Hannah Chung

May 24, 2020

IT FDN 100 Foundations of Programming: Python

Assignment 06

<https://github.com/hannahinissaquah/IntroToProg-Python-Mod06>

# **Assignment06: Working with Functions & Classes**

Introduction

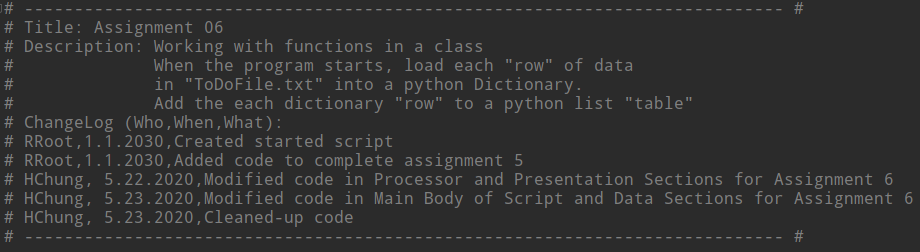
This week, we learned about creating scripts using Functions, and we also learned about creating simple Classes. Functions are a way of grouping one or more statements into useful chunks of code that make it easier to write, understand, and maintain, and classes are used to group functions or variables. Rather than creating a continuous series of instructions, using classes and functions make it easier for us to divide our code into three layers of concerns: data, processing, and presentation.

In this document, I will explain the steps I took to modify a script that manages a “To Do list” containing two columns of data: “Task” and “Priority”. I was provided with a file, “Assignment06\_Starter.py” that contained starter code. The starter code defines the functions to perform specific processing and presentation tasks, and my assignment was to add blocks of code where needed/missing to complete the assignment. Like last week’s assignment, this program allows the user to review current data, add a new item to the list, remove an existing item, save data to the file, reload data from the file, and to exit the program.

Assignment Steps

**Getting started**

First, I created a new PyCharm project that uses the \_PythonClass\Assignment06 folder as its location. I added the Assignment06\_Starter.py file to my project and loaded it into PyCharm. I then saved a copy of it and named the new file Assignment06.py. There were a few typos in the Script Header, so I corrected those and made my first entry into the change log as shown in Figure 1. Over time, I continued to make updates to the change log by recording the changes I had made to the previous version of the script.

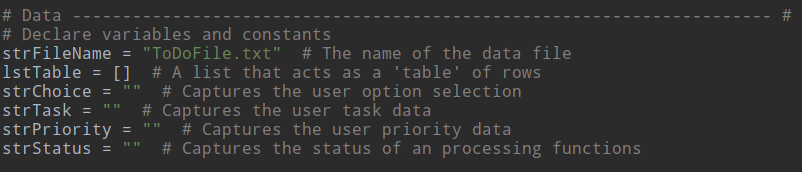


**Figure 1 Screenshot of Assignment06.py – Script Header**

Next, I ran the starter code to test if it works. I immediately got an error message stating the “ToDoFile.txt” could not be found. I realized it could not be found because it didn’t exist, so I created it. I re-ran the code and it appeared to be working, but not quite exactly as I expected it to with full functionality. So after I tested the starter code, I examined each line of the script.

**Data**

The program is separated into four main sections: Data, Processing, Input/Output (or Presentation), and the Main Body of the Script. The Data section was already populated with a list of declared variables to be used throughout the code. There were a couple of variables objFile and dicRow that I did not end up needing for the program so I removed those variables from this section. Figure 2 shows the final contents of my Data section.



