Kelsey Kawaguchi

Nov. 18, 2019

IT FDN 100

Assignment 06

Modifying To Do File Code

Introduction

This paper explains the logic on how I modified the To Do File Script, which allows the user to compile a “To Do” list based on task to be achieved and identify whether the task is a “high” or “low” priority. In the code, the user has the ability to add, delete, save, and print the list to an external text file. The goal and purpose of the assignment was to modify the code to utilize functions, which involved using existing code to create new functions, then reference those functions in the “main script” section. By doing this, the user has the ability to create one function and reference the function throughout the script as many times as needed.

Methodology

Figure 1 contains all of the declared data variables that are used in the To Do File code.

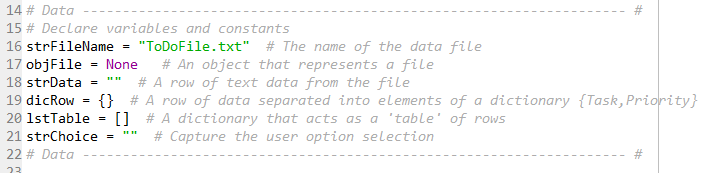


Figure 1 – Data Variables

In Figure 2, the first section in the program is “Processing”, which includes the DataProcessor class. The first function created is to read the textfile dictionary list. As identified below, a couple of the function parameters include the name of the file and the data file list of rows. The function will return a list of rows.

In the first function as shown in Figure 2, the only parameter is the file\_name. In the code, the list\_of\_dictionary\_rows is called out to be a list. Next, the file is opened as a “read” only file. Afterwards, for each line in the file, the data is split, based on the “,”. Next, a row in the dictionary associates “Task” with the first data string, then “Priority” with the second piece of data. Lastly, the list\_of\_dictionary\_rows adds the new data provided in the row to the file, then the file is closed, and the function returns the “appended” list\_of\_dictionary\_rows.

The write\_file\_from\_list\_of\_dictionaries requires the file\_name and list\_of\_rows as parameters. The purpose of this function is to write and save the new dictionary rows into the text file. At the end the function, if everything is successful, will return the list of tasks that were added to the textfile. The first line of the function success\_status = False identifies the absence of the success\_status. If the success\_status is true, then the file will open, write a new task and priority to the row in the lstTable, then close the file. Lastly, the return would be the result of the success\_status, which would be the appended textfile with new rows.

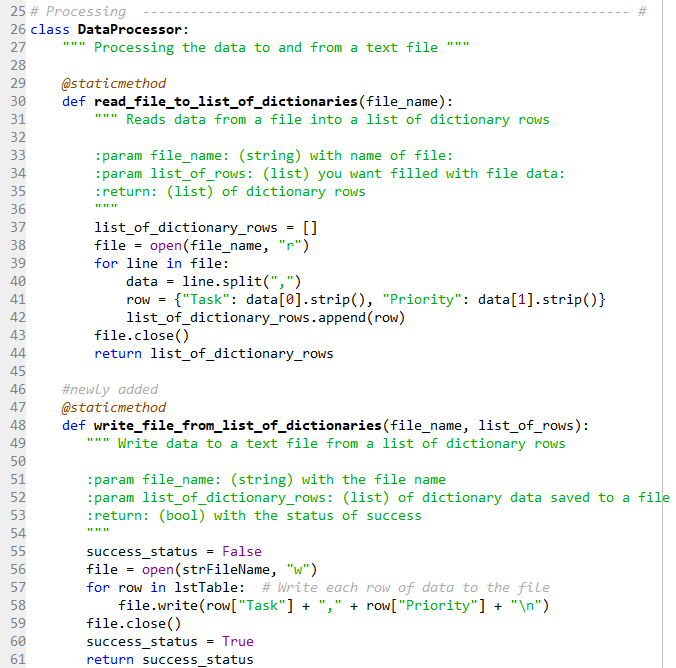


Figure 2 – Processing Functions Pt 1

In addition to the two functions as shown in Figure 2, two more functions were created as shown in Figure 3. The add\_data\_to\_list\_of\_dictionaries function involve the parameters of list\_of\_dictionary\_row, task, and priority. The function adds the task and priority strings and appends the row to the current list\_of\_dictionary\_rows.

