Justin Calhoun

Professor Summer Rae Elasady

Foundations Of Programming: Python

25 October 2018

Assignment 04 - Household Items Second Version

As this assignment is a variant on last week’s task, I was able to reuse portions of the code. However, in the last version, I had set up the script to save to the file immediately following each user entry, and to meet the requirements this week, the logic needed an overhaul.

Just like last week, I began by importing two modules. The first, re, provides a set of regular expression functions, which I’ll be using to do validation on the price string the user enters. The second, os, provides access to operating system interfaces, which I will be using to clear the screen.

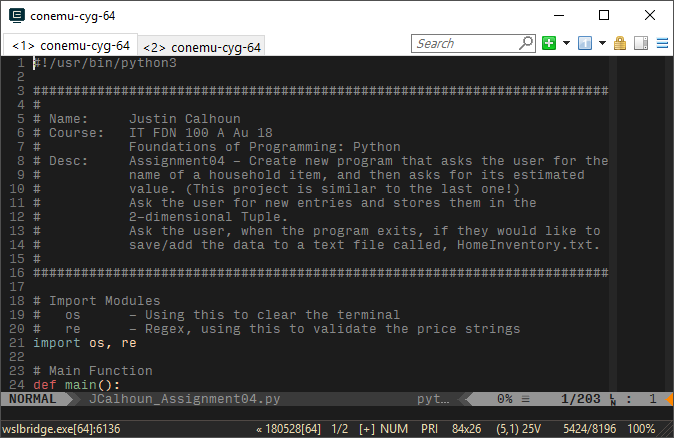


Figure 1: Import modules

I’ll be wrapping my script into a main() function. It’s not necessary for this assignment, but it’s a habit for me. At the beginning of the main() function, I’ll use re.compile to create a regular expression object. Later, we’ll use the .match function to test the user’s input. The specific expression I’m using will match the user’s price string if it meets one of the following criteria:

* The string is any number of digits, followed by a decimal point (period), and then followed by two more digits.
* The string is any number of digits.
* The string is a decimal point, and then two digits.

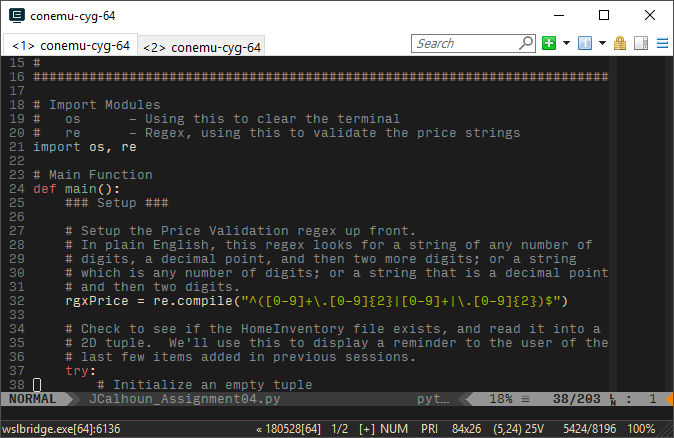


Figure 2: Regular expression

The next thing I chose to do was check to see if the HomeInventory.txt file exists. If it does, I’ll read the current data and save it into a two dimensional tuple, which I’ll use later to remind the user of the latest entries to the file. If HomeInventory.txt does not exist, I’ll create it and put in a header row. To do this, I’m using the try statement: python will “try” to open and read the file. In the event that there is a FileNotFoundError, we will use the except clause to catch the exception, preventing the program from exiting, and create the file fresh instead.