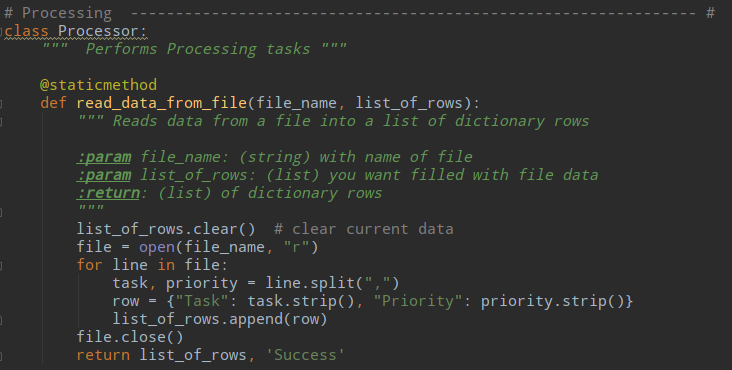
**Figure 2 Data Section**

**Processing**

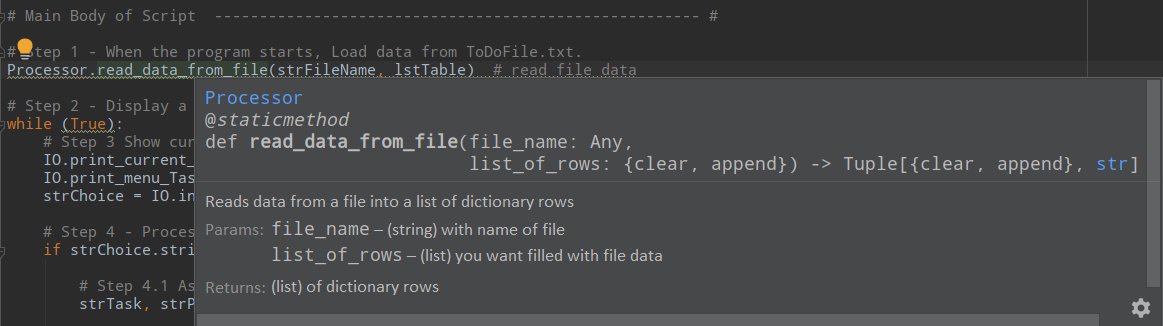
For the Processing and Presentation layers of the script, I chose to create and test my code for these two sections in a separate python file that I named “test\_classProcessor.py” and “test\_classIO.py”, respectively. This allowed me to focus on completing, testing, and trouble-shooting one block of code at a time before moving onto the next block of code.

The Processing Section is the layer that processes the information from the user. A class named “Processor” was created with four defined functions that perform different processing tasks: read\_data\_from\_file(), add\_data\_to\_list(), remove\_data\_from\_list(), and write\_data\_to\_file(). Before each function’s name, the starter code included a @staticmethod directive, which indicates you don’t need to make an object first, and the function can be called directly. We will learn more about this directive in a future module.

The read\_data\_from\_file() function as shown in Figure 3 was already completed and running correctly, so no changes were made to this code. The starter code also demonstrates a common Python practice of including a “docstring” (or document string), which is a header at the beginning of the function to provide additional notes. In PyCharm, to display the developer’s notes, use **ctrl + q** and a window will appear with the notes as shown in Figure 4. Inside the parenthesis of the read\_data\_from\_file() function are two parameters. *Parameters* are variable names that receive values sent to the function known as *arguments*. Functions can receive values from and it can return values back to the part of the program that called it. Here, we are returning two values: list\_of\_rows, which is the list you want filled with file data, and ‘Success’ to indicate it was successful. It is not necessary to return ‘Success’ status, but I have left the status inside the code for educational/validation purposes only.

