

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).

```
    return list_of_rows, 'Success'

    @staticmethod
    def add_data_to_list(task, priority, list_of_rows):
        # TODO: Add Code Here!
        return list_of_rows, 'Success'

    @staticmethod
    def remove_data_from_list(task, list_of_rows):
        # TODO: Add Code Here!
        return list_of_rows, 'Success'

    @staticmethod
    def write_data_to_file(file_name, list_of_rows):
        # TODO: Add Code Here!
        return list_of_rows, 'Success'
```

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).

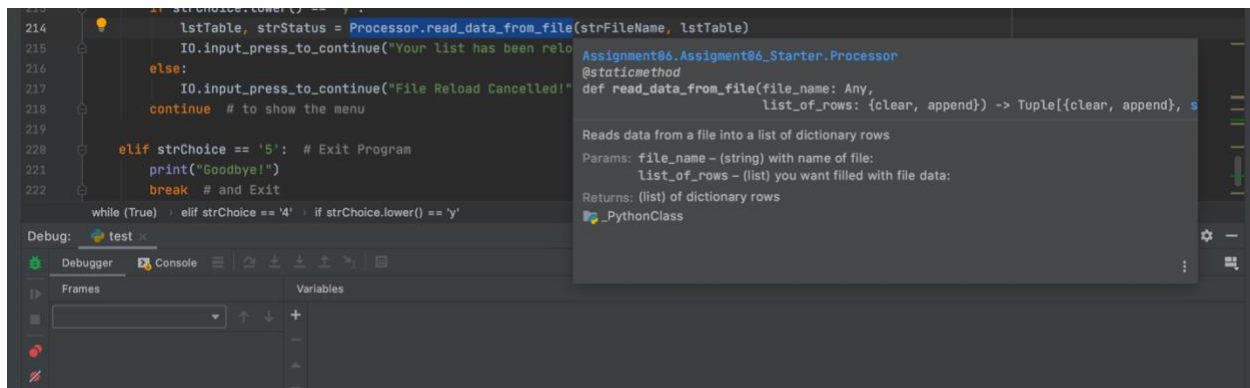


Figure 2. Using shortcuts (F1 on MacOS) to view docstring in PyCharm

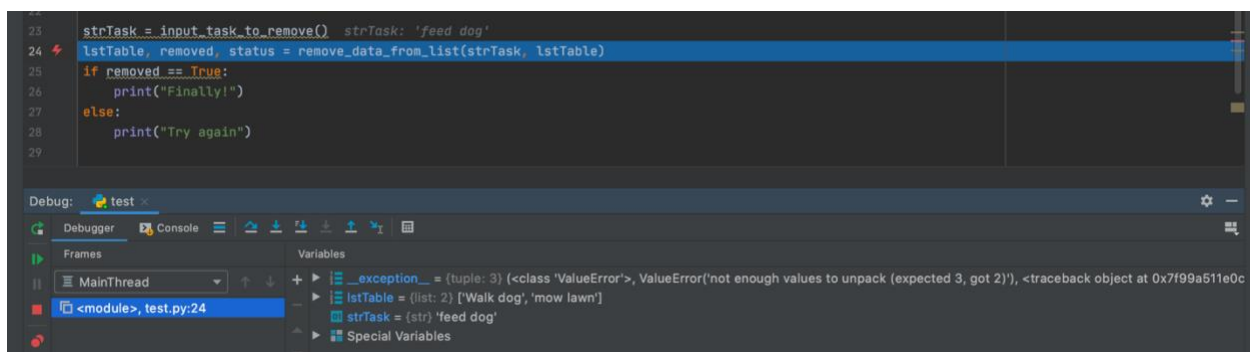


Figure 3. Using Debugger in PyCharm

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 method makes 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 **Processor** 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

