Alex Frain

March 8, 2021

IT FDN 110 B – Foundations of Programming, Python

Assignment08

https://github.com/alexfrain/IntroToProg-Python-Mod08

Working with Classes

# Introduction

In the assignment this week we start delving into the world of object oriented programming and look at classes, objects and creating our own custom classes. The script reads in a list of product objects from an external file into memory and then provides the user choices to add additional products to the list, view the current list of product objects in memory, save the updated list of product objects back to the file, and reload the list of product from the file. Since we are dealing with Python objects which is a complex form of data I also imported the pickle module so that I could serialize and de-serialize the list of product objects when reading from and writing to a file.

# What are Objects and Classes?

Our text book, Python Programming Third Edition, did a very good job of explaining objects and classes in an easy to understand manner. Objects in programming can be looked at as a means of representing real life objects in software. Like in real life, objects have properties that can be used to describe them and they have methods that give functionality to the objects. An object is created from a class which acts as a blueprint. You can create multiple objects from a class and each of these objects will have the same properties/characteristics but the property values for each object can differ from one another. The best representation of this from the assignment script this week is the Product class we had to complete. The class include two properties that provide characteristics to product objects created from the class, a name and a price. I didn’t create any class methods that would give these product objects functionality aside from a method to change the way the object is represented as a string (more on this in the Creating the Script section).

# Creating the Script

We were provided with a starter template that had three predefined custom classes (Product, FileProcessor, and IO) that we had to add properties and/or methods to and addition to completing the main body of the script. These classes also provide a means for us to easily handle Separation of Concerns where the Product class handles data, the FileProcessor class handles data processing, and the IO class handles presentation.

## Product Class

The Product class will allow the script to create product objects that have a name and price. The docstring for the Product class was included as part of the starter template which provided the properties this class contains and we were left to add code to fill out the remainder of the class. I started by adding comments for the 5 main components of a class, Fields, Constructor, Attributes, Properties, and Methods.

I started by creating the constructor and adding two attributes, product\_name and product\_price (Figure 1). I made them hidden by prepending double underscores to the attribute name since the docstring indicated we’d use properties to access those attributes.

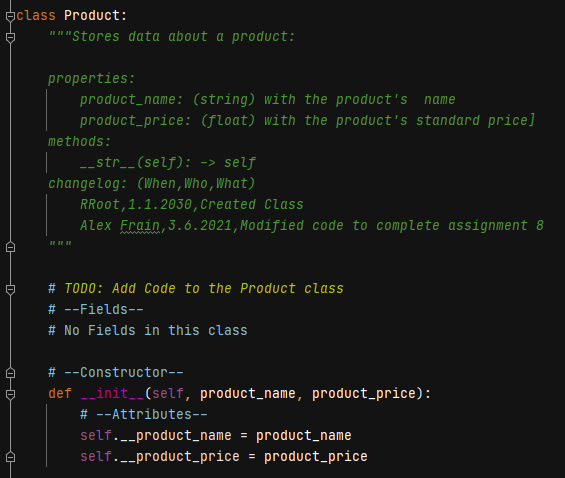


Figure 1: Product Class Constructor

I then added the class properties (Figure 2) for product\_name and product\_price. Each has their own getter property that returns the product object’s name and price respectively. There’s also a setter property for each that accepts a value as a parameter that will be used to set a product object’s name and price but I also included exception handling to check validity of the values. In the product\_name setter I check if the name only contains numbers and raise an exception if this is the case since the name should at least contain one letter. In the product\_price setter I use a try except clause to verify the value passed in can be converted to a float and provide an error if it cannot.