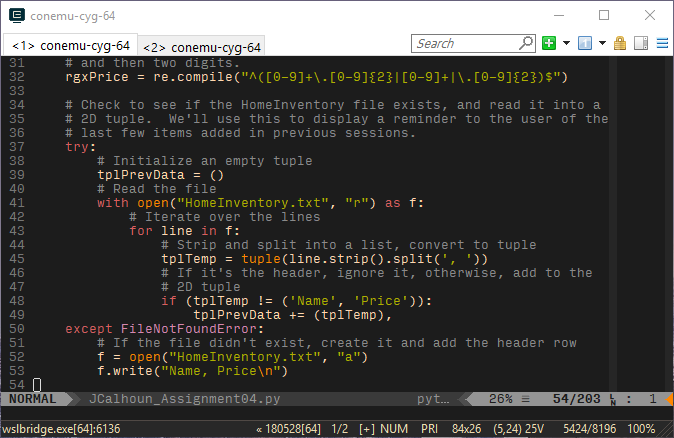


Figure 3: Trying to read the HomeInventory.txt file

With the file setup complete, I first clear the terminal. I accomplish this using the os module to pass “cls||clear” to the host terminal. This works because in both bash and the Windows Command Line, the OR operantor, “||”, will try to run the first command, and if it fails (exits with a non-zero status), attempt to run the second. Windows uses “cls”, while both Linux bash and macOS uses “clear”, so this should work on nearly any system you choose to run the script on. I then display some friendly welcome text to the user: they can press “Enter” to begin adding data, or “Exit” if they choose not to.

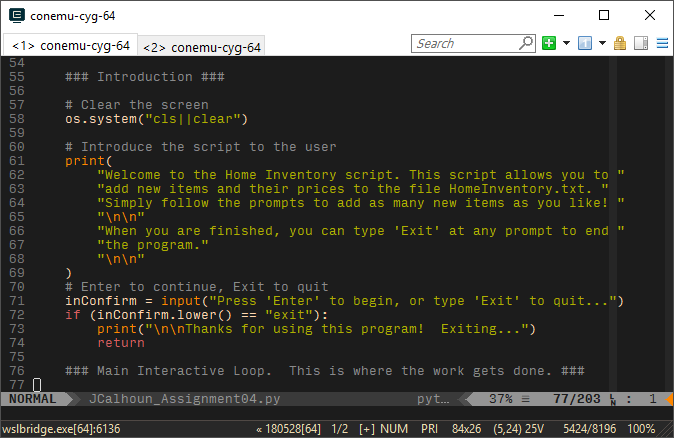


Figure 4: Welcome text

Assuming the user chooses to continue, we move into the meat of the script. I initialize a new empty tuple, then begin a while loop that will continue until the user is done entering data. At the top of the loop, I clear the screen, as I feel this is a bit of a nicer user experience: after each entry, they do not have a long terminal history of prior work, but a concise display of pertinent information.

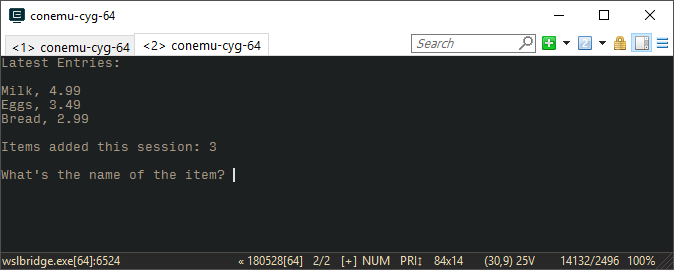


Figure 5: Data entry screen

To accomplish this output, I had to combine any data that might have already been in the HomeInventory.txt file with any new data the user has added during this session. I use a slice to only keep up to the last five items of the combined tuples. Then, if the length of the new tuple for the display data is zero, I print a special message about there being no data yet. Otherwise, I use a simple for loop to unpack and display the data to the user. Finally, I print a message to the user letting them know how many items they’ve entered this session, as a subtle reminder that they may wish to save at some point.

