

**Name:** Megan (Megs) Cambra  
**Date:** May 24th, 2023  
**Class:** IT FDN 110

## Assignment 06

# Unleashing the Magic of Functions: Transforming Code and Conquering Bugs

### Introduction

In our quest to understand the differences between the original "Assignment 5 – To Do List" code and the improved "Assignment 6 – To Do List," let's embark on a thrilling adventure through the world of code organization and bug hunting. Our journey begins by exploring how the code structure evolved from a procedural approach to a modular and organized design.

### Breaking Down the Code!

#### The Code Transformation: From Procedural to Modular

In "Assignment 5," code execution lacked organization and modularity, making it difficult to maintain. However, in the enchanted realm of "Assignment 6," *functions* emerged as powerful tools. By introducing *functions* in "Assignment 6," the code became more organized, with each *function* dedicated to a specific task. This modular design improved code comprehension and maintenance. *Functions* provided meaningful names to groups of instructions, promoting code readability. They also enabled code reusability, eliminating repetition, and allowing *functions* to be called from different program sections.

#### The Function-fortress: Unleashing the Power of Functions

In the quest to restore order to the chaos of the code, the revised "Assignment 6" code introduced a collection of essential *functions*, each assigned a specific purpose. One mighty *function*, known as 'read\_data\_from\_file,' rose to the occasion. The 'read\_data\_from\_file' *function* breathed new life into the code, by enabling the retrieval of the precious data from a file and instilling it within the list of *dictionary* rows.

```
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
```

**Figure 2.** The `read_data` function presenting the current items in the table.

### Forging New Artifacts: The Enchanting 'add\_data' Function

In the quest to empower adventurers with the ability to add new items to the table, a remarkably emerged—the 'add\_data\_to\_list' *function*. This enchanting *function* beckoned users to input the task name and priority, creating a *dictionary* to represent the new item, and gracefully appending it to the list. With the inclusion of this magical *function*, the code achieved newfound flexibility, enabling the addition of new items from various parts of the program. With the 'add\_data\_to\_list' *function* at their disposal, adventurers could easily contribute to the table's ever-growing collection. By providing the task name and priority as inputs, a new *dictionary* was conjured, representing the fresh addition. This mystical item was then seamlessly appended to the list, seamlessly expanding the table's realm of possibilities. The versatility of this *function* allowed the code to embrace a more dynamic nature, enabling the seamless integration of new items from any corner of the program.

```
@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)

    return list_of_rows
```

**Figure 3.** The 'add\_data\_to\_list' *function* allowing users to add new tasks to the table.

### Banishing the Shadows: The Formidable 'remove\_data' Function

In the epic quest to eliminate undesirable tasks from the table, a brave warrior emerged—the 'remove\_data\_from\_list' *function*. Armed with an *argument* known as the task name, this valiant *function* fearlessly embarked on a mission to search and destroy. An *argument* in programming is a *value* or *variable* passed to a *function* when called, providing specific information or data required to accomplish its task. In this case, the task name *argument* equips the 'remove\_data\_from\_list' *function* to target and banish a specific task.

With the task name *argument* in hand, the *function* ventured into the realm of the task list, tirelessly searching for a match. Upon discovering a task with a matching name, it promptly vanquished it, banishing it from the table forever. By encapsulating the logic for task removal within this *function*, the code achieved greater clarity and maintainability. The purpose of the *function* became well-defined and isolated, enabling easier comprehension, modification, and reuse.

```

1 usage
2 @staticmethod
3 def remove_data_from_list(task, list_of_rows):
4     """ Removes data from a list of dictionary rows
5
6     :param task: (string) with name of task:
7     :param list_of_rows: (list) you want filled with file data:
8     :return: (list) of dictionary rows
9     """
10
11     # TODO: Add Code Here!
12     # intRowNumber = 0
13     for row in list_of_rows:
14         t, p = dict(row).values()
15         if t == task:
16             # del list_of_rows[intRowNumber] # Deletes the matching item
17             # intRowNumber += 1
18             list_of_rows.remove(row)
19
20     return list_of_rows