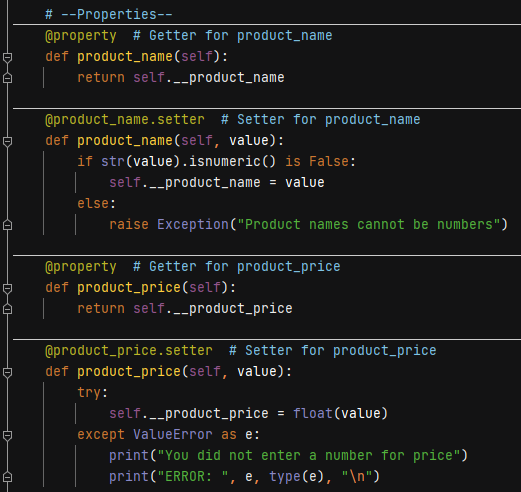


Figure 2: Product Class Properties

Lastly, I add one method to the class called *\_\_str\_\_*. This double underscore method (or dunder for short) is a Pythonic convention and is normally included as a part of all Python objects. This particular method returns a string representation of the object by default. By creating a *\_\_str\_\_* method in this class I am overwriting the default method and have it return the product object’s name and price as a string if, for instance, the product object were to be used in a *print()*.

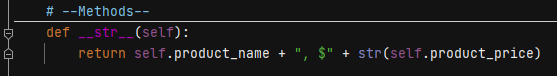


Figure 3: Product Class Methods

## FileProcessor Class

The FileProcessor class contains two methods that handle reading in a list of product objects from a file and saving a list of product objects to a file. Since these methods handle processing data I created them static methods as there isn’t a need to create multiple FileProcessor objects to handle reading and writing the product object lists nor is there a need to modify the class’ state.

Since we are asked to read and write a list of product objects to and from a file I decided the best way to address this is to pickle the data so I imported the pickle module at the beginning of the script. Pickling will allow the script to serialize the list of objects and it’s hierarchy to a bite stream that can then be written to a file and then de-serialized when read from a file to easily recreate the list and product objects contained within.

The *read\_data\_from\_file()* method (Figure 4) takes in a file name string as a parameter and uses the pickle’s *load()* method to read in the data from the file and de-serialize it. The de-serialized list of product objects is assigned to a list and the list is returned by the method. I also include exception handling in case the file name passed to the method does not exist and provide a custom message to the user indicating this.

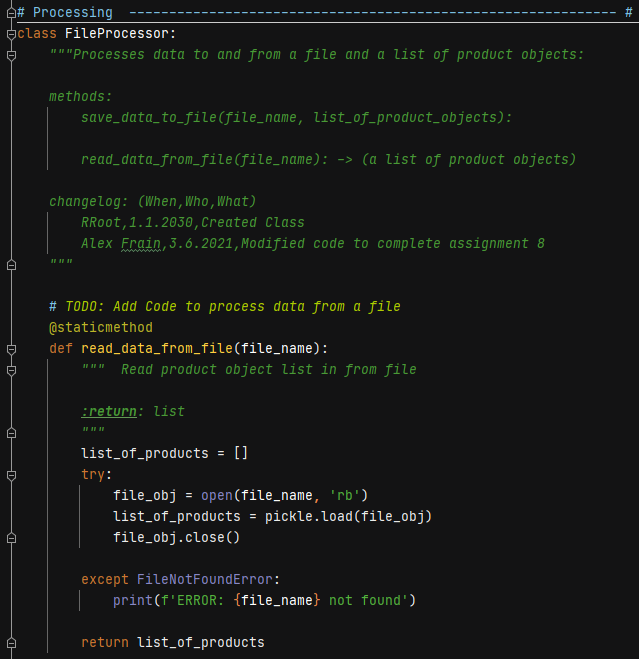


Figure 4: FileProcessor Class' read\_data\_from\_file method

The *save\_data\_to\_file()* method (Figure 5) takes in a file name string and list of product objects as parameters. It uses the pickle’s *dump()* method to serialize the list of product objects and write it to the file. Nothing is returned by this method.

