

Working with Functions

Introduction

In this week's lectures, modules and readings, we learned about classes, functions, parameters, arguments, global and local variables, doc strings, PyCharm debugging mode and Github pages. The application assignment in front of us was to take a starter template and with combination of code from previous assignment, convert the starter code into a fully working app that utilizes functions and better segregates processing vs input/output code.

Starter Template / Approach

As I did last week, I started by first reviewing the starter code to understand its structure. I did this to understand the intended behavior and the relationships between the main code and the functions that it would be referencing.

I immediately saw that the code would be broken down into two types of functions (organized in classes) that would either perform processing / calculation or interact with the user via soliciting input or presenting output.

With the number of functions, this immediately became overwhelming. I decided that rather than work within the starter template that building from scratch one piece of functionality at a time would allow me to better focus my efforts and allow for easier end to end testing.

Skeleton Code / Reading Data from File

I started the new Python document by copying the barebones elements. This included the header string (added my entry to the change log), the Data section (that initialized the global variables), the names of the two classes (Processor and IO, did not copy any functions yet), followed by one line of main code (that would read data from the file and store it in a final list).

This one line of main code is shown in **Figure 1** below.

```
Processor.read_data_from_file(file_name=file_name_str, list_of_rows=table_lst)
```

Figure 1 – Reading data from a file (called function)

This line utilizes the *read_data_from_file* function (within the Processor class, passing in the list and filename as arguments) so I copied in the function from the starter code into my code. The function code is shown in **Figure 2** below.

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

    @staticmethod
    def read_data_from_file(file_name, list_of_rows):
        """Reads data from a file into a list of dictionary rows..."""
        list_of_rows.clear() # clear current data
        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()
        return list_of_rows
```

Figure 2 – Reading data from a file (code within the called function)

This code would open the *ToDoFile.txt* (I had to initially create this file with sample data to have something to load), iterate through each line to create a dictionary row, append each row to a master list and return the final list.

Showing Current Data

With the data loaded in the list object, next step was to show the contents of the list / table to the user. This was done by adding the following code from the starter template, as shown in **Figure 3** below.

```
IO.output_current_tasks_in_list(list_of_rows=table_lst)
```

Figure 3 – Presenting List contents to the user (called function)

This line utilizes the *output_current_tasks_in_list* function (within the IO class, passing in the list as argument) so I copied in this function into my code.

```
@staticmethod
def output_current_tasks_in_list(list_of_rows):
    """ Shows the current Tasks in the list of dictionaries rows

    :param list_of_rows: (list) of rows you want to display
    :return: nothing
    """

    print() # Add an extra line for looks
    print("*****\u001b[1m Current ToDo List\u001b[0m *****")
    print("Task " + "|" + " Priority")
    for row in list_of_rows:
        print(row["Task"] + " (" + row["Priority"] + ")")
    print("*****")
```

Figure 4 – Presenting List contents to the user (code within the called function)

This code would iterate through the list and display the contents (Tasks and Priorities) to the console as shown in **Figure 5** example below.

```
***** Current ToDo List *****
Task | Priority
Task1 (High)
Task2 (Medium)
Task3 (Low)
Task4 (Medium)
Task5 (High)
Task6 (Low)
Task7 (Medium)
*****
```

Figure 5 - Presenting List contents to the user (UI) (Command Prompt)

Showing Menu / Getting Menu Choice

With the ability to load data from the file and display it, the next step was to add the capability that would present the menu to the user and ask the user to make a choice from a list of menu options. This was done by the two functions demonstrated in **Figure 6** below.

```
I0.output_menu_tasks() # Shows menu
choice_str = I0.input_menu_choice()
```

Figure 6 – Presenting menu options & Storing user choice (called function)

The output function will generate the “Menu of Options” text while the input function will generate the “Which Option” text and store the choice (this will then be used to code various behaviors), as shown in **Figure 7** below.

