David Petkov

May 29, 2023

IT FDN 100 A

Assignment 6

Link to GitHub

Making a To-Do in Python Using Classes and Functions

Preliminary Comment Before Beginning the Report

Last month I requested a 1-week extension for this assignment and it was granted by the professor. I will do what I can to have Modules 6 and 7 submitted by May 31, 2023.

Introduction

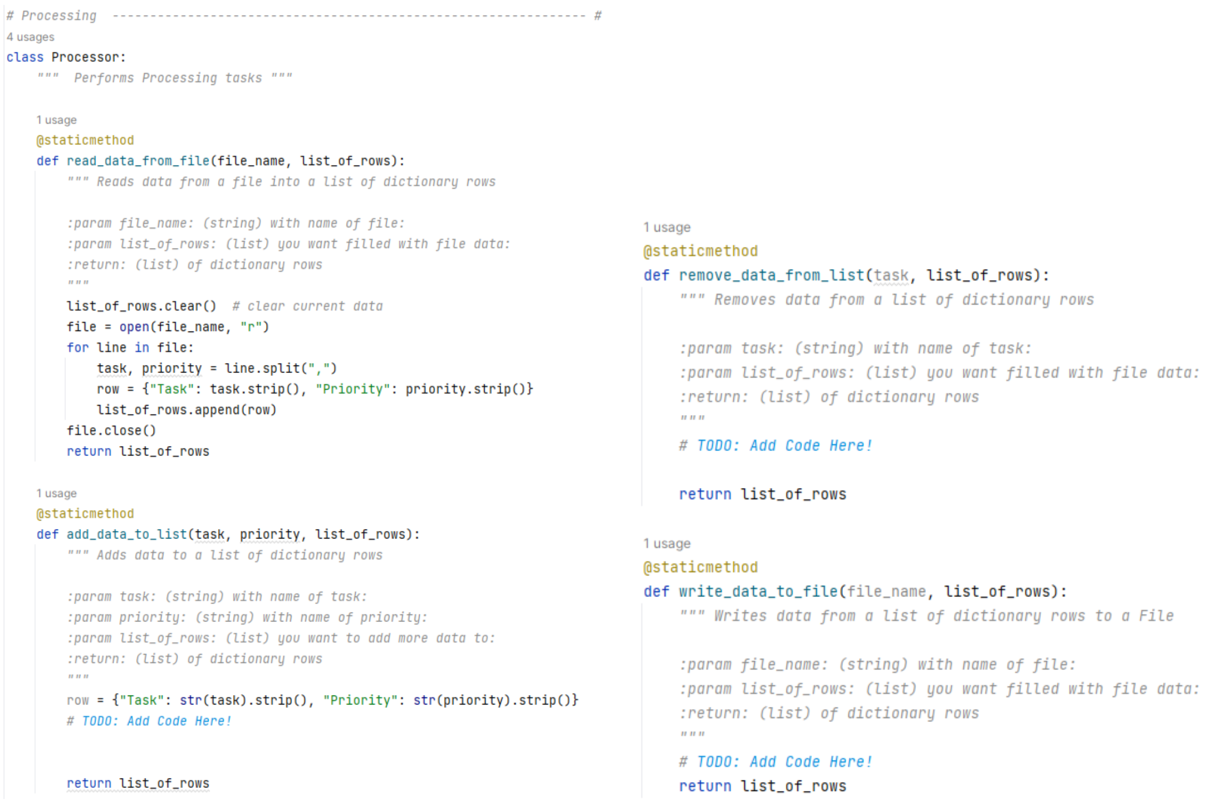
In this assignment, we build upon the scope of Module 5 by implementing the same functionality of the program this time using classes and functions. The reason for this is to learn how to organize our code. This will come in handy in the future as our programs start to become longer, certain aspects of the program need to be separated by what they do.

