Lauren Ferrier
November 22, 2022
Foundations of Programming, Python
Assignment 06
Git Hub Repository
Git Hub Website

Improving Upon a “ToDo” List: A Python Script

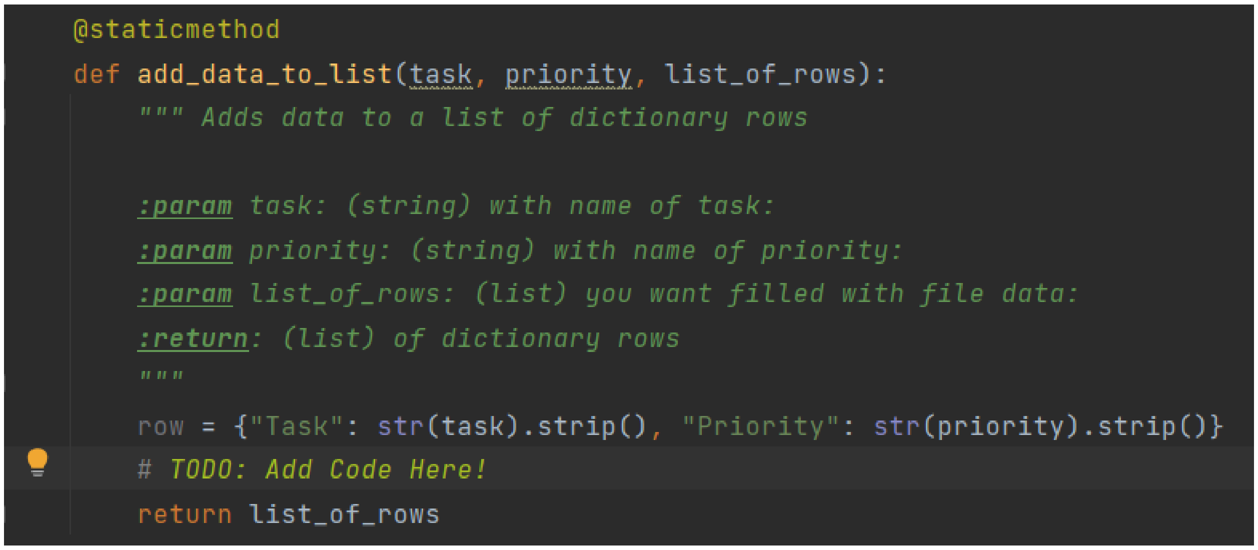
Introduction

This document details the steps taken to create a python script that builds off Module 5’s “ToDo” list script. While last week’s script emphasized the principles of dictionaries via a menu of options for our user to follow, this week takes this further by organizing the same script into classes and functions. We further break down these concepts by utilizing variables as arguments and implementing docstrings for further organization.

Steps Taken for Assignment 06

1. This assignment was completed within PyCharm (Jet Brains, 2022).

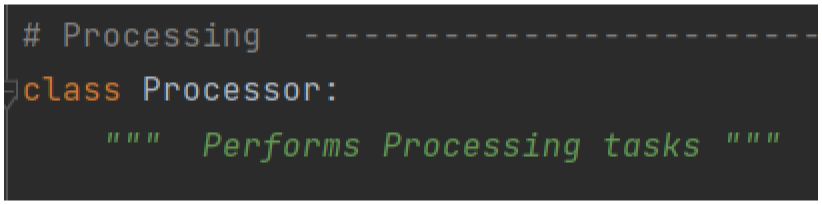
2. Within PyCharm, I opened the Assigment06_Starter.py that was provided with this week’s course materials since this would serve as the script template to get me started on the assignment. Several aspects of this week’s script have already been written, so I am focusing on editing and documenting where in our script template it specifies *# TODO: Add Code Here*. As shown below.



```
@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 filled with file data:
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
    # TODO: Add Code Here!
    return list_of_rows
```

Figure 1 An example of a script template that contains a docstring. Docstrings are designed for developers to include additional notes on functions. They also have the added benefit when used in integrated development environments to be hidden and called forth as needed for better visualization purposes. (_MOD6PYTHONPROGRAMMINGNOTES.DOCX 2022, PG 17).

