To-Do List Program

# Introduction

In this assignment I will give an overview of how I modified an existing program that allows the user to choose various actions to create a To Do list. The program reads an existing list and can add, delete, and print the tasks and their priorities, or save the information back to the file. When adding tasks, the program asks for the name of a task, and then asks for its priority. Both pieces of data are stored in a dictionary, where each task and priority are a row of data. Each dictionary entry is then added to a list to create a table of data which can then be printed or saved. This is similar to the To Do List program developed for the last assignment, but uses classes and functions to organize the script.

# Writing the script

To begin with, we were given a starting template to follow in developing the code as shown in Figure 1. The code was broken down into sections for data, processing, and presentation, along with the main body of the script. The processing and presentation sections were defined as classes called Processor and IO respectively. The variables had already been declared in the data section, and the functions and menu branches laid out. While developing the final code, many of the functions were copied to another file to be worked individually then combined to generate the complete script.

After any branch except option 5 is completed, the program loops back to the menu.

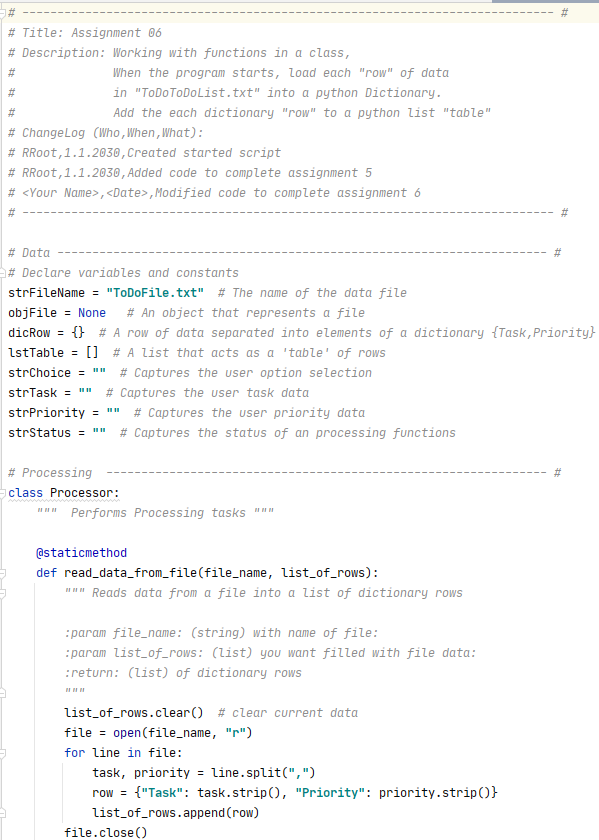


Figure : Starting code from assignment

## Step 1: Load the Data into Memory

The first step was to import the data from ToDoList.txt into memory so the program could act on it. As shown in Figure 2, the function ‘read\_data\_from\_file’ in the class ‘Processor’ was called with the file name and an empty list passed as arguments. The text file was opened and each row was extracted and split to create a dictionary element. Each dictionary element was then appended to the table, which was returned as lstTable. The string ‘Success’ had no variable to be returned to, so it did not appear in the main body of the script.

*# Main Body of Script ------------------------------------------------------ #  
  
# Step 1 - When the program starts, Load data from ToDoFile.txt.*Processor.read\_data\_from\_file(strFileName, lstTable) *# read file data*

…

*# Processing --------------------------------------------------------------- #*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  
  
 :param file\_name: (string) with name of file:  
 :param list\_of\_rows: (list) you want filled with file data:  
 :return: (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, **'Success'**

Figure : Reading from the file

Since the text file ToDoList.txt did not exist, the program gave the error shown in Figure 3.

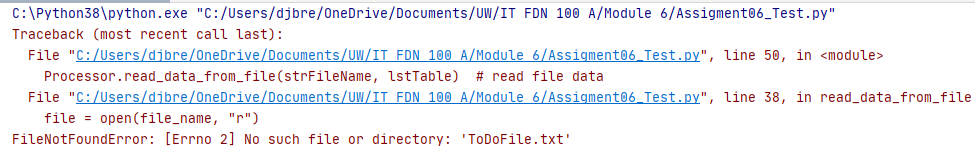


Figure : Missing file error

Rather than creating a placeholder text file, I imported os.path and checked to see if the file existed as shown in Figure 4.