Completing the Assignment

Our task is to take a starter code provided to us and implement our code wherever instructed to complete the assignment while also adding in another function to better organize the code. The program is split into several pieces:

1. Processing
2. Input/Output (I/O) Presentation
3. Main Body

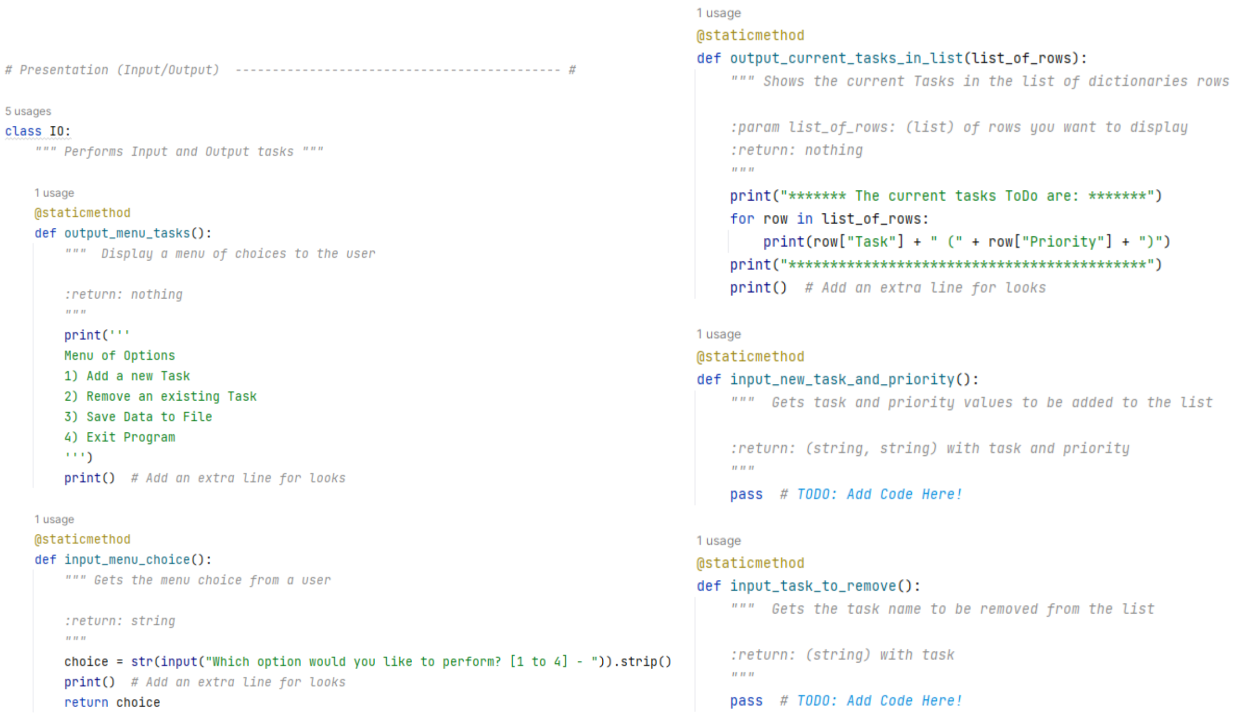
The first two pieces are classes of their own with their own respective functions. The processing section (Figure 1) is responsible for processing the data from the text file and encompasses all of the functions associated with processing the data. The I/O section (Figure 2) does the same thing but for the functions associated with the user interface. The main body of the script (Figure 3) holds the code that will reference the respective classes and functions depending on which option the user selects. As we can see, most of the work is already done for us, we simply need to add the code inside several functions.



```python
# Processing  --------------------------------------------------------------- #
4 usages
class Processor:
    """  Performs Processing tasks """

    1 usage
    @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
        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()
        return list_of_rows

    1 usage
    @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!


        return list_of_rows
```

```python
1 usage
@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!

    return list_of_rows

1 usage
@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!
    return list_of_rows
```

