Jordan Keating
May 18, 2023
Foundations of Programming: Python
Assignment 06
https://github.com/JSKeating/IntroToProg-Python

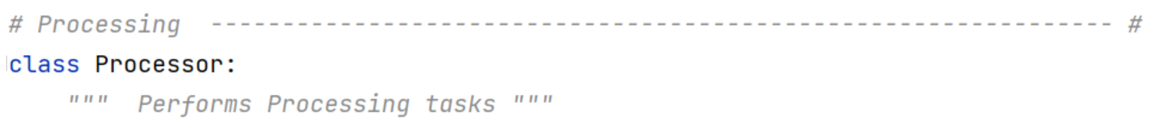
The To-Do List Program V2

Introduction

In this Assignment, we're going to be modifying our script from the To-Do List program in Assignment 5. We'll be making it easier using classes and functions to split up the code into several chunks based upon what the section of code is used for. This can be a really handy tool once scripts become longer and longer - you want your co-workers, classmates, and yourself (in some future date when you aren't wrapped up in the specific program!) to be able to easily read and follow along with how your code operates. We'll also be getting more practice working with starter code and the challenges that arise with reading and interpreting another coder's work.

Separation of concerns

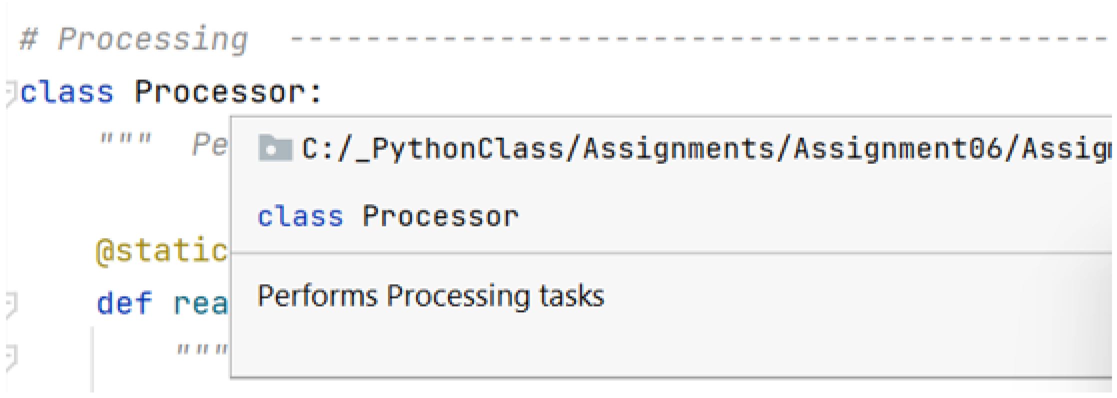
Our script is going to be separated based on a couple different classes to start us out based on what the scripts within that class are doing for us. **IO (for input/output)** is going to be for anything that manages inputs from the user or outputs that are displayed to the user. **Processing** is going to manage all of the behind the scenes processing of data to ensure our program runs smoothly. Since we're operating off of the Starter code, these classes have already been created for us, but for the future, we'll note that creation of a new class is as easy as the line below.



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

Figure 1.1 - Defining a class and adding docstring info

Class is used to define a new class - all you need to do is choose a name that makes sense for what you intend the code within this class to do. Also note that there is a triple-quoted description following the class-defining line - this is called a docstring. These docstrings are not displayed to the end user or used in the processing of the script itself, but are used as a reference item within Python - this is extremely useful when working in a group to ensure your classes (and functions) are used as intended. To view it, simply hover over the name of your class or function and you'll see the pop-up with your docstring as in Figure 1.2.



```
# Processing ----------------------------------------
class Processor:
    """  Pe
    @static
    def rea
        """
```

```
C:/_PythonClass/Assignments/Assignment06/Assig
class Processor
Performs Processing tasks
```

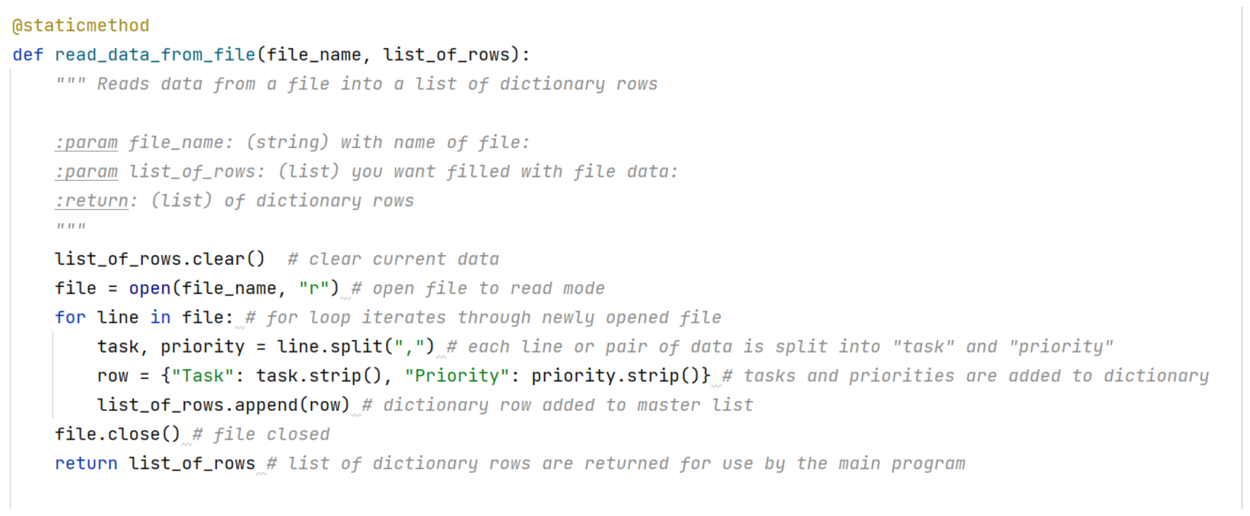
Figure 1.2 - Displaying docstring data

Processing

Let's get started with our first section. We'll start just like we did with the previous To-Do List, by calling in the existing data from ToDoList.txt (in this case, I created a new ToDoListV2.txt to make it more clear which version we're using). I created the text file in my Assignment 6 folder and added some tasks/priorities ('Complete Assignment 06, High' for example) to get us started.