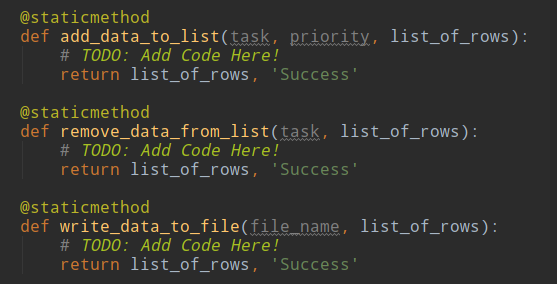
******

**Figure 3 Processor class and read\_data\_from\_file function**



**Figure 4 Showing the docstring text using PyCharm (press ctrl + q)**

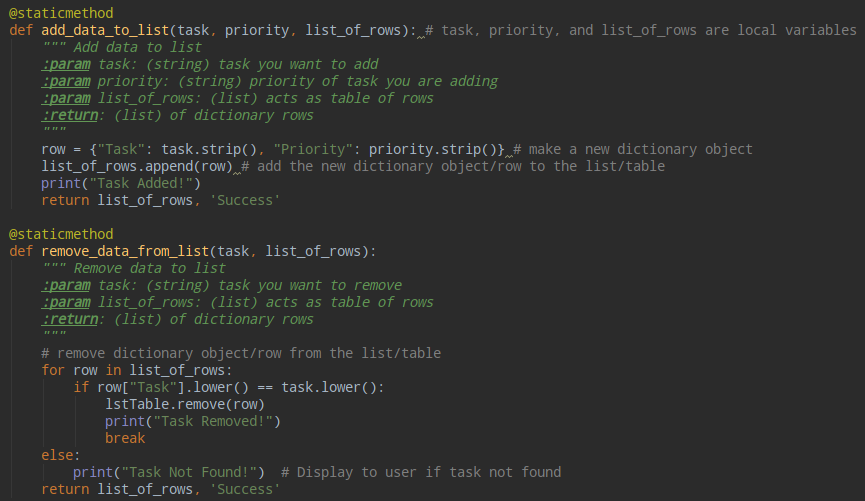
The first TODO item was to add code to complete the add\_data\_to\_list() function as shown in Figure 5. This function adds the new task item and priority to the list. By looking at the starter code, I can see that this function will receive values through its parameters: task, priority, and list\_of\_rows and will return the values of list\_of\_rows with the added dictionary row and ‘Success’.



**Figure 5 Showing a screenshot of the Assignment06\_Starter,py – “add\_data\_to\_list” function**

To complete the code for the add\_data\_to\_list() function, I first added a statement to make a new dictionary object based on the information provided by the user and then I created a second statement to add the new dictionary object/row to the list/table as shown in Figure 6.

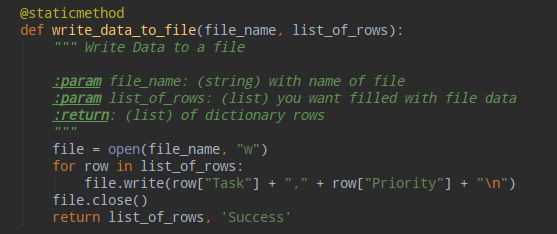
To complete the remove\_data\_from\_list() function, I created a for loop to remove the user-entered item from the table. The function will return the values of list\_of\_rows with the removed dictionary row and ‘Success’.



**Figure 6 Screenshot of completed code for functions: add\_data\_to\_list and remove\_data\_from\_list**

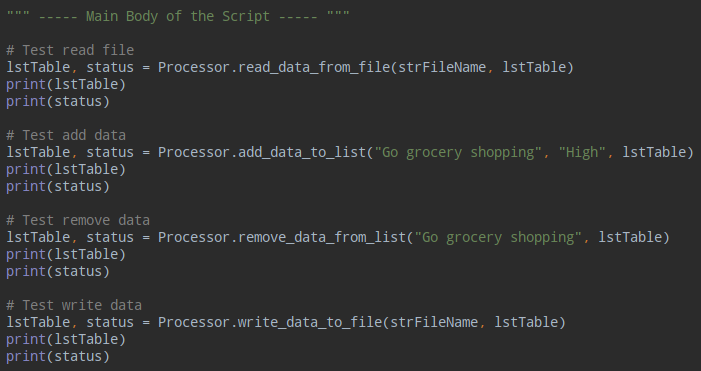
You may notice variables having the same name inside different functions (e.g. task). They are not the same with no connection to each other. No variable that you create in a function can be directly accessed outside its function because of a principle known as *encapsulation*. A variable created inside a function is called a *local variable* because it is local to that function. Any variable that is created in the global scope, which is all the area outside the function(s), is called a *global variable*.

The last function inside the Processor class to complete was the write\_data\_to\_file() function. When this function is executed, the ToDoFile.txt will be opened in write access mode, and the dictionary rows are written to the file using the write() method. The file is closed and the function returns the list of dictionary rows and again, ‘Success’.



**Figure 7 Screenshot of completed code for write\_data\_to\_file() function**

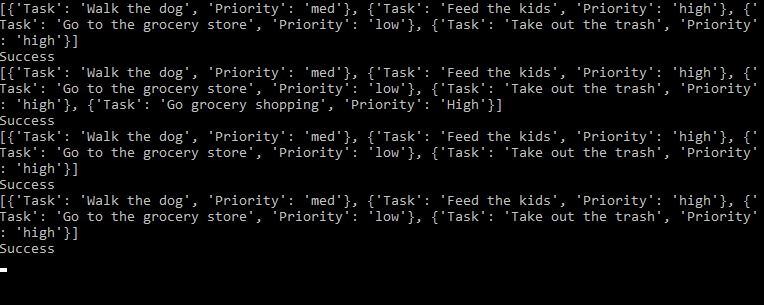
Figure 8 shows a screenshot of the Main Body of Script that I wrote inside the test\_classProcess.py file to test the class Processor functions.