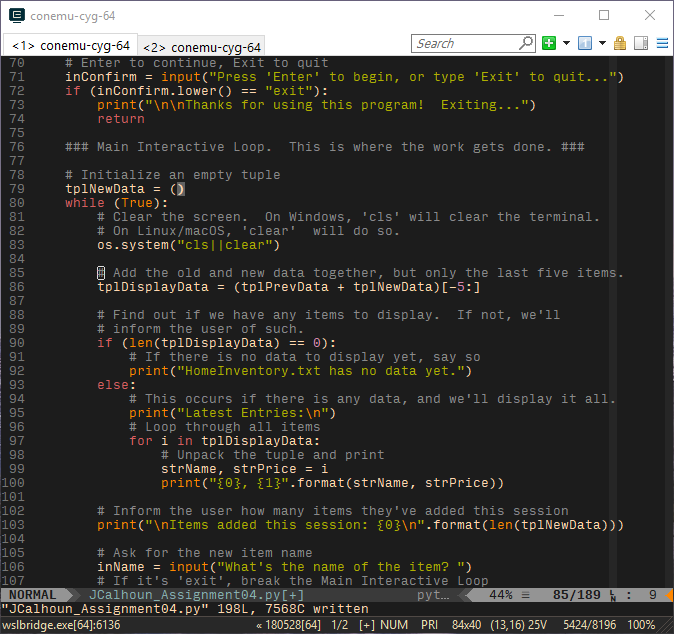


Figure 6: Display data

With the header information printed to the terminal, I finally ask them for new data. For the name of the new item, any string will do. If they type “exit”, however, I’ll break the current while loop and move on to asking them to save.

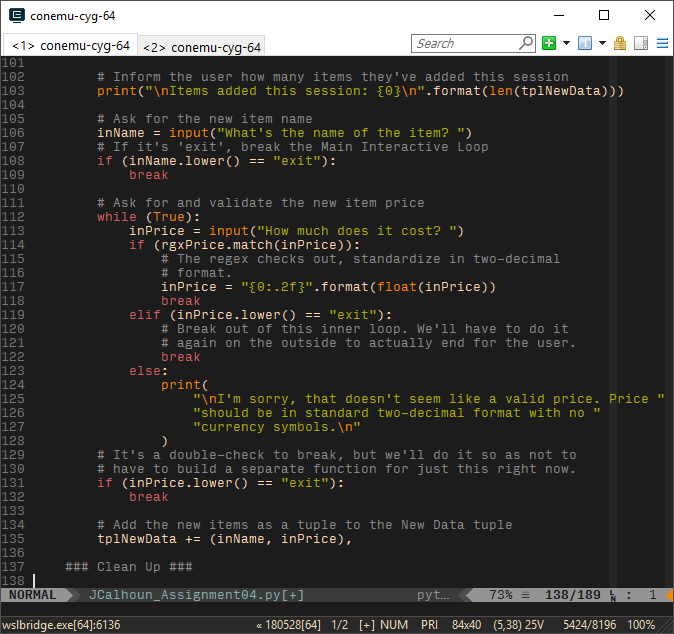


Figure 7: Input the name

Once I have the name, I nest another while loop, and start asking for the price. In this case, though, I want a string that is actually a price. Once the user enters their string, I use the re module’s .match function to test it against the regex string I compiled back at the beginning of the script. If it matches, I use some trickery to normalize the string: I convert the string into a float (which I know will work, because it passed the regex), then turn it back into a string, using the format function’s mini-language to set the precision to two decimal places. In this way, even though the user can enter the price in a variety of ways, I always commit it in a standard format.

If the user typed “exit”, I break the current loop, but have to break the outer loop later with another check. This is another similar if statement just past the price validation while loop.

Finally, if I don’t have a price string I can use, and they didn’t type “exit”, I simply let them know and start over.