We're going to create functions for each sub-section of code within our processing class. A function is created in a very similar fashion to the classes. Simply use *def* followed by the name of your new function. You can also add parameters that are required when using the new function later. I think of the *open* function - you need to pass the name of your file and what you intend to do with it ('r', 'w', or 'a') - the filename and action are parameters of the function *open*.

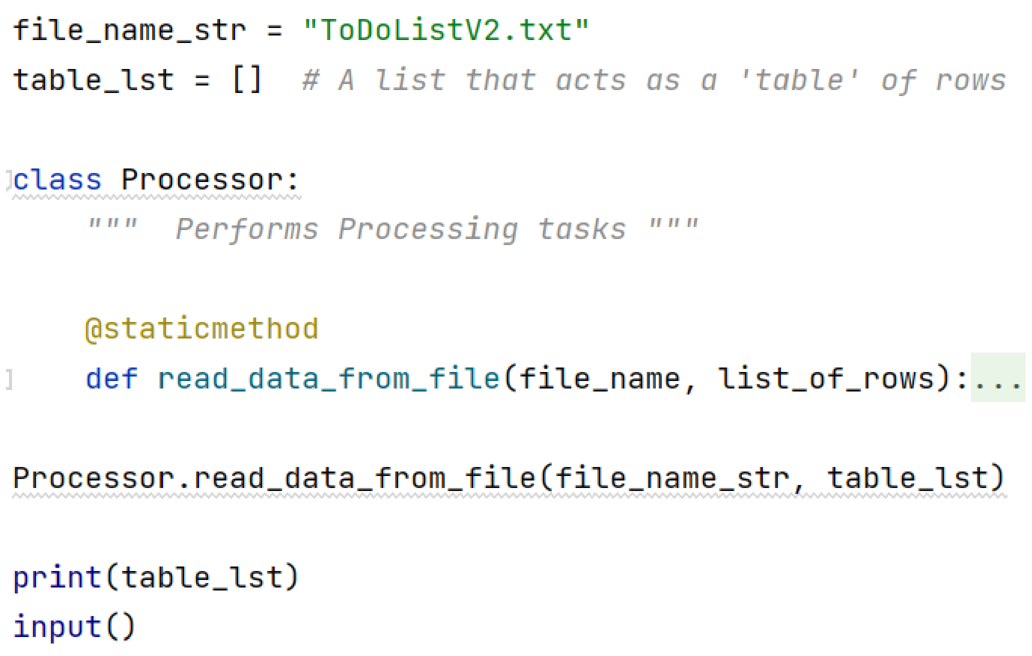
Comparing the existing code from our starter file and the code from our previous version, everything we need to pull in the existing data seems to already be here. The function *read_data_from_file* is defined and will has two parameters - file_name and list_of_rows.



```python
@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") # open file to read mode
    for line in file: # for loop iterates through newly opened file
        task, priority = line.split(",") # each line or pair of data is split into "task" and "priority"
        row = {"Task": task.strip(), "Priority": priority.strip()} # tasks and priorities are added to dictionary
        list_of_rows.append(row) # dictionary row added to master list
    file.close() # file closed
    return list_of_rows # list of dictionary rows are returned for use by the main program
```

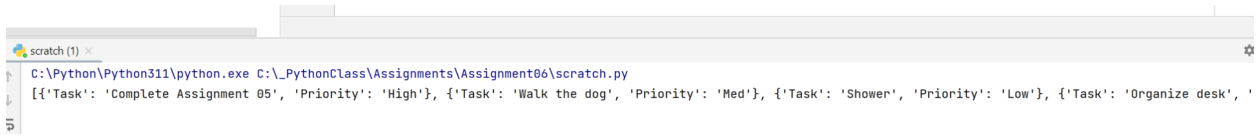
Figure 2.1 - Code from starter file w/ additional notes

I'm going to copy this code over to a scratch file and give it a try to make sure it's working. To run the function, we'll call it using processor.read_data_from_file and passing it our two arguments. For the arguments, we'll also want to initialize a couple of variables in our scratch file (file_name_str & table_lst) to make sure we have our text file ready to open and our table ready to copy over the data returned from our function. To make sure it worked as intended, I called the function, then used a print statement to display what table_lst looks like - it should have gathered all of the dictionary rows from our text file!



```
file_name_str = "ToDoListV2.txt"
table_lst = []  # A list that acts as a 'table' of rows

class Processor:
    """  Performs Processing tasks """

    @staticmethod
    def read_data_from_file(file_name, list_of_rows):...

Processor.read_data_from_file(file_name_str, table_lst)

print(table_lst)
input()
```

Figure 2.2 - Calling our function and printing the resulting dictionary rows (table_lst)



```
C:\Python\Python311\python.exe C:\_PythonClass\Assignments\Assignment06\scratch.py
[{'Task': 'Complete Assignment 05', 'Priority': 'High'}, {'Task': 'Walk the dog', 'Priority': 'Med'}, {'Task': 'Shower', 'Priority': 'Low'}, {'Task': 'Organize desk', '
```

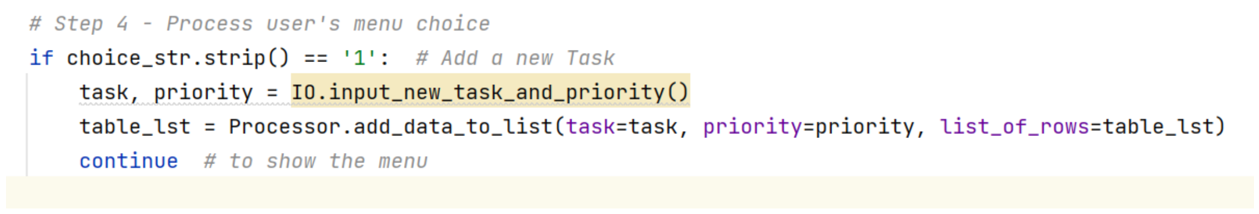
Figure 2.3 - We have a dictionary list! Woohoo!