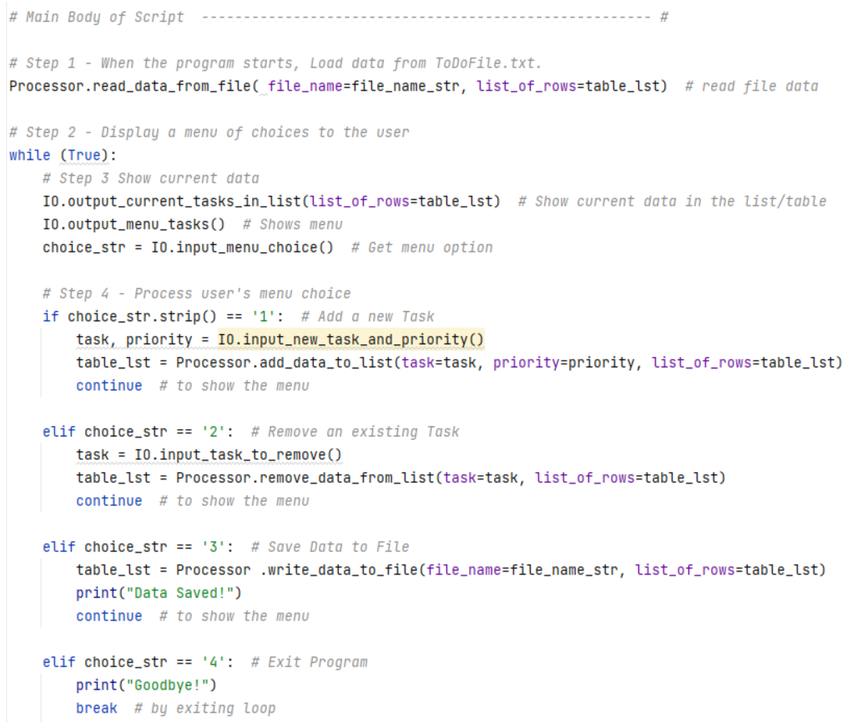
Figure 1: Processing Portion of the Program



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

5 usages
class IO:
    """ Performs Input and Output tasks """

    1 usage
    @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

    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
```

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

1 usage
@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
    """
    pass  # TODO: Add Code Here!

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

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

Figure 2: I/O Portion of the Program



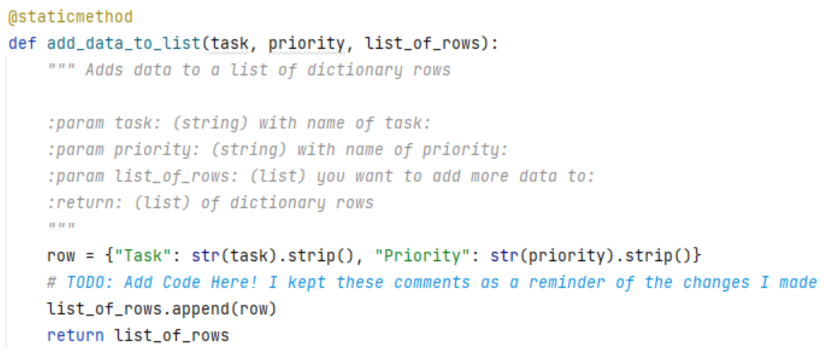
```
# Main Body of Script  ------------------------------------------------------ #

# 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)
        print("Data Saved!")
        continue  # to show the menu

    elif choice_str == '4':  # Exit Program
        print("Goodbye!")
        break  # by exiting loop
```

Figure 3: Main Body of the Program

The first section to modify is the processing section where my code was needed in three functions. The first function was responsible for adding data to a list of dictionary rows. In the code provided, the dictionary row was already defined so to add the data to a list, I would simply have to append a list with the dictionary row. The variables were already defined so all I needed was the append() function and that was it. Then it returns the newly updated list. (Figure 4)

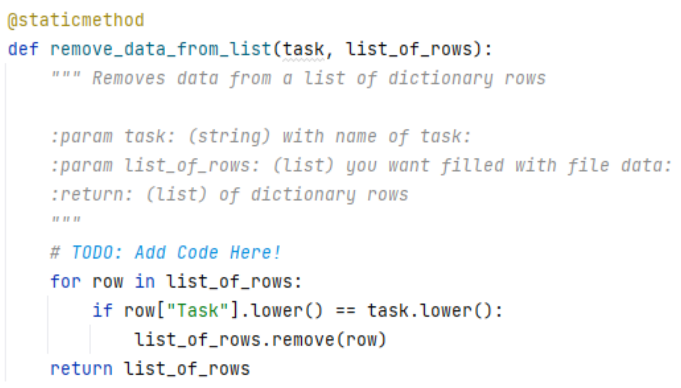
The second function was to remove a dictionary row from the list. A for loop would define every row in the list, and if the user input matched any dictionary row value, it would remove it using the remove() function and return the new list. (Figure 5)

The last function in this section writes data from a list to the file. The file object will be opened in write mode. Then a for loop would read every row in a list and write the rows to the file. (Figure 6)



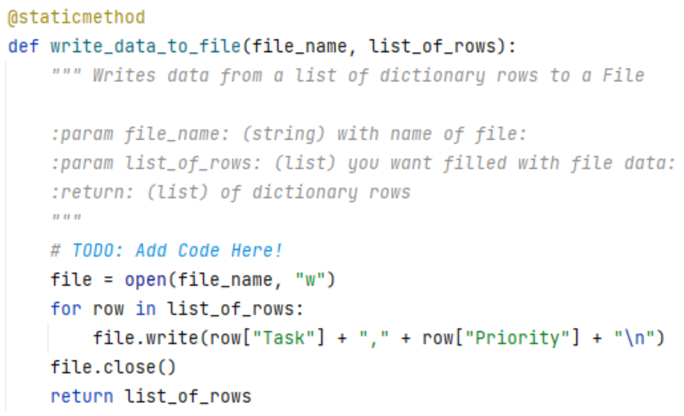
```
@staticmethod
def add_data_to_list(task, priority, list_of_rows):
    """ Adds data to a list of dictionary rows

