Name: Aimee Richardson
Date: 02/13/2023
Course: IT FDN 110 - Foundation of Programming: Python
Assignment: Assignment 5
GitHub URL: https://github.com/amrich1111/IntroToProg-Python

Assignment 05 - To Do List

Introduction

Assignment05 continues to explore the use of collections in Python and how they can be utilized to store and display user input, as well as output data to files. Previously, in Assignment04, lists were the main collection used, however in this assignment dictionaries are introduced and utilized to execute similar actions to store user input to build and modify a to-do list. One of the other main focuses of this assignment is successfully developing a program starting with code created by another user. In the following documentation, the process of understanding pre-made programs and developing the program further to allow users to create and modify their to-do-list is explained.

Creating the Script

Building Off of the Starter

An important part of completing Assignment05 successfully was building off of the starter code provided by another developer. In theory, this sounds like it may be simple, however during the process of creating the script it yielded some challenges. One mistake that probably made my process more difficult was writing out each step of the code prior to testing it. The starter code is presented with the important variables defined and the menu fleshed out with the related conditional statements. Each menu option was associated with an if/elif conditional statement in which code had to be added in order to successfully execute the task selected by the user. I chose to update a few things to make the code more consistent, such as the quotes since I tend to use double quotes in my programs.

Reading Files

In the start of the program, it was important to determine if the user's to-do-list already existed and had data. If the file already existed with previous input, the program needed to load the information from the file into a collection in Python. In order to execute this, a few different questions needed to be considered:

1. Is there a file? And if there isn't, how will the program handle this?
2. How can this data be stored in Python so that it can be utilized by the user later on if they choose?

For (1), we need to attempt to read the file if it exists. To do this, the file is opened using the open() method with the parameters set to the file (toDoFile = ToDoList.txt) and the action is set to "r", meaning the program will simply read the data in the file. This is the important step that determines whether or not the file exists. If there is not a file, the program would return an error to the user (Figure 5.1)

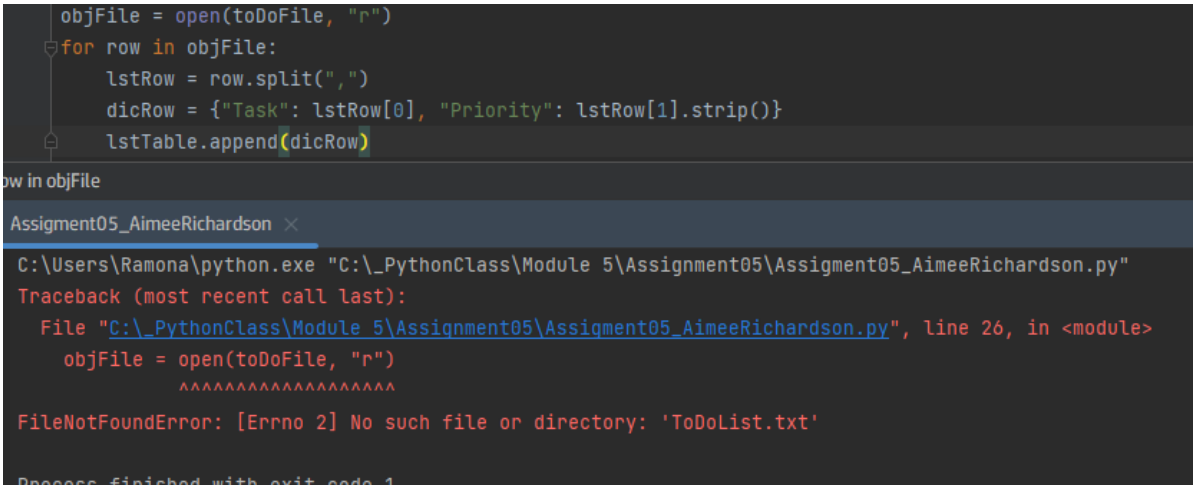

Figure 5.1: When trying to read a file that does not exist, the program will return an error

Reading the file in the beginning is necessary since the program intends to allow the user to continue to build upon their list if it has already been created. Each time the program runs, it does not retain memory of whether the user has used the program prior, therefore it does not know if the file exists. To account for this unknown situation, and considering that an error is possible if the file does not already exist, a try except block was used to handle any potential errors the program may encounter. By implementing a try except, the program is able to bypass errors at certain points in the code. In this case, it is beneficial considering the need to load all current data into the program. The code for reading the file was inspired by example code from Lecture 5.

