**Module 5: Lists and Dictionaries**

# Introduction

Module 5 delves two kinds of data structures: the lists and the dictionaries. Both are similar in that they organize data but each one has its way. Students are exposed to various methods to access data from these structures and to modify them (appending or removing elements, for example). The module also touches on ways to improve code so that’s it’s easier to read and understand. Finally, students are introduced to version control through GitHub. The homework included for this week asks students to take a script made by the instructor and fill in the functionality to create a ToDo list, prompting the user to enter a task with a priority; students must make use of lists and dictionary data structures to store the data and write to a file.

# Course Content

## Lists

As students have been working with lists for the past few weeks, lists were covered briefly. However, the strip() function to trim unwanted carriage returns was shown. The author made use of this for the homework assignment.

## Dictionaries

The author has very brief exposure to dictionaries (the data structure anyway) as it is not a data structure that’s used in his line of work. Classroom training for both his undergraduate degree and subsequent refresher courses explained the basics of the dictionary using Java and C#, and their respective methods. Python’s implementation is more straightforward as the use of curly braces indicates a dictionary, and the colon separates the key from the value; this contrasts with other languages where the keys and values must be explicitly declared one to one (example: key=foo, value=bar).

Creating a list of dictionary entries proved to be a step above what the author is normally used to seeing. The textbook reading was simple enough to understand how to access items from a dictionary but entering each dictionary entry into a list meant that careful attention had to be paid to not use list methods on dictionary items and vice versa.

## Improving Scripts

Throughout the author’s career, many of the techniques shown in this section have been used in the industry, though other techniques are simply preferences by the coder. For example, the Separation of Concerns is heavily employed in the author’s work, with variables declared at the top (which in turn are separated based on whether they’re constants, global, local, placeholder, or custom ones defined elsewhere in the directory), function and method declaration in the middle layer, and the script’s “main” processing section. The author has also advocated to follow the lead of one program where requirement numbers and associated work tickets are placed as a comment above the area that’s been worked on to represent what it’s trying to achieve.

Functions are also the norm in the author’s work, though they are defined in a “spec” and the “body” implements it. However, in other high level programming languages, they are usually defined and implemented as part of an object class and are called from elsewhere. Seeing that Python can declare a function as part of its main section is interesting.

Finally, Error Handling work in the same manner as other languages (though their syntax is Try-Catch) but allow a program to fail without interrupting runtime. In the author’s undergraduate courses, they made significant use of Try-Catch with Java once data structure and operating system concepts were introduced. The author found an instance where this will be used in the homework assignment.

## GitHub

The author is familiar with several programs of version control, but IBM’s ClearCase was the one he has been exposed to the most as programs throughout his career have used this. However, Git has made a splash and these programs have moved on to Git, though using Atlassian’s Bitbucket (which operates in a similar manner to GitHub as it also used Git). The author’s experience with GitHub is limited only to his undergraduate capstone project where students used it as a pilot program for easier review by the professors. However, the author is well versed with Git and the importance of version control (especially in the case of loss of local work or code gone awry).

# Assignment

This week students are to begin using dictionaries and file handlers to load data, modify it, and write it back to a file. The instructor has provided a starter template which students need only fill the various sections.

Step 1 was to load any data from a text file called ToDoList.txt into a list of dictionary rows. This meant making use of the file handler functions, loading each line as a dictionary, and then add each dictionary to a global list table. The author’s first attempt was simply follow what students have done in other homework assignments and use the open() function, with ‘r’ as an argument to read data. When executed the first time, an error came up: no file to load! This is where the Try-Except technique came into play: if the file exists and can be opened, then read its content and load it; otherwise, warn the user that the file does not exist and no data was loaded, while proceeding to the main menu. The author considered a case where the file exists but is empty, merely to let the user know that there was no data to load so a conditional statement was entered to print it out. The result for Step 1 is referenced in Figure 1 below.

# -- Processing -- #

# Step 1 - When the program starts, load the any data you have

# in a text file called ToDoList.txt into a python list of dictionaries rows (like Lab 5-2)

#If the file exists, load the data...

try:

fileHandler = open(objFile,'r')

print("\nNow loading items from "+objFile)

for strData in fileHandler:

lstRow = strData.split(',')

dicRow = {"Task":lstRow[0],"Priority":lstRow[1].strip()}

lstTable.append(dicRow)

#Let the user know whether the file was empty or not

if len(lstTable) == 0:

print("\nFile was empty. Nothing was loaded.")

else:

print("\nItems loaded.")

fileHandler.close()

#...and if not, warn the user and continue.

except:

print("\nNo file to load. Continuing...")

## Figure 1 Implementation of Step 1

Step 2 was already implemented as the menu of options was already defined as a print statement. The author briefly considered copying the text and pasting it as part of the strMenu variable declared above, and simply print the variable, but as the functionality would remain unchanged, the author decided against it.

Step 3 asked the students to print out the local data. Using the same procedure as in previous homework assignments, a for loop was used, except this time the data called is a dictionary. The author became stumped for a bit trying to grasp the concept of iterating through a list but that each item is a dictionary; instead, the items were being treated as tuples. While working on other sections of the code, eventually he was able to understand how to call the data to be printed. Figure 2 below shows the implementation of Step 3.