The second function in Figure 3 is the remove\_data\_From\_list\_of\_dictionaries, which include the: list\_of\_dictionary\_rows, and task\_to\_remove. The return will be the Boolean that executes the successful removal of the row. The first line states a Boolean operator to identify that the code is “off”. Next, the row number is initially stated to be 0, which will be changed as the user inputs a row\_number to remove. Next, there is a while loop to initiate the loop if the row\_number is less than the number of rows in the lstTable. Next, the if statement states that if the task\_to\_remove is equal to the row\_number in the lstTable, then to delete the row\_number in the lstTable. This statement will be turned on. For if statement to be true, the row number must have a count greater than or equal to 1. The return is the successful deletion of the row.

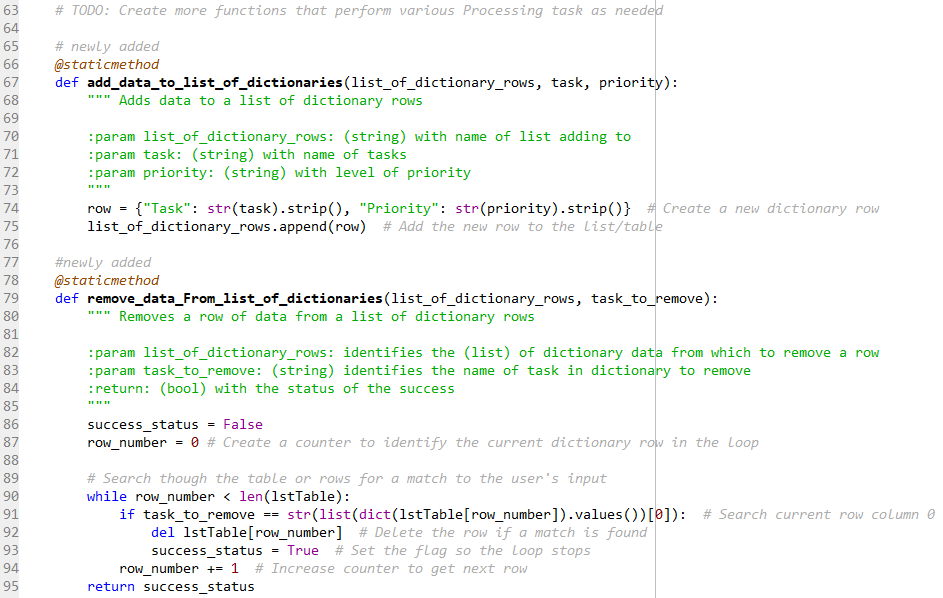


Figure 3 – Processing Functions Part 2

Figure 4 shows the Presentation (input/output) section of the code. The class is stated as “I/O”.

The first function created is to display the menu items. When OutputMenuItems is called, the menu options are printed.

The second function named InputMenuChoice allows the user to input which option they would like the program to perform and return the user input.

The third function named print\_current\_list\_items prints the current tasks and priority back to the user.

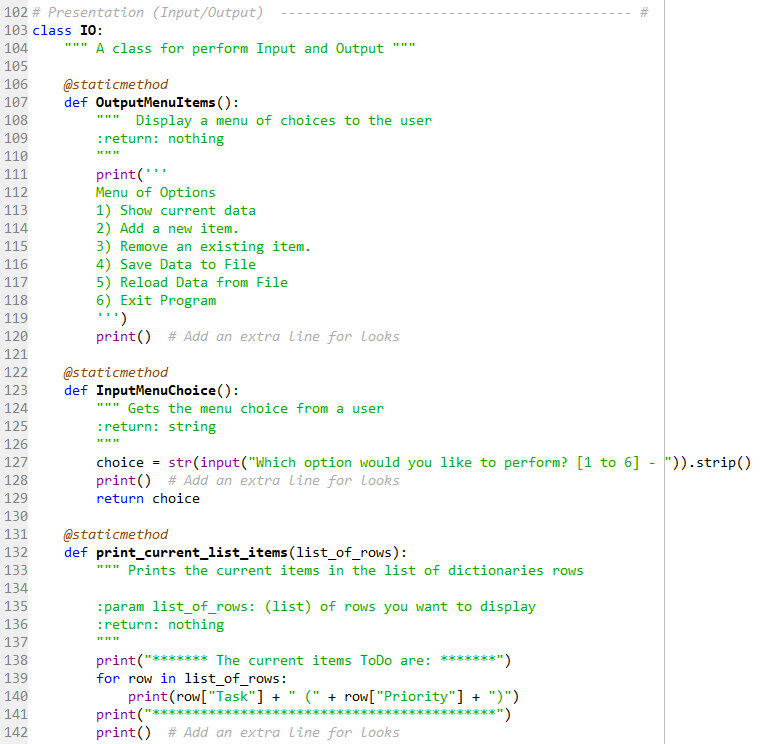


Figure 4 – Presentation I/O Part 1

Figure 5 below shows other IO tasks created under the I/O class.