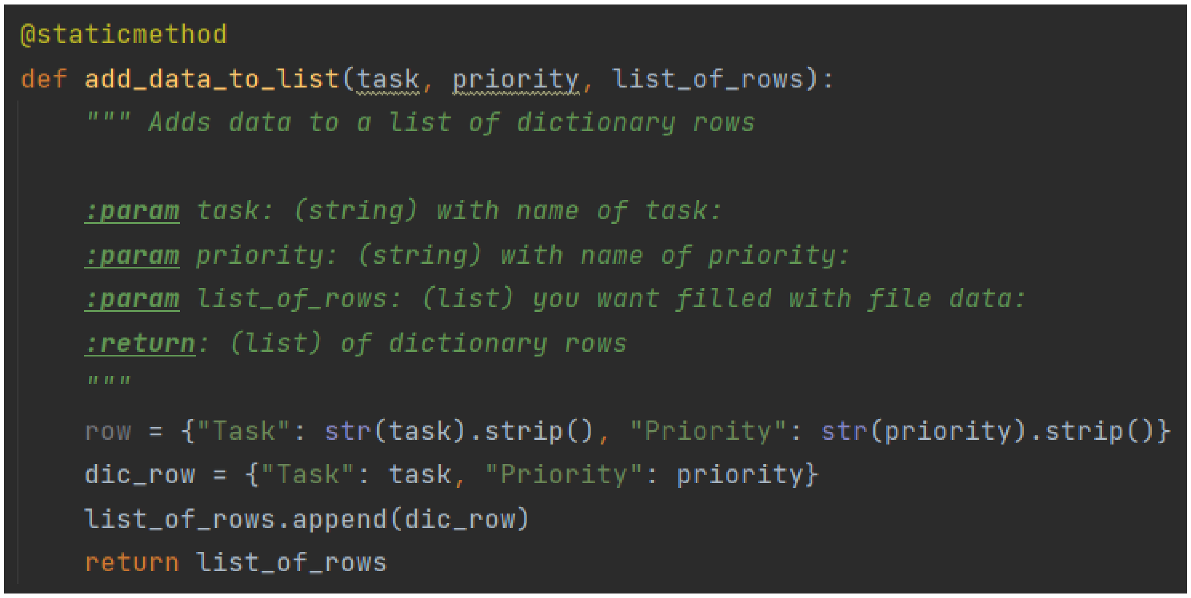
3. This week's script template has defined a framework of classes and associated functions. Our first class is *Processor*, where all the functions that process our user's eventual tasks will live.



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

Figure 2 Above is example of a class, which is a method of grouping functions, variables, and constants. (_MOD6PYTHONPROGRAMMINGNOTES.DOCX 2022, PG 18).