In reviewing the rest of the supplied starter code, the opening & reading of our text file is completed now (in Processing class) while the initial display of our user menu and collection of the first user input to select a menu option are also completed for us (as part of the I/O Class).

Adding functions for each menu option

Here's the real meat of the assignment - updating our code to move our menu options into their own function. The real key here is to understand the parameters set by our starter code and initial variables and ensure that our created function is using those parameters and return values to match. Let's start with the add task function.

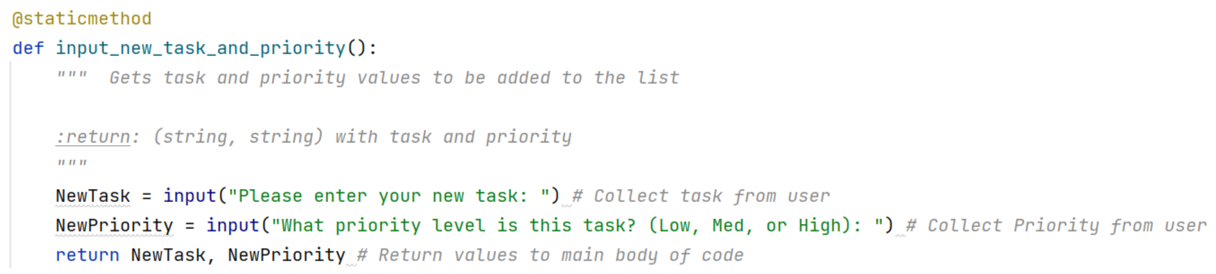
From the starter code provided, when a user chooses that option, we'll want to ask them to input two things - the new task for the list and it's priority level. As we're asking for inputs, this will fall under the I/O class (which has already been called for us as IO.input_new_task_and_priority()). You'll see that the task and priority values are expected to be returned from that function, so we'll have to make sure we do that!



```python
# 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
```

Figure 3.1 - Choice 1 calls for IO.input_new_task_and_priority AND Processor.add_data_to_list

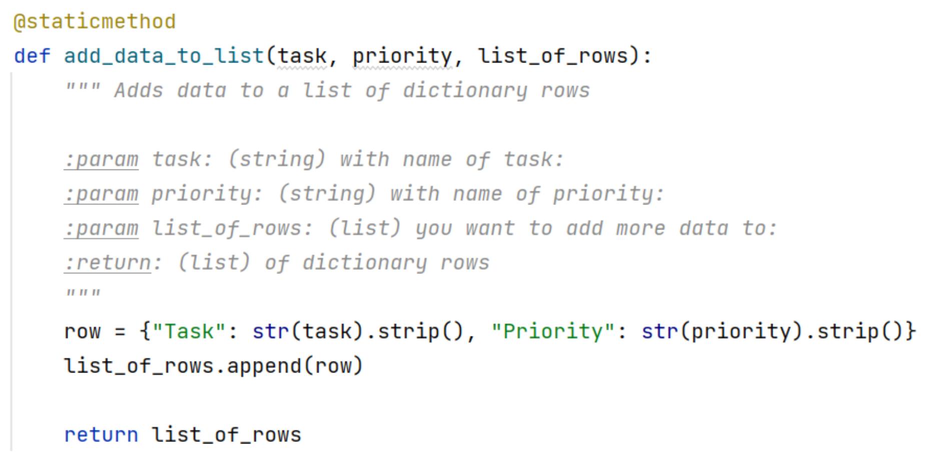
To finish input_new_task_and_priority from what's already created for us, we simply want to capture the user inputs into two variables, then use *return* to send them back to the main body of the script. This will pass those NewTask & NewPriority inputs from the user into the task & priority variables prepared for us in Figure 3.1.



```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
    """
    NewTask = input("Please enter your new task: ") # Collect task from user
    NewPriority = input("What priority level is this task? (Low, Med, or High): ") # Collect Priority from user
    return NewTask, NewPriority # Return values to main body of code
```

Figure 3.2 - Creating NewTask and NewPriority to collect inputs from user and using *return* to send those inputs to Main

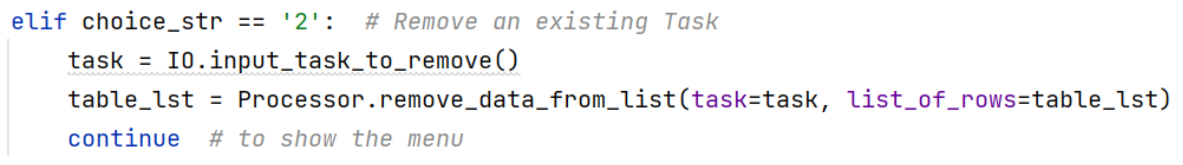
Similarly, for the Processor.add_data_to_list function, a majority of the work has been done for us - we already have docstring notes, and we now have arguments to fill out the required parameters. Task will end up with NewTask the user input, Priority will end up with the NewPriority user input, and list_of_rows already refers to our existing table_lst where we read our existing data into. With the first piece sorting our task and priority arguments into a dictionary row, we really only need to append the existing list_of_rows with the new row and we're set to return that to the main script!



```python
@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()}
    list_of_rows.append(row)

    return list_of_rows
```

