Kyle O’Donnell

August 16, 2020

Foundations of Programming: Python

Assignment06

https://github.com/kylenod/ITFDN110-Mod06/blob/master/docs/index.md

Modifying the To-Do List Program in Python

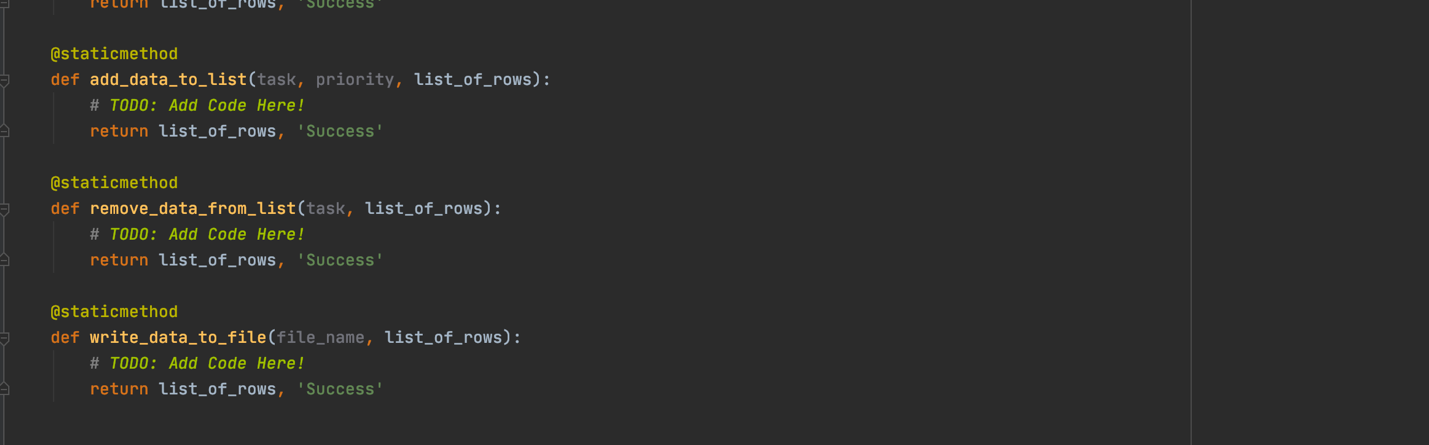
# Introduction

This paper documents the process of updating the to-do list program in Python in completion of week 6’s assignment for Randal Root’s ‘Foundations of Programming: Python.’ The process includes building out the Assignment06\_Starter.py template with code for several custom functions and ***if*** statements. This week’s script provides greater separation of processing and presentation layers compared to the previous version of the to-do list program. The new format is more consistent with the principle of *Separation of Concerns* and standard Pythonic coding conventions. Updating the script involved many of the concepts covered in week 6, such as classes, functions and return values.

# Writing the Script

## Template

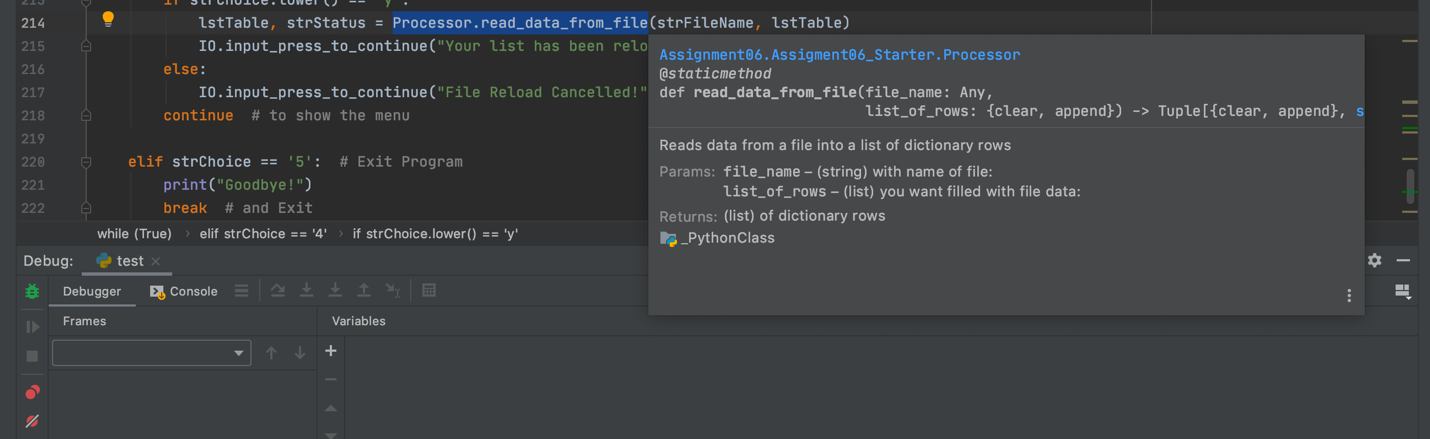
The week’s script was adapted from the Assignment06\_Starter.py template provided by the professor. The template contained an outline for custom functions, classes and the main body of the script, as well as instructions on where to add code (Figure 1).