    :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! I kept these comments as a reminder of the changes I made
    list_of_rows.append(row)
    return list_of_rows
```

Figure 4: Adding data to a list of dictionary rows



```python
@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:
        if row["Task"].lower() == task.lower():
            list_of_rows.remove(row)
    return list_of_rows
```

Figure 5: Removing data from a list

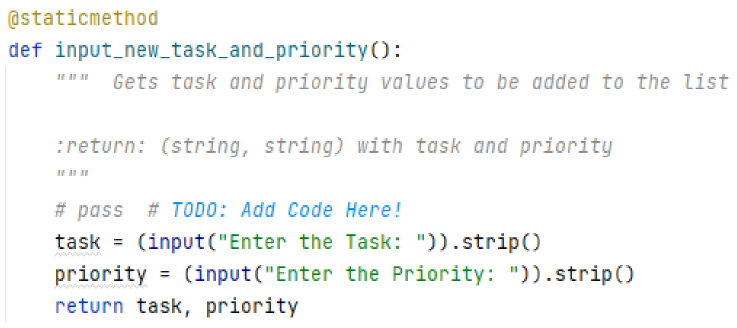


```python
@staticmethod
def write_data_to_file(file_name, list_of_rows):
    """ Writes data from a list of dictionary rows to a File

    :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()
    return list_of_rows
```

Figure 6: Writing data to a file

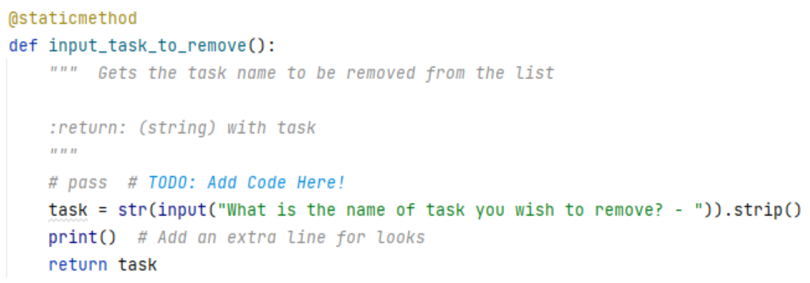
The second section to modify is the I/O portion of the program. The first function is to take user inputs so they can be added to a list. This was simply done with two input() functions. Then, the function returns the two inputs which are then used by another function in the processing class which actually adds the data to the list(this is done in the main body of the program). (Figure 7)

The second function is removing a dictionary row from a list. A simple input() function is needed so the user can type the task they want to be removed and the function returns the variable. Then, another function in the processing class is used to actually remove the row (this is done in the main body of the program). (Figure 8)



```python
@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
    """
    # pass  # TODO: Add Code Here!
    task = (input("Enter the Task: ")).strip()
    priority = (input("Enter the Priority: ")).strip()
    return task, priority
```

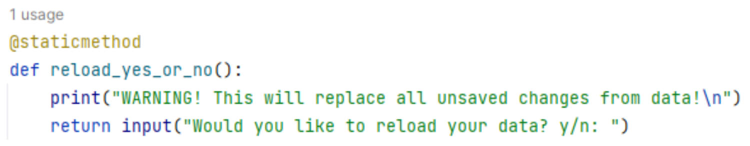
Figure 7: Enabling users to input a task and priority



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

    :return: (string) with task
    """
    # pass  # TODO: Add Code Here!
    task = str(input("What is the name of task you wish to remove? - ")).strip()
    print()  # Add an extra line for looks
    return task
```

