

Phillip Thompson
5/17/23
IT FDN 110 A
Assignment 5
<https://github.com/philthom10/IntroToProg-Python>

Assignment 5 – To Do List – List and Dictionaries

1. Introduction

Module 05 this week focused on the basics of Dictionaries, comparing and contrasting with Lists. The assignment this week was the first time taking an existing code and revising it to complete a certain function. The To Do List program demonstrates reading data from a file, adding data to dictionaries and lists, and writing data from dictionaries and lists to files.

2. Writing the Code

The code for this assignment was written in the PyCharm IDE. I first created a Project called Assignment 05, and then copied the Assignment05_starter.py file into that project folder. See below (Figure 1), showing the project structure in PyCharm.

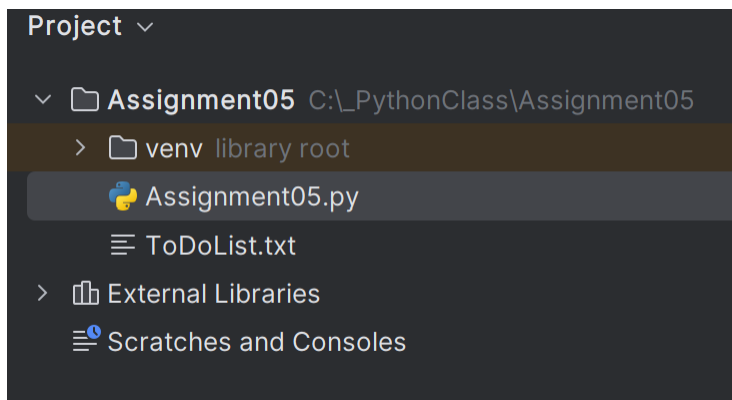


Figure 1: Project structure in PyCharm

In order to keep it clear which comments were newly added by me rather than existing from the starter script, I started each comment I added with my initials PT.

2.1. Processing

2.1.1. Loading Existing Data (Step 1)

The first step and first section to add code to was to load any existing data from the text file into a list of dictionary rows in Python. This was done similarly to the Module 05 lecture notes Lab 5-2. The code I added for this step is shown below (Figure 2).

```

objFile = open(objFile, "r") # PT: opening file in read mode
for row in objFile: # PT: goes through each line in file
    strData = row.split(",") # PT: returns a list with two elements, items that were sep by comma file
    dicRow = {"Task":strData[0], "Priority":strData[1].strip()} # PT: indexes string created by file and keys to dict.
    lstTable.append(dicRow) # PT: makes list "table" with dictionary elements as "rows"
objFile.close()

```

Figure 2: Code for Step 1, loading existing data

Using lab 5-2 and previous assignments as a guide, this code was straight-forward. The `.split()` method takes the string from text file with a comma as a separator and makes it into a list, assigned to variable `strData`. From there, the elements from the list are made into a dictionary assigned to the variable `dicRow`. The keys for the two elements in the dictionary are "Task" and "Priority". The For loop appends the `lstTable` list with a dictionary row for each row of data in the text file.

I attempted to write a more robust code for this section that would check if the text file existed first, and then if it did, create the list as shown above (Figure 2), but if it did not exist, it would create the file and add the headers in it. This attempt used If and Elif methods along with True and False comparison with the file variable. This did not work correctly, so after some time of trial and error, I decided to use the simpler code shown above (Figure 2) that relies on the file already existing.

2.2. Input / Output

The next step in writing the code was handling the Input / Output section. The display of the menu in Step 2 was already completed in the starter code, so Step 3 was the next portion that I added code to.

2.2.1. Show the Current Items in the Table (Step 3)

In order to show the current items in the table back to the user, I used the code shown below (Figure 3).

```

# Step 3 - Show the current items in the table
if (strChoice.strip() == '1'):
    print("The current data is: \n") # PT
    for row in lstTable: # PT
        print(row["Task"] + " | " + row["Priority"]) # PT: calls elements of row by key in dictionary
    continue

```

Figure 3: Code for showing current table items to user

This code uses another for loop that prints the data in `lstTable` by row (element). Since each row (element) in `lstTable` is a dictionary, each of the two elements in the dictionary (row) are called out by key: "Task" and "Priority".