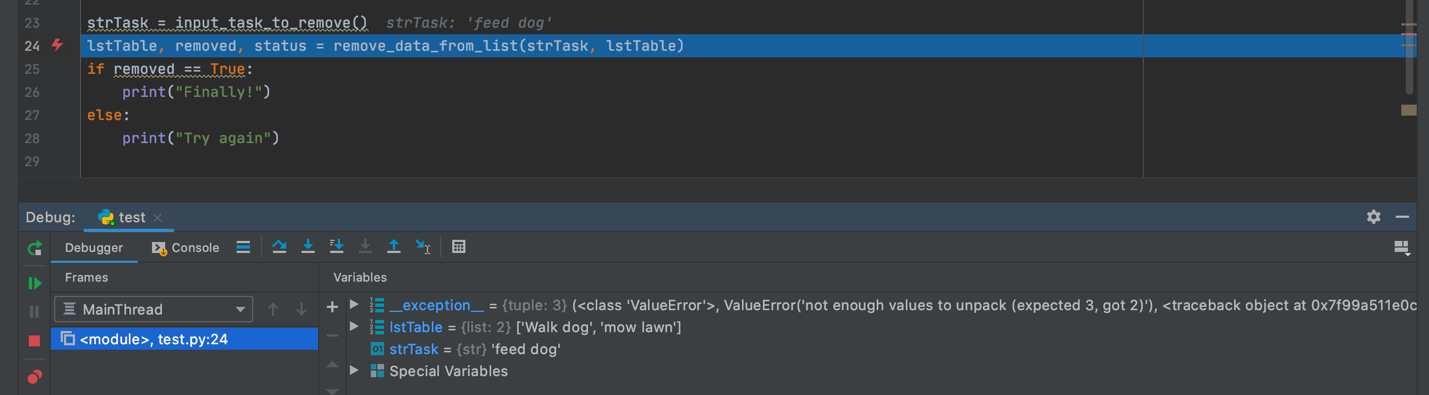
**Figure 1.** Template for Assignment06\_Starter.py

## Pycharm

Using PyCharm, I created a new project for Assignment06\_Starter.py in the Assignment06 directory. The advanced capabilities in PyCharm are becoming increasingly useful as scripts become more complex. It was especially helpful to be able to view docstrings for custom functions without having to scroll to the top of the script (Figure 2) and to debug different parts of code in a test file (Figure 3).



**Figure 2.** Using shortcuts (F1 on MacOS) to view docstring in PyCharm



**Figure 3.** Using Debugger in PyCharm

## Data

The first section of code declares the variables and constants that will be used throughout the program. Similar to the previous week, this portion was provided with the template. I added the value boolRemove to help capture the status of removed items in option 2.

strFileName = "ToDoFile.txt" # The name of the data file

objFile = None # An object that represents a file

dicRow = {} # A row of data separated into elements of a dictionary {Task,Priority}

lstTable = [] # A list that acts as a 'table' of rows

strChoice = "" # Captures the user option selection

strTask = "" # Captures the user task data

strPriority = "" # Captures the user priority data

strStatus = "" # Captures the status of processing functions

boolRemove = None # Captures status of removed row

Listing 1

## Processing

To help organize the script, the custom functions defined in the processing section are divided into two classes: **Processor** (for processing) and **IO** (for Input/Output) functions. Each function has its own docstring, which provides metadata on what the function does, its parameters and what it returns. Per Python conventions, the name of each function is formatted with snake casing (underscores separating each word), and the parameter names do not reference types (in contrast to variables). All functions have the @StaticMethod directive.