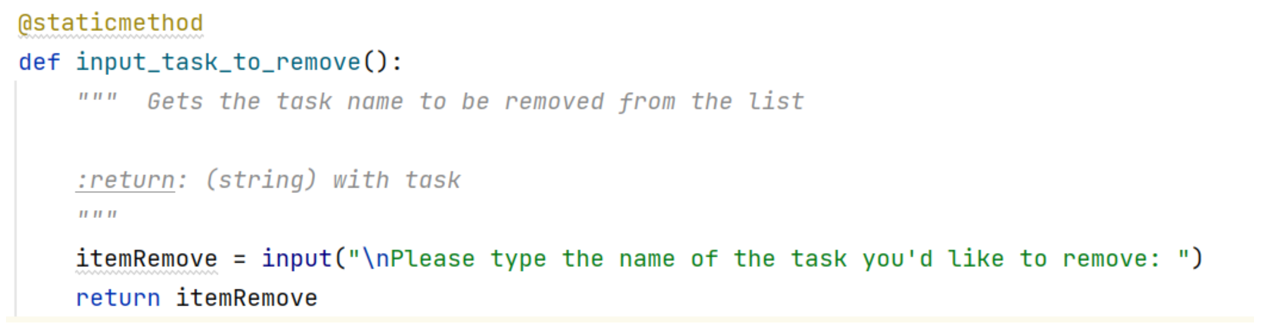
Figure 3.3 - Adding list_of_rows.append to add new task/priority to existing list

Moving on to option 2, we'll see a similar set-up as option 1 in our starter code. We have an I/O function prepared where we'll need to ask the user which task will be removed, then a processor function to do the actual removal.



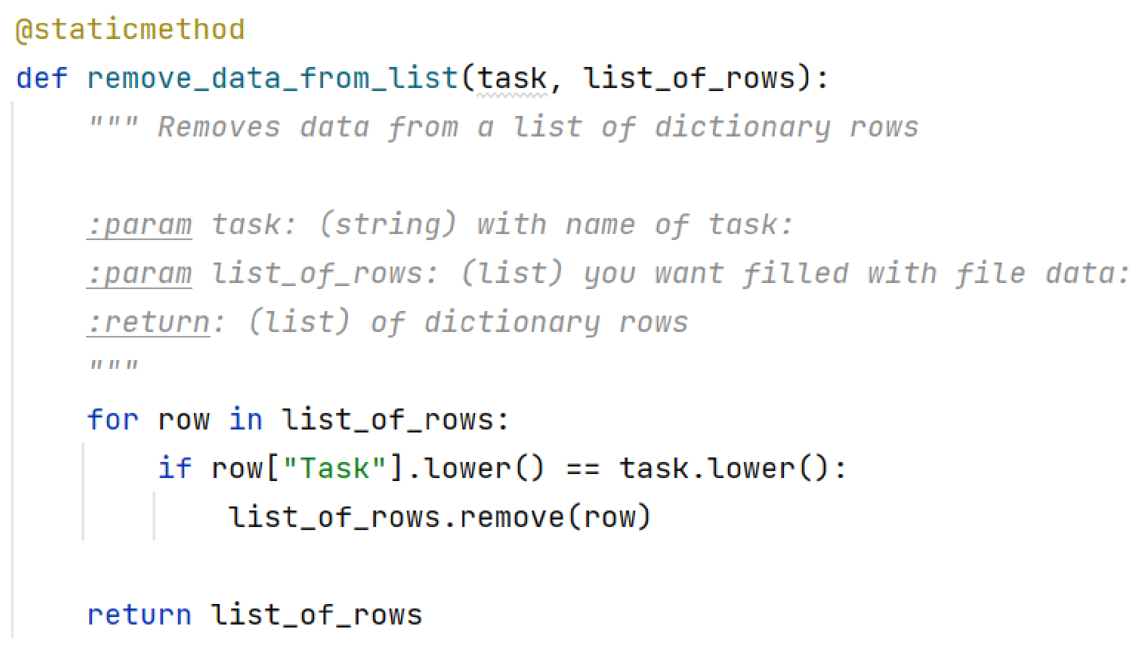
```python
elif choice_str == '2':  # Remove an existing Task
    task = IO.input_task_to_remove()
    table_lst = Processor.remove_data_from_list(task=task, list_of_rows=table_lst)
    continue  # to show the menu
```

Figure 3.4 - Main body option 2 script



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

    :return: (string) with task
    """
    itemRemove = input("\nPlease type the name of the task you'd like to remove: ")
    return itemRemove
```

Figure 3.5 - Adding code for I/O function

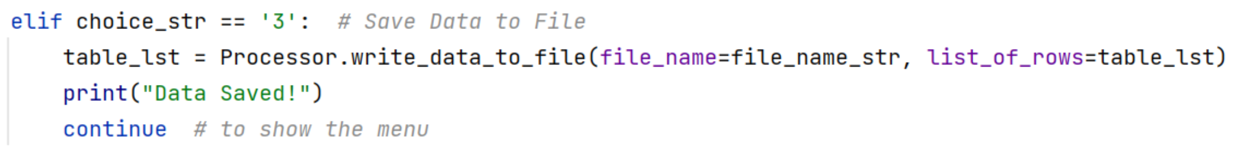


```python
@staticmethod
def remove_data_from_list(task, list_of_rows):
    """ Removes data from a list of dictionary rows

