Dave Phillips

2021-08-02

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

Module 5 – Assignment 1

So Much To-Do, So Little Time…

# **Introduction:**

In this assignment I’ll be coding a script to keep a basic to-do list for the user. The module will accept three input fields – task, due date, and priority – and dynamically save that information into an external file for later access. Additionally, the user has the option of viewing and/or removing individual saved records.

# **Code Overview:**

The code for this app is split into six major sections. File handling, main loop/input, add records, view records, remove records, and save & exit. The code is written as one giant block as opposed to utilizing functions in order to demonstrate the “long way” to perform actions pertaining to the records.

# **Section 1 - File Handle:**

Lines 12 – 25 are where we open (or create) the To-Do List text file. At the beginning of this section I set up two local variables – ‘**mainList**’ and ‘**count**’. It’s important, and much cleaner, to explicitly state these at the beginning this section so I can alter them further down in the code. ‘**mainList**’ will be used to store a list of all to-do list items, including the ones previously saved to the .txt file. ‘count’ will help us determine the current number of records, which in turn allows us to assign sequential ID numbers to each record.

Lines 17 – 22 deal with the initial reading and parsing of the external file. This particular open statement utilizes the “**r**” parameter to signify that we will only be reading the contents of the file. Following the initialization of the file handle, the script loops through each record in the file (delimited by ‘\n’ new line characters), increases the previously defined ‘**count**’ variable by one for each item currently existing in the file. Following that

1. import json
2. import pandas as pd

Figure 1. Import statements

# **Load Existing File Data:**

Before getting into the method definitions and the main loop, I want to load the currently existing inventory data stored in the external JSON file into the script. This will allow me to apply ID numbers to each entry. Additionally, it should make the process of viewing existing records much simpler than querying the file each time. (Figure 2)

The count object on line 19 acts as a sort of id number “tracker”. I will explain this in a moment. Lines 21 & 22 utilize the built-in **open** function to read the contents of ‘homeinventory.json’ and, using the json module imported earlier, parsing the raw string of data into a JSON object for processing.

Lines 24 & 25 simply cycle through all items within the JSON object and keeps a count. This is be useful later when adding ID numbers to any additional records added to the file.

Finally, as is good practice, line 27 closes the file handler connection so no accidental changes are made.

1. # Load JSON file
2. count = 1
3. with open("homeinventory.json") as json\_file:
4. inventory = json.load(json\_file)
5. for i in inventory:
6. count += 1
7. json\_file.close()

Figure 2. Load current JSON records

# **ADDING RECORDS:**

The next portion of code (Figure 3) deals with adding new items to the database. Notice that lines 29-50 start with the **def** keyboard. This is a new technique that allows me to more easily separate out various code functionality which, in turn, keeps my main script loop clean and readable. After the function name (AddRecord) is **def**ined, there is a set of parentheses with contains the *parameters* of the function. That is, the variables being passed in for processing within that portion of the script. In this case, I’ll be using the ‘inventory’ JSON object as well as the ID number counter established in the previous step.

On line 30 we establish a new dictionary that will contain the pieces of user\_input received during the next step. This needs to be established within the function so that each time said function is called it starts with a fresh dictionary which is simply a bit more “Pythonic” than overwriting the entries each time. (I think.)

The following portion, lines 32-37 are where the user is prompted for their inventory information. Because the ‘name’ object will be a string, which is already the object type returned by the **input** function, this line can be placed outside of the **try/except** block.

The **try/except** block is important in this circumstance as it helps filter out invalid input – specifically values that are *not* numbers – that might be entered into the ‘quantity’ and ‘value’ prompts. The purpose of this is to attempt (**try**) a chunk of code and handle any errors in a specific way (**except**). In this case, lines 35-37 are attempting to change the object types of the aforementioned prompts into **integer** and **float** values, respectively. If either of these processes throws an exception, the script jumps directly to the **except** statement (lines 49-50) and executes the code there before continuing as normal. This will prevent the script from “crashing and burning”.

1. def AddRecord(inventory, count):
2. new\_record = {}
3. name = input("Enter Name: ").title()
4. try:
5. quantity = int(input("\nQuantity: "))
6. val = float(input("\nValue: "))
7. value = str(round(val, 2))

Figure 3. Define function and request input

Lines 40 – 43 are where the data is committed to the dictionary that was initialized on line 30. Since dictionaries are sets of key/value pairs, the data is added by simply assigning the value to the [KEY] of the dictionary. This is also where the count variable comes into play. ‘Count’ has been set to one more than the number of items already in the JSON file, which corresponds to the ID number of the new record, and so we can simply add it into the dictionary as it was passed in with no alterations. Lastly, the ‘inventory’ list object that contains the current inventory data will have the new record added to the end, which keeps it in numerical order sorted by the ID number. The inventory list is now returned to the outside of the function so it can be manipulated by other methods in the program if needed. (Figure 4)

1. new\_record["ID"] = count
2. new\_record["Name"] = name
3. new\_record["Quantity"] = quantity
4. new\_record["Value"] = "$" + value
5. inventory.append(new\_record)
6. return inventory

Figure 4. Add new values to dictionary and append to inventory list

# **VIEWING RECORDS:**

In the definition of the ViewRecords function on line 52 the only parameter required is the inventory object that was created during the initial read of the file (and that also contains any additional records added during the current session). Lines 53 – 61 are simple print statements that show the user their options for viewing the data and an input statement to store their response. There is a **try/except** block on lines 58 – 61 to ensure the user doesn’t try to enter text at the prompt.

