Dave Phillips

2021-07-19

IT FDN 110 A Su 21: Foundations Of Programming: Python

Module 3 – Assignment 1

Taking Stock!

# **Introduction:**

In this assignment I’ll be coding a module to keep a basic home inventory for the user. The module will accept three input fields – Item, Quantity, and Value/ea. – and dynamically save that information into an external JSON file for later access.

# **Code Overview:**

The full block of code for this assignment can be viewed at the end of this document (Appendix 1). However relevant sections of code are provided throughout. We will break it down into chunks to examine the functionality of each section.

# **Initial Statements:**

Beginning at line 7 – lines 1-6 are introductory information – we have a few important initial statements. (Figure 1) The `**import** json` statement on line 7 allows the script to use the Python JSON module to organize and store the data later on. Lines 10-12 are simply print statements with a titleLine 15 sets up our link to the destination file (with a parameter of “a” for appending data).

1. import json

##########

1. fhandle = open("homeinventory.json", "a")

Figure 1. Initial statements

# **Input Loop:**

Following the initial statements, we move to the main input loop. This is where the user will respond to provided prompts that will ask for the required information (item, quantity, and value). Notice that the entirety of the script is wrapped in a `**while True**` loop on line 18. This allows the user to enter multiple items during one session instead of requiring them to restart the script each time. Additional discussion of this further on.

At the beginning of the loop – line 19 – we initialize an empty dictionary. It’s important that this be located inside the loop, as opposed to being initialized at the start, since we want to clear the information from it each time the loop iterates otherwise the values added further down will never change and the export file will simply have a lot of the same records, which is obviously not the goal. Next the user is prompted for the required information. I’ve omitted blank lines for brevity.

Let’s examine each input statement, as each addresses the received data in a different way. The first statement on line 20 is simply asking for a string name of the item being recorded. Since **input** already returns a string object, no type changes are required. However lines 22 and 24 are a different matter. Line 22 is asking for a *number* of items in question. In order to manipulate this later on, it’s prudent to simply convert this to an integer immediately upon entry. Line 24, on the other hand, is asking for a price, which typically displays two decimal places. To avoid any confusion that might arise from placing too many conversions on one line, I’ve split the processing of the value over two statements. Initially the string object derived from the user input is converted to a **float** number (with decimals) and line 25 ensures that only two decimal places are shown regardless of how many the user entered.

1. item = input("\nEnter an item. || ")
2. qty = int(input("How many? || "))
3. val = float(input(f"Enter the value of one '{item}'. || "))
4. value = round(val, 2)

Figure 2. Initial user input

Something worth noting is the use of an “f-string” on line 24. This is a convenient way to include my variables in the **print** statement, using curly braces, in a more natural way. The same logic applies to using an f-string on line 27 when printing the values that were entered for confirmation. Line 28 is a simple “Y”/”N” input from the user confirming the info they just entered. (Can’t have an incorrect inventory hanging around!)

# **Entry Confirmation and Processing:**

As an additional feature, I’ve included a way for the user to cancel their entry in case of an incorrect value. This is accomplished through a simple **if/else** set of statements. The easiest place to start is checking whether the user, on line 28, entered “n” to void their entry. The **continue** keyword simply sends the user back to the top of the infinity-loop and begins again. At this point, the `itemDict` dictionary would be cleared and ready for a new entry (which is why it was important to include it *within* the loop).

The goal of this prompt is to get a “y” entry. So next the script will run the **elif** statement on line 34 just to ensure the user didn’t type “Bob” or something else outside the scope of the “Y”/”N” entries. It simply tells the user that their entry was wrong and sends them back to the beginning.

Finally, if those two conditions are met, on lines 38-40 the script will commit the three objects we have created from the user input and place them in the previously created dictionary so it can be stored in the JSON file during the next step. To do this, the dictionary keys are simple hard-coded names, and we use the **=** operator to assign the corresponding values. (Figure 3)

1. else:
2. itemDict["item"] = item
3. itemDict["quantity"] = qty
4. itemDict["value-each"] = value

Figure 3. Data being committed to dictionary

# **Storing into File:**

Lastly, lines 41 & 42 use the **dumps** method from the json module to store the current values of **itemDict** in a more organized, or “pretty”, fashion. (Figure 4) Note that the final line in the app (line 57) performs the **close** method on the file we opened at the beginning. This is good practice and should always be observed.

Lines 45-55 allow the user to return to the beginning and enter more records. After all, an inventory of a single item won’t do anyone any good.

Text

Description automatically generated

Figure 4. – “Pretty” JSON output

# **Summary:**

Upon executing the script, the command line window will display steps that result in the following. Note how the values with decimal places exceeding two are rounded down. (Figure 5)

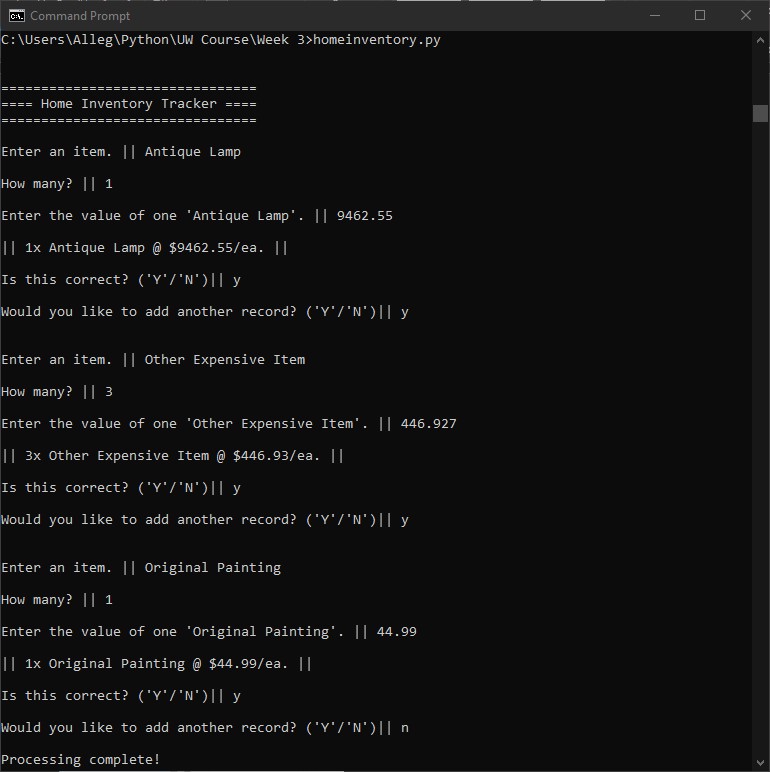


Figure 5. – Successful execution of the script via command line.