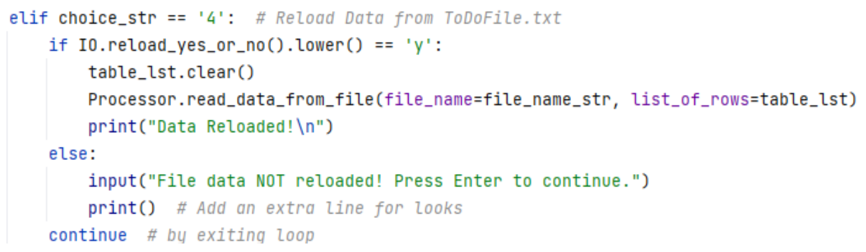
Figure 8: Enabling a user to enter a task they want to be removed

I created a new function in the I/O class which would return a user input whether the user wanted the data reloaded or not (Figure 9). In the main body(Figure 10), I added another elif statement for the option to reload. Inside the elif statement, an if statement would check what the user input was using the new function I created. If the user wants the data reloaded, the data from the list currently in the session would be lost, but the data in the file would remain intact. This is because the clear() function is used only to clear list data, file data is not overwritten. The if statement also implements two other functions that already exist to ensure no further data loss. After the list contents are cleared, data from the file is read to the list and then output to the user.



```
1 usage
@staticmethod
def reload_yes_or_no():
    print("WARNING! This will replace all unsaved changes from data!\n")
    return input("Would you like to reload your data? y/n: ")
```

Figure 9: Function for reloading data



```
elif choice_str == '4':  # Reload Data from ToDoFile.txt
    if IO.reload_yes_or_no().lower() == 'y':
        table_lst.clear()
        Processor.read_data_from_file(file_name=file_name_str, list_of_rows=table_lst)
        print("Data Reloaded!\n")
    else:
        input("File data NOT reloaded! Press Enter to continue.")
        print()  # Add an extra line for looks
    continue  # by exiting loop
```

Figure 10: Code in the main body for either clearing the list or continuing with no changes

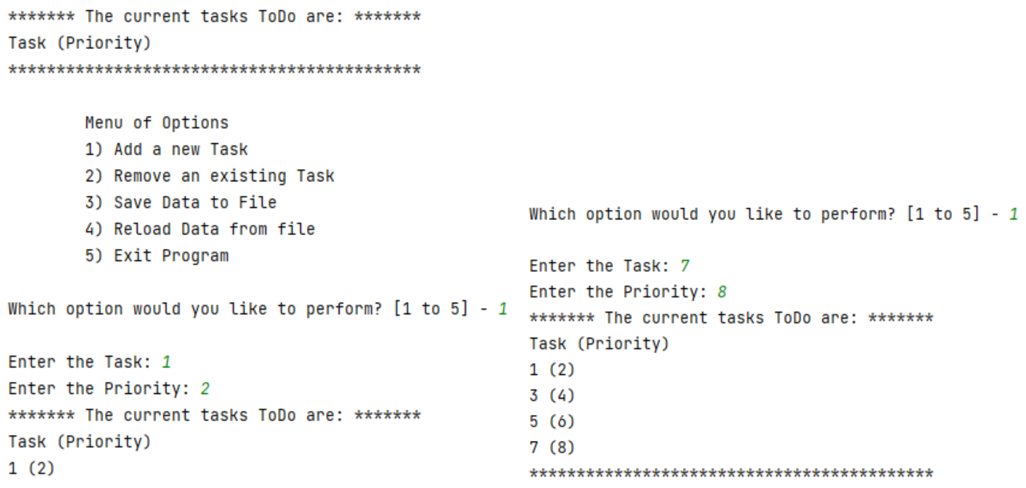
I could've also added another option to allow the user to display file data, but I wouldn’t be creating any new functions since they were already created for us, so I decided not to add a sixth option.

The Program in Action

In this section, I will demonstrate how the program works with each user option:

1. Add data
2. Remove data
3. Save data
4. Reload data
5. Exit

Adding data allows the user to add as many tasks to the list as they would like. Figure 11 demonstrates this.