    :param task: (string) with name of task:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    for row in list_of_rows:
        if row["Task"].lower() == task.lower():
            list_of_rows.remove(row)

    return list_of_rows
```

Figure 3.6 - Updating Processor.remove_data_from_list

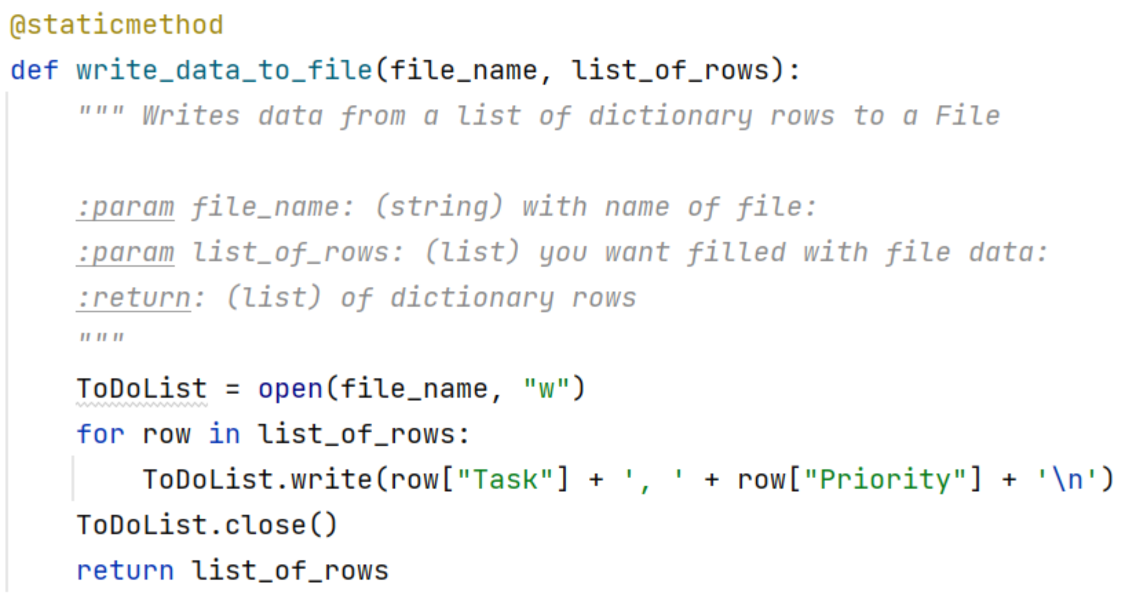
Our I/O function is simple again - we just ask the user for the task they'd like removed and send that string input back to the main body and passed into the awaiting task variable. The Processor function also utilized the same scripting as the previous version of our ToDoList - we iterate through each row in list_of_rows (again this directs toward the global table_lst variable) looking for a match for the user input. If a match is found, we use *remove* to delete the completed task. The updated menu is then automatically displayed for the user to confirm the removal.

Finally, we'll take care of option 3. In this option, we only have a processing function to write our data to our ToDoListV2 text file. We can again just copy and paste our existing code from the previous version of this program - really the only thing to do is make sure our variable names are all matched up and we're good to go! Nothing to return on this one!



```python
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
```

Figure 3.7 - Option 3 has one Processor function to update

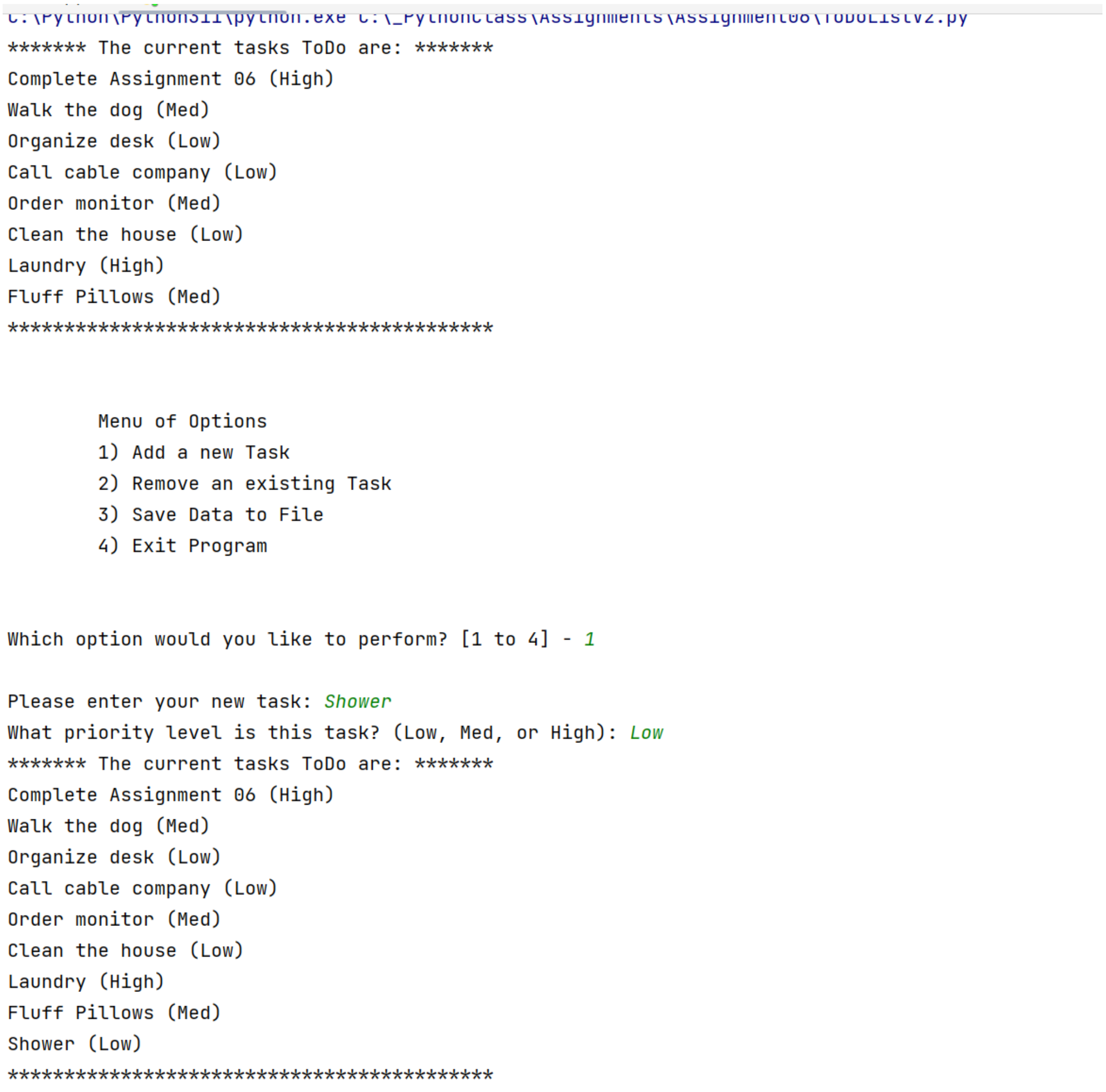