The first function, input\_task\_and\_priority in Figure 5 created allows the user to input a task and priority. The function will then return the task and priority entered by the user.

The second function update\_status\_of\_removal updates the user if the blnItemRemoved is true, then “the task was removed” will print. If the blnItemRemoved did not occur, then “I’m sorry, but I could not find that task” would print.

Lastly, print\_user\_file\_status is a function that prompts the user to save or not save their data. If “y” is input by the user, then DataProcessor.write\_file\_from\_list\_of\_dictionaries will become active and save the new lstTable into the strFileName file. If the data was not saved, then the user is prompted with a message that the data was not saved.

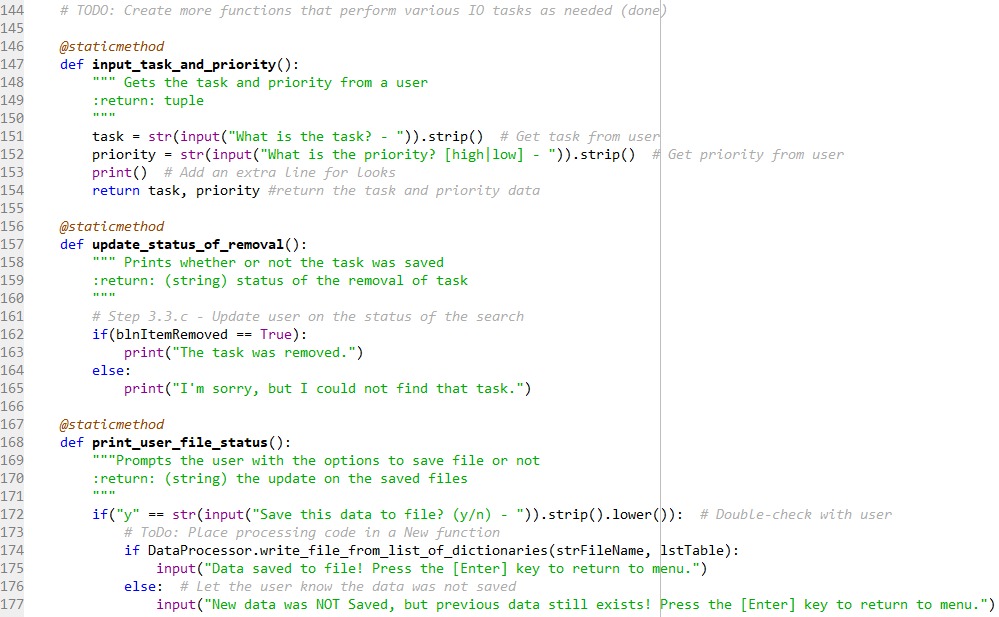


Figure 5 – Presentation I/O Part 2

The last section of the code contains the Main Body of the Script. The first line of the script identifies the action that will occur for the variable of lstTable, which will be for the DataProcessor.read\_file\_to\_list\_of\_dictionaries function to read the list of data in the To Do file.

Step 2 as shown in Figure 6 starts with a while loop. As long as the while loop is active, the IO.OutputMenuItems function will prompt the user with the printed list of menu options. After this, the user will be able to select a menu choice as the strChoice variable calls on the function IO.InputMenuChoice().

Step 3 involves the code as the user selects the different menu options (1 thru 6).

In step 3.1, if the user selects 1, then the IO.print\_current\_list\_items function will run while using the data in the strFileName variable. After the function runs, then program will loop to the top of the code and display the menu options.

If option 2 is selected, as shown in Figure 6, step 3.2.a, then the IO.input\_task\_and\_priority() function will run and the return data will be entered into the tplData variable. In step 3.2.b, the DataProcessor.add\_data\_to\_list\_of\_dictionaries will take the parameters of the current lstTable, tplData [0] = task, and tplData [1] = priority into the strFileName. At the end, the updated menu (lstTable) will print out to the user.

