Name: Aimee Richardson
Date: 02/19/2023
Course: IT FDN 110 - Foundation of Programming: Python
Assignment: Assignment 6
GitHub URL: https://github.com/amrich1111/IntroToProg-Python

Assignment 06 - Function(al) To Do List

Introduction

In Assignment06, the use of functions to organize the To Do List program is explored. Beginning with starter code, this assignment tests the ability of one programmer to understand and build upon another's pre-existing code. The goal of this assignment is to successfully utilize the ToDoList code from Assignment05 and reimagine it to fit into the given starter code, reorganizing the conditional statements into functions within classes. In the following documentation, the process of understanding the starter program and restructuring the code from Assignment05 into functions is explained further.

Creating the Script

Understanding the Starter Code

Similar to previous Assignment05, in Assignment06 a key element of creating the program was being able to understand and build upon a pre-existing program. In this case, the starter code given had a few sections to unpack. First, being the data variable decelerations. It was important to understand these since some would be utilized to store data within functions throughout the program. Specifically file_obj, which would be used to read, store, and write data to the to-do list in the processing step of the program and choice_str which is used to capture user input in the input/output step of the program.

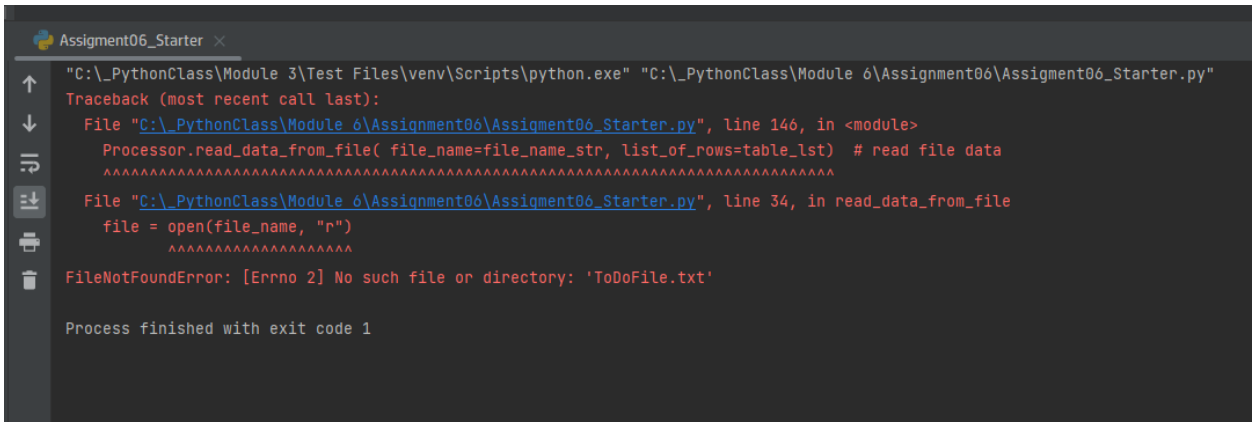
Further on, two classes were predefined which each held functions to execute certain pieces of code depending on what the code intended to do. In this case, a Processor class is defined to hold functions related to processing the data – this being either adding, removing, or saving data. The other, named IO, handles prompting and storing all instances where user input/output is needed. Thinking back to Assignment05, both the IO and processing steps happened within the conditional statements – this indicates that we will need to break apart our code from Assignment05 and reorganize the different parts into the correct functions. For each menu choice, there is both a processing function and/or an IO function to successfully prompt the user and execute given the task.

In the main body of the script, the same menu and conditional statement from Assignment05 is looped through and allows the user to either add to, remove, or save their data and exit the program at any time. Within each conditional statement, each class's function is utilized to both capture the user's input, if needed, and execute the task they have selected.