**Figure 8 Main Body of Script of test\_classProcessor.py**

Figure 9 shows the results of the test\_classProcessor.py running in the Command Shell. After successfully testing the code for the Processor class, I copied and pasted the code from the test file to the Assignment06.py file.

I have now completed the Processing layer of Assignment06 and will take similar steps to complete and test the Presentation (Input/Output) layer.



**Figure 9 test\_classProcessor.py script running in Command Shell**

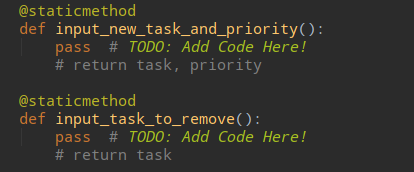
**Presentation**

The Presentation Section is the layer that performs input and output tasks. A class named “IO” was created with seven defined functions that perform different input/output tasks as shown in Table 1.

**Table 1 List of Functions in class IO**

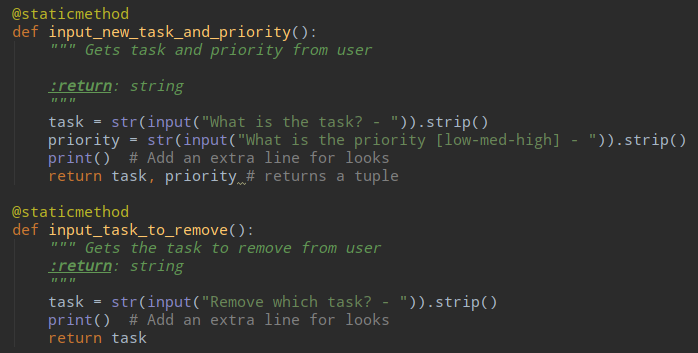
|  |  |
| --- | --- |
| **Function** | **Description** |
| print\_menu\_Tasks() | Displays a menu of choices to the user |
| input\_menu\_choice() | Gets menu choice from the user |
| print\_current\_Tasks\_in\_list(list\_of\_rows) | Shows the current tasks in the list of dictionary rows |
| input\_yes\_no\_choice(nessage) | Gets a yes or no choice from the user |
| input\_press\_to\_continue(optional\_message=‘ ’) | Pauses program and shows a message before continuing |
| input\_new\_task\_and\_priority() | Gets task and priority from user |
| input\_task\_to\_remove() | Gets the task to remove from the user |

Like the Processing section, I created and tested my code for the Presentation section in another python file named test\_classIO.py. Of the seven functions defined, two functions needed a block of code: input\_new\_task\_and\_priority() and input\_task\_to\_remove() as shown in Figure 10.



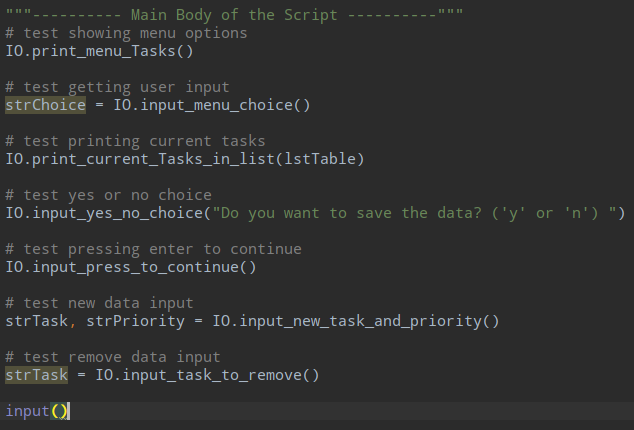
**Figure 10 Screenshot of functions in class IO that needed code added**

The input\_new\_task\_and\_priority() is getting the task and priority information from the user and assigning its values to local variables task and priority. The function will then return the value of task and priority back to the part of the program that called it. Similarly, the input\_task\_to\_remove() function gets the task to remove from the user and assigns it to a local variable task. The function will then return the value of task back to the part of the program that called it. Figure 11 shows a screenshot of the completed code for input\_new\_task\_and\_priority() and input\_task\_to\_remove() functions.