In step 3.3, if the user enters 3, then the user will be prompted with which task they would like to remove. With the task identified and held in the variable strKeyToRemove, the DataProcessor.remove\_data\_from\_list\_of\_dictionaries function is run with the parameters of the lstTable and strKeyToRemove). The value returned from this program will be input as the blnItemRemoved variable. In 3.3.c, the IO.update\_status\_of\_removal function is run to print to the user if the line was removed or not. Lastly, in step 3.3.d, the IO.print\_current\_list\_items function is run to print the current list to the user. The code is then looped to the top.



Figure 6 – Main Body of Script Part 1

Figure 7 contains the last 3 options that the user can select. In step 3.4, if the user selects option 4, then they have the ability to save their data to the text file. In step 3.4.a, the IO.print\_current\_list\_items in the lstTable are printed to the user so the user will know what data is saved to the file. Next, in step 3.4.b, the user is prompted to save the data and runs the IO.print\_user\_file\_status function to save the data. At the end of step 3.4, the user is then looped to the top of the code.

When the user selects 5, the user clears the current data from the list/table, and uploads the text file data. In step 3.5, the user is prompted that all of the data may be lost if the user wants to upload the text file. If the user selects y, then the table will be wiped and replaced with the text file. If the data was not uploaded, the user is made aware and the current list of items in the table is printed. Lastly, the code loops to the top.

Step 3.6 includes the break of the code, where the user would want to end the program and insert 6.

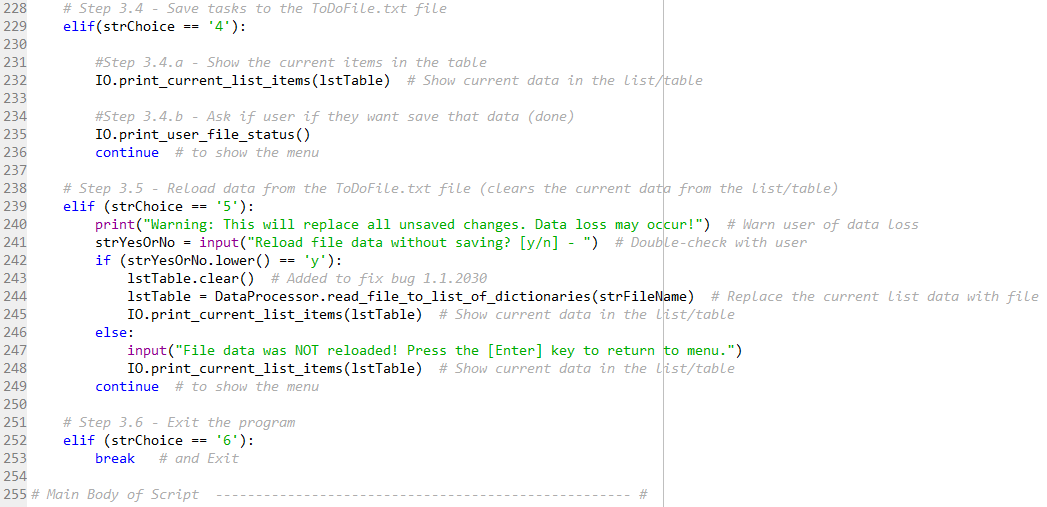


Figure 7 – Main Body of Script Part 2

Command Results

In the command window shown in Figure 8 below, once the user runs the program, they are prompted to select an option. If a text file is not already created, the user will need to first input some data into a text file, and then can select option 2 as shown below. As expected, when option 1 is selected, the To Do items are printed.

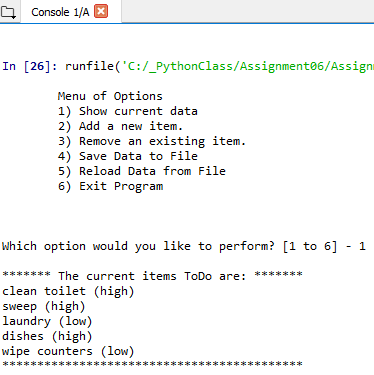


Figure 8 – Spyder Python Console Part 1

Figure 9 then shows program response when 2 is entered and the user wants to input a new task and priority. After entering the task and priority, the To Do list is then updated with the new data.