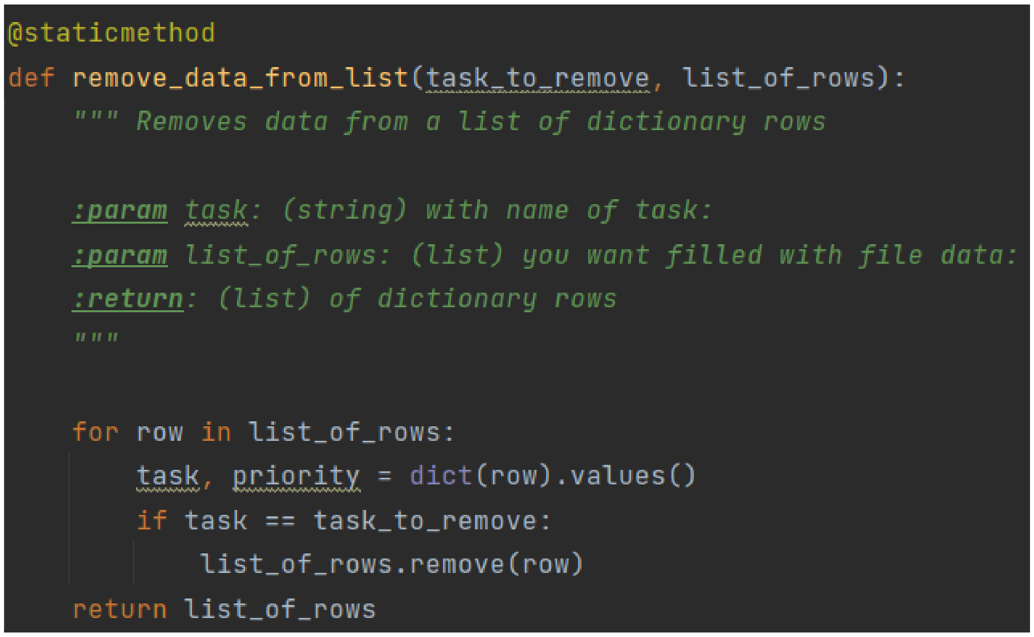
4. The first function within the above class that requires editing relates to our user adding a new task to our list of dictionary rows and is titled *add_data_to_list*. We have given this function three parameters: task, priority, and *list_of_rows*. Task and priority are what our user will eventually define, and we will expand upon this more later, but the principal here is that we want to take an eventual task and priority, set them up within our structured dictionary row within our stored list (*dic_row*). We then will append this row, returning it as *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 filled with file data:
    :return: (list) of dictionary rows
    """
    row = {"Task": str(task).strip(), "Priority": str(priority).strip()}
    dic_row = {"Task": task, "Priority": priority}
    list_of_rows.append(dic_row)
    return list_of_rows
```

Figure 3 A demonstration of a function using multiple parameters. These parameters permit us to pass values into our function for processing (_MOD6PYTHONPROGRAMMINGNOTES.DOCX 2022, PG 2).

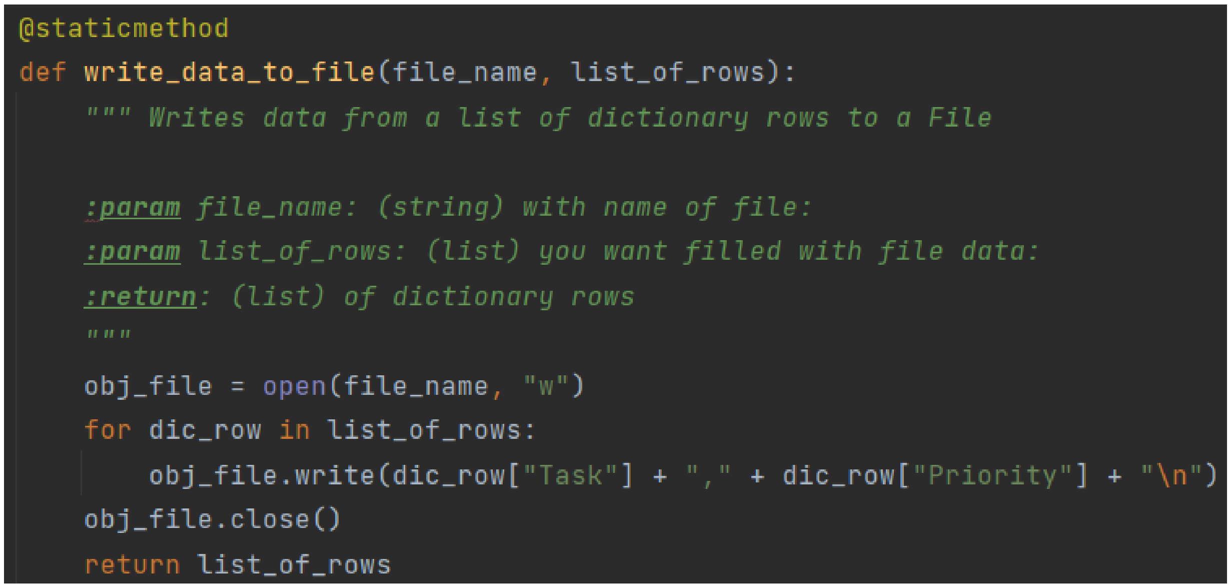
5. Moving through our script, we still have several functions left to edit within our *Processor* class, the next being *remove_data_from_list*. Like above, we have multiple parameters moving through our function. Our first parameter is *task_to_remove*, which is the task that will eventually be provided by our user (more on this later!) to, just as it sounds like, be removed from their to do list. To complete this, our function will look for a task within our list of dictionary rows that matches the task (*task_to_remove*) provided by our user. If there is a match, that matching task's corresponding row is removed, and the user is returned the resulting to do list (*list_of_rows*).