```
Please select from Menu of Options
0) Refresh Data from File
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] - x
```

Figure 7 – Presenting menu options & Storing user choice - UI – PyCharm

This code presenting the menu and storing user’s latest choice will execute every time after the previous user action has completed.

Adding a New Task

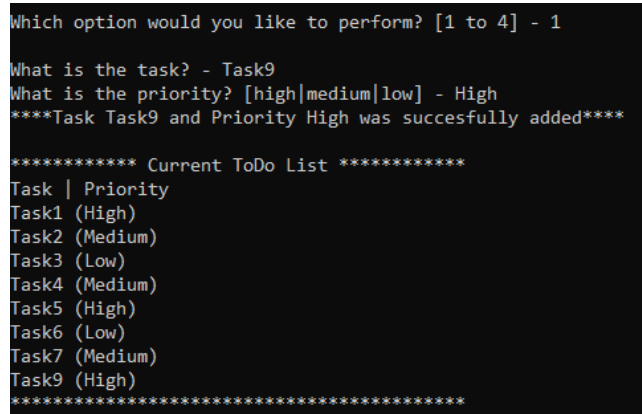
With this basic functionality validated in the Command Prompt and PyCharm (heavily used the debug mode) to validate that the choice was indeed stored, the next step was to handle addition of new task (option 1).

When the user entered “1” as the option (to make it more user friendly, added “add” and “a” as viable options to select this option), the code in **Figure 8** below would execute.

```
elif choice_str.strip() == '1' or choice_str.strip().lower() == 'add' or choice_str.strip().lower() == 'a':
    task, priority = IO.input_new_task_and_priority()
    if (Processor.validate_task_to_add_not_existing(task_to_add=task, list_of_rows=table_lst)):
        IO.output_failed_task_add()
        continue
    else:
        table_lst = Processor.add_data_to_list(task_to_add=task, priority=priority, list_of_rows=table_lst)
        IO.output_successful_task_add()
        continue # to show the menu
```

Figure 8 – Adding New Task (called functions)

This code would provide the user with ability to enter the *Task* and *Priority* to add, add it to the list, provide confirmation message and show latest contents of the list. The happy path is shown in **Figure 9** below. The unhappy path was optional but was coded to reject the add if the Task already existed in the list (to prevent duplicate tasks). In attempt to truly separate processing and IO, I removed all the print statements within the main code and put them into extra output functions.

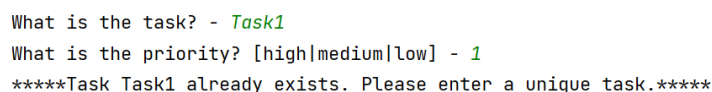


```
Which option would you like to perform? [1 to 4] - 1
What is the task? - Task9
What is the priority? [high|medium|low] - High
****Task Task9 and Priority High was succesfully added****

***** Current ToDo List *****
Task | Priority
Task1 (High)
Task2 (Medium)
Task3 (Low)
Task4 (Medium)
Task5 (High)
Task6 (Low)
Task7 (Medium)
Task9 (High)
*****
```

Figure 9 – Adding New Task (UI) – Happy Path – Command Prompt

Unhappy path (as tested in PyCharm) is shown in **Figure 10** below.



```
What is the task? - Task1
What is the priority? [high|medium|low] - 1
*****Task Task1 already exists. Please enter a unique task.*****
```

Figure 10 – Adding New Task (UI) – Unhappy Path – PyCharm

Removing a Task

The next step was to add the ability to remove a task. The code in **Figure 11** below provided this capability.

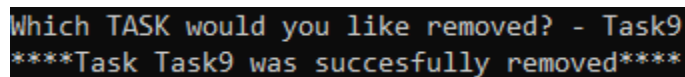
```
# Remove an existing Task
elif choice_str.strip() == '2' or choice_str.strip().lower() == 'remove' or choice_str.strip().lower() == 'rem':
    task = IO.input_task_to_remove()
    table_lst, status = Processor.remove_data_from_list(task_to_remove=task, list_of_rows=table_lst)
    if(status == True):
        IO.output_successful_task_removal()
    else:
        IO.output_failed_task_removal()
    continue # to show the menu
```

Figure 11 – Removing a Task (called functions)

This code is executed when the user enters “2” , “remove” or “rem” as the option. It first captures the user input to know which task the user wants to remove. It then runs the *remove_data_from_list* function which attempts the removal and sends back the new list and status code.

Removal is successful if the Task was in the list and failed if it wasn't. Both output scenarios are coded in separate output functions.

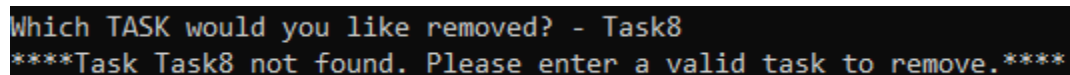
Outputs of successful removal is capture in **Figure 12** below.