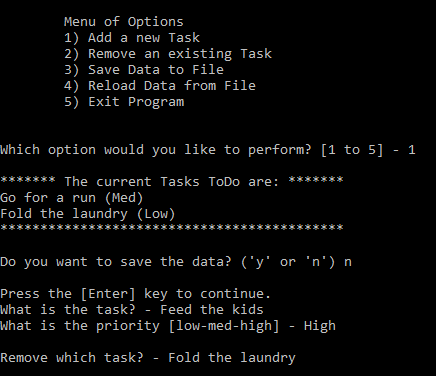


**Figure 11 Screenshot of code added to input\_new\_task\_and\_priority() and input\_task\_to\_remove() functions**

Figure 12 shows a screenshot of the Main Body of Script that I wrote inside the test\_classIO.py file to test the class IO functions.

******

**Figure 12 Main Body of Script of test\_classIO.py**

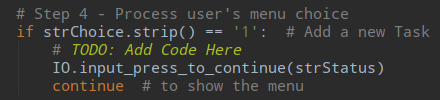


**Figure 13 test\_classIO.py script running in Command Shell**

After successfully testing the code for the IO class, I copied and pasted the code from the test file to the Assignment06.py file.

**Main Body of Script**

In Python, you must define a function before you can use code to call the function. The Main Body of the Script is the layer of the code where I call the functions defined earlier. There are four TODO items to complete in this section. I will explain in detail how I completed the first TODO item shown in Figure 14 and since the same approach was taken for the remaining 3 TODO items, I will just include a brief description and screenshots of the completed code for the remaining items.

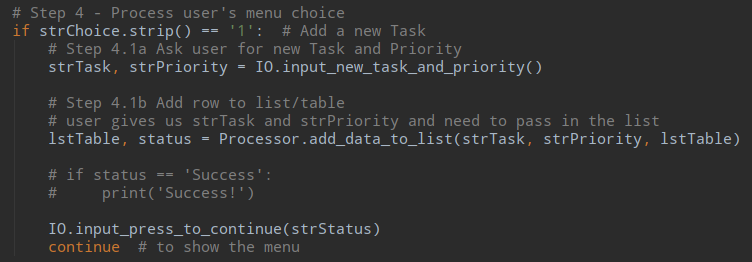


**Figure 14 TODO item for Option 1 (Add New Task)**

If the user chooses Option 1 (Add a new Task), the program needs to *get* (I/O) information from the user about the task to add and its priority. It then needs to *add* (process) the new row to the list/table. To first get the information from the user, I will call the IO.input\_new\_task\_and\_priority() function and assign the returned values of task and priority as shown in Figure 11, to the global variables strTask and strPriority (Step 4.1a).

Next, to add the data to the list/table, I will call the Processor.add\_data\_to\_list() function. You will notice that inside the parenthesis are strTask, strPriority, and lstTable. These are the *arguments* that will get passed into the function’s *parameters*, which are task, priority, and list\_of\_rows, in that order. Here, I am using what is known as *positional parameters* and *positional arguments*, meaning that the parameters are getting their values based solely on the position of the values sent. Another way to assign certain values to specific parameters, regardless of order, is by using *keyword arguments*.

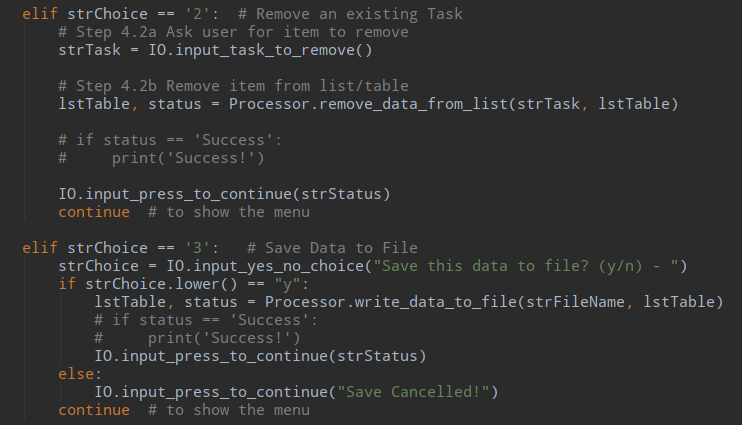
This function will append the new item to list\_of\_rows and return values of list\_of\_rows and ‘Success’ to the part of the program that called it (Step 4.1b) which are then assigned to the variables lstTable and status, respectively. Figure 15 shows a screenshot of the completed code for Option 1. I have boxed in yellow the new code with comments that were added.