*# Data ---------------------------------------------------------------------- #  
# Declare variables and constants*from os import path

…

list\_of\_rows.clear() *# clear current data*if path.exists(file\_name):  
 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, **'Success!'**

Figure : Code to check if file exists

If it existed, the data was read from the file. If not, the function bypassed the file request and returned lstTable as empty.

## Step 2 and Step 3: Display a Menu and Show Current Data

Before the next steps were developed, a ‘while’ loop was initialized to repeat main body commands. Steps 2 and 3 would then display the current data and show the user the possible actions they could choose.

To show the current data, the function ‘print\_current\_Tasks\_in\_list’ in the class ‘IO’ was called with lstTable as the argument as shown in Figure 5. The ‘print’ function gave the requirements for the format of the printed list.

*# Step 3 Show current data*IO.print\_current\_Tasks\_in\_list(lstTable) *# Show current data in the list/table*

…

@staticmethod  
def print\_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(**"\*\*\*\*\*\*\* The current Tasks ToDo are: \*\*\*\*\*\*\*"**)  
 for row in list\_of\_rows:  
 print(row[**"Task"**] + **" ("** + row[**"Priority"**] + **")"**)  
 print(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 print() *# Add an extra line for looks*

Figure : Code to show current data

The program then called the function ‘print\_menu\_Tasks’ in class ‘IO’ to display the menu (see Figure 6).

IO.print\_menu\_Tasks() # Shows menu

…

@staticmethod  
def print\_menu\_Tasks():  
 *""" Display a menu of choices to the user  
  
 :return: nothing  
 """* 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*

Figure : Code to display menu

Once the menu was displayed, the program called the function ‘input\_menu\_choice’ in class ‘IO’ to request the user to input 1, 2, 3, 4, or 5 as shown in Figure 7. Since the data would be used later in the program, the returned result was assigned to a variable.

strChoice = IO.input\_menu\_choice() *# Get menu option*

…

@staticmethod  
def input\_menu\_choice():  
 *""" Gets the menu choice from a user  
  
 :return: string  
 """* choice = str(input(**"Which option would you like to perform? [1 to 5] - "**)).strip()  
 print() *# Add an extra line for looks* return choice

Figure : Code to request user input

The result is shown in Figure 8.

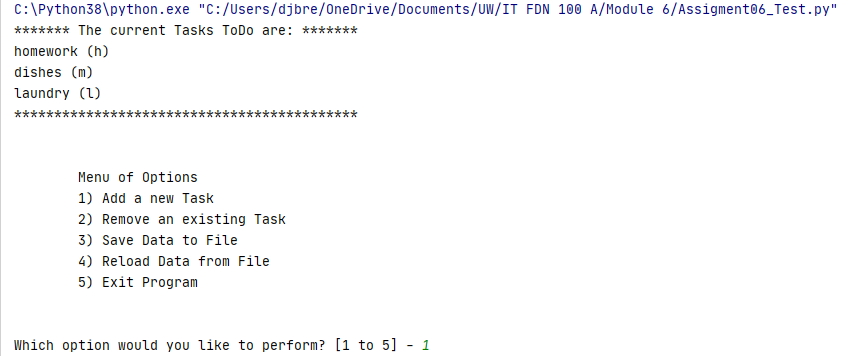


Figure : Display current list and menu

## Step 4: Process User’s Menu Choice

The next step used If-Else If branches based on the value entered by the user from the prior step to determine the next actions. Many of these options required additional coding to be developed. The last action of each option (except option 5) is to call the ‘input\_press\_to\_continue’ function in class ‘IO’. This prints the status of each option and waits for the user to press Enter to continue running the program and loop back to Steps 2 and 3.

### Option 1: Add a New Task

To add a new task, the function ‘input\_new\_task\_and\_priority’ in the class ‘IO’ was called with input prompts for the arguments and variables for the results as shown in Figure 9. The function asks for a task name and priority, then returns them to the associated variables.

strTask, strPriority = IO.input\_new\_task\_and\_priority(**"Enter task name: "**, **"Enter task priority (L,M,H): "**)

…

@staticmethod  
def input\_new\_task\_and\_priority(task\_message, priority\_message):  
 *""" Gets a task name and priority from the user  
  
 :param task\_message: (string) request for task name  
 :param priority\_message: (string) request for task priority  
 :return: (string) task name, task priority  
 """* return str(input(task\_message)).strip().lower(), str(input(priority\_message)).strip().lower()

Figure : Option 1 Get task information

The program then called the function ‘add\_data\_to\_list’ in the class ‘Processor’ passing the task and priority variables and lstTable as arguments. The task name and priority were added as a dictionary row, which was then appended to the existing data table list as shown in Figure 10. The data table and a status were returned to the program.