2.2.2. Adding a New Item to the List / Table (Step 4)

The next code added to the program was for Step 4, which is adding a new To-Do item to the list / table. I accomplished this with the code shown below (Figure 4)

```
# Step 4 - Add a new item to the list/Table
elif (strChoice.strip() == '2'):
    strNewTask = input("Task name: ").title().strip() # PT: captures user input and uses strip method
    strNewRank = input("Task priority (High, Medium, Low): ").capitalize().strip() # PT
    lstTable.append({"Task": strNewTask, "Priority": strNewRank}) # PT: appending table with new entries
    saveStatus = False # PT: Toggles back to false if updated after saving to file
    continue
```

Figure 4: Code for showing current table items to user

This code is similar to what was done in Assignment 04 in terms of appending the list *lstTable* with the *append()* method, but it appends each row (element) of the table (list) with a dictionary, rather than a string. I originally defined a new dictionary variable with elements made of the user input variables and then appended *lstTable* with that, but I decided that was unnecessary since this block of code is not being used in a For loop. The code was consolidated to the what is shown above (Figure 4). For front-end aesthetics, I chose to use the title case method, as described in table 2.3 of the textbook (**DAWSON, 38**).

Note: the *saveStatus* variable is covered later in this document—it is not critical for the Step 4 addition of a new item to the list / table.

2.2.3. Removing an Item from the List / Table (Step 5)

In order to allow the user to remove an item from the list / table of To-Do tasks, I used the code shown below (Figure 5).

```
# Step 5 - Remove a new item from the list/Table
elif (strChoice.strip() == '3'):
    strRemoval = input("Which task would you like to delete? ").strip().title() # PT
    counter = 0 # PT: For counting removals to aid in giving user feedback
    for row in lstTable:
        if row["Task"].lower() == strRemoval.lower(): # PT: searches each row in table for entered value
            lstTable.remove(row) # PT: removes row from table if it has selected value
            counter += 1
    if counter > 0: # PT: counter will be over 0 only if a row item was removed
        print("\n The task '" + strRemoval + "' was removed from the list!")
    elif counter == 0: #PT: if counter is zero, no removals were made
        print("\n The task '" + strRemoval + "' was not in the original list. Please select a menu option.")
    saveStatus = False # PT: toggles back to false if updated after saved
    continue
```

Figure 5: Code for removing an item from the list/table

The critical, required function of Step 5 is accomplished by receiving a user input string of a task to be deleted, and assigning it to the variable *strRemoval*. A For loop then uses the *==* comparison operator to see if the “Task”-keyed element is the same as the user input in each row. The row with the matching “Task” element then is removed using the *remove()* method.

I originally tried to accomplish this using the *delete* method described in Chapter 5 of the textbook (**DAWSON, 146**).

That code soon became complex and challenging to get to work. Revisiting the Module 05 lecture and QA session reminded me the *remove()* method, which created a much simpler and concise code.

I wanted to provide user feedback on whether or not a matching task was found and removed. To accomplish this, I created a variable called *counter* and assigned the integer value of 0 to it. If a row is deleted in the For loop, the *counter* variable is increased by 1. Then simple If and Elif statements are used to print back feedback to the user if an item was deleted or not. There is likely a more concise way to accomplish providing the feedback to the user, but with my existing Python knowledge, I found this to be a simple way to do it.

2.2.4. Saving Tasks to File (Step 6)

The code used to save the data to a file is similar to that in Assignment 04, just with added complexity of the rows (elements) in the table (list) *lstTable* being dictionaries. I converted the values called out by key to a string in order to concatenate with a comma separator within the file *write()* method. The code for this step is shown below (Figure 6).

```
# Step 6 - Save tasks to the ToDoList.txt file
elif (strChoice.strip() == '4'):
    objFile = open("ToDoList.txt", "w") # PT
    for row in lstTable:
        objFile.write(str(row["Task"]) + "," + str(row["Priority"]) + "\n") # PT: saves each row to file
    objFile.close()
    saveStatus = True # PT: toggles save status to true once data has been saved to file
    print("Tasks successfully saved to file.")
    continue
```

Figure 6: Code for saving to a file

2.2.5. Exiting the Program (Step 7)

The final step that I added code for was exiting the program. I wanted to alert any user with unsaved changes who chooses option 5 that they have unsaved data before the exit the program, giving them an option to save before exiting. I accomplished this with the code shown below (Figure 7).

```

# Step 7 - Exit program
elif (strChoice.strip() == '5'):
    # PT: This block checks to see if file has been updated without saving before closing
    # if the file has been updated without saving it asks user if they would like to save before closing
    if saveStatus: # PT: evaluates if true then breaks and exits
        break
    elif not saveStatus: # PT: evaluates if not saved (false) then prompts for save choice below
        strSavePick = input(" ATTENTION: You have unsaved changes. " # PT: prompts for saving if unsaved data
                             "Would you like to save before exiting? (Y or N) ").lower().strip()
        if strSavePick == "y":
            objFile = open("ToDoList.txt", "w")
            for row in lstTable:
                objFile.write(str(row["Task"]) + "," + str(row["Priority"]) + "\n") # PT: saves to file
            saveStatus = True # PT: toggles save status to true once data has been saved to file
            print("\nChanges successfully saved to file.")
            break # and Exit the program
        elif strSavePick == "n":
            break
        else:
            print("\n\t*** Invalid choice, what would you like to do next? *** ") #PT: goes back to menu
    input("\n\nPress the enter key to confirm and exit.") # PT: Exits whole program upon user input

