Data Saved!\n")
        save_status = True  # PT: sets save status flag to false
        continue  # to show the menu

    elif choice_str == '4':  # Exit Program
        if save_status:  # PT: evaluates if true then breaks and exits
            break
        elif not save_status:  # PT: evaluates if not saved (false) then prompts for save choice below
            str_save_pick = IO.input_yes_or_no(" ATTENTION: You have unsaved changes."
                                               " Would you like to save before exiting? (Y or N) ")
            if str_save_pick == "y":
                table_lst = Processor.write_data_to_file(file_name=file_name_str, list_of_rows=table_lst)
                print()  # PT: Blank line for looks
                print("Data Saved!")
                break  # and Exit the program
            elif str_save_pick == "n":
                break
            print("Goodbye!")
input("\n\nPress the enter key to confirm and exit.")  # PT: Exits whole program upon user input
```

Figure 14: Main Body of Script

3. Running the Program

The program eventually ran successfully both in PyCharm (Figure 15) as well as the Windows command window (Figure 16). Initial runs produced some errors that I was able to fix using the error messages and stepping through some lines in the PyCharm debugging mode. The debugging mode in PyCharm was not necessary, but it was a good opportunity to use it when there were errors. The few errors I got were generally due to missing replacing variables from code I copied from Assignment 05 to replace them with the local variables in the functions. This is something that I need to pay better attention to going forward when it comes to copying from existing code as well as adding to existing starter code. It was not difficult to spot the errors in this assignment, but it will likely be much more difficult in longer, more complex code.

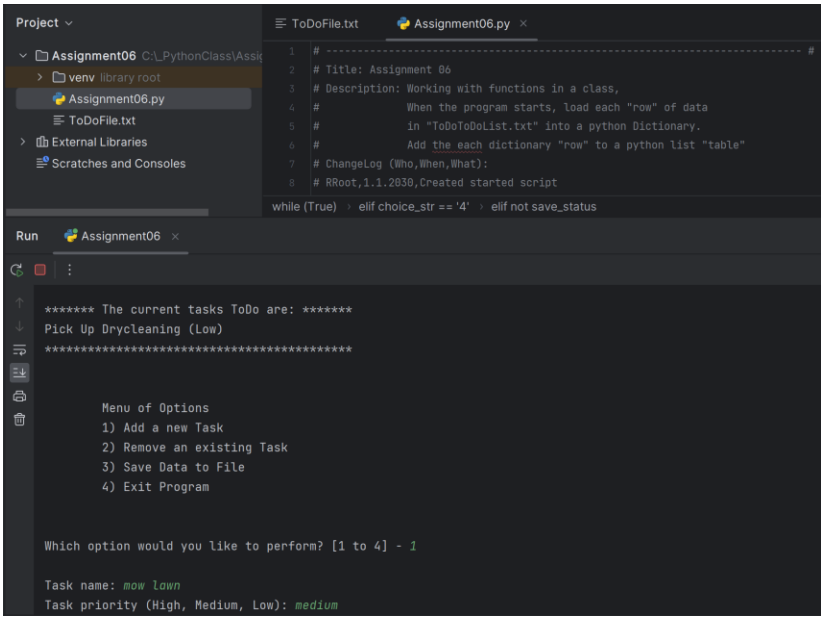

Figure 15: Running Program in PyCharm

```
C:\_PythonClass\Assignment06>Assignment06.py

******* The current tasks ToDo are: *******
Pick Up Drycleaning (Low)
Mow Lawn (Medium)
*****


        Menu of Options
        1) Add a new Task
        2) Remove an existing Task
        3) Save Data to File
        4) Exit Program


Which option would you like to perform? [1 to 4] - 1

Task name: Wash dishes
Task priority (High, Medium, Low): high


******* The current tasks ToDo are: *******
```

Figure 16: Running Program in Windows command window

I verified that all data was saved in the text file, as shown below (Figure 17).

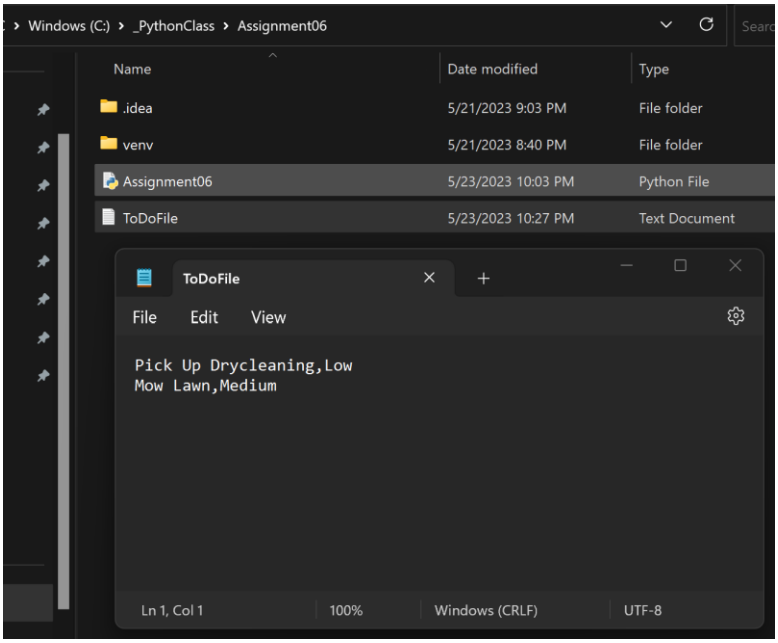

Figure 17: Text file with data saved from program, folder shown in background

4. Summary

This program was a good introduction into using classes and functions to organize code. Since the algorithm and theory was the same at the core for this program as Assignment 05, the focus was on being able to complete the functions in the starter code. While the code added to the functions was simple and familiar, completing the program required being able to understand arguments, parameters, local variables, and returning values from functions.

Because the starter code contained the arguments for the calling of the existing functions, there was not a lot of opportunity to make a mistake on those—the main thing in this assignment was to understand how those arguments were passed to the functions via parameters and use them correctly within the functions.

The knowledge learned in Module 06 and this assignment will be useful in making more creative programs from start to finish, while also being able to successfully use code from other sources without much error fixing.

Works Cited

Dawson, Michael. *Python Programming for the Absolute Beginner*, Third Edition, Cengage Learning, 2010