**Figure 15 Screenshot of completed code for Option 1 (Add New Item)**

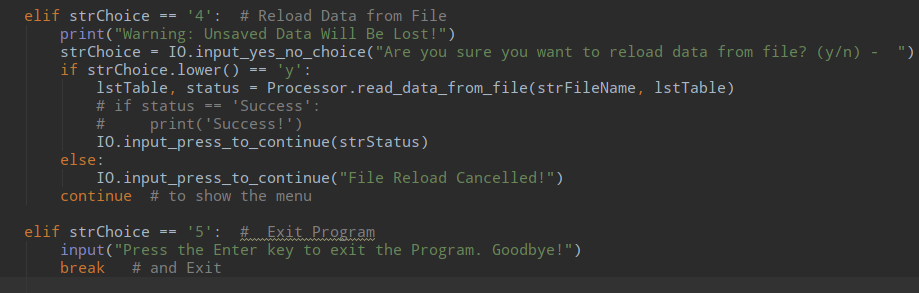
If the user selects Option 2 (Remove an existing Task), then, the program will call the IO.input\_task\_to\_remove() function to ask the user “Remove which task?” and the returned value will be assigned to the variable strTask. Next, the Processor.remove\_data\_from\_list() function, receiving the strTask and lstTable arguments that get passed into the task and list\_of\_rows parameters, will remove the item from the list/table and the returned values of list\_of\_rows and ‘Success’ will be assigned to lstTable and status, respectively.

If the user selects Option 3 and verifies he/she would like to save the data to the file, the Processor.write\_data\_to\_file() function will be called and the strFileName with the lstTable are the arguments that get passed into the parameters file\_name and list\_of\_rows, respectively. See Figure 16 for a screenshot of the completed code for Options 2 and 3.



**Figure 16 Screenshot of completed code for Option 2 (Remove Task) and Option 3 (Save Data to File)**

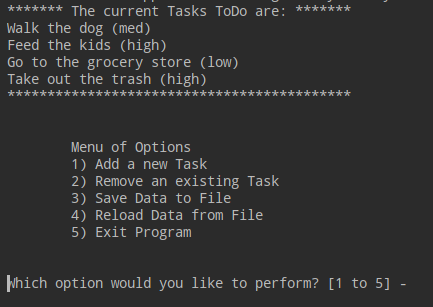
Figure 17 shows the completed code for Option 4. I also replaced the print statement for strChoice = 5 with an input() function to add a pause in the program before exiting.



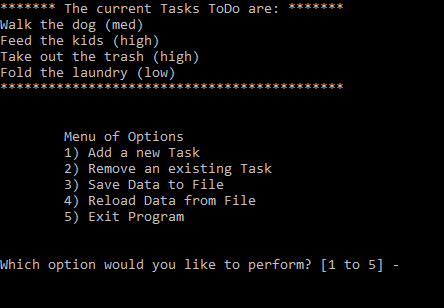
**Figure 17 Screenshot of completed code for Option 4 (Reload Data from File) and Option 5 (Exit Program)**

**Running the Script**

When the program starts, any data that is inside the ToDoFile.txt will be loaded and displayed under current tasks. Figure 18 shows the script running in PyCharm and Figure 19 shows the script running in the Command Window.