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

…

@staticmethod  
def add\_data\_to\_list(task, priority, list\_of\_rows):  
 *""" Adds a new dictionary row into a list of dictionary rows  
  
 :param task: (string) name of task  
 :param priority: (string) priority of task (L,M,H)  
 :param list\_of\_rows: (list) you want filled with file data  
 :return: (list) of dictionary rows, (string) status  
 """* lstTable.append({**"Task"**: task, **"Priority"**: priority})  
 return list\_of\_rows, **'Success!'**

Figure : Option 1 Add data to list

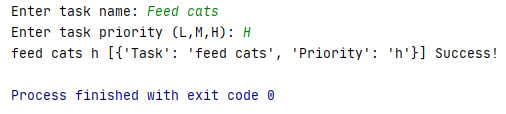


Figure : Results of Option 1 test

### Option 2: Remove a Task

To remove a task, the function ‘input\_task\_to\_remove’ in the class ‘IO’ was called with an input prompt for the argument and a variable for the result as shown in Figure 12. The function asks for a task name, then returns it to the associated variable.

strTask = IO.input\_task\_to\_remove(**"Which task do you want to remove? "**)

…

@staticmethod  
def input\_task\_to\_remove(message):  
 *""" Gets a task name from the user for removal  
  
 :param message: (string) request for task name  
 :return: (string) task name  
 """* return str(input(message)).strip().lower()

Figure : Option 2 Get task information

The program then called the function ‘remove\_data\_from\_list’ in the class ‘Processor’ passing the task variable and lstTable as arguments. The list of rows was rewritten row by row excluding the entered task name as shown in Figure 13. The data table and a status were returned to the program.

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

…

@staticmethod  
def remove\_data\_from\_list(task, list\_of\_rows):  
 *""" Removes a specific dictionary row from a list of dictionary rows  
  
 :param task: (string) name of task  
 :param list\_of\_rows: (list) you want filled with file data  
 :return: (list) of dictionary rows, (string) status  
 """* list\_of\_rows = [dicRow for dicRow in list\_of\_rows if dicRow[**"Task"**] != task]  
 return list\_of\_rows, **'Success!'**

Figure : Option 2 Remove data from list

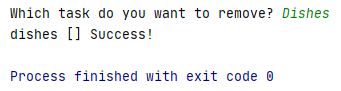


Figure : Results of Option 2 test

### Option 3: Save the Data

To save the data list as a text file, the program gave an additional prompt by calling function ‘input\_ yes\_no\_choice’ in the class ‘IO’ with an input prompt for the argument and a variable for the result as shown in Figure 15. The function asks for a ‘y’ or ‘n’ input, then returns it to the associated variable.

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

…

@staticmethod  
def input\_yes\_no\_choice(message):  
 *""" Gets a yes or no choice from the user  
  
 :return: string  
 """* return str(input(message)).strip().lower()

Figure : Option 3 Get information

If the result was ‘y’, the program called function ‘write\_data\_to\_file’ in the class ‘IO’ with the file name and lstTable as arguments as shown in Figure 16. The function would save the list data to the declared file with each row on a new line. The file was opened in Write mode rather than append because all of the data had been loaded into memory.

if strChoice.lower() == **"y"**:  
 lstTable, strStatus = Processor.write\_data\_to\_file(strFileName, lstTable)  
 IO.input\_press\_to\_continue(strStatus)  
else:  
 IO.input\_press\_to\_continue(**"Save Cancelled!"**)

…

@staticmethod  
def write\_data\_to\_file(file\_name, list\_of\_rows):  
 *""" Writes data from a list of dictionary rows into 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, (string) status  
 """* file = open(file\_name, **"w"**)  
 for row in list\_of\_rows:  
 file.write(str(row[**"Task"**]) + **","** + str(row[**"Priority"**] + **"**\n**"**))  
 file.close()  
 return list\_of\_rows, **'Success!'**

Figure : Option 3 Write data to file

If the user did not enter ‘y’, the function ‘input\_press\_to\_continue’ was called with the message “Save Cancelled!” as the argument.