Building the Functions

When working with starter code that has multiple sections, it is important to test the program in small pieces while building it. For this assignment, actually writing the code to execute each task was not difficult since it was previously built in Assignment05. The more challenging part, however, was figuring out how to allocate it correctly within the different functions and ensuring the entire program was still functional. I started building the program by going through each menu option and determining which part of the conditional statements fit into each function. As said above, the IO class executes functions which prompt, store, and return user input and the Processor class takes care of all the actions which need to be performed on or using the data.

Reading Data from the File



For this assignment, new code did not need to be added in order to read the data from the existing ToDoList.txt file. However, when running the code as is, the program would throw an error if the called file did not exist in the current directory (Figure 6.1). To avoid this error, I ended up applying a try-except block, similar to Assignment05 (Figure 6.2). This will allow the program to attempt to read the file if it exists, but if it does not, it will continue as expected and create the file later on when the user saves their data.

```
list_of_rows.clear()  # clear current data
try:
    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:
    pass
return list_of_rows
```

Figure 6.2 - Assignment06 script, implementing a try-except block to avoid an error if the called file does not exist in directory

Adding Data to the List

```
strTask = input("Task: ").strip().title()
strPriority = input("Priority: ").strip().title()
lstTable.append({"Task": strTask, "Priority": strPriority})
```

Figure 6.3 - Assignment05 code to add a task to the to do list. For the IO class, only the highlighted input statements need to be added.

In Assignment05, when adding data to the list, the user was asked to input a task and a priority, and then the program would store and append the entry. For the IO class, only the input lines are needed to execute the function, the action of appending the entry to the table would be done in the Processor class (Figure 6.3). Overall, these lines just needed to be copied and pasted into the input_new_task_and_priority() function. The only additional thing to consider is the return statement. In the main script of the program, this function is unpacked and the output is passed into the related Processor class function. To ensure this runs as expected, the function is set to return the inputted variables in a tuple (Figure 6.4).

```
@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!
    strTask = input("Task: ").strip().title()
    strPriority = input("Priority: ").strip().title()
    print()
    return strTask, strPriority
```

Figure 6.4 - Assignment06, IO class function which prompts user for input returns the user inputted task and priority

In the Processor class, there is a function which executes the task of adding the data to the list. In the starter code, storing the task and priority in a dictionary is already done, the only addition that needed to be made was adding the final row of code from Figure 6.1, which appends the user input to the table, and updating the variable passed to the append statement to be consistent with the starter code.

Removing Data from the List

My previous code, which removed data from the list, was a bit more complicated to translate into the functions. I wanted to retain the functionality, which allows the user to select which task to remove in the case that there are multiple with the same name. The entire conditional statement could have been inputted into the Processor class, however in doing this, there were multiple cases where user input/output was embedded within the processing which contradicts the purpose of the class. In hopes of improving this, I decided to create additional functions in Assignment06 to clean it up a bit.

This required a few additional IO and Processor class functions:

1. **New IO Class Functions**
    a. **output_multiple_tasks_to_remove** → function to display a list of tasks that match the user's inputted task.
    b. **input_index_to_remove** → function to ask user which number task they would like to remove from the list outputted (above function), returns an integer with the index of the task they would like to remove, which is later used in the main remove function to .pop() the desired task from the to-do-list.
    c. **output_was_task_removed** → function to print a message to the user, stating whether or not the action was successful, takes a boolean as a parameter to indicate which message to print.
        i. This was implemented to display whether or not the task was removed. In case the user entered an invalid task name OR an invalid index, the program will print "Task was not found".
2. **New Processor Class Functions**
    a. **count_number_of_tasks** → function to iterate over the current list and count how many times the user inputted task appears in the table, returns the total number of tasks.

