The To-Do List Script

# Introduction:

This week we worked with a start script to create a to-do list program. This allowed me to focus on the main components of the script: the actions the code was taking with the data the user input. However, I did have to adjust some of my script to match what already existed in the code. We leveraged a list of dictionaries in this assignment which I liked because dictionaries make it very clear what data is being stored within them.

# Topic 1: Reading files and loading data into memory

The first step in making this program work was to load any current data from the file associated with this program so we could begin working with it. Because the data existed as comma-separated values in text form, we needed to read each row of data in the file, split it at the comma, and separate each value into two items in a dictionary. Each row would translate to one dictionary, then that collection of dictionaries would be placed into a list.

I first opened the file in read mode, then iterated over each row with a for loop (Figure 1):

objFile = open("ToDoList.txt", "r")  
for row in objFile:  
 strData = row.split(",")  
 dicRow = {"Task": strData[0], "Priority": strData[1].strip()}  
 lstTable.append(dicRow)

Figure 1: Reading data from the file and placing each row into a dictionary then a list.

Splitting the row, placing in a dictionary, and placing that dictionary into a list needed to happen within the for loop because each of those tasks needed to happen for each row in the file. I assigned the first and second values in the row to the keys “Task” and “Priority” respectively, and made sure to remove the invisible carriage return using the strip() method as recommended in the Module05 Programming Notes.

Using descriptive keys is important as I’ll need the keys to access individual dictionary items later on (Python Programming For The Absolute Beginner 3rd Addition, Michael Dawson, Cengage Learning 2010, Pg 142).

Now that the data had been read from the file and stored in memory, my user could begin interacting with. At this point, my program moved on to its next code block: the while loop.

# Topic 2: Working with lists and dictionaries to show current data to user

The starter script we were provided had already outlined the if/elif statements that needed to be used to continue showing the user the menu of options until they quit. I just needed to add in the actual actions that would be taken for each possible option.

I started with option 1: show the current data to the user. This was a straightforward process – I needed to write some code that would iterate through my list of dictionaries, then format the dictionary values in a way that was easy to read for my end user (Figure 2).

for row in lstTable:  
 print(f"{row['Task']}, {row['Priority']}")  
continue

Figure 2: A for loop that iterates over each dictionary in the list, then removes the values from each dictionary and prints them nicely.

Each item in the list lstTable was a dictionary, so rather than using the index numbers for each value, I used the keys I had set up when initially reading the data from the file (Python Programming For The Absolute Beginner 3rd Addition, Michael Dawson, Cengage Learning 2010, Pg 142). I was able to leverage f-string formatting in my print statement to replicate the comma-separated values the user would expect to see in the final file output.

I tested this code, and it worked successfully!

# Topic 3: Adding new dictionary to current list

The next action I worked on was adding a new item to the To-Do list. In order to do this, I needed to capture user input on what they wanted to add, place those values into a dictionary, then add that dictionary to the end of the current list holding all of the data.

Step one was creating two new variables to capture the user’s desired inputs (Figure 3):

strTask = input("Please enter a new task: ") # Get task from user  
strTask = strTask.title() # Format entry  
intPriority = int(input("Please enter priority: ")) # Get priority from user

Figure 3: Getting user input

I chose variable names that reflected the keys I would be using in the dictionary and used a few other methods to format the string input and ensure the correct data type was being captured.

Next, I placed each value into a dictionary with the appropriate keys, and finally appended that dictionary as an additional item in my list lstTable (Figure 4).

dicRow = {"Task": strTask, "Priority": intPriority} # Add user input to dict  
lstTable.append(dicRow) # Add new row to table

Figure 4: Adding the user inputs to a new dictionary, then appending that dict to my list

I tested this function by selecting option 2 to add a new row, then after that was complete viewing the current data. Success! My new row had been added.

# Topic 4: Removing dictionary from list

The final major manipulation a user might make to the To-Do List data was to remove an entry. I assumed the user would want to have control over which record they were deleting (rather than the most recently added, for example). This required an additional instance of user input (Figure 5).

strTask = input("Task to be removed: ") # Determine what task the user wants to remove

Figure 5: Created variable to capture user input on what task they want to remove

The first step was for my program to determine exactly what row it needed to remove. This required a for loop iterating over each “row” of data in my list lstTable and a the right method to remove the appropriate item. Within that for loop, I added an if statement to check for the user input strTask in that row’s “Task” value (Python Programming For The Absolute Beginner 3rd Addition, Michael Dawson, Cengage Learning 2010, Pg 137), then removed the row if strTask was found in it . W3 Schools provided a good list of methods I could use to remove the item (List/Array Methods, <https://www.w3schools.com/python/python_ref_list.asp>, 2020)(External Site).

I used “in” instead of “==” because I wanted to allow more flexibility for users to only enter partial tasks to remove. I also used the lower() method on both the user input variable and “task” value to ensure capitalization did not trip anything up (Figure 6).

for row in lstTable: # iterate over current entries to find task user entered  
 if strTask.lower() in row["Task"].lower(): # if task in a row matches user input, remove row  
 lstTable.remove(row)

Figure 6: An if statement nested within a for loop to check each row for user input and remove rows that match.

After that loop was complete, I added one more if statement to give the user some notice if the task they tried to remove could not be found using the in operator (Python Programming For The Absolute Beginner 3rd Addition, Michael Dawson, Cengage Learning 2010, Pg 125)(Figure 7).

if strTask not in lstTable:  
 print("Task not found, please enter task in list")

Figure 7: A final if statement to let user know if their task could not be found.

# Topic 5: Exiting the program and general user experience

Now that I had functioning code for each of the main actions a user could take, I needed to include a way for the user to exit the program. The code we started with already had code that functioned, but I wanted to add some features to improve the usability.

Namely, I wanted to make sure the user had one last chance to save their changes before exiting the program in case they chose exit by accident. To do this, I included an input statement prompting the user to choose what to do next, then an if statement for each choice (Figure 7).

strSave = input("Save any changes? [Y/N]: ") # Check if the user needs to save  
if strSave.lower() == "y": # If yes, save everything to the file then quit  
 objFile = open("ToDoList.txt", "w")  
 for row in lstTable:  
 objFile.write(f"{row['Task']},{row['Priority']}\n")  
 objFile.close()  
 print("Changes saved!")  
 break

Figure 8: Prompting the user to save or not

If the user chose Y, then the save process mimicked the code for Option 4, “Save Data to File”. If the user chose N, then the program simply closed via the break statement.

# Conclusion

Working with someone else’s code was both helpful at points and difficult at others. It was nice that the basic layout of the script was already there. However, some things, like variable names were different than what I would have chosen. Ultimately, using dictionaries made this easier because of how clear the keys made the actual data stored within the dictionary. This project made it very apparent how important good comments, variable names, etc. are when working on a program collaboratively.