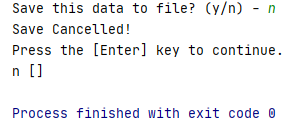


Figure : Results of Option 3 test

### Option 4: Reload Data from File

To save the data list as a text file, the program printed a warning about data loss and an additional prompt by calling function ‘input\_yes\_no\_choice’ as described in Option 3.

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) - "**)

Figure : Option 4 Get information

If the result was ‘y’, the program called function ‘read\_data\_from\_file’ described in Step 1. This reset lstTable back to the file configuration (see Figure 19.

if strChoice.lower() == **'y'**:  
 lstTable, strStatus = Processor.read\_data\_from\_file(strFileName, lstTable)  
 IO.input\_press\_to\_continue(strStatus)  
else:  
 IO.input\_press\_to\_continue(**"File Reload Cancelled!"**)

Figure : Option 4 Reload data from file

If the user did not enter ‘y’, the function ‘input\_press\_to\_continue’ was called with the message “Save Cancelled!” as the argument.

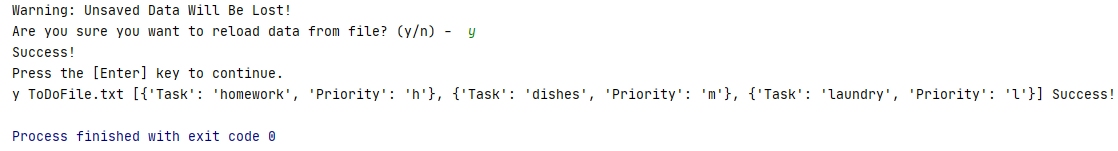


Figure : Results of Option 4 test

### Option 5: Exit the Program

Finally, if the user entered “5”, the program printed “Goodbye” and exited as shown in Figure 21.

print(**"Goodbye!"**)  
break *# and Exit*

Figure : Option 5 code

# Testing the script

To see whether the code functioned as intended, I first ran it in PyCharm. The data from ToDoFile.txt is printed along with the menu as shown in Figure 22.

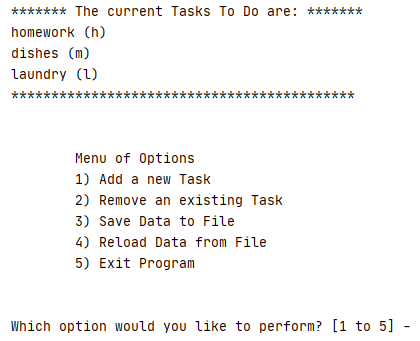


Figure : Show current data - PyCharm

Choosing option 1 lets me input a task and its priority. The program prints the updated list and reprints the menu and query (see Figure 23):

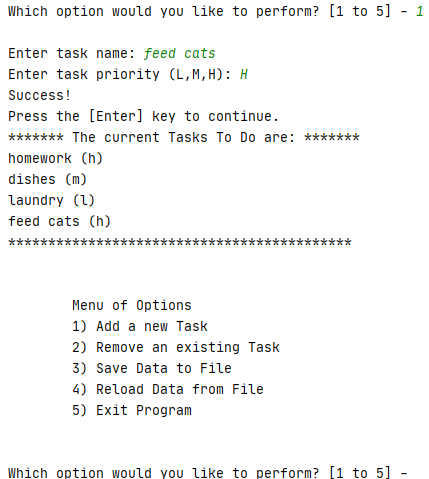


Figure : Adding a task - PyCharm

When option 2 was entered, the program requested a task name, then printed the updated list and returned to the menu as shown in Figure 24.

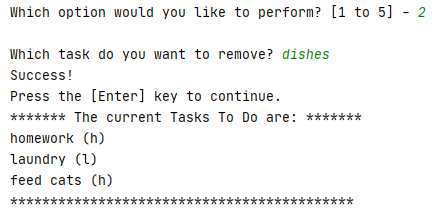


Figure : Removing a task - PyCharm

Next, I entered option 4 to reload the file data, but also entered ‘n’ when asked if I wanted to reload the data. The reload was cancelled and the data remained the same as shown in Figure 25.