### Processor

The first set of functions are grouped into the **Processor** class.

class Processor:

""" Performs Processing tasks """

Listing 2

#### Loading Data

This function reads existing data from a text file and loads it into a table. It has two parameters, the file name and a list of rows (or the ‘table’). It returns the modified table with the loaded data. I added ***try-except*** error handling so that if the file does not exist, it will skip this step and proceed to the rest of the program with an empty table.

def read\_data\_from\_file(file\_name, list\_of\_rows):

try:

list\_of\_rows.clear()

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()

except: pass

return list\_of\_rows, 'Success'

Listing 3

All **Processor** functions return the string “Success.” The template makes it easy to print this message as confirmation that a function has run successfully. However, I ended up using more specific messages for each ***if*** statement. I retained the “Success” string in my final script as it could be used with future updates.

#### Adding New Data

The next function adds new tasks to the to-do list. It has three parameters: the new task, its priority, and the table. It returns the modified table after new data has been appended to it as a dictionary “row.”

def add\_data\_to\_list(task, priority, list\_of\_rows):

dicRow = {"Task": task, "Priority": priority}

list\_of\_rows.append(dicRow)

return list\_of\_rows, 'Success'

Listing 4

#### Removing Data

For this section, I used the Boolean value *removed*. At the beginning of the function, *removed* is assigned *False* and only changed to *True* if an input matches a key in the lstTable rows. I also used kept\_items to create a new table for the tasks the user does *not* want to remove. Because of the properties of **for loops,** this methodmakes it easy to remove more than one row with the same task name. Both the *removed* value and the kept\_items are returned by the function.

def remove\_data\_from\_list(task, list\_of\_rows):

removed = False

kept\_items = []

for row in list\_of\_rows:

if row["Task"].lower() == task.lower():

removed = True

else:

kept\_Items.append(row)

return kept\_items, removed, 'Success'

Listing 5

#### Saving Data

The last **Proccessor** function writes data to a text file. It takes the file name and table as parameters and uses the ***for loop*** to write each row to the file. It returns the table (which should actually be unmodified as its data was only copied to a new file).

def write\_data\_to\_file(file\_name, list\_of\_rows):

objFile = open(file\_name, "w")

for row in list\_of\_rows:

objFile.write(row["Task"].capitalize() + ", " + row["Priority"].upper()+"\n")

objFile.close()

return list\_of\_rows, 'Success'

Listing 6

### IO

The next set of functions is part of the **IO** class.

class IO:

""" Performs Input and Output tasks """

Listing 7

Listing 7

These functions perform input/output tasks, reducing redundancy in the presentation layer.

#### Print Options Menu

The first **IO** function prints the options menu. This function was included with the template and is fairly straightforward. It does not return any values as none of the data encapsulated within the function needs to be accessed globally.

def print\_menu\_Tasks():

print('''

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

''')

print() # Add an extra line for looks

Listing 8

#### Input Menu Choice

The next function uses an input statement to elicit a menu option from the user and returns

their choice as a string. In this case, returning the *choice* value is critical to the rest of the program, because it will determine the flow of the presentation layer.

def input\_menu\_choice():

choice = str(input("Which option would you like to perform? [1 to 5] - ")).strip()

print() # Add an extra line for looks

return choice

Listing 9

#### Print Current Data

This function displays the current list to users by using a ***for loop*** to iterate over the table and print each dictionary stored in lstTable. I modified this function to match the list format I had in Assignment 5. Similar to the menu function, there are no returned parameters.