```
@staticmethod
def write_data_to_file(file_name, list_of_rows):
    """ Writes data from a list of dictionary rows to a File

    :param file_name: (string) with name of file:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    ToDoList = open(file_name, "w")
    for row in list_of_rows:
        ToDoList.write(row["Task"] + ', ' + row["Priority"] + '\n')
    ToDoList.close()
    return list_of_rows
```

Figure 3.8 - Adding our script for iterating through the row and writing data to the text file

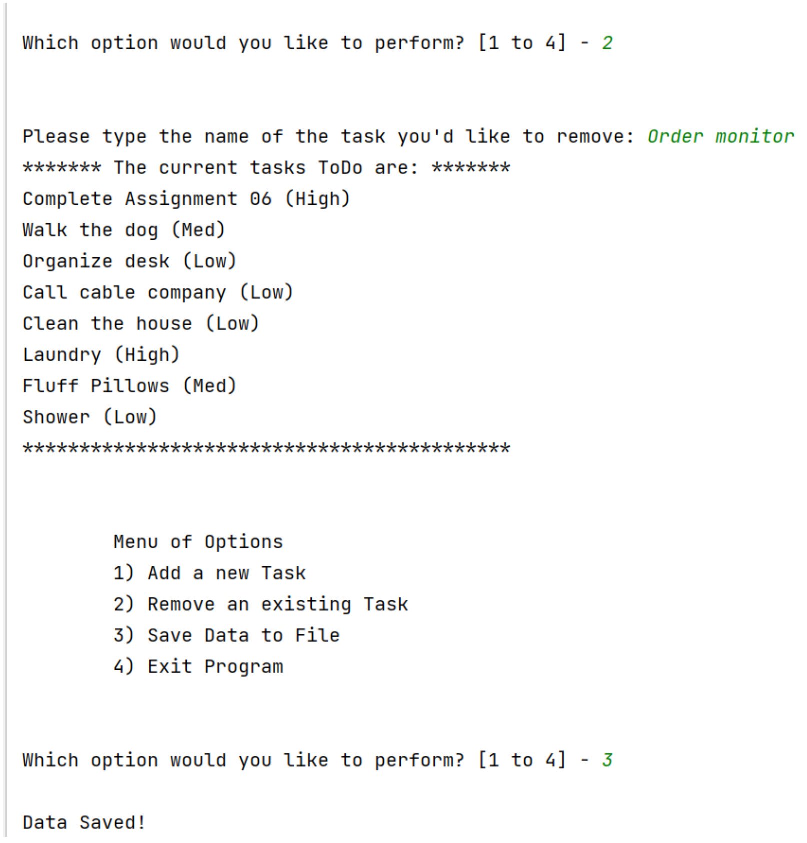
Testing

Alright, we'll now test all features in both PyCharm and CMD to ensure our script is working as intended.



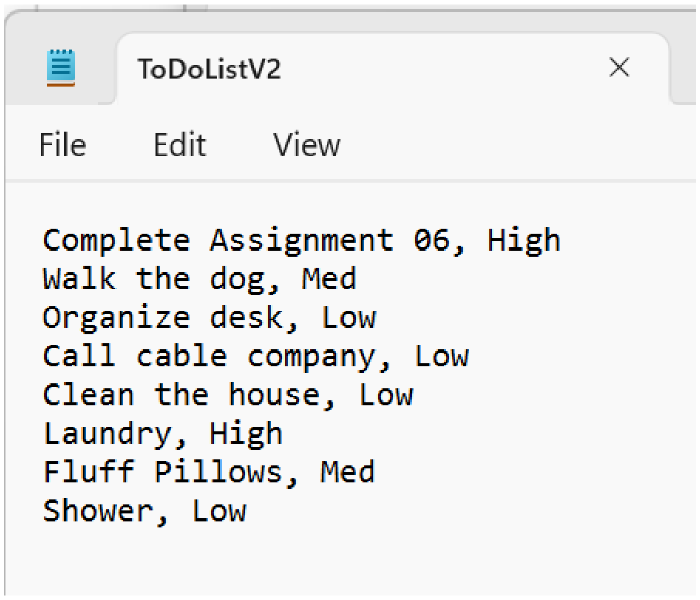
```
C:\Python\Python311\python.exe C:\_PythonClass\Assignments\Assignment08\ToDoListv2.py
******* The current tasks ToDo are: *******
Complete Assignment 06 (High)
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Order monitor (Med)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
*******************************************