```

Figure 7: Code for saving to a file

As can be seen above (Figure 7), the user feedback and option to save unsaved data uses the *saveStatus* variable. I defined that as shown below (Figure 8), with the value of True assigned to it.

```

# -- Data -- #
# declare variables and constants
objFile = "ToDoList.txt" # An object that represents a file
strData = "" # A row of text data from the file
dicRow = {} # A row of data separated into elements of a dictionary {Task,Priority}
lstTable = [] # A list that acts as a 'table' of rows
strMenu = "" # A menu of user options
strChoice = "" # A Capture the user option selection
saveStatus = True # PT: status of if file has been saved since last change to data

```

Figure 8: Variables

The *saveStatus* variable has the value of False assigned to it in steps 4 and steps 5 (see Figures 4 and 5 in this document). Therefore, the variable starts as True, and then if tasks are added and deleted it changes to False (data is not saved). When the user saves the data in step 4 (see Figure 6), the *saveStatus* variable has the value of True assigned to it again.

So, when it comes to step 7 for exiting the file, the *if saveStatus* statement confirms the variable is True, then the code is broken from the while loop and the user can press enter to exit the program. The *elif not saveStatus* statement checks if the variable is False, in which case it gives the user the option to input a selection to choose to save the data or not, similar to the code used in Assignment 04. This would be a good candidate for a custom function for For loop that saves the data so it doesn't have to be fully typed again, but we were not to use custom functions in this assignment.

I originally wrote the If and Elif statements as follows:

```
if saveStatus == true:
    break
elif saveStatus == false:
    <<con't>>
```

However, I took the PyCharm suggestion to change to the simpler code using Boolean values of the variable that is shown in Figure 8.

3. Running the Program

The program ran successfully both in PyCharm (Figure 9) as well as the Windows command window (Figure 10). I did have an error the first few times when trying to run in the command window due to not being able to find the ToDoList.txt file. This error was solved after some troubleshooting, with the solution being to change my working directory in the command window to my Assignment 05 folder that the .py file was in.