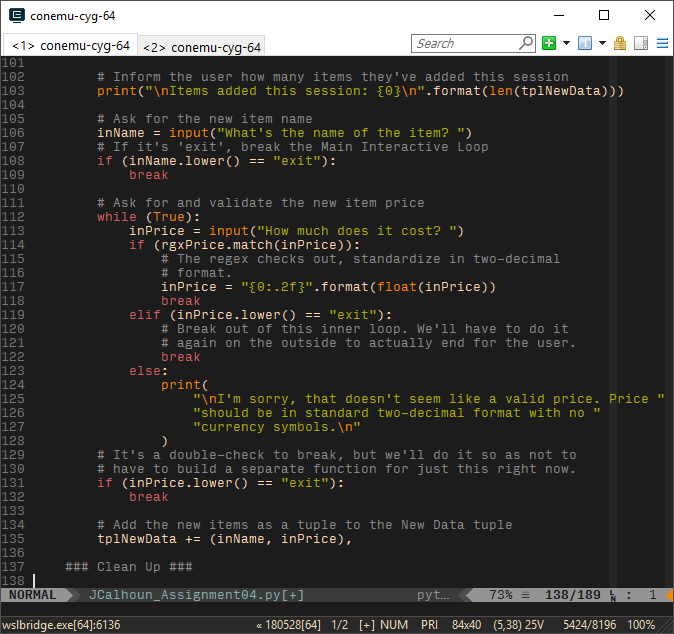


Figure 8: Price string validation

With everything else done, the final action in the loop is to add the user’s input to the two dimensional tuple tplNewData.

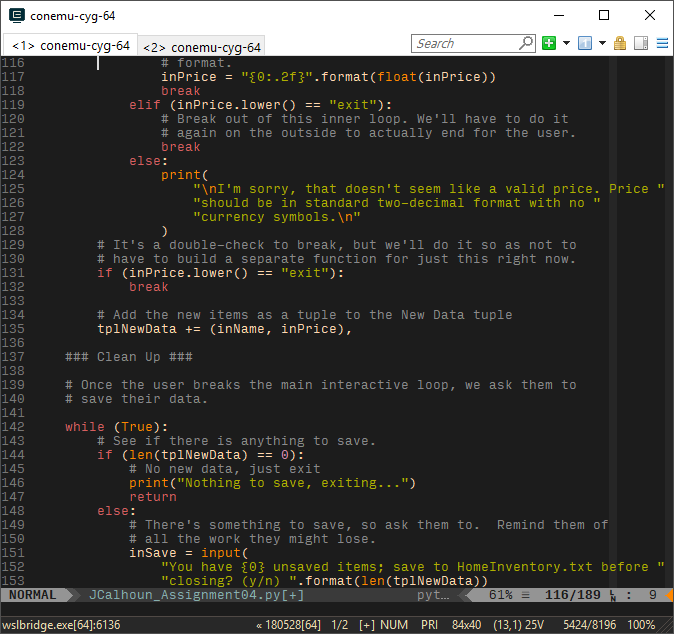


Figure 9: Stash the input

Once the user finally breaks the data entry loop, I start the cleanup and exit. I begin another while loop, and then an if statement: check if there’s anything to save at all. If they never entered any data, I print a quick message and exit the function (and the script) with a return statement.

If they have entered data, I ask the user if they want to save. I will only accept an explicit answer, so using another if statement, I check to see if they said “yes”. If so, I open the HomeInventory.txt, then use a for loop to unpack each tuple in the tuple and write the data to the file, one line at a time. After that’s complete, I print a note confirming, and return to exit the script.

In the event they said “no”, I instead print a message that the script is exiting without saving, and then return.

Finally, if I don’t get an answer from the user that is explicitly a “yes” or “no”, I tell them so, let them know the valid options, and the loop will restart to ask again.