```
@staticmethod
def remove_data_from_list(task_to_remove, 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:
        task, priority = dict(row).values()
        if task == task_to_remove:
            list_of_rows.remove(row)
    return list_of_rows
```

Figure 4 Our remove_data_from_list function continues our use of multiple parameters, passing our user's provided value through our function to remove any corresponding tasks from our stored list of dictionary rows.

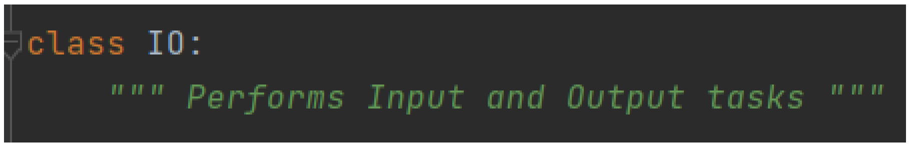
6. Our last function within our *Processor* class is *write_data_to_file*. This is where any of the changes that our user implemented, whether it be adding or removing a task, is taken from temporary memory, and then stored into permanent memory via our *ToDoFile.txt*. We again have two parameters, *file_name* and *list_of_rows*. We are specifying that we want to write to our file, the dictionary rows (*dic_row*) stored in our overall list (*list_of_rows*), setup to respect the row formatting of our eventual user inputted task and corresponding priority. Then like our previous functions, we return our saved data within our *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
    """
    obj_file = open(file_name, "w")
    for dic_row in list_of_rows:
        obj_file.write(dic_row["Task"] + "," + dic_row["Priority"] + "\n")
    obj_file.close()
    return list_of_rows
```

Figure 5 Our write_data_to_file function that will be committing our user's changes into stored memory.

7. We are now moving on to our next class! This class relates to the presentation that our user will see to execute all the above functions we just detailed and is titled *IO*.

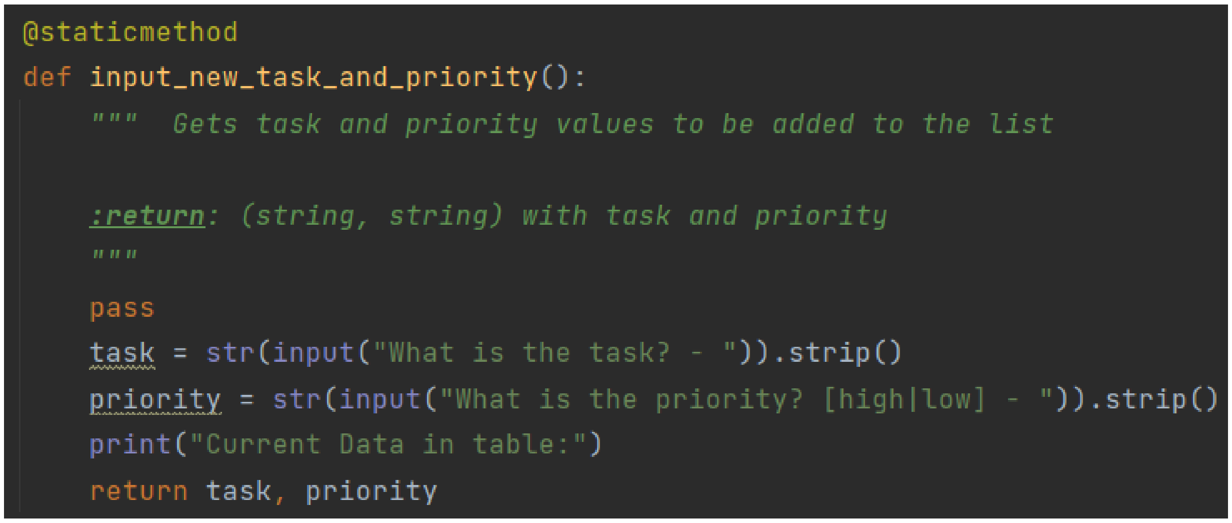


```
class IO:
    """ Performs Input and Output tasks """
```

Figure 6 Our IO class is where our user presentation script is warehoused. This area of our script deals more directly with our user's given inputs and their corresponding outputs.