Displaying the Current Data

The first option in the menu tells the program to display all the data in the user's to-do-list. This must include all the modifications the user has made throughout using the program as well as any items that already existed in the saved file. Reading the file at the very start of the program helps this piece of the code run accurately. To execute this, a conditional statement was used to check the length of the current list table. If the list's length is 0, indicating that there is no data, the program will display a message to the user. If data is available, the program will loop through each row in the list table and print it in a user-friendly format.

Adding Tasks to the List

The second option in the menu allows the user to input new tasks to their list. To execute this, two new variables were defined to save each of the user's input. The strip() method was also utilized to ensure the input saved did not include any extra white space potentially inputted by the user – this ensures the data is as clean and consistent as possible for later use.

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

Figure 5.2 - User input is saved in variables and added to the list within a dictionary

Finally, to add this new row to the table, the new task and priority are added onto the list. In order to save each item along with their relative key, the input variables are grouped into a dictionary which is then added to the larger list using the append() method (Figure 5.2).

Removing Tasks from the List

The third option in the menu allows the user to remove a task. Perhaps if the user mistakenly adds a task or if they've completed a task, it is useful to have the ability to remove records from the list. To execute this, a few things had to be considered:

1. Which task does the user want to remove?
2. How to find the inputted task within the list to remove

First, a variable is defined which prompts the user to input a task they would like to remove, utilizing the same .strip() method mentioned above. Utilizing this method in both instances allows this step to execute as intended. For example, if the user had accidentally inputted white space for one input, the program would not be able to identify the correct task to remove since the string values would be slightly different.

```
for row in lstTable:
    if (row["Task"].lower() == strRemoveTask.lower()):
        lstTable.remove(row)
```

Figure 5.3 - Utilizing a for loop and a conditional statement to iterate over the current list and remove the selected task.

Once the task to remove is identified and saved in a variable, the program must find the correct task to remove in the table. My first thought to execute this was to loop through the table using a for loop and then nesting a conditional statement within the for loop to determine if the task inputted matched the current row's task and remove it (Figure 5.3). Upon further research, I found that modifying a list while iterating over it is not recommended since it modifies the indexes while looping over it, which could potentially cause the loop to skip (Loeber, 2021). In an effort to find a better solution, I decided to change my method of removing the 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")
```

Figure 5.4 - Iterated over list to find task and save the index, then removed the task utilizing the pop() method

To execute this, I chose to iterate over the list with the same conditional statement, but rather than removing the item while iterating over the list, the program saves the index of the task in a variable. Once the iteration ended, the task was then removed using the pop() method since it can remove an item from a list based on its index (Figure 5.4).

To further experiment, I chose to also account for the situation where multiple tasks have the same name in the list. To account for this, I implemented a counter to iterate over the list and count how many tasks matched the user's input. If the count was greater than 1, the user would then be prompted with the option to select which task they would like to delete by displaying them in a list. In order to do this successfully, I used similar logic as mentioned above, using the index of the task to remove it (Figure 5.5). To execute this, a few additions were made to the code in Figure 5.4:

1. A print statement was added within the conditional statement to list the matched tasks to the user
2. A removeChoice variable was defined so the user could input which task they would like to remove
3. A try except block was added in case the user entered an invalid choice