        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

Please enter your new task: Shower
What priority level is this task? (Low, Med, or High): Low
******* The current tasks ToDo are: *******
Complete Assignment 06 (High)
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Order monitor (Med)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
Shower (Low)
*******************************************
```

Figure 8.1 - Running in PyCharm - Current tasks, menu and option 1 all working correctly



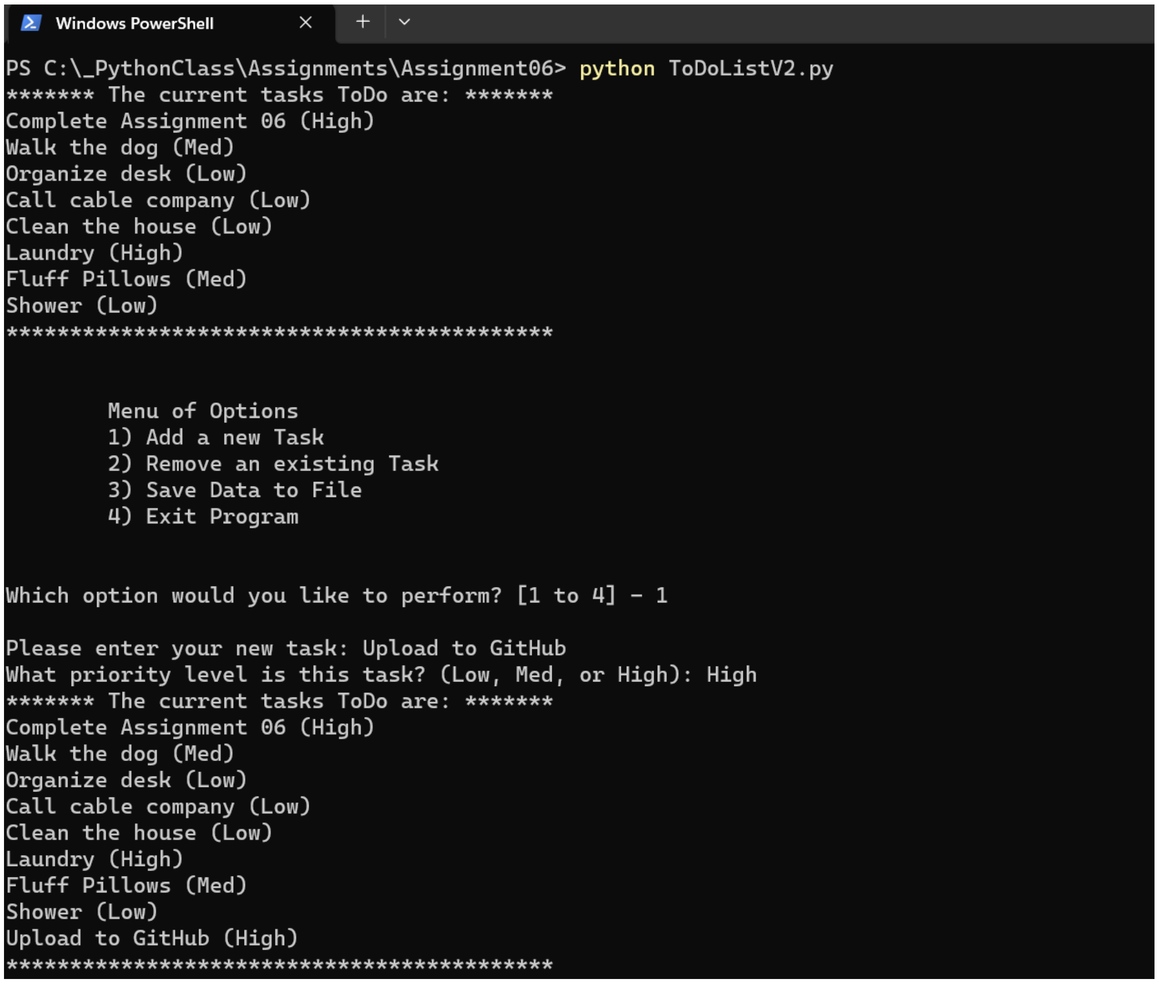
```
Which option would you like to perform? [1 to 4] - 2


Please type the name of the task you'd like to remove: Order monitor
******* The current tasks ToDo are: *******
Complete Assignment 06 (High)
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
Shower (Low)
*******************************************


        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 8.2 - Running in PyCharm - Option 2 successfully removes task from list & option 3 displays “Data Saved!”