8. Relating back to our *add_data_to_list* function in our *Processor* class, we now want to setup the function where the user is presented with the option to provide the task and priority that will pass through this aforementioned function. Our *input_menu_choice* function is how we will do this! Unlike all our previous functions, we do not have any parameters defined here, since the user is the one that will be providing us our dynamic values via string inputs, *task* and *priority*.

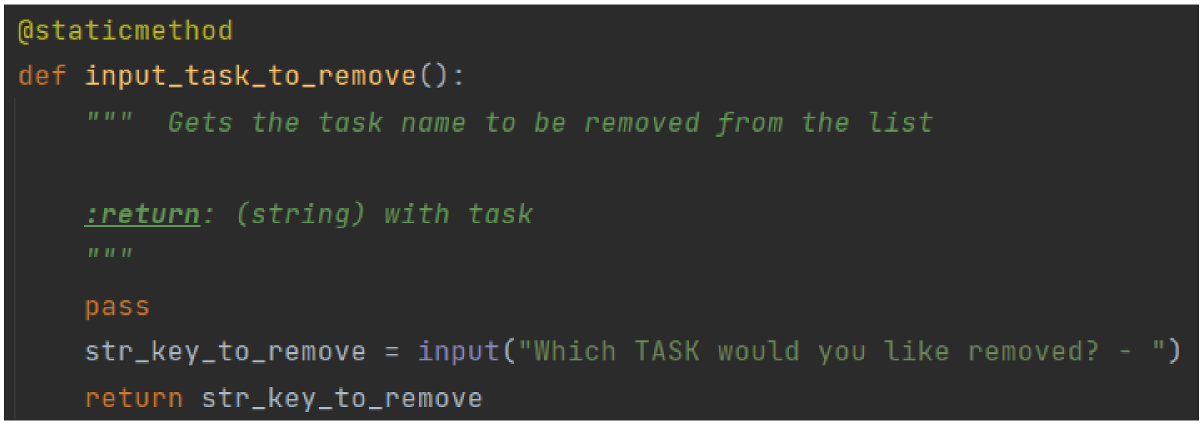
Once these values are provided by our user, they are returned their values for review.



```
@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
    task = str(input("What is the task? - ")).strip()
    priority = str(input("What is the priority? [high|low] - ")).strip()
    print("Current Data in table:")
    return task, priority
```

Figure 7 The inputs of task and priority are defined by our user. These are then eventually passed through our add_data_to_list function, as shown previously in figure 3.