```

**Figure 4.** The 'remove\_data\_from\_list' function facilitating the removal of tasks from the table.

### Saving the Quest Progress: The Heroic 'write\_data' Function

In our daring escapades, where the fate of our adventures' progress hung in the balance, emerged the gallant 'write\_data\_to\_file' function. With unwavering determination, this noble function shouldered the responsibility of preserving our heroic deeds by persisting the updated data back to the file. By accepting the file name and list of rows as *parameters*, it harnessed its power to write the data from the list of *dictionary* rows to the specified file. This act of separating this crucial task from the other *functions* not only enhanced code organization but also bestowed upon us the gift of easier maintenance and expansion of our quests.

```

1 usage
2 @staticmethod
3 def write_data_to_file(file_name, list_of_rows):
4     """ Writes data from a list of dictionary rows to a File
5
6     :param file_name: (string) with name of file:
7     :param list_of_rows: (list) you want filled with file data:
8     :return: (list) of dictionary rows
9     """
10
11     # TODO: Add Code Here!
12     file = open(file_name, "w")
13     for line in list_of_rows:
14         file.write(line["Task"] + ", " + line["Priority"] + "\n") # Writes
15     file.close()
16
17     return list_of_rows

```

**Figure 5.** The 'write\_data\_to\_file' function preserving the updated data by writing it to a file.

### The Grand Unifier: The Magnificent 'main\_menu' Function/Class

The combination of the IO class and the commands within it serves as a **main menu** for the program, providing options and facilitating user interaction, handling input and output tasks. The main menu functionality consists of several methods.

The `output_menu_tasks()` method displays a menu of options to the user. It presents a clear and organized list of choices, such as adding a new task, removing an existing task, saving data to a file, or exiting the program. The `input_menu_choice()` method prompts the user to input their choice from the menu. It retrieves and returns the selected option as a string, allowing the program to determine the subsequent actions based on the user's decision. The `output_current_tasks_in_list(list_of_rows)` method is responsible for displaying the current tasks stored in the list of dictionaries. It iterates over the list and prints each task along with its corresponding priority. This provides the user with a visual representation of the existing tasks, enhancing clarity and comprehension.

Overall, this main menu functionality of the `IO` class creates an interactive experience for the user. It presents options, captures user input, and provides visual feedback on the current state of the tasks. By encapsulating these menu-related tasks within the `IO class`, the code achieves modularity, readability, and improved user interaction.

```
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] - ")).s
        print() # Add an extra line for looks
        return choice
```

*Figure 6. Showing part of the `IO` class, serving as the program's entry point, and directing the flow of execution.*

## Unlocking the Code!

In our exploration, we have uncovered answers to fundamental questions that unlock the mystical world of coding. Let us shed light on these secrets and unravel their significance in more detail.

### Functions: The Versatile Tools

*Functions* serve as versatile tools that allow us to encapsulate sets of instructions and reuse them throughout our code. They enhance modularity, readability, and reusability. *Parameters* act as

placeholders in a *function's* definition, while *arguments* are the actual values passed to the *function*. By utilizing *parameters* and *arguments*, *functions* can handle different inputs and execute tasks accordingly. Return values enable *functions* to provide results or information to the caller, fostering communication and leveraging *function* outputs in other parts of the program.

### Global vs. Local: The Battle of Variables

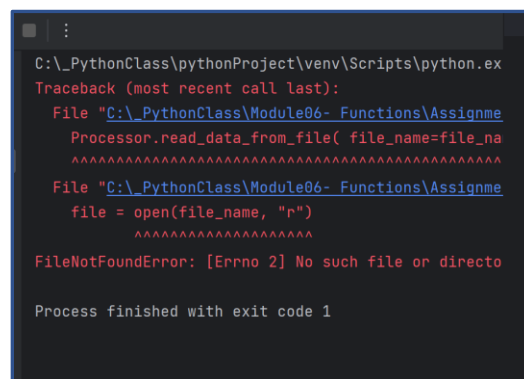
Within the realm of *functions*, we encounter two types of variables: *global* and *local*. *Global variables*, defined outside any *function*, are accessible from anywhere in the program. In contrast, *local variables* are declared within specific *functions* and have limited visibility. *Functions* play a vital role in organizing code by breaking it down into modular units. This improves code readability and maintainability by isolating specific tasks within functions, making them easier to understand and modify. Furthermore, *functions* promote code reusability by encapsulating commonly used operations, enabling us to summon their power whenever needed.

### Functions and Classes: Different Dimensions

While *functions* offer modularity and reusability, *classes* introduce a higher level of abstraction. *Classes* define blueprints for creating objects with shared properties and behaviors. They enable the creation of multiple object instances based on a single blueprint, facilitating the management of complex systems and interactions.