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

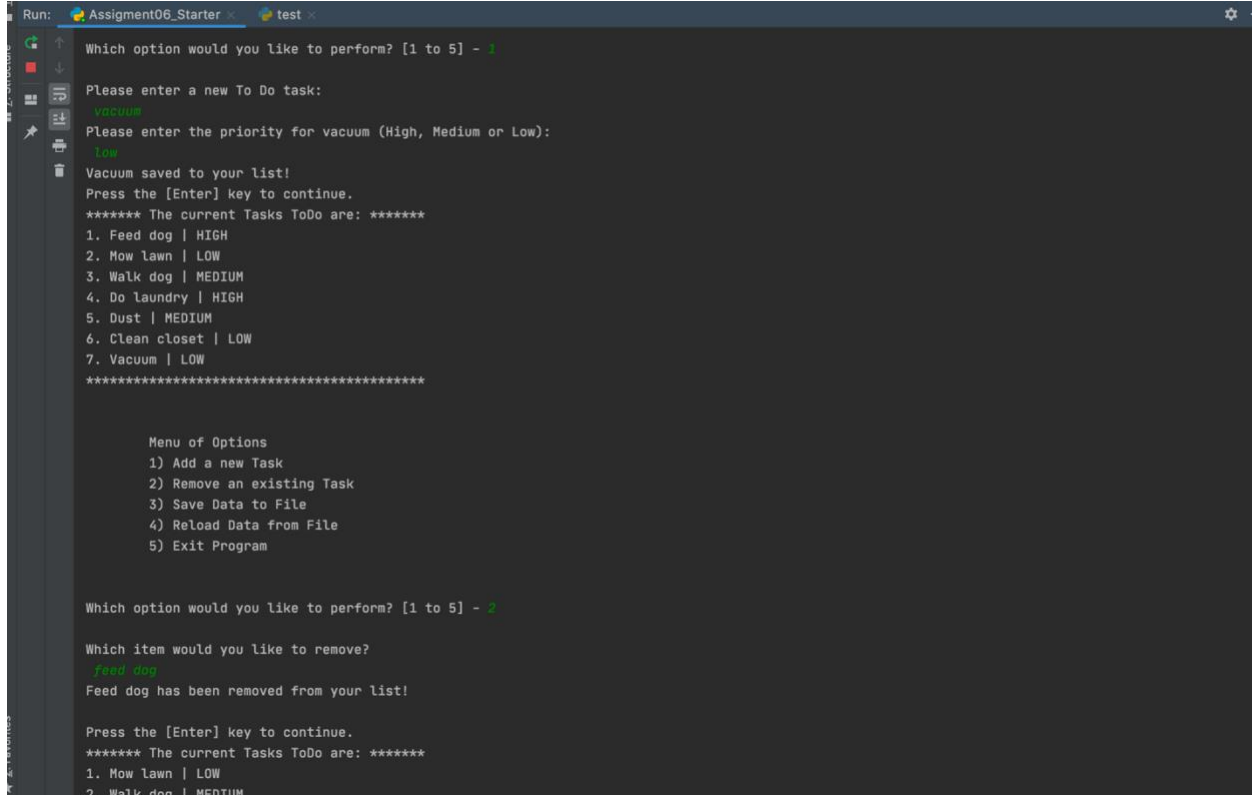
The last option allows users to exit if they enter 5. The program prints a goodbye message and breaks out of the loop to close the program. I also added a final **else** clause to the program that handles inputs that are not valid menu options.

```
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.



```
Run: Assignment06_Starter.py test
Which option would you like to perform? [1 to 5] - 1
Please enter a new To Do task:
vacuum
Please enter the priority for vacuum (High, Medium or Low):
Low
Vacuum saved to your list!
Press the [Enter] key to continue.
***** The current Tasks ToDo are: *****
1. Feed dog | HIGH
2. Mow lawn | LOW
3. Walk dog | MEDIUM
4. Do laundry | HIGH
5. Dust | MEDIUM
6. Clean closet | LOW
7. Vacuum | LOW
*****

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

Which option would you like to perform? [1 to 5] - 2
Which item would you like to remove?
feed dog
Feed dog has been removed from your list!
Press the [Enter] key to continue.
***** The current Tasks ToDo are: *****
1. Mow lawn | LOW
2. Walk dog | MEDIUM
```

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 Assignment06_Starter.py”. Once the program opened, each step worked as expected (Figure 5).

```
Assignment06 — python Assignment06_Starter.py — 200x59

1) Add a new Task
2) Remove an existing Task
3) Save Data to File
4) Reload Data from File
5) Exit Program

Which option would you like to perform? [1 to 5] - 3

Save this data to file? (y/n) - n
Save Cancelled!
Press the [Enter] key to continue.
***** The current Tasks ToDo are: *****
1. Mow lawn | LOW
2. Walk dog | MEDIUM
3. Do laundry | HIGH
4. Dust | MEDIUM
5. Clean closet | LOW
6. Vacuum | LOW
*****

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

Which option would you like to perform? [1 to 5] - 2

Which item would you like to remove?
vacuum
Vacuum has been removed from your list!

Press the [Enter] key to continue.
***** The current Tasks ToDo are: *****
1. Mow lawn | LOW
2. Walk dog | MEDIUM
3. Do laundry | HIGH
4. Dust | MEDIUM
5. Clean closet | LOW
*****

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

Which option would you like to perform? [1 to 5] - 3

Save this data to file? (y/n) - y
Your data has been saved to 'ToDoList.txt'
Press the [Enter] key to continue.
```

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.

```
ToDoFile.txt v

Mow lawn, LOW
Walk dog, MEDIUM
Do laundry, HIGH
Dust, MEDIUM
Clean closet, LOW
```

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.