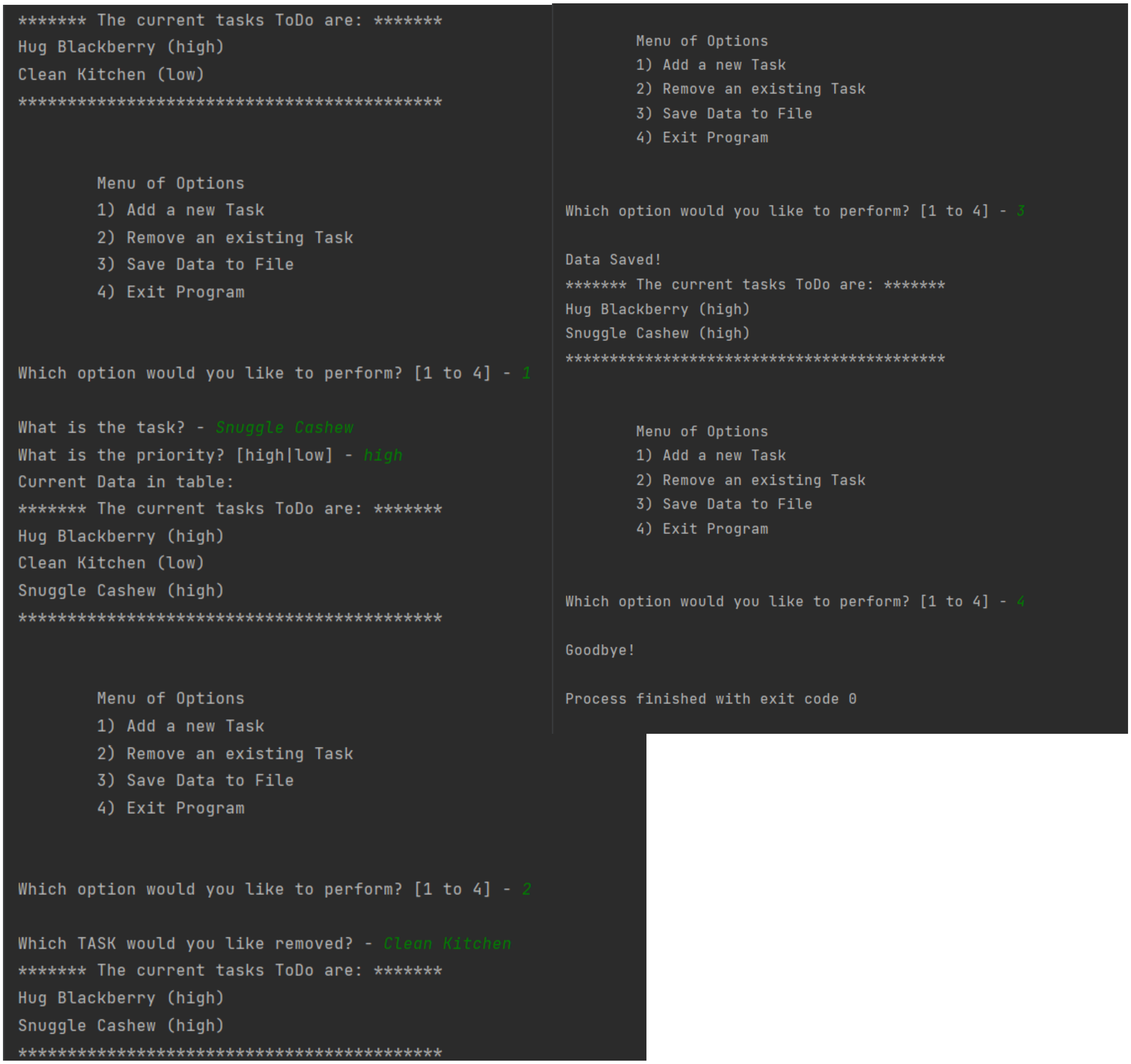
9. Next is our function that setups up the interface for our user to remove a given task, as defined within our *input_task_to_remove* function. Like our above function, we do not have any parameters defined since our user will provide our value here. We have defined this value as str_key_to_remove, which is the input our user will provide.