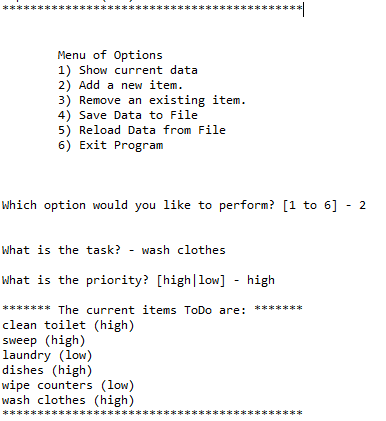


Figure 9 – Spyder Python Console Part 2

In Figure 10, the user inputs option 3, which removes a task input by the user. The user is then presented with the information that the task has been removed, and the updated To Do list is printed.

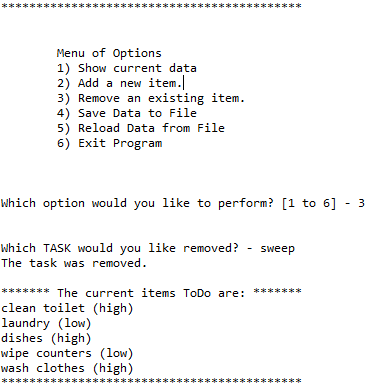


Figure 10 – Spyder Python Console Part 3

Figure 11 shows the results when the user inputs 4 and would like to save the data. As expected, the current list that will be saved to the textfile is printed, and the user is prompted with whether or not they would like to save the data. When “y” is input, the user is presented with the message saying that the data has been saved, and the menu is again displayed.

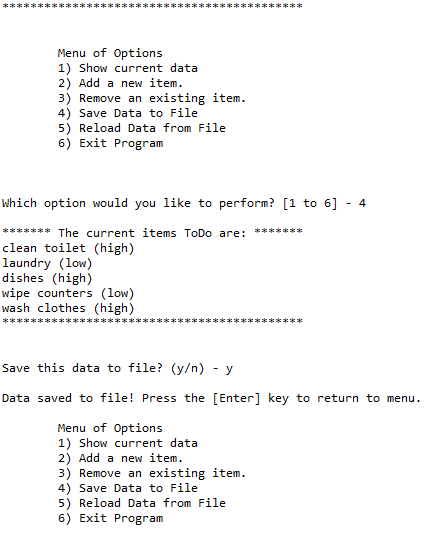


Figure 11 – Spyder Python Console Part 4

When option 5 is input as shown in Figure 12, the user is given the option to upload the data in the text file without saving the data. Once “y” is input, the file in the To Do list file are printed to the user. As always, once the step is executed, the menu of options is printed to the user.

Lastly, when the user inputs option 6, the program ends.

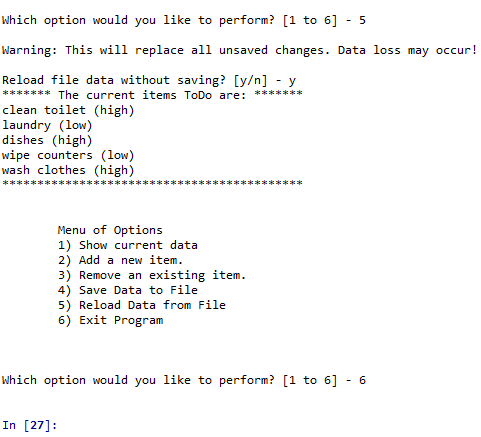


Figure 12 – Spyder Python Console Part 5

Figure 13 displays that the output text file is located in the same file as the program.

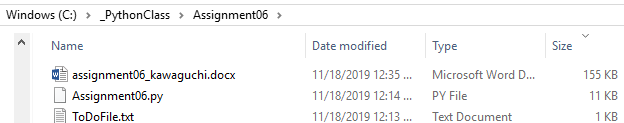


Figure 13: Proof of the Text File location

Figure 14 shows the data stored in the ToDoFile text file. The data matches what was seen last in the command window, so the program was successful!

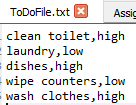


Figure 14: To Do List Data

Summary

In Conclusions, I successfully modified the To Do file by creating functions and referencing the functions throughout the code. After modifying the program, the user is still able to successfully add, remove, print, and save the data input.