Chloe Wei
June 6, 2020
IT FDN 100 A
Assignment 08
GitHub Mod08 Repository

Utilizing An Object Class

For this module, I will show the difference in the backend without effecting the front of the user interface, when indirectly utilizing class methods or functions as well as directly from the main source code.

Product Database

Layout the Intention of the Script

I will need to take in a new produc/price, present the database of products/prices, and save the database to a text file (which must also be read back to the systems memory for augmentation).

Create a Pseudocode & Flowchart

This project will have three classes, one of which will act as the object class (Product), processing class (FileProcessor) and input/output class (IO) and data in memory will be compiled as a list:

```
Run Assignment08.py file
Read 'Product.txt' to RAM
Close file
Load DB Menu
        If 'View Inventory' selected
            Read DB from RAM
            Return DB list
        If 'Add Product' selected
            Try/Except:
                Add <Product_Name>          # letters not numbers
                        Exception as e: 'must enter a product name'
                Else: add product to list
            Add <Price> # flt or str
        If 'Save List' selected
            Overwrite RAM to 'Product.txt'
        If Exit, dbl check
            Return 'Save data or lose changes to db'
            if 'y': save before exit
            else: Return 'new data will be lost'
```

End session
Close Terminal/Window prompt

Based on the pseudocode, the flowchart for my Product database will look like the following:

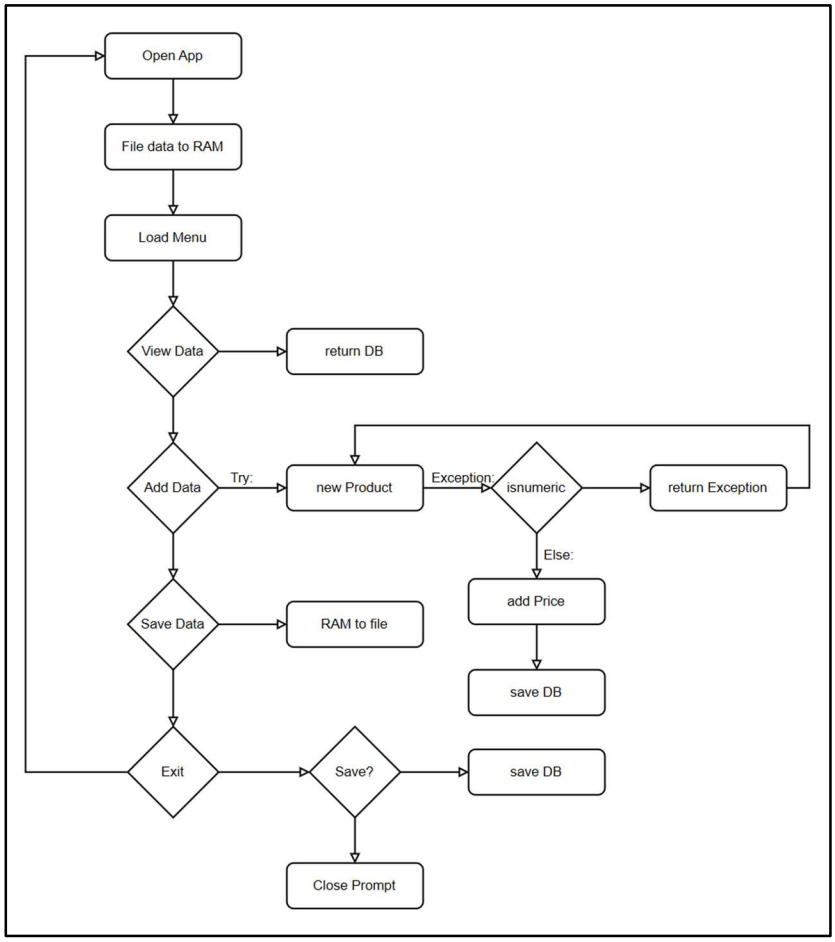

Figure 1: Product flow chart

Starting with the `class FileProcessor()`, I will start by testing the functions to read and overwrite the data to/from the file ('product.txt'), then using class 'IO', to view or manipulate the database, and finally exploring the syntax for our object class, 'Product'.

File Data

My previous codes (and data) were small, so I only accessed the stored data from the text file when 'View Data' option was selected. However, every 'view' instance requires an unnecessary step and use of memory to open, read, then close the file. To rectify this, I am using the 'with' statement (figure 2) prior to loading the user interface, UI, with the Product Menu.

```
list_of_product_objects = []
try:
    with open(file_name, "r") as objF:
        for row in objF:
            list_of_product_objects.append(row.strip().split(","))
except IOError:
    print("\tProduct database is empty.\nAdd new products.")
    with open(file_name, "a") as objF:
        lstrow = ["Product", "Price"]
        objF.write(str(lstrow[0]) + "," + str(lstrow[1]))

return list_of_product_objects
```

Figure 2: Read data from 'Product.txt'