def print\_current\_Tasks\_in\_list(list\_of\_rows):

print("\*\*\*\*\*\*\* The current Tasks ToDo are: \*\*\*\*\*\*\*")

count = 0

for row in list\_of\_rows:

count += 1

print(str(count) + ". " + row["Task"].capitalize() + " | " + row["Priority"].upper())

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print()

Listing 10

#### Input Yes/No/Enter

The *input\_yes\_no\_chose* function takes a “message” as a parameter and returns the string in all lowercase characters without extra spacing. This allows the program to handle variation in inputs for the yes/no prompt without having to write out this statement separately each time.

def input\_yes\_no\_choice(message):

return str(input(message)).strip().lower()

def input\_press\_to\_continue(optional\_message=''):

print(optional\_message)

input('Press the [Enter] key to continue.')

Listing 11

The *input\_press\_to\_continue* function simply prints an empty line followed by a prompt telling users to press *enter* to continue. This enhances the user experience by providing a small break between tasks. The message parameter has a default empty string value. This can be replaced by entering a string as an argument when the function is called.

#### Input Task to Remove

The last custom function prompts the user to provide a task to be removed. The function takes no parameters but returns the Task string.

def input\_task\_to\_remove():

Task = str(input("Which item would you like to remove?\n "))

return Task

Listing 12

## Presentation

The Presentation layer is the main body of the script. This section combines custom functions with argument inputs and provides a logical flow for the program’s user experience.

The first line of this section calls the *read\_data\_from\_file* function, which takes the file name and lstTable as arguments. The arguments used with custom functions in this section were declared in the Data section.

Processor.read\_data\_from\_file(strFileName, lstTable)

while (True):

IO.print\_current\_Tasks\_in\_list(lstTable)

IO.print\_menu\_Tasks()

strChoice = IO.input\_menu\_choice()

Listing 13

Next, an ***infinite while loop*** is initiated to ensure the script continuously prompts users for menu choices until they choose to exit. Below the ***while loop,*** three custom functions are utilized, all part of the **IO** class. The first function takes the lstTable as an argument and prints the current data in lstTable. The second prints the menu options, and the last one prompts users for a menu choice, assigning the returned value to the variable strChoice.

#### Option 1 – Add Task

There are three custom functions nested within option 1. The first function, *input\_new\_task\_and\_priority*, prompts the user for a new task and its priority. The returned values are assigned to variables, which are subsequently used as arguments in the function *add\_data\_to\_list*. Note that multiple returned values are returned as a tuple. When the tuple is assigned to multiple variables (as it is here), it is an example of *tuple-unpacking*.

Finally, the **IO** function *input\_press\_to\_continue* prints a message and prompts to hit *enter* to continue. This function is similarly called after each menu option is completed.

if strChoice.strip() == '1':

strTask, strPriority = IO.input\_new\_task\_and\_priority()

lstTable, strStatus = Processor.add\_data\_to\_list(strTask, strPriority, lstTable)

IO.input\_press\_to\_continue("{} saved to your list!".format(strTask).capitalize())

continue

Listing 14

#### Option 2 – Remove Task

Option 2 is slightly more complex. The **IO** function prompts users for the task they want to remove. The processing function then returns the new lstTable and the Boolean Value ‘boolRemoved’. If the Boolean value is False, the program prints a message that the item was not found. If the value is true, it confirms the item has been removed. If the Boolean value was not returned by the function, it would always evaluate to False (based on how it was declared as a global variable), and the user would not get confirmation their task had been removed (even when it had been).

elif strChoice == '2':

strTask = IO.input\_task\_to\_remove()

lstTable, boolRemoved, strStatus = Processor.remove\_data\_from\_list(strTask, lstTable)

if boolRemoved == False:

print("{} is not on your your list.".format(strTask).capitalize())

else:

print("{} has been removed from your list!".format(strTask).capitalize())

IO.input\_press\_to\_continue("")

continue

Listing 15

#### Option 3 – Save Data

Option 3 allows users to save the current list to a file. Before saving they are prompted with *input\_yes\_no\_choice* to confirm they want to write new data to the file. If they enter ‘y’ for yes, the function *write\_data\_to\_file* takes the file name and lstTable as arguments and writes the data to the file. If users enter ‘n’ for no, they are returned to the options menu.