### Illuminating the Path: Debugging Tools

In our pursuit of flawless code, we must navigate the challenging path of debugging. PyCharm, our trusted ally, equips us with powerful debugging tools. By setting breakpoints, analyzing variables, and tracing code execution, we can swiftly identify and resolve bugs. Continuous code reruns and focused error localization are effective strategies for pinpointing the root causes of issues. Through these debugging practices, we ensure our code shines brightly, free from doubts and shadows.

A screenshot of a PyCharm console window with a dark background. It displays a Python traceback error. The text is as follows:

```
C:\_PythonClass\pythonProject\venv\Scripts\python.exe
Traceback (most recent call last):
  File "C:\_PythonClass\Module06- Functions\Assignme
    Processor.read_data_from_file( file_name=file_na
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\_PythonClass\Module06- Functions\Assignme
    file = open(file_name, "r")
    ^^^^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directo

Process finished with exit code 1
```

**Figure 7.** Example of debugging using PyCharm.

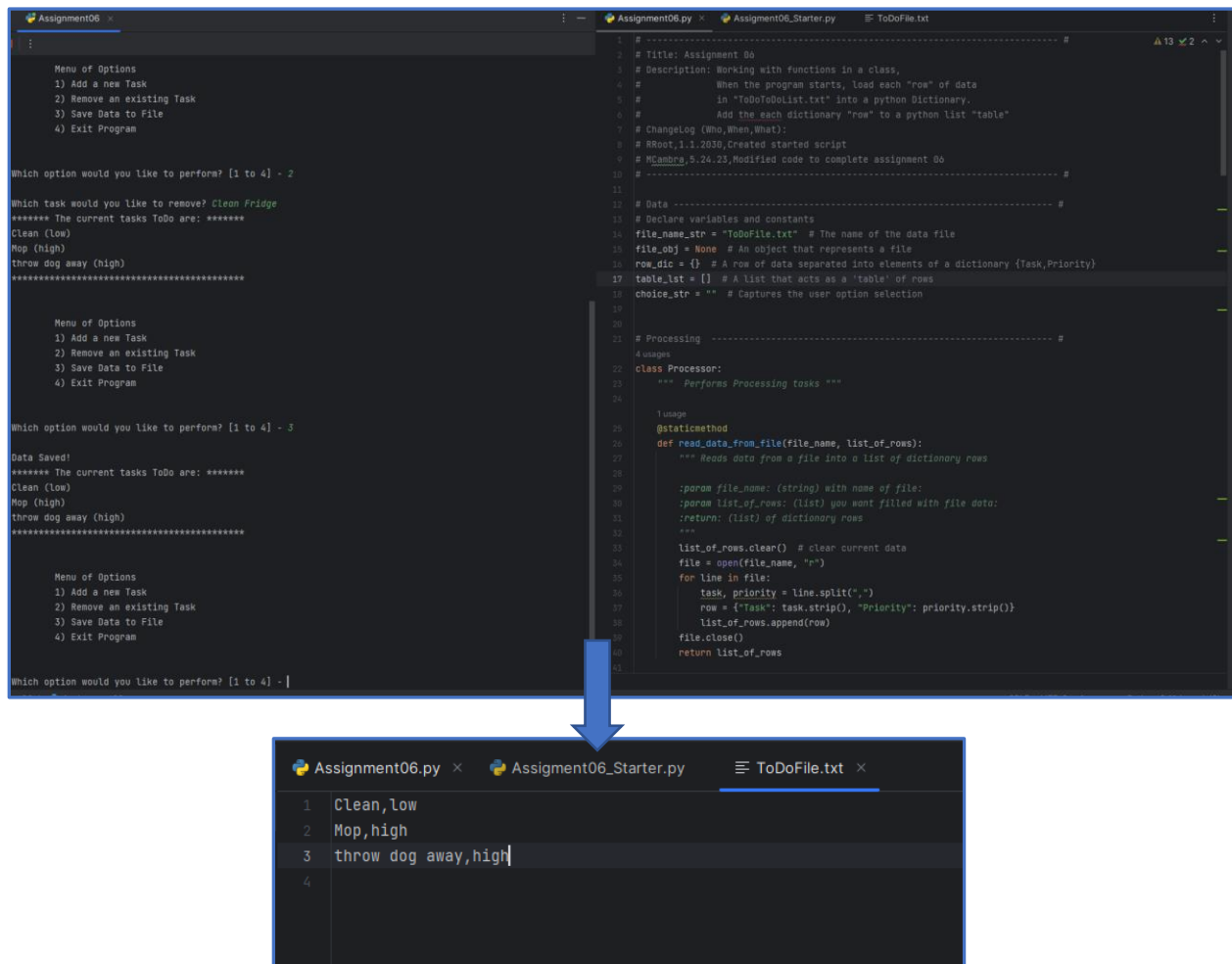
As we unveil these coding insights, we gain a deeper understanding of the intricacies within. *Functions* empower us to harness the full potential of code, while debugging tools empower us to refine and optimize our creations.