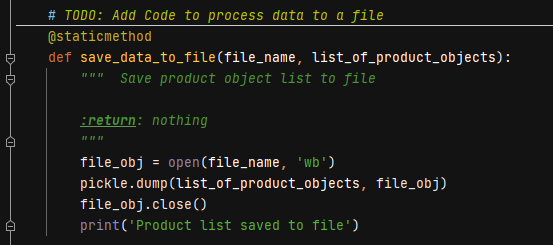


Figure 5: FileProcessor Class' save\_data\_to\_file method

## IO Class

The IO class addresses the presentation layer of the script handling input and output. In the starter template we had to create the class’ docstring and provide code to handle four pieces of functionality, show a menu to the user, get the user’s menu choice, show the current data to the user, and get product data from the user. Each of these pieces equated to a class method that I created as static methods like I did in the FileProcessor class for the same reasoning explained there.

I started with the docstring which includes a brief description about the class and the method names included in the class. I followed that up by creating the *print\_menu\_product()* method (Figure 6) to handle showing a menu to the user. It prints a string of 5 menu choices and returns nothing.

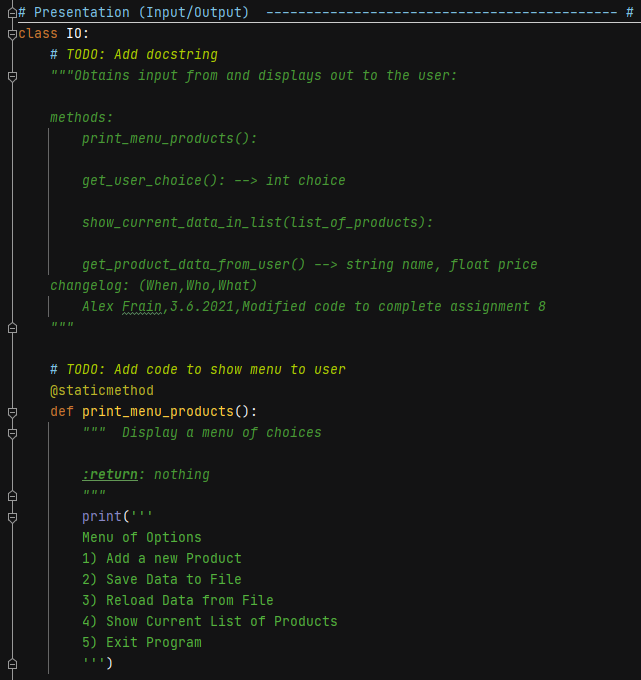


Figure 6: IO class' docstring and print\_menu\_products method

I created the *get\_user\_choice()* method (Figure 7) to handle getting the user’s menu choice. I use a while loop and ask the user to input a number between 1 to 5 which corresponds to a specific menu choice. I convert their input to an integer and use the *range()* function to confirm the number they entered falls within the valid range. If it does the loops is exited via a *break* statement and the user’s integer choice is returned by the method. I also use exception handling in the method to catch any non integer input from the user and provide them with a custom message indicating a number needs to be entered.