elif strChoice == '3':

strChoice = IO.input\_yes\_no\_choice("Save this data to file? (y/n) - ")

if strChoice.lower() == "y":

lstTable, strStatus = Processor.write\_data\_to\_file(strFileName, lstTable)

IO.input\_press\_to\_continue("Your data has been saved to 'ToDoList.txt")

else:

IO.input\_press\_to\_continue("Save Cancelled!")

continue

Listing 16

#### Option 4 – Reload Data

Users can select option 4 to reload data from the file. This option was not available in the previous version of the script. Like in option 3, they have to confirm their choice before the program completes the task. The function *read\_data\_from\_file* is then called to read data from the text file. The returned table is saved to lstTable, overwriting any changes that had been made since the file was last loaded from the text file.

elif strChoice == '4':

print("Warning: Unsaved Data Will Be Lost!")

strChoice = IO.input\_yes\_no\_choice("Are you sure you want to reload data from file? (y/n) - ")

if strChoice.lower() == 'y':

lstTable, strStatus = Processor.read\_data\_from\_file(strFileName, lstTable)

IO.input\_press\_to\_continue("Your list has been reloaded from ToDoList.txt!")

else:

IO.input\_press\_to\_continue("File Reload Cancelled!")

continue

Listing 17

#### Option 5 – Exit Program

The last option allows users to exit if they enter 5. The program prints a goodbye message and breaks out of the loop to close the program. I also added a final ***else*** clause to the program that handles inputs that are not valid menu options.

elif strChoice == '5':

print("Goodbye!")

break

else:

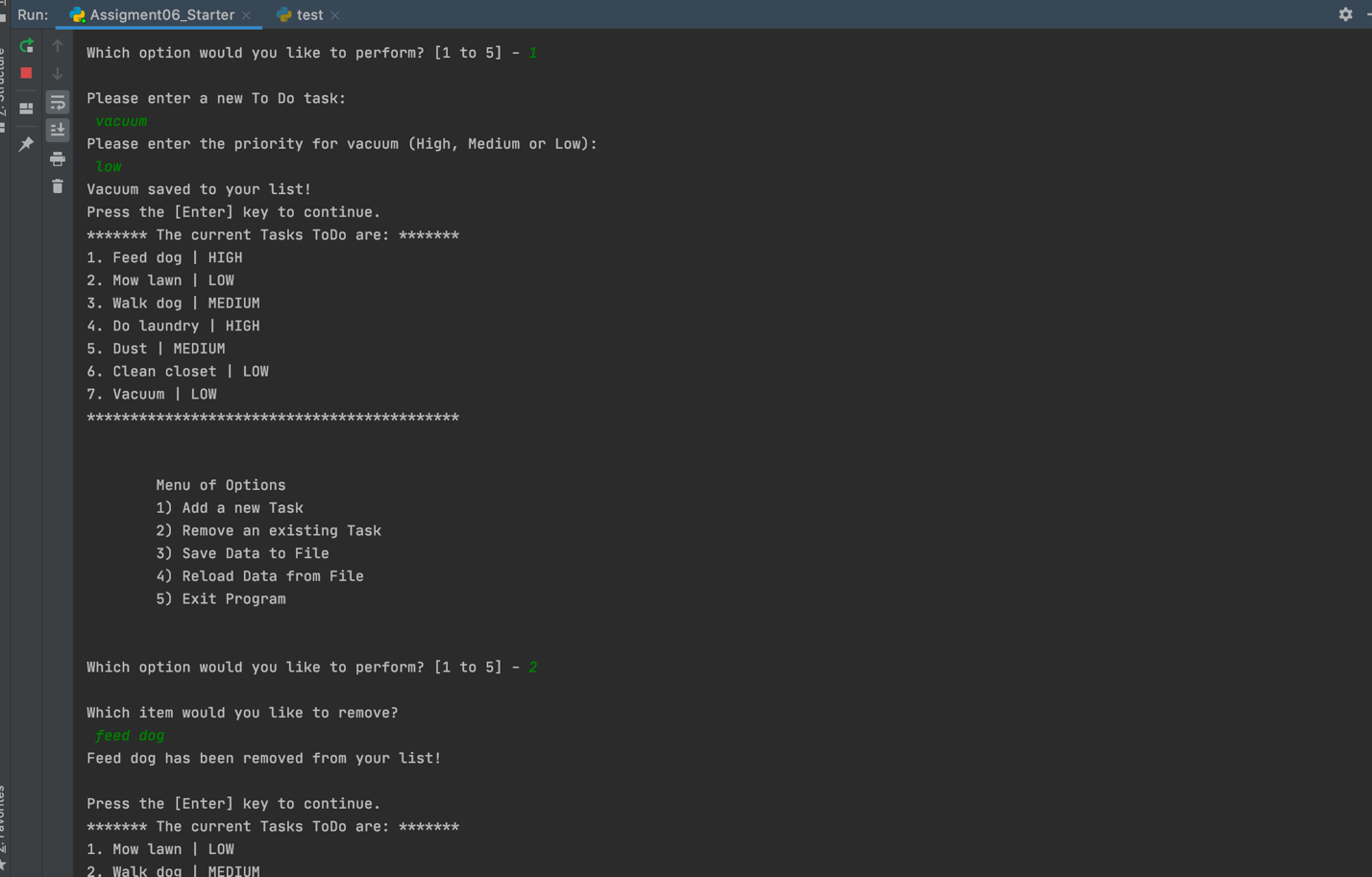
IO.input\_press\_to\_continue("Please select an option from the menu.")