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

    :return: (string) with task
    """
    pass
    str_key_to_remove = input("Which TASK would you like removed? - ")
    return str_key_to_remove
```

Figure 8 This function sets the scene for the task that eventually will be removed in our remove_data_from_list function, as shown previously in figure 4.

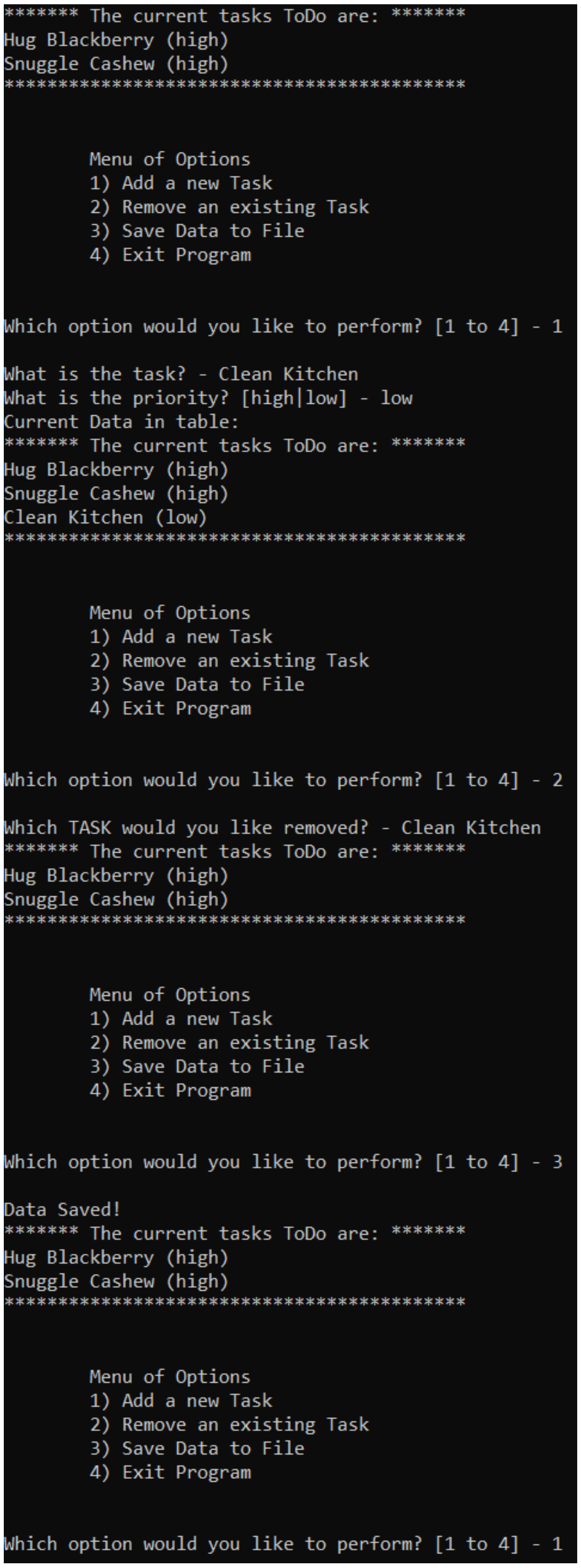
10. We now have our script template completely filled out!



```
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Clean Kitchen (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

What is the task? - Snuggle Cashew
What is the priority? [high|low] - high
Current Data in table:
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Clean Kitchen (low)
Snuggle Cashew (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] - 2

Which TASK would you like removed? - Clean Kitchen
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (high)
*******************************************
```

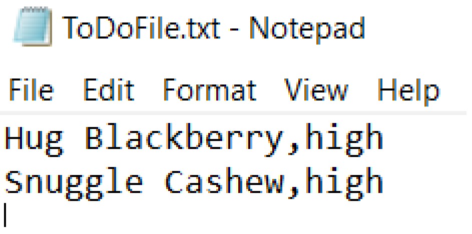
```
        Menu of Options
        1) Add a new Task
        2) Remove an existing Task
        3) Save Data to File
        4) Exit Program


Which option would you like to perform? [1 to 4] - 3

Data Saved!
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (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] - 4

Goodbye!

Process finished with exit code 0
```

Figure 9 Completed Assignment 06. As shown in PyCharm.



```
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (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] - 1

What is the task? - Clean Kitchen
What is the priority? [high|low] - low
Current Data in table:
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (high)
Clean Kitchen (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] - 2

Which TASK would you like removed? - Clean Kitchen
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (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!
******* The current tasks ToDo are: *******
Hug Blackberry (high)
Snuggle Cashew (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] - 1
```

Figure 10 Completed Assignment 06, as shown in a command window.



```
Hug Blackberry,high
Snuggle Cashew,high
```

Figure 11 Resulting ToDoFile.txt from Assignment 06's executed script.

Summary

Using our provided course materials, I created a successful script that permits a user to engage with a menu that presents multiple options of moving forward. While this script was also used during Module 05 to demonstrate dictionary keys, saving this dictionary data as a list, and then subsequently writing all inputted data into a file for storage, this week's version has reorganized this script to implement classes and functions that contain multiple parameters. The result is a much more streamlined script that does not rely on linear execution, but instead, follows a path organizer per functional classes of processing and presentation that eventually will be executed via our main body of code.