These new functions were implemented within the original remove_data_from_list function which required a few updates as well:

```
elif choice_str == '2':  # Remove an existing Task
    task = IO.input_task_to_remove()    # Ask user which task to remove
    task_count = Processor.count_number_of_tasks(task=task, list_of_rows=table_lst)     # Count instances of task
    table_lst, removed_bool = Processor.remove_data_from_list(task=task, task_count=task_count, list_of_rows=table_lst)
    print(IO.output_was_task_removed(removed_bool))
    continue  # to show the menu
```

Figure 6.5 - Assignment06 script, updates made to conditional statement in main script

1. A new line was added to the main script which saves the task count returned by the Processor function, count_number_of_tasks, in a variable (Figure 6.5).
2. A new task count parameter was added to the remove function.
3. A new variable was defined within the remove_data_from_list function to assess whether the task was removed.
4. The return statement of the remove function was updated to return a tuple, now outputting both the list of rows and the boolean value, which are unpacked in the main script conditional statement (Figure 6.5).

There were a few iterations of this until the final code was achieved. I started by breaking out each piece of the code based on the task that was being executed. As mentioned in Lecture 6, each function should be responsible for executing a specific task. In this case, the main remove data function does call upon multiple other functions, but, it is overall more concise compared to how it started.

```
taskCount = 0
for row in lstTable:
    if (row["Task"].lower() == strRemoveTask.lower()):
        taskCount += 1
if(taskCount > 1):  # If there are multiple tasks with the same Task name, ask user which to delete
    print("\nMultiple tasks found. Indicate which task you would like to remove!\n")
    for row in lstTable:
        if (row["Task"].lower() == strRemoveTask.lower()):
            taskIndex = lstTable.index(row)
            print(str(taskIndex + 1) + ". " + row["Task"] + " | " + row["Priority"])
    removeChoice = input("\nMultiple tasks found. Please enter the task number you would like to remove: ")
    try:    # Try to remove inputted item from list, if index does not exist display message to user
        lstTable.pop(int(removeChoice) - 1)  # Using pop to remove item from list based on index
        print("The task has been removed!\n")
    except:
        print("\nInvalid entry! Task entered does not exist.")
elif(taskCount == 1):   # If only 1 task has the inputted name, remove item
    taskIndex = ""
    for row in lstTable:
        if (row["Task"].lower() == strRemoveTask.lower()):
            taskIndex = lstTable.index(row)
    lstTable.pop(taskIndex)
    print("The task has been removed!\n")
else:   # If no task has the inputted name, display message to user
    print("Task not found!")
continue
```

Figure 6.6 - Assignment05 script, remove data conditional statement

```
    # TODO: Add Code Here!
    removed_bool = True
    if task_count > 1:  # If there are multiple tasks with the same Task name, ask user which to delete
        IO.output_multiple_tasks_to_remove(list_of_rows)
        remove_choice = IO.input_index_to_remove()
        try:  # Try to remove inputted item from list, if index does not exist display message to user
            list_of_rows.pop(int(remove_choice) - 1)  # Using pop to remove item from list based on index
        except:
            removed_bool = False
    elif task_count == 1:  # If only 1 task has the inputted name, remove item
        task_index = ""
        for row in list_of_rows:
            if row["Task"].lower() == task.lower():
                task_index = list_of_rows.index(row)
        list_of_rows.pop(task_index)
    else:  # If no task has the inputted name
        removed_bool = False
    return list_of_rows, removed_bool
```

FIgure 6.7 - Assignment06 script, remove data function

Overall, a lot of changes had to be made to the original starter script to work this step out. Although it was most time intensive, the resulting code (Figure 6.7) versus the previous code (Figure 6.6) looks much cleaner and reduced the total number of lines within the function.

Saving Data to the File

Saving the user inputted data to the file required less effort to implement. The same logic was used as the previous assignment, utilizing the open() method along with the ‘w’ parameter and a for loop to iterate over the list to write the data to the file. The code used in the Assignment06 function was essentially the same as Assignment05 – the only notable changes that were needed were:

1. Updating variable names to be consistent with the starter code (i.e. using file_obj vs objFIle).
2. Implementing a return statement in the function which returns the current list of dictionaries.