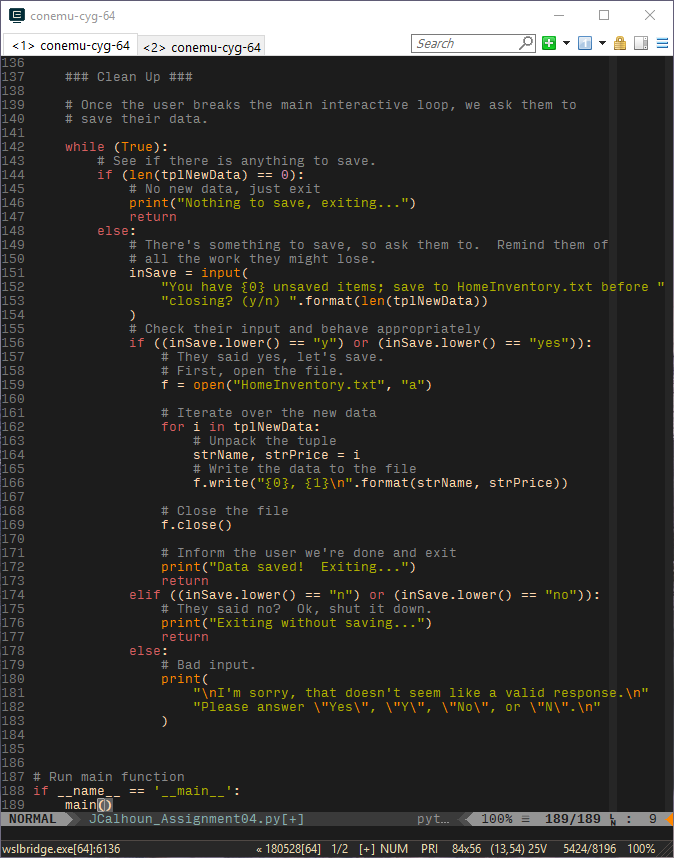


Figure 10: Save logic

This is the end of the main() function. All that’s left to do is add some code outside the function that will ensure the main() function is run. \_\_name\_\_ is a special variable that the python interpreter sets when executing a script. If this file is directly run, \_\_name\_\_ will equal “\_\_main\_\_”; essentially, if you run this script directly, it will always execute the main() function, but if you instead wanted to import this script as a module into another python script, it’s \_\_name\_\_ would be something else, and thus not run this code until your other script asks it to.

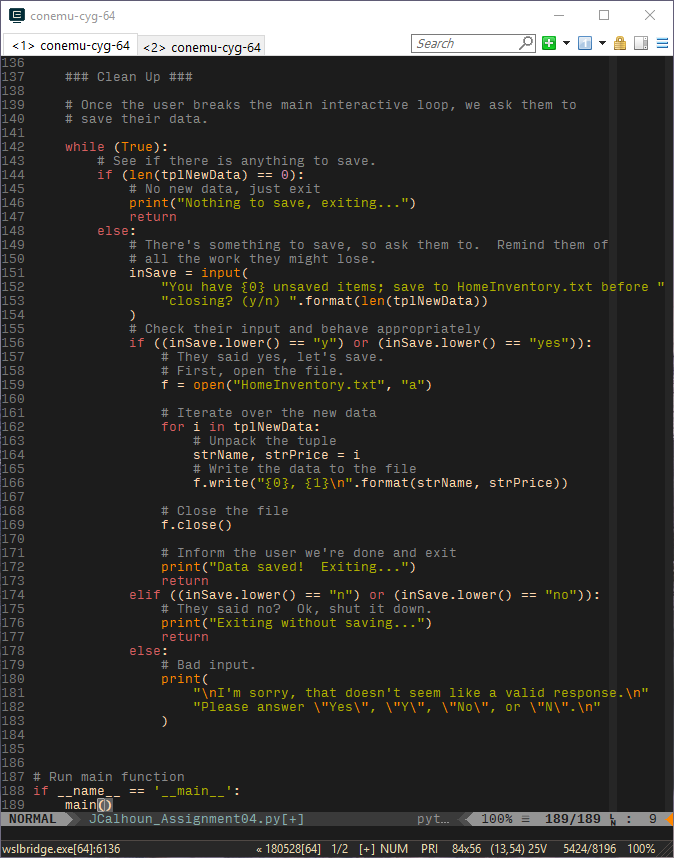


Figure 11: Running the main function

With that, the script is complete. Were I to continue working on it, there are some small issues with how the text is displayed at certain terminal sizes that I’d like to find fixes for. I’d also add more logic to save without having to exit, cancel/fix an entry if you make a typo, and perhaps better ways to browse all the existing data.