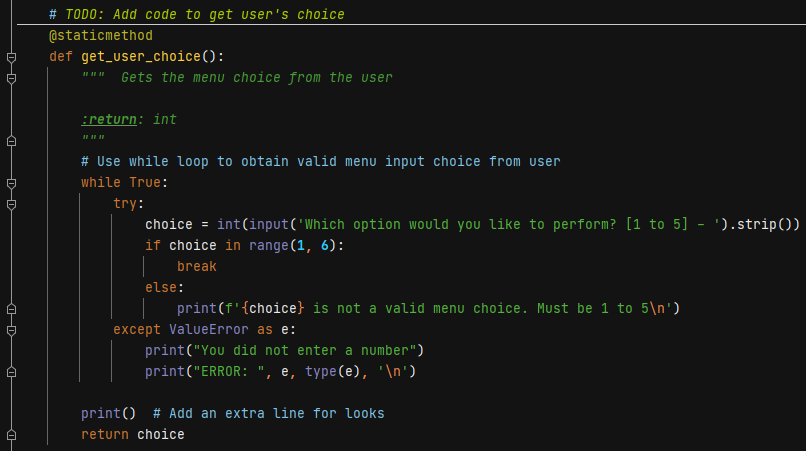


Figure 7: IO class' get\_user\_choice method

The *show\_current\_data\_in\_list()* method displays the current list of product objects in memory. It takes in the list of products as a parameter and first checks if the list is empty. If it is a message is printed stating this. If this list is not empty it is looped through and each product object is printed which uses the product class’ *\_\_str\_\_()* method that I created to display the product name and price. Nothing is returned by this method.

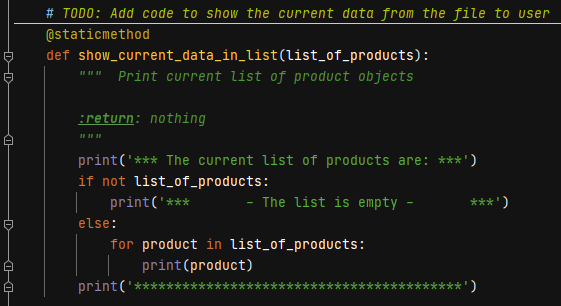


Figure 8: IO class' show\_current\_data\_in\_list method

The last method in this class is the *get\_product\_data\_from\_user()* method (Figure 9) which gets a product name string and float value for price from the user and returns them. I use two separate while loops to get the different input values from the user and break out of the loops if the input is valid. When they enter a product name I check if the name contains just numbers by using the string’s *isnumeric()* method or if they enter any characters that don’t include a letter by using the string’s *islower()* method to see if it’s False which would indicate there are no letters in the name. I raise exceptions in both cases and provide custom messages for each. I also check if the user enters an empty string which is also not valid for a name and provide a message indicating this. In the loop where the user enters the product’s price a *try except* block is used to catch a *ValueError* in case the user enters a non numeric value.

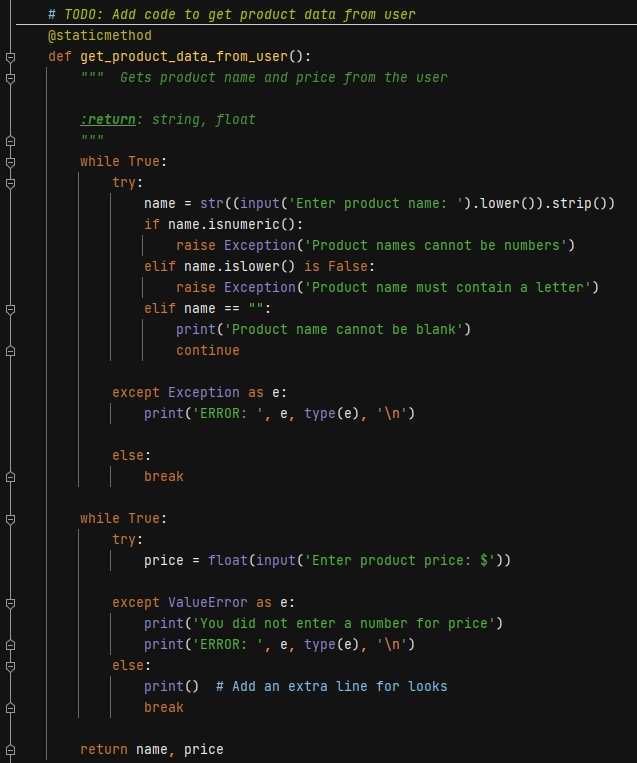


Figure 9: IO class' get\_product\_data\_from\_user method

## Script Main Body

I start the main body of the script by reading in a list of product objects from a file into memory by calling the FileProcessor’s *read\_data\_from\_file()* method (Figure 10). The script then either displays a message stating the list is empty if the list returned by the method is empty or the IO’s *show\_current\_data\_in\_list()* method is called to display the list of product objects.