```
Which TASK would you like removed? - Task9
****Task Task9 was succesfully removed****
```

Figure 12 – Removing a Task (UI) – Success – Command Prompt

Outputs of failed removal is capture in **Figure 13** below.



```
Which TASK would you like removed? - Task8
****Task Task8 not found. Please enter a valid task to remove.****
```

Figure 13 – Removing a Task (UI) – Failure – Command Prompt

Saving Data To File

The next step was to add the ability to save data back to the file. The code in **Figure 14** below provided this capability.

```
# Save Data to File
elif choice_str.strip() == '3' or choice_str.strip().lower() == 'save' or choice_str.strip().lower() == 's':
    if(IO.input_save_confirmation_choice() == "y" ):
        table_lst = Processor.write_data_to_file(file_name=file_name_str, list_of_rows=table_lst)
        IO.output_successful_data_save()
    else:
        IO.output_rejected_data_save()
    continue # to show the menu
```

Figure 14 – Saving the Data To File (called functions)

Like the above capabilities, specific input (in this case input of “3”, “save” or “s”) would force execution of this code block.

The code would first ask user to confirm that they indeed wanted to save the data (confirmation requiring a “y” option).

If confirmed, it would call the *write_data_to_file* function and output a successful “Data Saved” message as shown in **Figure 15** below. I validated the *ToDoFile.txt* after this behavior and indeed contained the correct tasks and priorities from the latest save.

```
Save this data to file? (y/n) - y

****Data Saved to File****

***** Current ToDo List *****
Task | Priority
Task1 (High)
Task2 (Medium)
*****
```

Figure 15 – Saving Data to File (UI) – Confirmed – PyCharm

I validated the *ToDoFile.txt* after this behavior and indeed contained the correct tasks and priorities from the latest save.

If cancelled, it would output a “Data Not Saved” message, as shown in **Figure 16** below. Both of these output scenarios to the user are coded in their respective output functions.

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

Save this data to file? (y/n) - n

****Data NOT saved, but previous data still exists. Press the [Enter] key to return to menu. ****
```

Figure 16 – Saving Data to File (UI) – Cancelled – PyCharm

Exiting the Application

The last step was to add the capability to exit the application. This was very simple and simply included an output function that said “Goodbye” to the user and stopped the script by breaking out of the while loop as shown in **Figure 17** below.

```
elif choice_str.strip() == '4' or choice_str.strip().lower() == 'exit' or choice_str.strip().lower() == 'x':
    IO.output_successful_program_exit()
    break # by exiting loop
```

Figure 17 – Exiting the application (called functions)

Enhancing the User Experience / Application

As already shown, I had already enhanced the experience by doing the following:

- Allowing multiple ways of calling each option, depending on user preference
- Adding user validation to ensure duplicate tasks are not added
- Adding clarity to the user that a task that wasn't in the list couldn't be removed.
- Giving user the ability to cancel the save
- At every point, try to provide a status update and the latest contents of the list

I also decided to enhance the formatting by using a combination of bold and underline to make headers and user behaviors stand out more, as shown in **Figure 18** below.

```
***** Current ToDo List *****  
Task | Priority  
Task1 (High)  
Task2 (Medium)  
*****
```

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

Figure 18 – Using bold and underlines to enhance the UI - PyCharm

Summary

This week, we built off our existing knowledge of working with lists, dictionaries and files by introducing classes, functions, parameters, arguments, global and local variables, doc strings, PyCharm debugging mode and Github pages.

A function's ability to abstract the coded logic out of the main script simplified the main code and allowed better separation of the processing code versus the code that would interact with the user. The classes allowed a way to group related functions (as well as the ability to hide that group of functions when not in use).

Arguments (usually utilizing global variables) were regularly passed into parameters of the function, with local variables being used inside the function. For every function, I documented the doc string to make it clear the purpose of the function, the parameters it took and a return value (if any).

PyCharm debugging mode was especially helpful to step through the code whenever I encountered issue or when I needed to validate contents of a variable where I hadn't yet coded a function to display it to the console.

Boris Unigovskiy

August 17, 2022

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

IT FDN 100 B

Assignment 6

This week we started by creating a GitHub page on our repository – looking forward to exploring how they can be used in the upcoming weeks as well as learning new concepts that will allow me to build more complex and useful applications.