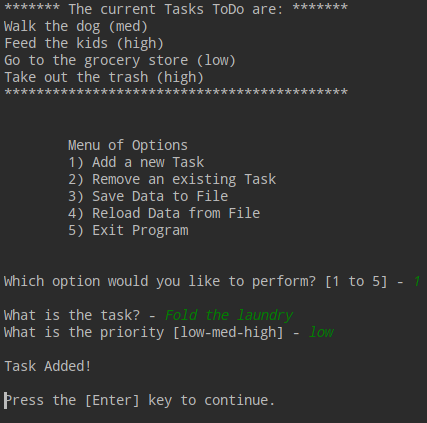
****

**Figure 18 PyCharm - Menu of Options**

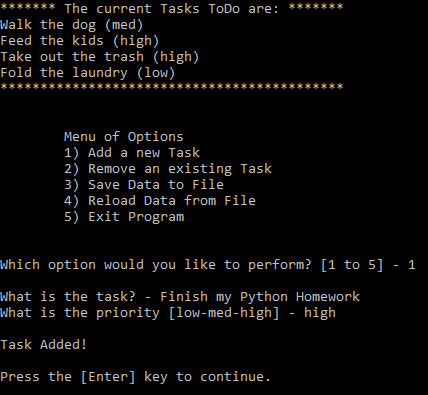


**Figure 19 Command Window - Menu of Options**

Figure 20 shows the script running in PyCharm when Option 1 is selected, and Figure 21 shows the script running in the Command Window when Option 1 is selected.

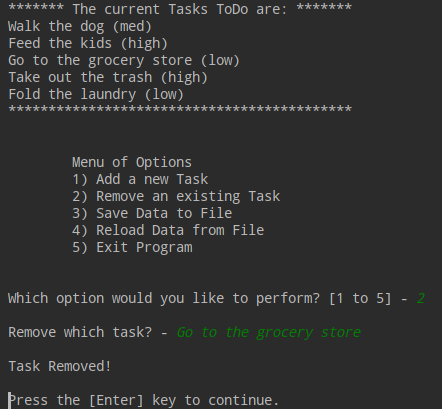
****

**Figure 20 PyCharm - Selecting Option 1**

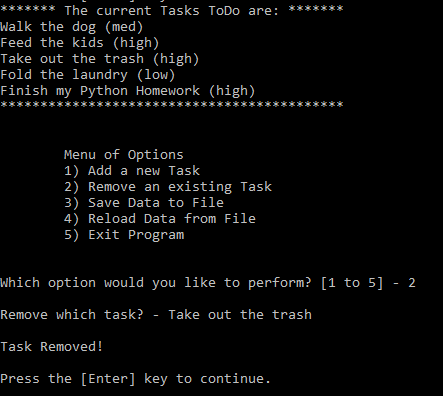


**Figure 21 Command Window - Selecting Option 1**

When user selects Option 2, he/she is prompted to enter an existing task to remove. See Figure 22 to see script running in PyCharm when the user selects Option 2, and see Figure 23 to see the script running in the Command Window.

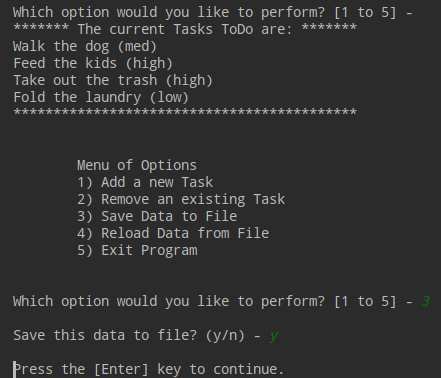
******

**Figure 22 PyCharm - Selecting Option 2**

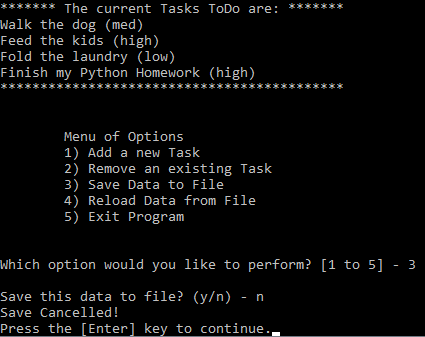
******

**Figure 23 Command Windowl - Selecting Option 2**

When the user selects Option 3, the user is asked whether he/she would like to save the data to the file. Figure 24 shows the script running in PyCharm when the user selects Option 3, and Figure 25 shows the script running in the Command Window. Figure 24 shows the user choosing to save to the file whereas Figure 25 shows the user choosing not to save the data to the file.