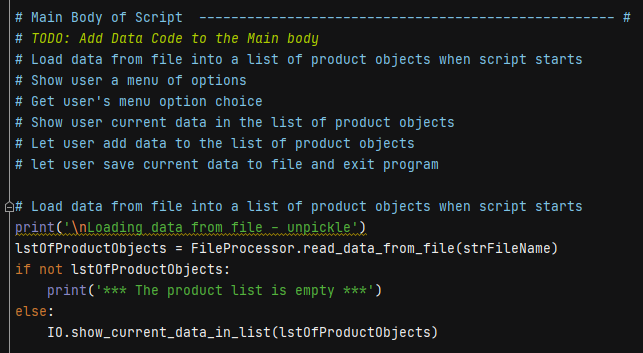


Figure 10: Beginning of Main Body

The remainder of the script body is contained within a *while* loop (Figure 11) that first calls the IO class’ *print\_menu\_products()* method to display a list of menu options to the user. I then get the user’s menu choice by calling the *get\_user\_choice()* method and assign the returned *int* to the *choice\_int* variable.

I use a block of *if elif* statements to compare the integer value entered by the user to determine which menu selection code to run. If the user entered a 1 to add a new product I call the IO’s get\_product\_data\_from\_user() method to get the product data from the user and I assign them to a *name* and *price* variable. I create a new product object called *prod* passing in the *name* and *price* variables so they can be set as the new object’s attributes. This new product object is then appended to the list of product objects.

If the user entered a 2 I call the FileProcessor’s *save\_data\_to\_file()* method to save the list of product objects to a file.

If the user entered a 3 I call the FileProcessor’s *read\_data\_from\_file()* method again which overwrites the list of product objects in memory with the list from the file.

If the user entered a 4 I call the IO’s *show\_current\_data\_in\_list()* method to display the list of product objects.

Finally, if the user entered a 5 to exit the script I use a *break* statement to exit the while loop and the script is finished.

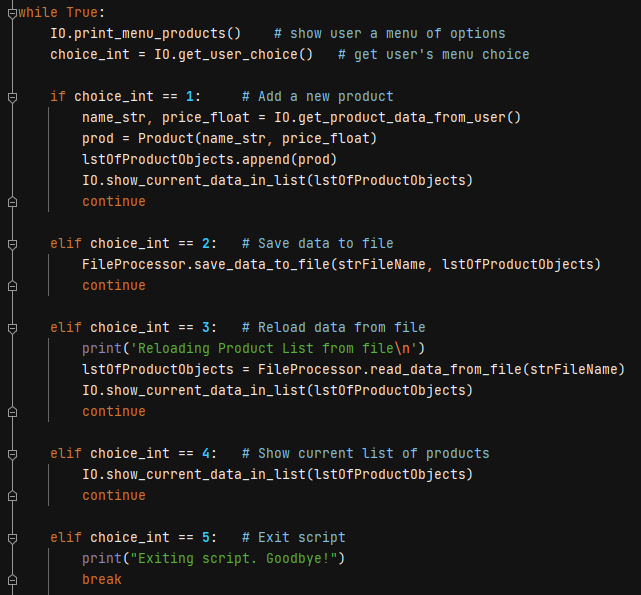


Figure 11: Main Body While Loop

# Running the Script

I ran the script in both PyCharm and Windows Command line to confirm it worked with both. I went through the same verification in both environments by running through each menu option verifying results with both valid and invalid input.

I started the script without having a products.dat file on my computer to confirm I would receive the custom exception error (Figure 12). Since the file does not exist the list of product objects is empty and a message indicating this is displayed followed by displaying the menu of options available to the user.