```
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("\nPlease 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.")
```

Figure 5.5 - Implementing a task counter to determine if multiple tasks match user's input and a for loop to display tasks so the user can decide which to remove.

By implementing this option, the user has further control over which item they want to remove from their list. If the remove() method was used, the code likely would have been simpler, but it would remove the first occurrence of the value which may not be what the user intended (Dawson, 2010).

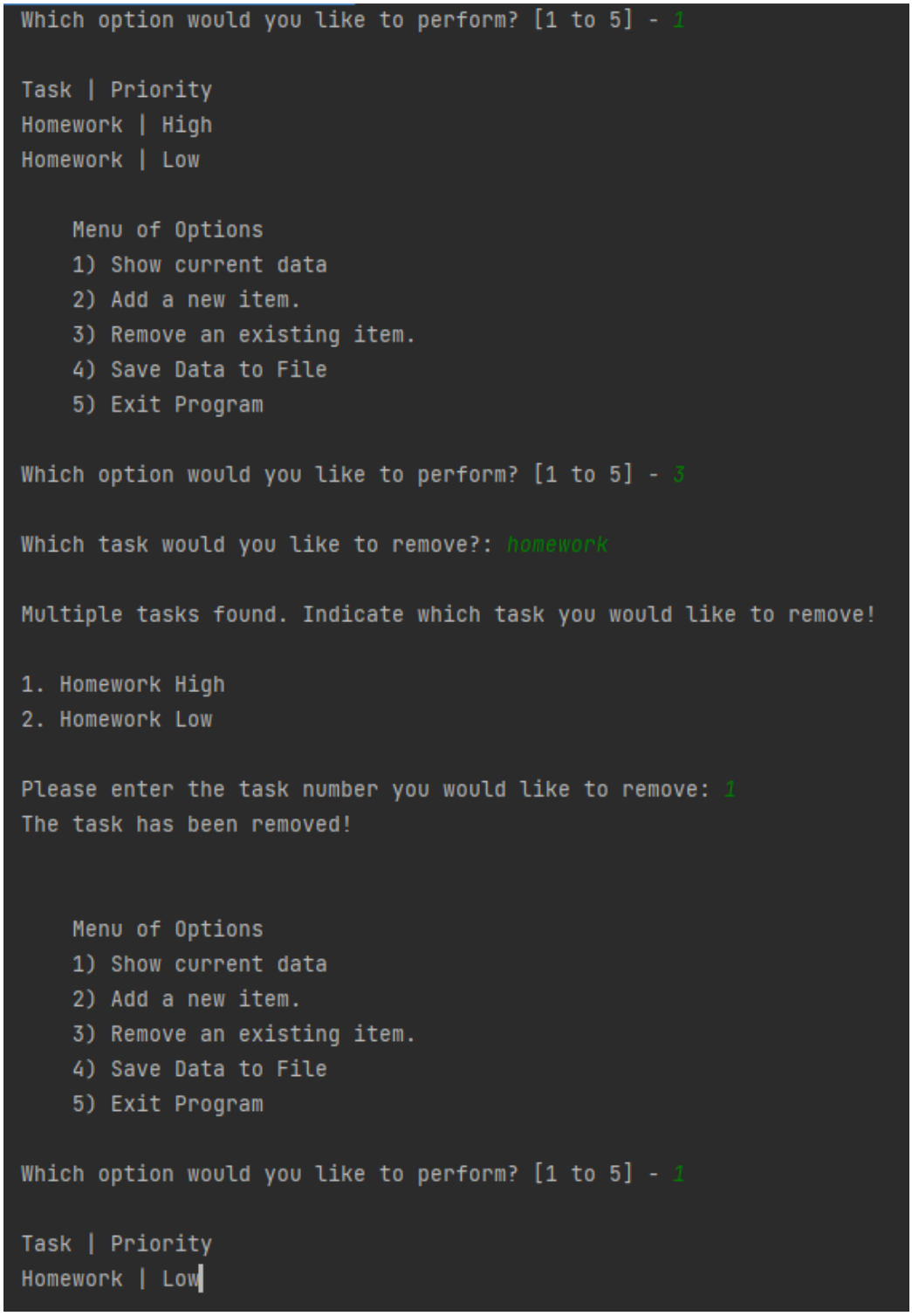

Figure 5.6 - Multiple items in task list with the same name, program displays tasks to user and asks which to delete

When a user enters multiple tasks, such as ‘Homework’, the program will display a message indicating multiple Homework tasks have been found and ask the user to select which to remove from the list (Figure 5.6).

Saving Current Data to the File

The fourth option in the menu tells the program to save the current data to the file. Saving the data to the file allows all the user's progress to be saved and accessed again after exiting the program. To do this, the program must go through the entire list of tasks either added or modified by the user and add it to the file.