## Embarking on the Code Quest

### Unleashing the Magic in PyCharm and the Command Window

To run the "Assignment06.py" code, you have two options: PyCharm or the Command Window. If you choose PyCharm, open the project containing the code in the PyCharm IDE. Then, locate the main file or entry point of the program and click on the run button. PyCharm will execute the code, and you will see the program's output in the console.

Alternatively, if you prefer the Command Window, navigate to the directory where the code files are located using the command line interface. Once in the appropriate directory, type the command to run the code, which may vary depending on the programming language. For example, if it's a Python script, you can use the command "Assignment06.py" to execute the program. The Command Window will display the output of the program, allowing you to interact with it as needed.



on the user's choice, the script performs the corresponding action by calling different functions from the `Processor` and `IO` classes. The script ensures smooth navigation through the menu, allowing users to interactively manage their tasks until they choose to exit 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 9.** Main body of the script showing how condensed a script can get when incorporating functions and classes into the code.

## Summary

In conclusion, the transformation from "Assignment 5 – To Do List" to the improved "Assignment 6 – To Do List" code was a journey of code organization and bug hunting. Through the power of *functions*, the code gained modularity, readability, and reusability. Each *function* had a specific purpose, from loading data to displaying, adding, and removing items, to saving progress. The *main\_menu* function acted as the central orchestrator of the adventure, providing a clear structure. This modular approach laid a solid foundation for scalability and adaptability. With the introduction of *functions*, the enchanted realm of "Assignment 6" transformed the chaotic landscape of "Assignment 5" into an organized and maintainable codebase.

With all this newfound knowledge, we can embark on future coding quests, armed with *functions*, *classes*, and debugging techniques, confident in our ability to conquer any challenge that comes our way. May your code always be organized, reusable, and bug-free as you continue your coding adventures.

Check out the above code by clicking this link:  
<https://github.com/mcambra56/ITFnd100-Mod06>



## APPENDIX I. Command Prompt

```
Command Prompt - python Assignment06.py
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\Users\megan>cd C:\_PythonClass\Assignment06

C:\_PythonClass\Assignment06>python Assignment06.py
***** The current tasks ToDo are: *****
Clean (low)
Mop (high)
throw dog away (high)
*****

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] -
```

*Figure 10a. Changing directory and running Assignment06.py program using the Command Prompt window instead of PyCharm.*

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

Please name the task: Wash Car
What is the priority? [high|low]: high
***** The current tasks ToDo are: *****
Clean (low)
Mop (high)
throw dog away (high)
Wash Car (high)
*****
```

*Figure 10b. Testing out option 1 using the Command Prompt.*

```
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] - 3

Data Saved!
```

*Figure 10c. Testing out option 3 using Command Prompt. (Save Data)*



## APPENDIX II. Python Script of “Assignment6.py”

```
# -----
- #
# 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
# MCambra,5.24.23,Modified code to complete assignment 06
# -----
- #

# 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

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

    @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

    @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)

    return 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!
        # intRowNumber = 0
        for row in list_of_rows:
            t, p = dict(row).values()
            if t == task:
                # del list_of_rows[intRowNumber]          # Deletes the matching
item
                # intRowNumber += 1
                list_of_rows.remove(row)

        return list_of_rows

    @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 line in list_of_rows:
            file.write(line["Task"] + "," + line["Priority"] + "\n")      #
Writes only the items
        file.close()

        return list_of_rows

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

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

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

    @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!

        strTask = str(input("Please name the task: ").strip())
        strPriority = str(input("What is the priority? [high|low]:
").strip())

        return strTask, strPriority

    @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!

        rmvTask = str(input("Which task would you like to remove? ").strip())

        return rmvTask

```

```

# 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

```