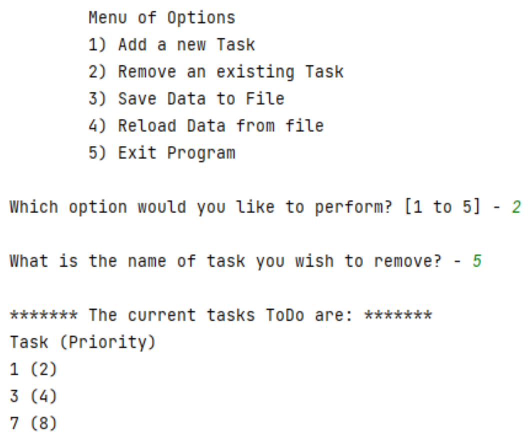


```
******* The current tasks ToDo are: *******
Task (Priority)
*******************************************

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

Which option would you like to perform? [1 to 5] - 1

Enter the Task: 1
Enter the Priority: 2
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
```

```
Which option would you like to perform? [1 to 5] - 1

Enter the Task: 7
Enter the Priority: 8
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
5 (6)
7 (8)
*******************************************
```

Figure 11: What the user interface shows when adding data

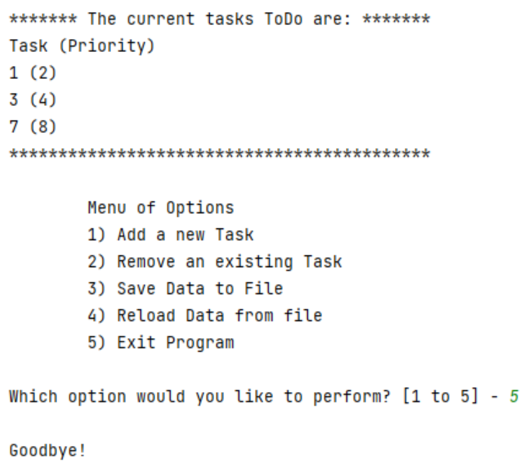
Next, to remove data from the list, we select option 2, then the user types the task they want to be removed. (Figure 12)

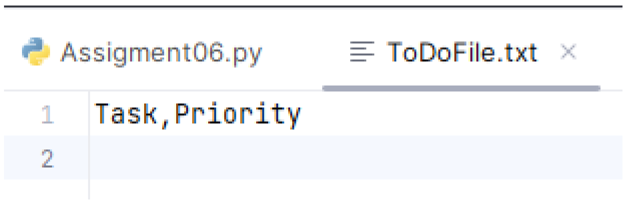


```
        Menu of Options
        1) Add a new Task
        2) Remove an existing Task
        3) Save Data to File
        4) Reload Data from file
        5) Exit Program

Which option would you like to perform? [1 to 5] - 2

What is the name of task you wish to remove? - 5

******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
7 (8)
```

Figure 12: What the user interface shows when removing data

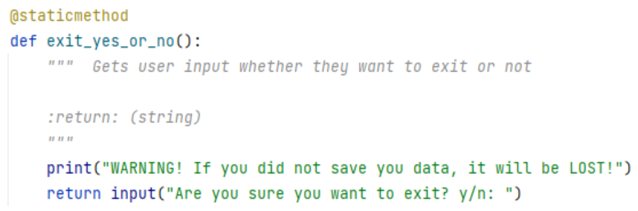
The third option, to save, is crucial. If I were to exit the program after what is shown in Figure 12, none of my progress would be saved (Figure 13). To mitigate this, I decided to create another function in the I/O class for the user to input whether they want to exit or not. Then, in the main body, the program will either exit or stay depending on what option the user selected. (Figure 14)


```
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
7 (8)
*******************************************

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

Which option would you like to perform? [1 to 5] - 5

Goodbye!
```


Assigment06.py   ToDoFile.txt

```
Task,Priority

```

Figure 13: Data from the session being lost when the user doesn't save


```python
@staticmethod
def exit_yes_or_no():
    """  Gets user input whether they want to exit or not

    :return: (string)
    """
    print("WARNING! If you did not save you data, it will be LOST!")
    return input("Are you sure you want to exit? y/n: ")
```