continue

Listing 18

# Testing in PyCharm

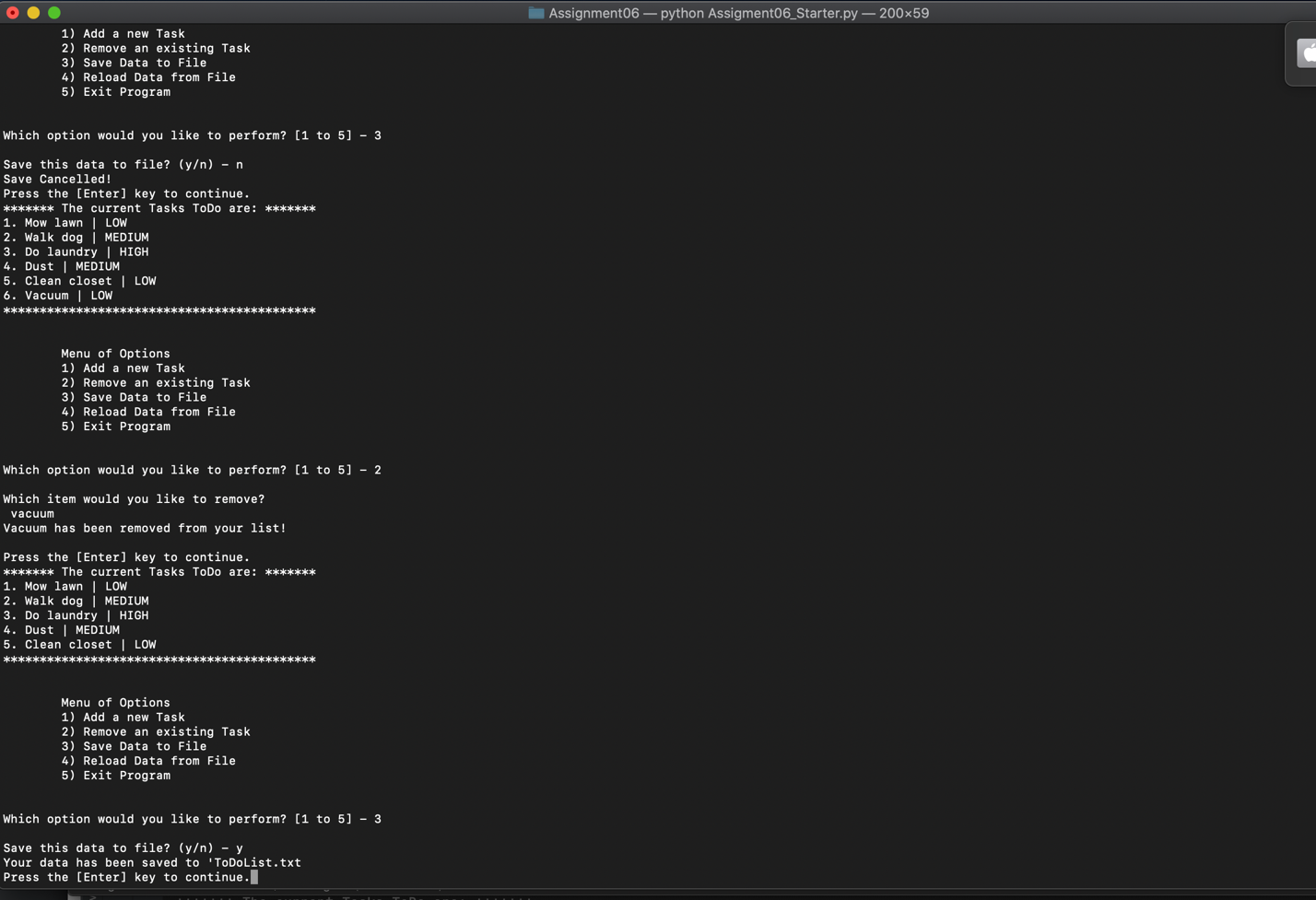
Once the script was complete, I tested the program in PyCharm (Figure 4). Each step worked as expected.