Executing the Script

Executing in the Command Shell

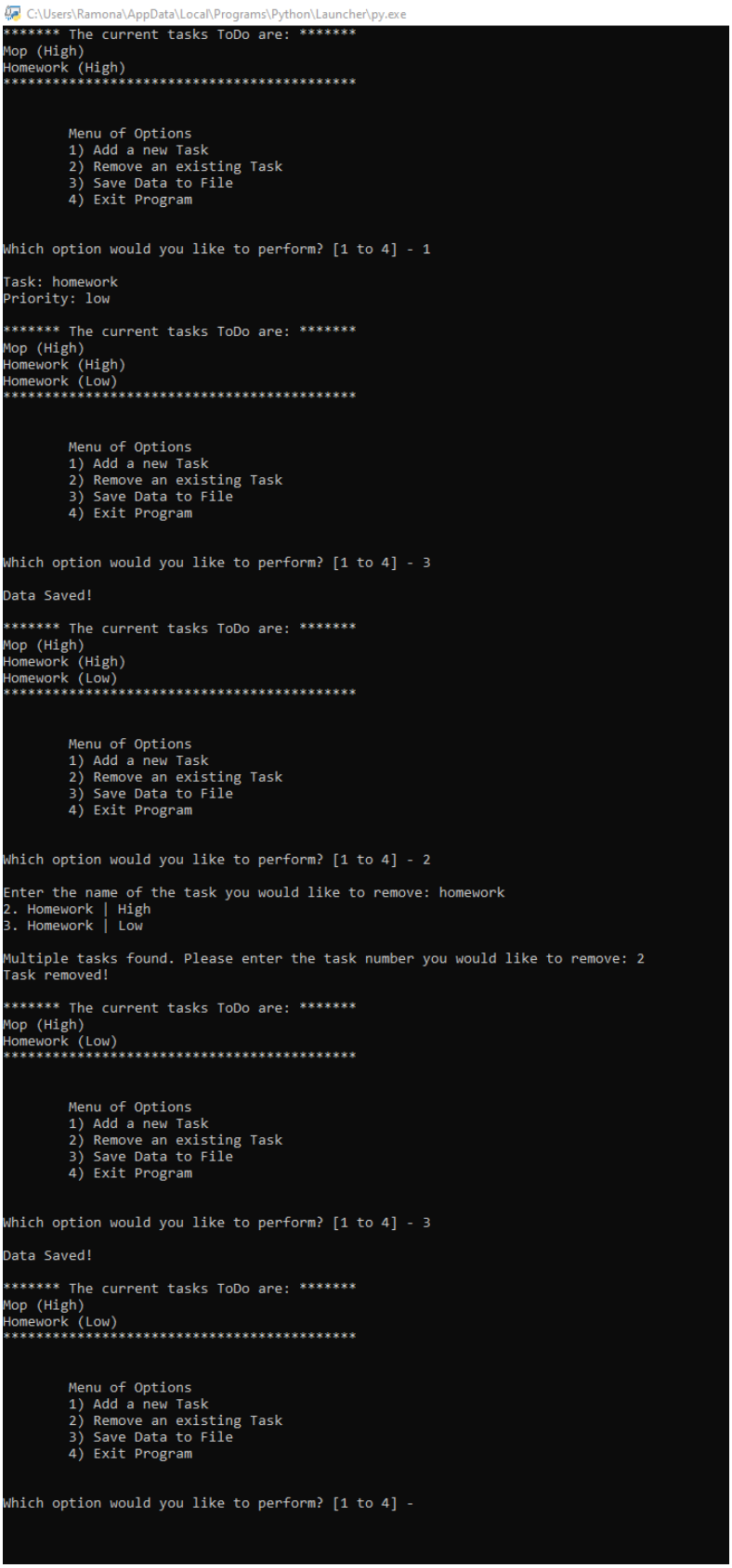


Figure 6.8 - ToDoList script executing in the Command Shell

Executing in the PyCharm

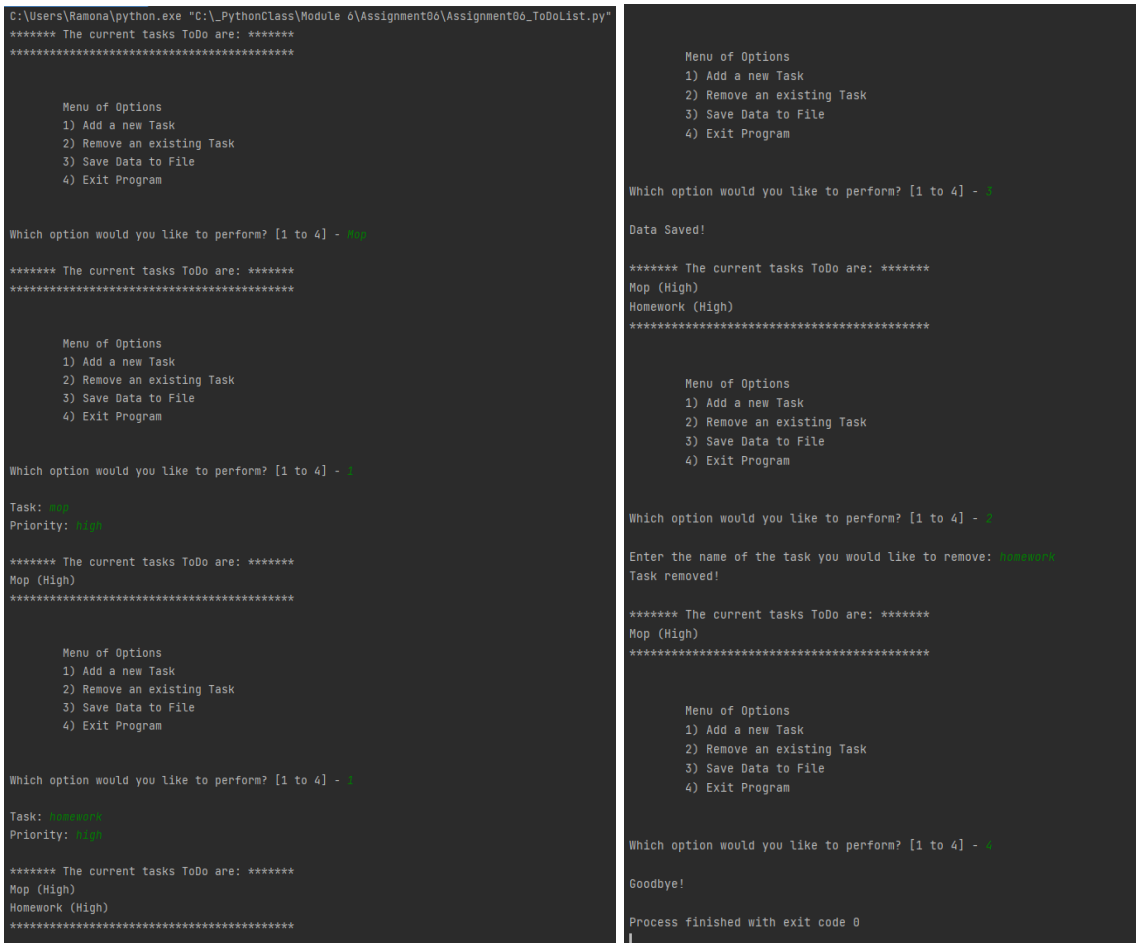


Figure 6.9 - ToDoList script executing in PyCharm

The ToDoList script was able to run successfully in the Command Shell (Figure 6.8) and PyCharm (Figure 6.9), executing all menu functions as intended. Various user input was also tested to ensure the program ran consistently in both environments.

Summary

Assignment06 built on code from Assignment05, with the challenge of reorganizing the code into functions. Further, these functions were separated into two separate classes based on the overall function of the code. By working with the starter code and breaking up the original script into functions, we were able to successfully build a more organized and concise program with the same capabilities, allowing users to modify a to-do-list by adding and removing items, as well as save their data to files for later use.