```
objFile = open(toDoFile, "w")
for row in lstTable:
    objFile.write(str(row["Task"]) + "," + str(row["Priority"] + "\n"))
objFile.close()
print("To Do List Saved!")
continue
```

Figure 5.7- The ToDoList.txt file is opened and a for loop is utilized to iterate over the current list data and each Task-Priority pair is written to the file.

To execute this, the file needs to be opened and the write ("w") parameter must be defined to indicate that new data is being added to the file. In order to add all of the new data from the list, a for loop is utilized to iterate over the entire list table row by row, and write each task and priority to the file (Figure 5.7). After the entire list is processed, the file is closed and a message is displayed to the user indicating that their data has been saved.

Exiting the Program

The final menu option allows the user to exit the program. If the user exits the program without saving, all the data they've currently added/edited will be lost. This part of the program was simple to execute, the most important component was the break which was provided in the starter code. This was essential because it allows the program to break out of the while loop. The part that was added was a print() statement displaying a message to the user saying they have exited the program.

Executing the Script

Executing in the Command Shell

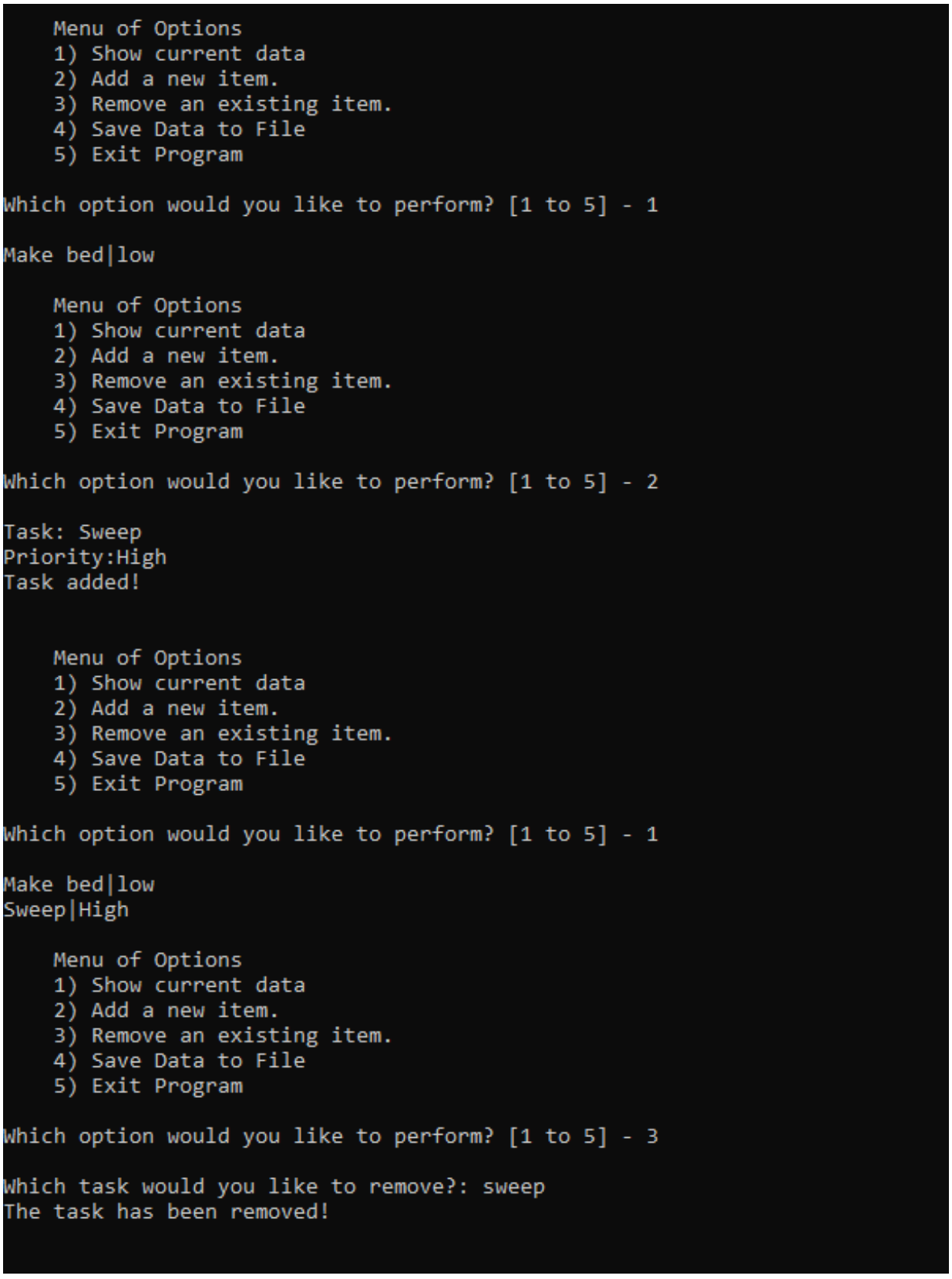

Figure 5.8 - Executing the script in the command shell

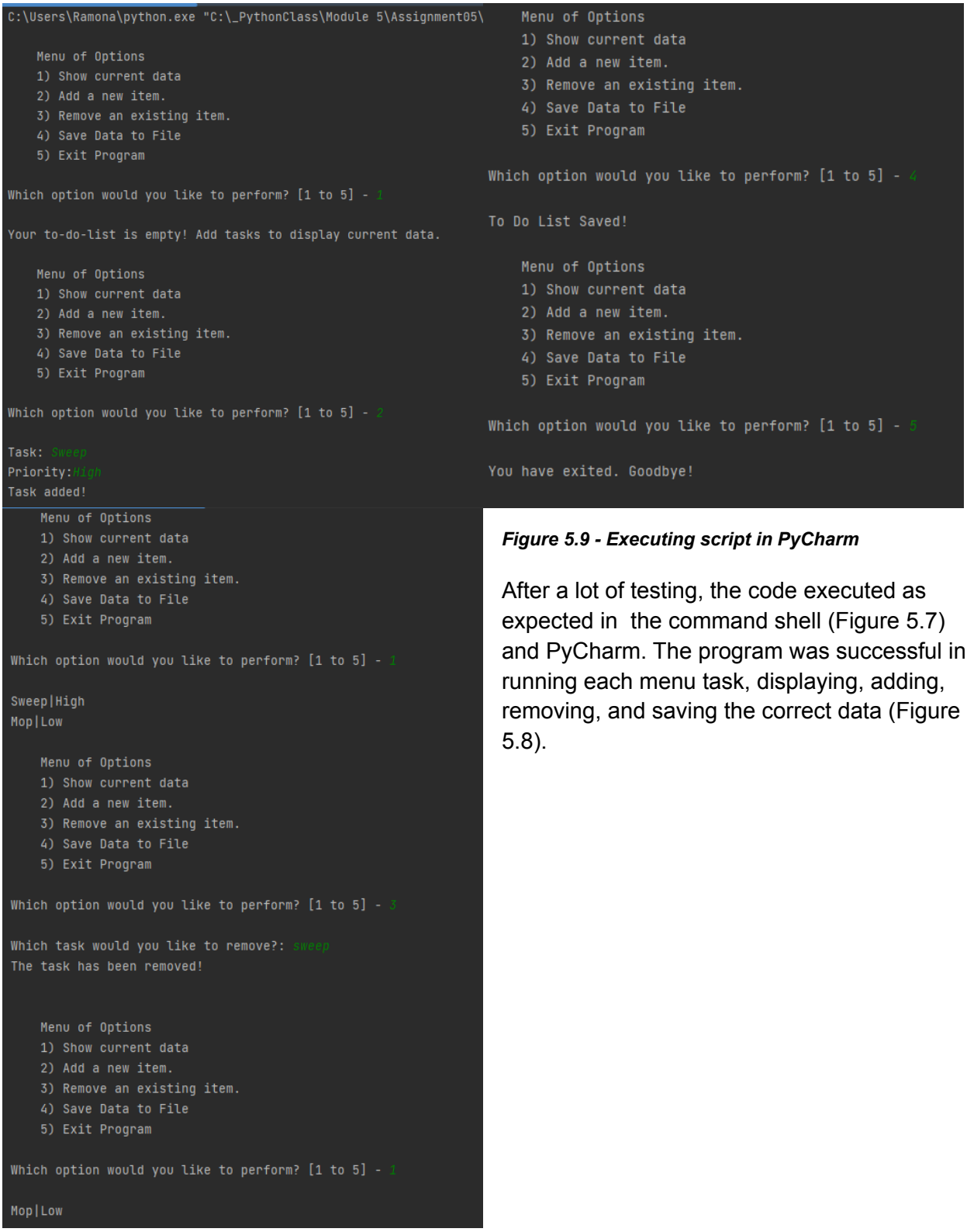

Figure 5.9 - Executing script in PyCharm

After a lot of testing, the code executed as expected in the command shell (Figure 5.7) and PyCharm. The program was successful in running each menu task, displaying, adding, removing, and saving the correct data (Figure 5.8).

Summary

Assignment05 highlighted the use of collections, specifically lists and dictionaries, and how they can be utilized to add, manipulate, and store tables of data. Multiple new concepts were applied in the creation of this program, such as the use of try except blocks when attempting to read existing files, storing lists within dictionaries to create tables of data, and loading data from files into the program to build upon the existing data. In the end, a successful program was created that allows users to view, add, and remove tasks from their to-do-list while also having the ability to build upon them after quitting the program.

Citations

Dawson, M. (2010). Python programming for the absolute beginner (3rd ed.). Course Technology.

Loeber, P. (2021, July 31). How to remove elements in a python list while looping. Python Engineer. Retrieved February 13, 2023, from https://www.python-engineer.com/posts/remove-elements-in-list-while-iterating/