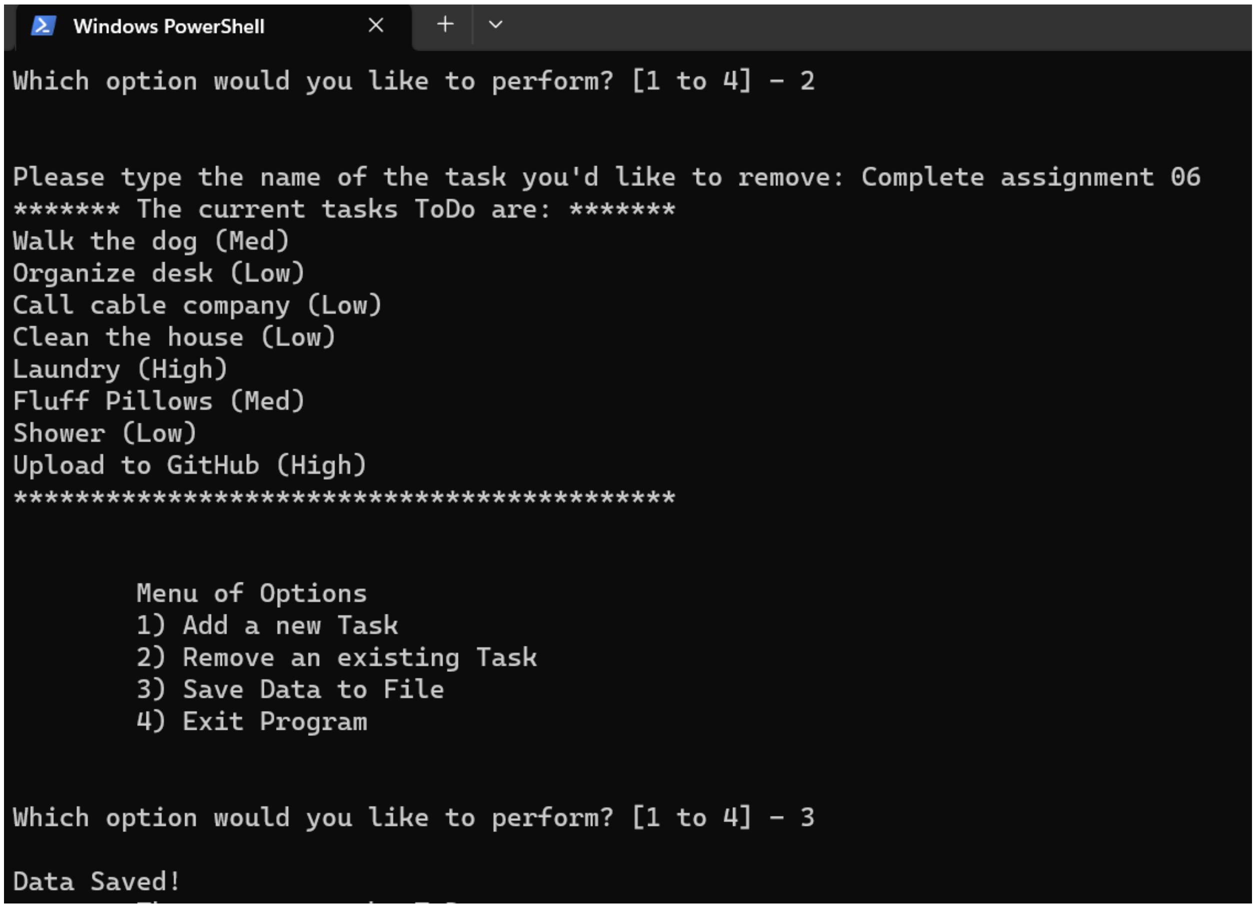
```
Complete Assignment 06, High
Walk the dog, Med
Organize desk, Low
Call cable company, Low
Clean the house, Low
Laundry, High
Fluff Pillows, Med
Shower, Low
```

Figure 8.3 - Confirming all changes appear in ToDoListV2.txt - bingo! Shower is added and Order Monitor has been removed!



```
PS C:\_PythonClass\Assignments\Assignment06> python ToDoListV2.py
******* The current tasks ToDo are: *******
Complete Assignment 06 (High)
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
Shower (Low)
*******************************************


        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

Please enter your new task: Upload to GitHub
What priority level is this task? (Low, Med, or High): High
******* The current tasks ToDo are: *******
Complete Assignment 06 (High)
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
Shower (Low)
Upload to GitHub (High)
*******************************************
```

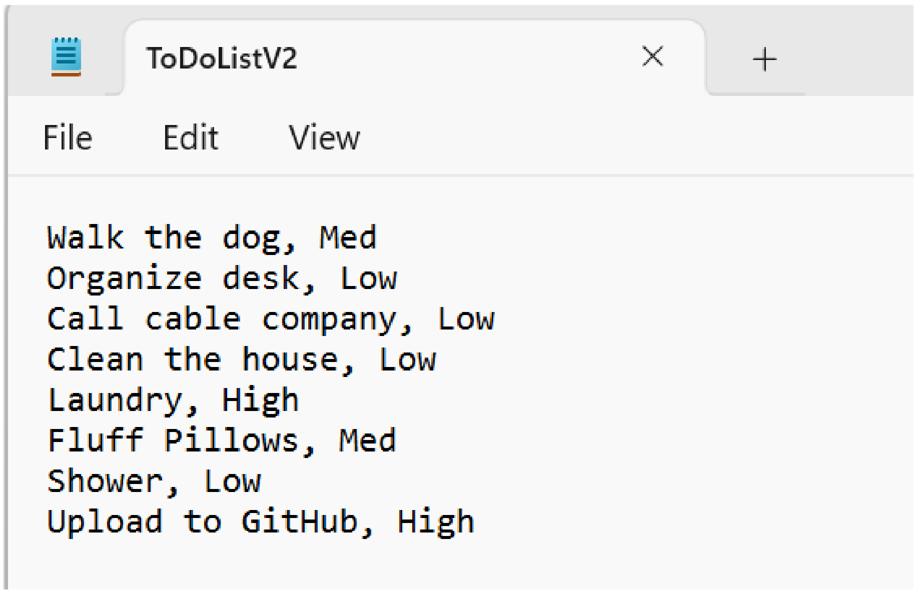
Figure 8.4 - Running in CMD - Option 1 adds task successfully



Windows PowerShell

```
Which option would you like to perform? [1 to 4] - 2


Please type the name of the task you'd like to remove: Complete assignment 06
******* The current tasks ToDo are: *******
Walk the dog (Med)
Organize desk (Low)
Call cable company (Low)
Clean the house (Low)
Laundry (High)
Fluff Pillows (Med)
Shower (Low)
Upload to GitHub (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] - 3

Data Saved!
```

Figure 8.5 - Running in CMD - Option 2 removes Assignment 06 from list & option 3 returns “Data Saved!”



ToDoListV2

```
Walk the dog, Med
Organize desk, Low
Call cable company, Low
Clean the house, Low
Laundry, High
Fluff Pillows, Med
Shower, Low
Upload to GitHub, High
```

Figure 8.6 - Confirming ToDoListV2.txt changes

Summary

The addition of functions and classes to our knowledge base is huge! This really opens up how we can organize code and keep it easily readable for future users. I found this assignment to be relatively simple, but it definitely takes a certain amount of concentration to ensure you are properly following the variables and parameters between the main body of your script and the various functions. Once you have the basic concept figured out though, it really was just copying and pasting the same functions we used from the previous assignment to get this one working properly. I'm excited to start a new project from scratch incorporating this new knowledge - it will be a totally different beast tackling it on my own!