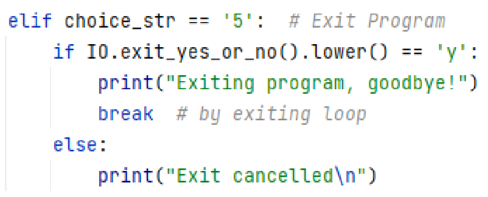
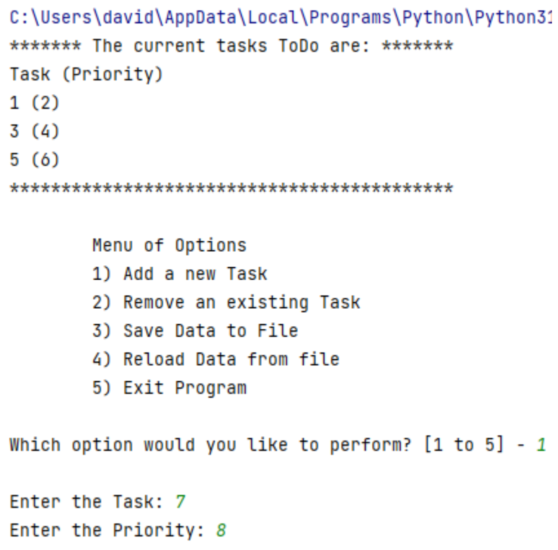

```python
elif choice_str == '5':  # Exit Program
    if IO.exit_yes_or_no().lower() == 'y':
        print("Exiting program, goodbye!")
        break  # by exiting loop
    else:
        print("Exit cancelled\n")
```

Figure 14: Added functionality allowing the user to choose whether they want to exit or not, a safeguard

The fourth option is to reload the data from the current session. Selecting this option will delete any unsaved data from the session, but the file data will remain untouched. Figure 15 is a demonstration of loading a file, adding a few items to the list, then reloading the list and losing the items.


```
C:\Users\david\AppData\Local\Programs\Python\Python3
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
5 (6)
*******************************************

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

Which option would you like to perform? [1 to 5] - 1

Enter the Task: 7
Enter the Priority: 8
```

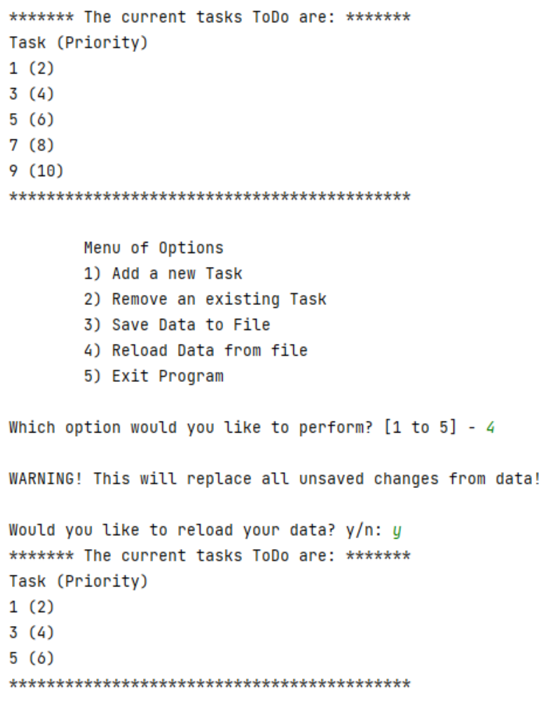

```
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
5 (6)
7 (8)
9 (10)
*******************************************

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

Which option would you like to perform? [1 to 5] - 4

WARNING! This will replace all unsaved changes from data!

Would you like to reload your data? y/n: y
******* The current tasks ToDo are: *******
Task (Priority)
1 (2)
3 (4)
5 (6)
*******************************************
```

Figure 15: A demonstration that reloading data only affects data currently in the session and not the file

Lessons Learned

The most important lesson learned from this module was how to implement classes into code. Functions are familiar to me since I have programming experience and we technically have been using functions this entire class (print(), input(), etc.). Having classes allows me to organize my code depending on the functionality of the script, which is why we have two classes, one for processing data and another for I/O.

The second thing I learned was what @staticmethod means. At first glance, I was very confused since the title is not self-explanatory. By adding a static method, you are bounding a function specific to a class in the script.

The last lesson I learned was how to better work with lists and dictionaries. In modules 4 and especially 5, I had to rush through the program because I was leaving for vacation, so I did not take the time to really learn how to manage data between a file and lists. As a result, my result in module 5 was very wacky and only worked when operated in a very specific order, which means I was the only person who knew how to work the program properly, not a great design if others cant use it. I learned this on my return 3 weeks later when reviewing the answers to 4 and 5, and on this module as well. By reading the data from a file to a list, you don't have to constantly open and close a file object, instead, you only have to work with a list, which is much easier to make changes to.