In order to view the records in an attractive and understandable way, I’ve opted to use the **pandas** Python library, which is preferred when working with frames of data. Line 63 takes the raw JSON inventory object that was passed in and formats it in a way that allows for “pretty printing”. Following that, the dataframe object is created.

Lines 65 – 78 are where the code differs depending on the user’s choice. If the user opted to view the most recent records, the script will print the 10 **tail** records from the dataframe that was just created. Otherwise, if the user requested a list of all records, we use the **to\_string** method to convert the *entire* dataframe to something printable in the command line. (Figure 5)

1. records = pd.json\_normalize(inventory)
2. df = pd.DataFrame.from\_dict(records)
3. if user\_input == 1:
4. print("\n")
5. print(df.tail(10))
6. elif user\_input == 2:
7. print("\n")
8. df = df.to\_string(index = False)
9. print("")
10. print(df)

Figure 5. Process JSON for viewing and display records based on user selection

# **Main Loop:**

The reason I’ve coded this as a loop is to avoid making the user re-run the script every single time they want to take an action. This is particularly useful when adding multiple records.

Lines 79 – 80 initiate the loop and ensure that it continues to run until the value of “running” is set to **False** or a **break** statement is reached.

Lines 90 – 111 handle the user input from the main prompt (on lines 82 – 88) and will run the appropriate function (defined above) corresponding to the selection. Notice that in measure 95 the count variable is increased by one. This ensures that as the loop continues, if the user decides to add additional records, the ID number will continue to increase without having to reread the inventory object every time.

The third option (lines 105-111) write the current inventory object to a file for storage. Notice the process is very similar to opening the file one lines 21 & 22. However, instead of using the json.load method, we use the **dump** method, which, as you would expect, dumps the data into the specified file. On the last line “running” is set to **False** which ends the loop and exits the script. The end.

# **Summary:**

Upon executing the script, the command line window will display the following prompt requesting the user’s input. (Figure 6) Depending upon the option selected, there will be different prompts and information shown. (Figures 7 & 8) After performing each task, note how the menu always returns to the initial Add/View/Exit options. This is why the infinite main loop is important.

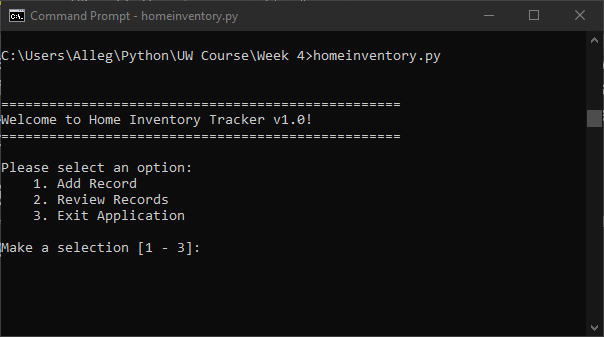


Figure 6. Initial menu via command line

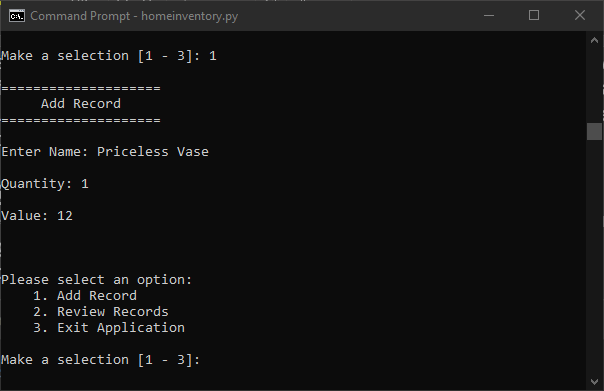


Figure 7. Add Record

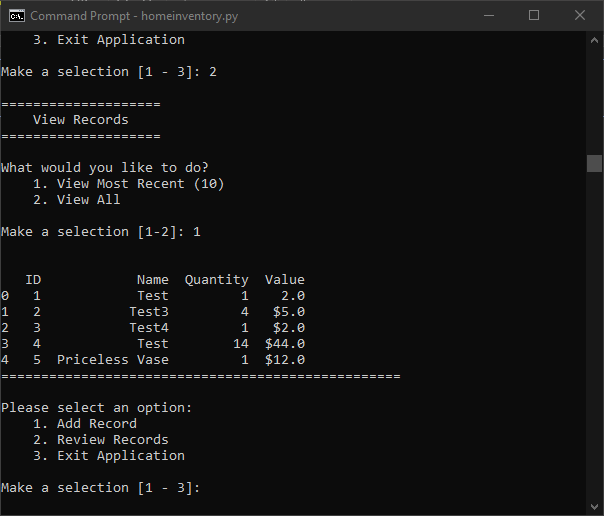


Figure 8. View Records