If the ‘products.txt’ file already exists in the folder, it will load the data into the local list variable, ‘list_of_product_objects’, in the system’s memory for user interface, UI, manipulations. If the file does not yet exist in the folder, a product file will be created to take in new data from the UI (figure 3). To test my write to file, I also put in starting data: “Product”, “Price” which was removed after testing. Saving the data from memory will only occur when selected from the UI when ‘Save Data to File’ is selected or prior to exiting the UI (but as an optional selection).

```
Product database is empty. Add Products using Menu.

        **********
        Product Menu:
        1) View Products
        2) Add A New Product
        3) Save Data to File
        4) Exit
        **********



What would you like to do? [1-4]:
```

Figure 3: PyCharm UI return exception

The UI will have 4 options: ‘View Products’, ‘Add A New Product’, ‘Save Data to File’, and ‘Exit’. Our first option, ‘View Products’, will return the data to the UI if products exist in the list table, ‘lstOfProductObjects’. If the table is clear of data (because a new text file was generated, or no previous data was entered) the UI will prompt for new entry and return the user menu.

```
C:\Users\WeiVar20\AppData\Local\Programs\Python\Pyth
    Product database is empty.
Add new products.

        Menu
        1 - Display Product Price Inventory
        2 - Add New Product
        3 - Save Session
        4 - Exit Session

Which option would you like to perform? [1 - 4]:
```

Figure 4: Empty Database in RAM

However, now that the session has begun, a new file was created in the background with our sample data. The return to the UI is to let users know that the database is brand new and clear. Now that the file is loaded into the shared path, I can now access my sample data:

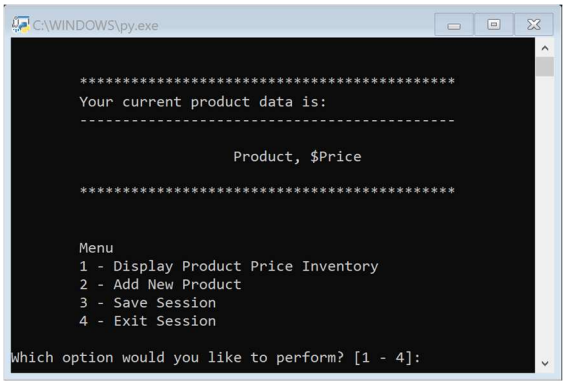

Figure 5: Sample data read from file

Now that I know creating a new text file, reading the data from the file to memory, and displaying the data works, I can now focus on passing in data before I streamline the display to the UI.

Indirect Methods

Because I only need the functionality in the ‘FileProcessor’ and ‘IO’ class, we will keep static method for the functions. However, I will need to access the method in the ‘Product’ class, so I use the variable ‘objProd’ to take on a new object from the ‘Product’ class.

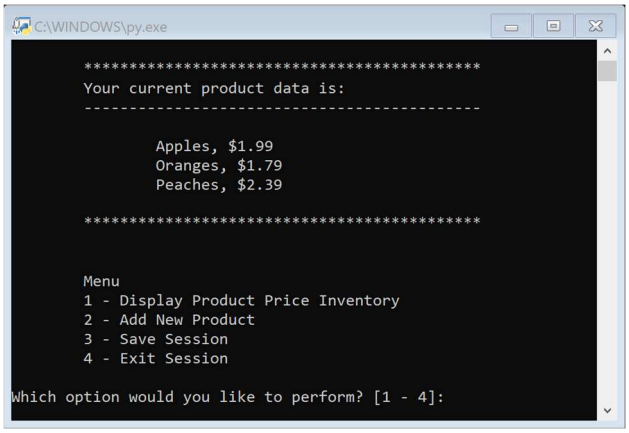

Figure 6: New list comprised of 3 new data objects

Product Class

The Product class will convert the different data types for our input variables ‘product_name’ and ‘price_name’, from a string and float respectively, to one unified string.

```
# -- METH --
def __str__(self):
    return self.prodName + "," + self.prodPrice
```

Figure 7: Returns a new string every instance

It will also verify the arguments pass to the ‘product_name’ are not numbers and the ‘product_price’ is a number (float).

```
# -- PROP --
# Property getter/setter for Product Name
@property
def prodName(self):
    return str(self.__prodName).title()     # Title case

@prodName.setter
def prodName(self, pName):
    """ Ensure the Product Name is not a number

    :param pName: (string)
    :return:
    """
    if str(pName).isnumeric() == False:
        self.__prodName = pName
    else:
        raise Exception("Product Names cannot be numbers")

# Propert getter/setter for Product Price
@property
def prodPrice(self):
    return str(self.__prodPrice)

@prodPrice.setter
def prodPrice(self, pPrice):
    if str(pPrice).isnumeric == True:
        self.__prodPrice = pPrice
    else:
        raise Exception("Product Price must be a number.")
```

Figure 8: Properties of 'Product' class

When the ‘Add New Product’ is selected, the application will access the functions in the IO class, pass the arguments to the ‘Product’ class. The values will then be checked that they are the desired types, then assembled as a string of data which is then passed as to ‘objProd’ as a new object to add to the database.

Summary

Regardless of the approach towards the methods and functions in the class, the effects to the UI is not noticeable. However, my organizing my definition functions according to how data is handled, allows me to break them into manageable chunks of code, incorporate those same functions in other scripts, but now also manage how I store new data in memory.