# Step 3 - Show the current items in the table

if (strChoice.strip() == '1'):

#If there are items to print, then print them...

if len(lstTable) != 0:

intCounter = 1

print("Here is the list of items:\n")

#Get the dictionary item and print its task and priority

for row in lstTable:

print(str(intCounter)+". Task: "+row["Task"]+", Priority: "+row["Priority"])

intCounter+=1

#...otherwise, display that the list is empty

else:

print("You have no items; try adding some.")

## Figure 2 Implementation of Step 3

Step 4 was to implement adding a new item to the list. The first attempt was simply to ask for a task and priority and add it. However, the author took a page from the assigned chapter reading and also implemented input validation by checking existing data to see if the “task” being added existed; if so, then the program would return to the main menu without asking for the priority; otherwise, after ensuring that the task was not on the list, it allowed the user to input a priority and print out a confirmation of having been added. As there were no guidelines on what “priority” should be (high, medium, or low; scale of 1-5 or 1-10; etc.) the author stuck to a simple 1-5 scale. Also, when tasks are added, they’re added in lowercase for easier verification. Figure 3 shows the implementation of Step 4.

# Step 4 - Add a new item to the list/Table

# Step 4 - Add a new item to the list/Table

elif (strChoice.strip() == '2'):

exists = False

strTask = input("What task needs to be added? ")

#Isolate each dictionary row of the table to check if it exists

for row in lstTable:

if strTask.lower() in row["Task"]:

#Key (Task) exists; return to main menu

print("Task already exists; returning to main menu.")

exists = True

break

#If the key does not exist, continue

if not exists:

strPriority = str(input("From 1-5, what is the priority? "))

lstTable.append({"Task":strTask.lower(),"Priority":strPriority})

print("Added \""+strTask+"\" with priority "+strPriority)

## Figure 3 Implementation of Step 4

Step 5 required implementation of removing an item from the list so the author used similar logic to adding an item: check to see if the item already exists and if it does, remove it; otherwise, warn the user that the item does not exist. It also uses the lowercase() function to compare against a key. Figure 4 shows the implementation.

# Step 5 - Remove a new item from the list/Table

elif (strChoice.strip() == '3'):

exists = False

strTask = input("What task should be removed? ")

#Get the dictionary item

for row in lstTable:

#Check if the key (Task) exists and if it does, remove it

if strTask.lower() in row["Task"]:

exists = True

lstTable.remove(row)

print("Removed \""+strTask+"\" from the list.")

break

#Key does not exist; return to main menu

if not exists:

print("Task does not exist; returning to main menu.")

## Figure 4 Implementation of Step 5

Step 6 was to save the local data to a file. It is barely any different to how students have been writing data to a file, except each dictionary item must be unpacked. For this step, the author also decided to add the safeguard of when the list is empty (the contents of the file will be erased but a warning will inform the user that, because the local list was empty, the items on the file will also be deleted). Initially, the author noticed that when writing the items to the file, there would always be a newline at the end; the author implemented a way to ensure that when the last item of the list is added, there is no newline added at the end. Figure 5 below shows the implementation of Step 6.

# Step 6 - Save tasks to the ToDoToDoList.txt file

elif (strChoice.strip() == '4'):

fileHandler = open(objFile,"w")

#If the list is empty, no data to save so file contents are deleted

if len(lstTable) == 0:

print("List is empty; all items on file are deleted.")

fileHandler.close()

else:

#There are items to save

print("Saving current data to "+objFile)

counter = 1

#Get each dictionary

for row in lstTable:

#If the item is not the last one, add to file with newline at the end

if counter != len(lstTable):

fileHandler.write(row["Task"]+","+row["Priority"]+"\n")

counter += 1

#If item is the last, no new line at the end

else:

fileHandler.write(row["Task"]+","+row["Priority"])

fileHandler.close()

## Figure 5 Implementation of Step 6

Finally, Step 7 had students implement an exit for the program. The author only added an input() statement to inform that the program was exiting and to press any key to fully exit, as Figure 6 shows.

# Step 7 - Exit program

elif (strChoice.strip() == '5'):

input("Exiting the program. Press any key to continue...")

break # and Exit the program

## Figure 6 Implementation of Step 7

In the interest of saving space, and in order to show a full uninterrupted execution, the following Word files attached will show the execution from the Terminal from VSCode (File 1) and the Terminal app on the MacOS (File 2), which is the IDE the author has decided to use going forward. Figures 7 and 8 show screenshots of the corresponding terminals outputs.



## File 1 Output from VS Code



## File 2 Output from Terminal

Text

Description automatically generated

## Figure 7 VS Code Terminal output

Text

Description automatically generated

## Figure 8 MacOS Terminal output

# Summary

This week students learned how to use dictionaries, which is usually a data structure reserved for later in other languages. In a way, the author feels that Python’s way of dictionary is both easier and more complex: easier because of how it’s declared, but more complex in that its key-value pairs are not as explicit (though it may have been as a result of the homework assignment using a list of dictionary entries).

Most of the other contents of the module were not new to the author, though the author felt some nostalgia when he revisited his old GitHub account (which was still active) and found it still housed his capstone project; he hopes this time he’ll know how to properly use the repository.