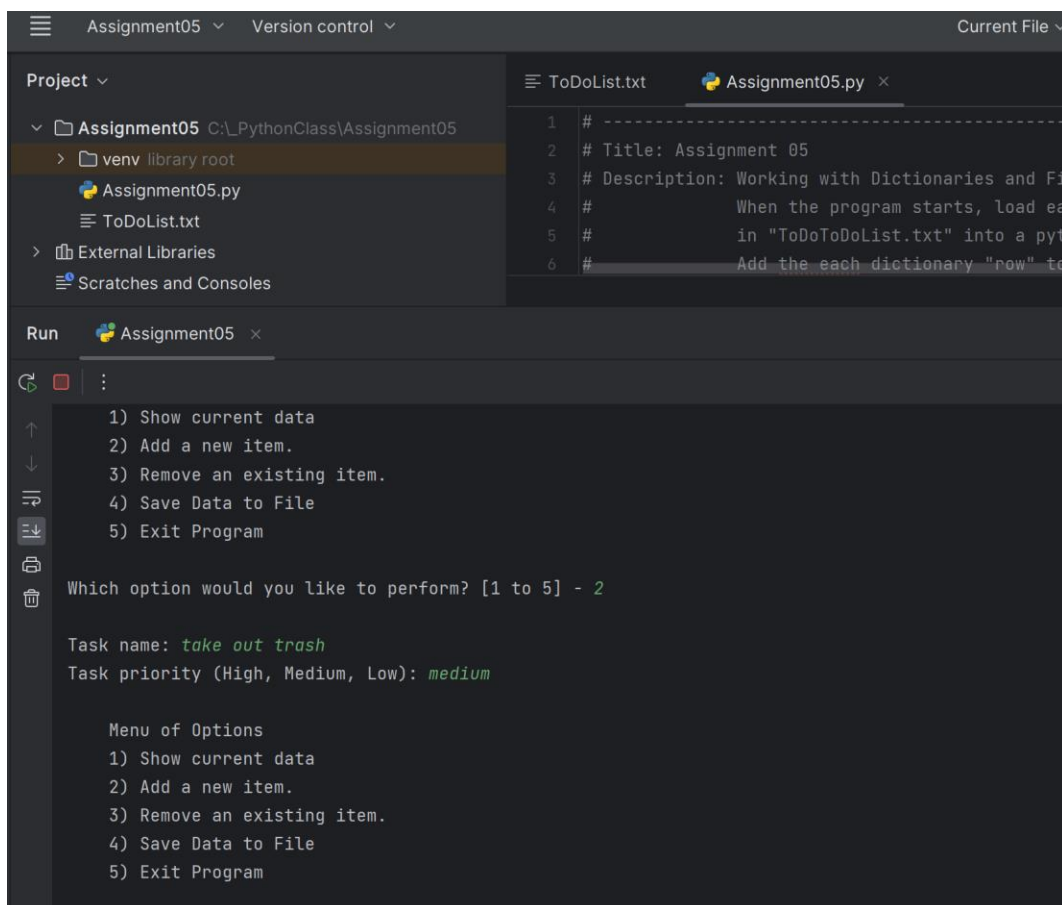
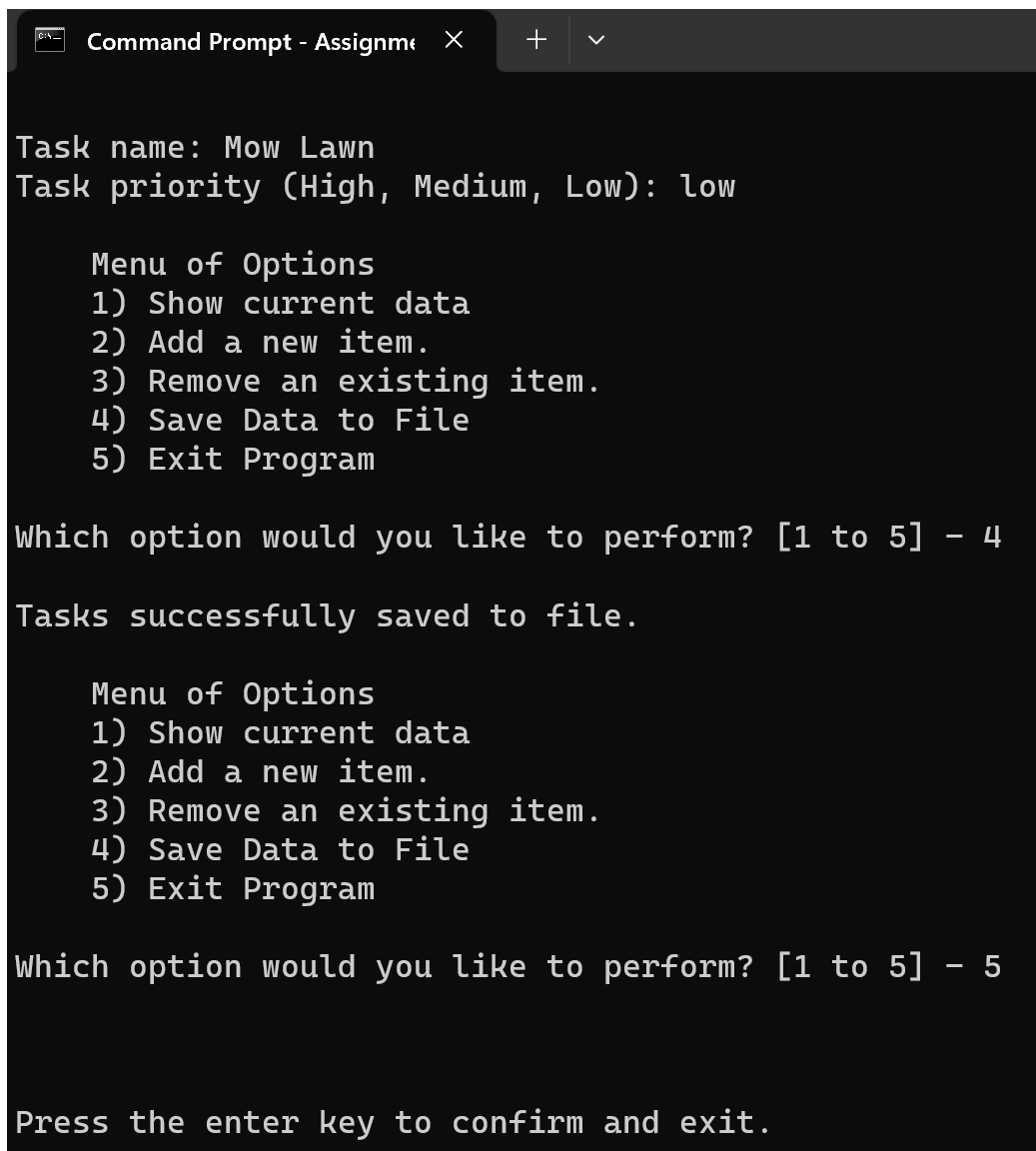