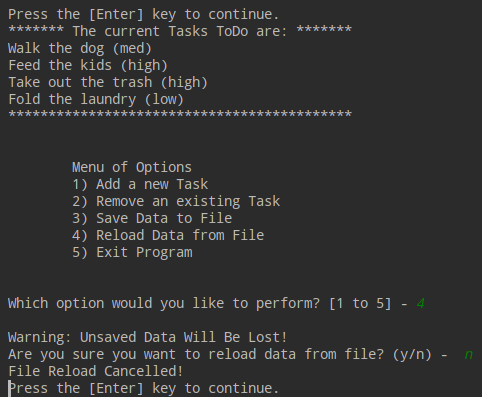
******

**Figure 24 PyCharm - Selecting Option 3**

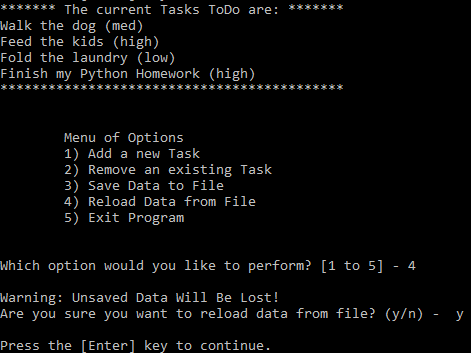
****

**Figure 25 Command Window - Selecting Option 3**

When Option 4 is selected, the user is asked to verify that he/she wants to reload data from the file. Figure 26 shows the script running in PyCharm and the user types ‘n’ for no. Figure 27 shows the script running in the Command Window when Option 4 is selected and the user types ‘y’ for yes.

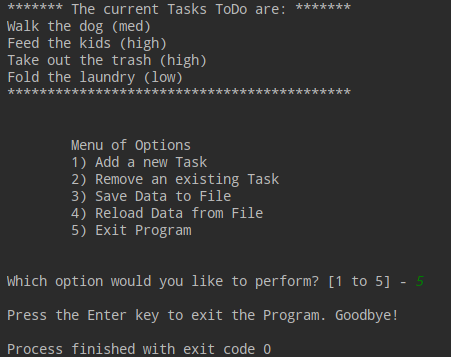
******

**Figure 26 PyCharm - Selecting Option 4**

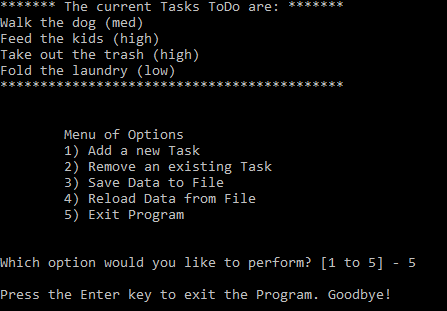
******

**Figure 27 Command Window - Selecting Option 4**

When user selects Option 5, he/she is asked to press the ‘Enter’ key to exit. Figure 28 shows the script running in PyCharm when the user selects Option 5, and Figure 29 shows the script running in the Command Window when Option 5 is selected.

******

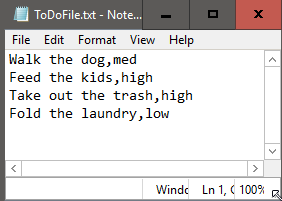
***Figure 28 PyCharm - Selecting Option 5***

******

***Figure 29 Command Window - Selecting Option 5***

**Verifying the Script Worked**

I verified that the script worked by opening the ToDoFile.txt located in the \_PythonClass\Assignment06 folder. Figure 30 shows a screenshot of its contents.

******

**Figure 30 Screenshot of ToDoFile.txt**

Summary

This week, we learned about creating scripts using Functions, and we also learned a little about creating simple Classes. Using classes and functions make it easier for us to divide our code into three layers of concerns: data, processing, and presentation.

For this assignment, I modified a script that manages a “To Do list” containing two columns of data: “Task” and “Priority”. I was provided with a file, “Assignment06\_Starter.py” that contained starter code. The code defined many functions that were divided into two classes: Processor and IO (input-output), and my assignment was to complete the blocks of code that were missing in the different layers of the script. Working with starter code has its advantages and disadvantages. It was helpful to have the separated sections with pseudo-code, defined functions, and TODO comments. I did not find any bugs in the starter code but there were some extraneous pieces of code that I did not believe were needed.