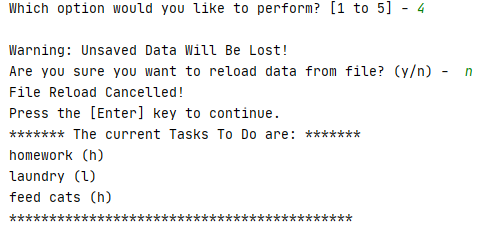


Figure : Reload data from file - PyCharm

I chose option 4 again, but this time entered ‘y’ to reload the data. This time, the new task I had added and the old task I had removed were over-written by the file data (see Figure 26).

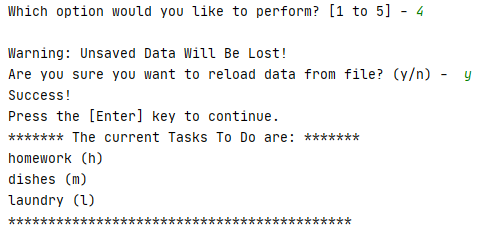


Figure : Really reload data from file - PyCharm

I used option 5 to exit the program so that I could check that the text file was created (see Figure 27).

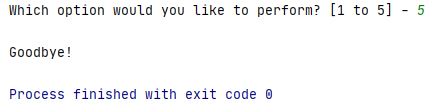


Figure : Exit program - PyCharm

The ToDoList.txt file was created in the working folder as shown in Figure 28.

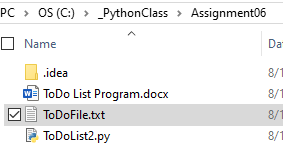


Figure : ToDoList.txt file

The data showed in the text file as expected (see Figure 29).

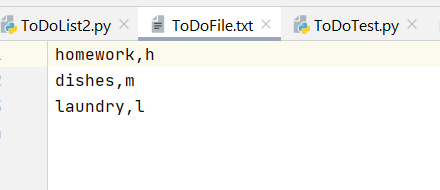


Figure : Text file data

Finally, I ran the program from the command window. I both added and deleted a task as shown in Figure 30.

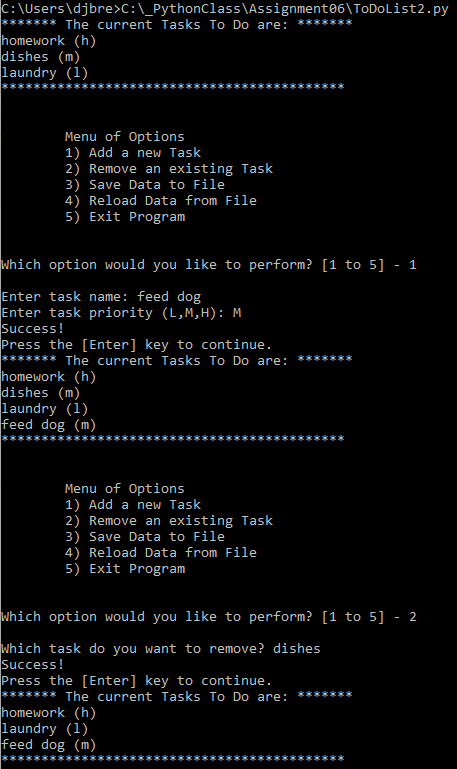


Figure : Run program – cmd

I entered option 3 to save to list data back to the file, but also entered ‘n’ when asked if I wanted to save the data. The save was cancelled and the data remained the same as shown in Figure 31.

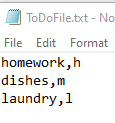
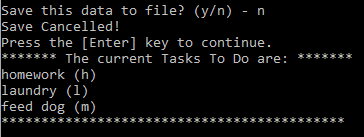


Figure : Save data to file – cmd

I chose option 3 again, but this time entered ‘y’ to save the data. This time, the new task I had added and the old task I had removed were over-written in the file (see Figure 32).

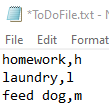
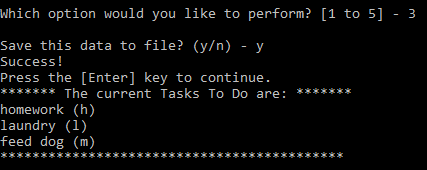


Figure : Really save data to file - cmd

The program worked per the requirements of the assignment, but the removal function could be further modified to give an error message if the data does not exist.

# Summary

Using the textbook and the Module 6 documentation, I created the ‘To Do List2’ program and successfully ran it in PyCharm and the OS command window.