Molly Weaver

February 15, 2021

Foundations of Programming: Python

Assignment 05

https://github.com/molweave532/IntroToProg-Python

The ToDo List Program

# Introduction

For assignment 05 we introduced dictionaries. This time we started with a fragment of program written by the professor and then filled in the code for each menu option. We took user input to create a to do list. The user could add or delete tasks and save the data to create a running list that could be returned to and edited again and again. The difficulty level definitely increased this week.

## The Program

This week in class we learned about separating our programs into three sections – data, processing, and presentation. As we are not using functions yet, the processing and presentation are mixed into the while loop, but all the variables were declared at the beginning of the program along with a brief description of each (***figure 1***).

Text

Description automatically generated

***Figure 1.*** Declaring data at the beginning of the program.

I did create a counter and a variable to be used for testing in the processing section. I assumed that that was acceptable.

The professor had already created an outline using pseudocode and set up the menu options, so the first step was to create the to do list text file. I created the file using objFile as in previous assignments (***figure 2***).

A picture containing text

Description automatically generated

***Figure 2.*** Creating the to do list.

The file was created as expected (***figure 3***).

Graphical user interface, application

Description automatically generated

***Figure 3.*** ToDoList.txt file created.

The first time through the file is empty but on subsequent runs there will be data saved. The next thing would be to read the data from that file and add it to the to do list, but I decided to write that part of the code later.

I started with option 2 “Add a new item.” I got user input and saved the data into a dictionary, where the task name was the key and the priority was the value (**figure 4**).

Graphical user interface, text, application

Description automatically generated

***Figure 4.*** Adding data as a dictionary item.

I included some print statements so I could see what was happening at each step (***figure 5***).

Text

Description automatically generated

***Figure 5.*** Adding tasks.

The task was accepted and saved as a dictionary. Next, I added the dictionary row to the table. The table is a list created by rows of dictionary items (***figure 6***).

Graphical user interface, text, application

Description automatically generated

***Figure 6.*** Adding the dictionary items to a list.

I included another print statement so I could see my list after a task was added (***figure 7***).

Text

Description automatically generated

***Figure 7.*** List created.

The dictionary item was created and added to the list. I added a few more tasks to the list (***figure 8***).

Graphical user interface, text, application

Description automatically generated

***Figure 8.*** Creating a list of dictionaries.

I realized that instead of saving each row as a dictionary inside of a list, I had created one dictionary of all the rows inside of a list. After multiple false starts I created a test file to just run this task (as the professor had suggested in class. Eventually, I will learn to listen to him!). I started with a pre-made list of dictionaries and searched to see if the tasked already existed (***figure 9***).

Graphical user interface, text, application

Description automatically generated

***Figure 9.*** Searching the list for a task.

I pulled out each dictionary row one at a time and saved it as “test.” I checked to see if the task entered matched the key for that row. If it did, I told the user that the task already existed (**figure 10**).

Text

Description automatically generated

***Figure 10.*** The task already exists.

I should have given the user the option here to overwrite the task and give it a new priority. I decided I could do that later if I had time. I needed to get the basics working first.

I added my new code to the homework code. I created an else statement to give the user the option to add a new task to the list and added that row to the table (***figure 11***).

Text

Description automatically generated

***Figure 11.*** Adding a task to the table round 2.

Graphical user interface, text

Description automatically generated

***Figure 12.*** New task added.

Text

Description automatically generated

***Figure 13.*** The task already exists!

Both my if and my else statements now functioned as I expected.

The next item on my personal to do list was to remove a task from the list. I tried a few list methods in my test program – remove and pop, but they of course didn’t work, so I tried removing a slice. The code ended up being similar to adding an item (***figure 14***).

Graphical user interface, text, application

Description automatically generated

***Figure 14.*** Removing an item from the list.

I was able to remove an item from my list (***figure 15***).

Text, letter

Description automatically generated

***Figure 15.*** Dishes removed from to do list.

Now that I had a functional to do list, I went back to option 1 – printing the to do list. I used a simple for loop to print each dictionary in the list individually (***figure 16***).

Text, letter

Description automatically generated

***Figure 16.*** Printing the to do list.

Text

Description automatically generated

***Figure 17.*** To do list printed.

The list was printed one task to a line, but I wanted to clean up the formatting a bit (***figure 18***).

Graphical user interface, text, application

Description automatically generated

***Figure 18.*** Formatting the print statement.

This was trickier and took several attempts in my test environment, but eventually it looked like what I wanted (***figure 19***).

Text

Description automatically generated

***Figure 19.*** The formatted list.

I essentially used the same combination of loops from option 2 to go through each dictionary row and pull out the key and the value and print them in a string. It would have looked better if the tasks were sorted by priority but again, that is something I could come back to if I had time.

The final menu option was to save the data to the ToDoList.txt file. First, I tried to use append in my test environment. I realized that my pre-made list items were being added twice. I changed from append to write (***figure 20***).

Graphical user interface, text, application

Description automatically generated

***Figure 20.*** Writing items to the file.

The first step each time the program was launched would be to read any items that had previously been stored and write them into the list. They would then be written back into the file along with any new items, so no old data would be overwritten (***figure 21***).

Graphical user interface, text, application

Description automatically generated

***Figure 21.*** List saved into the text file.

The final step was to read the saved data into the list the next time the file is opened (***figure 22***).

Graphical user interface, text, application, email

Description automatically generated

***Figure 22.*** Reading data from the file into the list.

I took most of this code from lab 5-3. Thank you, professor! I added several print statements to be sure everything was functioning as I expected (***figure 23***).

Text

Description automatically generated

***Figure 23.*** Reading data from the file and using it to create the list of dictionaries.

Each line is read from the file into a list. The line is converted to a dictionary and the new line character is removed using strip (). Then each dictionary is added to the final list table.

I ran through the menu options to be sure everything was still functioning as expected.

Text, letter

Description automatically generated

***Figure 24.*** Printing previously saved items.

Graphical user interface, text, application

Description automatically generated

***Figure 25.*** Adding a new item to the list.

Graphical user interface, text, application

Description automatically generated

***Figure 26.*** Re-saving the data to the file.

The final step was to get rid of all my extra print statements and run the program using the command line.

Text

Description automatically generated

***Figure 27.*** Cmd – Print list.

Text

Description automatically generated

***Figure 28.*** Cmd – Add a task.

Text

Description automatically generated

***Figure 29.*** Cmd – Task already exists.

Text

Description automatically generated

***Figure 30.*** Cmd – Remove a task.

Text

Description automatically generated

***Figure 31.*** Cmd – Updated task list.

Graphical user interface, text, application

Description automatically generated

***Figure 31.*** Cmd – Updated task list saved.

### Conclusion

The homework definitely got more complex this week. We started with a fragment of code written by someone else and had to add to it. The code had all the variables declared at the top and then the processing steps followed. We used a lot of knowledge from previous homework assignments – getting user input, creating lists, saving data to a file. There were also a lot of new concepts – creating and searching dictionaries, combining lists and dictionaries, deleting items from a list, reading data from a file into a specific format. I struggled at first until I took the professor’s advice and created a test file to run bits of my code until I got it working as expected. I also took a lot of the code from the class notes and labs, whereas before I had tried to write it out on my own. There were things I didn’t do that would have made the program more user friendly, but I didn’t have the time – sorting the list by priority, giving the user the option to overwrite a task to change the priority, and ensuring the user had saved the data before exiting. In the end the program performs the expected tasks, just maybe without a few bells and whistles.