Figure 9: Running Program in PyCharm



```
Command Prompt - Assignme X + v

Task name: Mow Lawn
Task priority (High, Medium, Low): low

Menu of Options
1) Show current data
2) Add a new item.
3) Remove an existing item.
4) Save Data to File
5) Exit Program

Which option would you like to perform? [1 to 5] - 4

Tasks successfully saved to file.

Menu of Options
1) Show current data
2) Add a new item.
3) Remove an existing item.
4) Save Data to File
5) Exit Program

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

Press the enter key to confirm and exit.
```

Figure 10: Running Program in Windows command window

I verified that all data was saved in the text file in the format that I desired, as shown below (Figure 11).

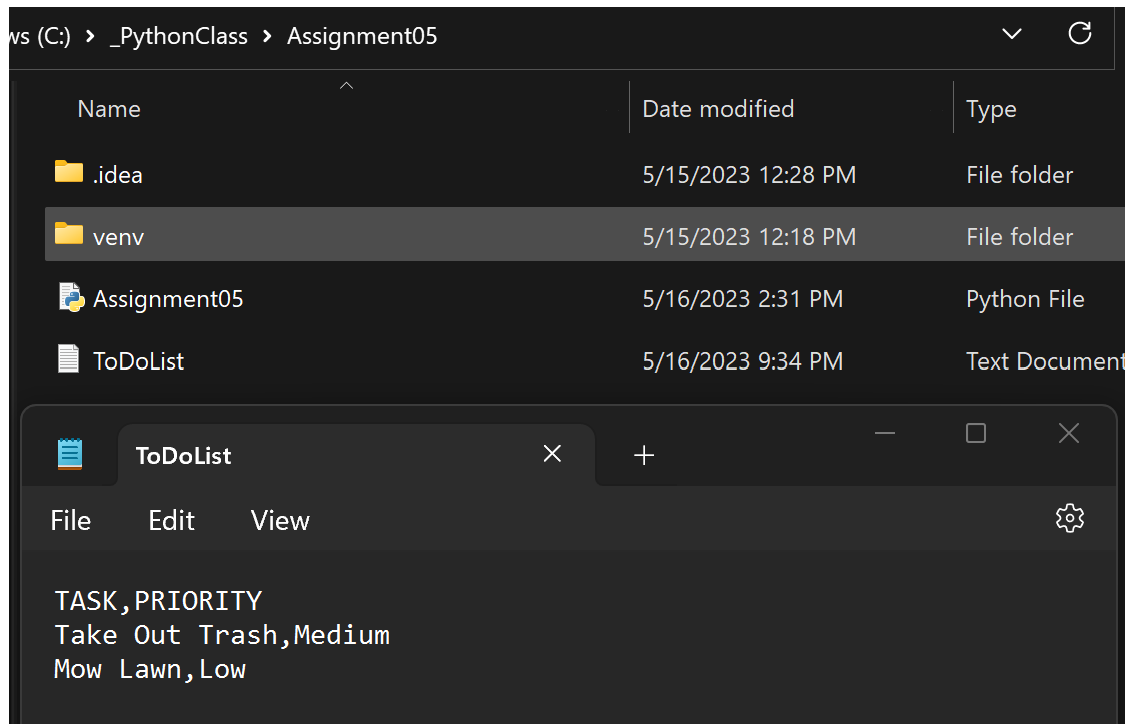


Figure 11: Text file with data saved from program, folder shown in background

4. Summary

This ToDoList program demonstrated the new knowledge of working with Dictionaries, as well as introducing the methods that can be used to delete data from a file, not just write and append to one. While the core methods used to accomplish the required steps in the program are similar to those used in the previous assignment, this assignment was an introduction in writing code as an edit to an existing program. Working from an existing code with still simpler required methods allowed me to experiment with adding features to improve the user experience within the bounds of an existing code framework. This is clearly a skill that is necessary for practical applications of using Python, as it is not always the case that one is starting from scratch to write a program from beginning to end.

Works Cited

Dawson, Michael. *Python Programming for the Absolute Beginner*, Third Edition, Cengage Learning, 2010