**Figure 4.** Testing Assignment06\_Starter.py in PyCharm

# Running in Terminal

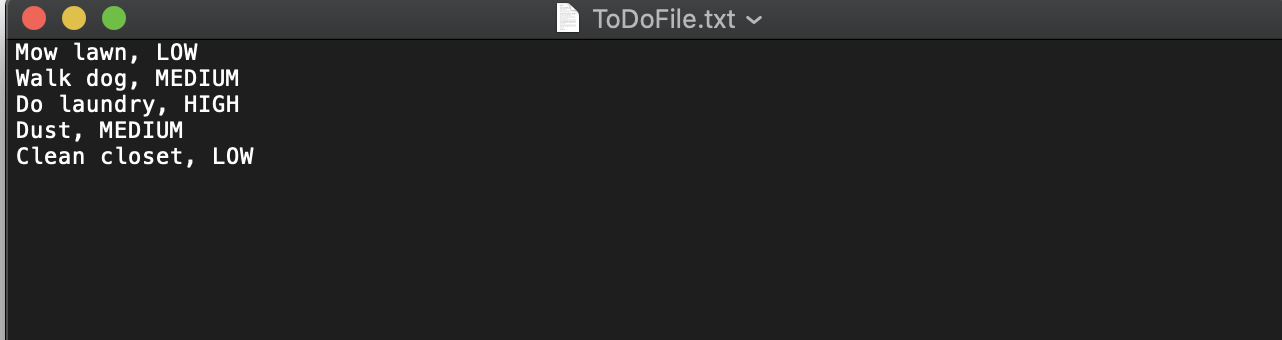
Next, I ran the program as a Console application in Terminal. I first navigated to the relevant directory with the command “cd Documents/\_PythonClass/Assignment06” and launched the program with the command “python Assigment06\_Starter.py”. Once the program opened, each step worked as expected (Figure 5).



**Figure 5.** Running Assignment06\_Starter.py in Terminal

# To-Do List Text File

As the final step in this assignment, I checked ToDoFile.txt in the Assignment06 directory. The most recent data had successfully been saved as a list to the text file.



**Figure 6.** Confirming data saved in ToDoFile.txt

# Summary

This paper details the process of updating the to-do list program to incorporate custom functions. In the previous version of the program, there was little separation of processing and presentation aspects of the code. The script now has distinct sections for the different layers, making it easier to read and update without as much redundancy in the presentation layer. While updating the script, I became more comfortable with PyCharm features and gained an understanding of how functions work in Python.