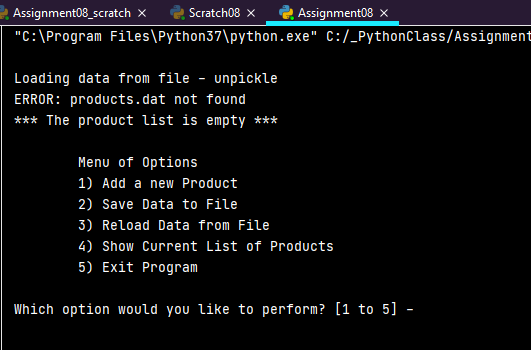


Figure 12: Initial start of script with no file to read in from

I then test the handling of invalid menu option input which includes integers out of range and non numeric characters (Figure 13).

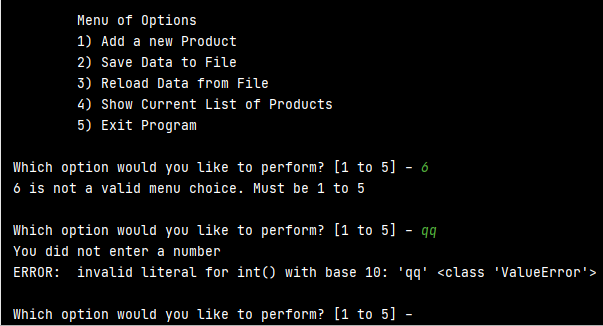


Figure 13: Handling invalid menu option input

Next I test option 1, add a new product, to confirm I could add a product as well as handle exceptions for invalid name and price input (Figure 14).

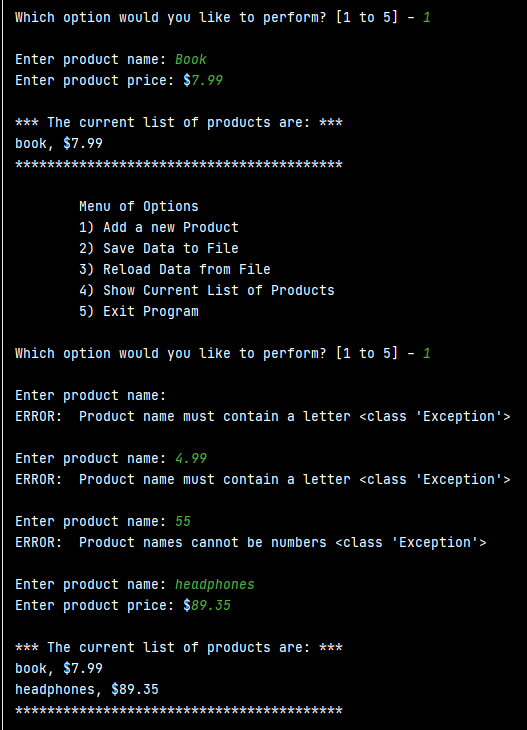


Figure 14: Adding a new Product and handling exceptions

I test option 2, save data to file, and option 5, exit the program, to confirm the products.dat file gets created in my local directory and the list of product objects is saved to it (Figure 15).

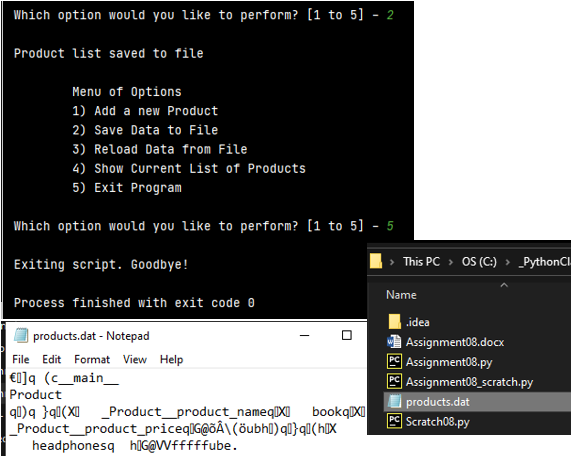


Figure 15: Save data to file

I started the script again and now am able to the file is opened, the list of product objects is unpickled, and loaded into memory, and then displayed to the screen (Figure 16).

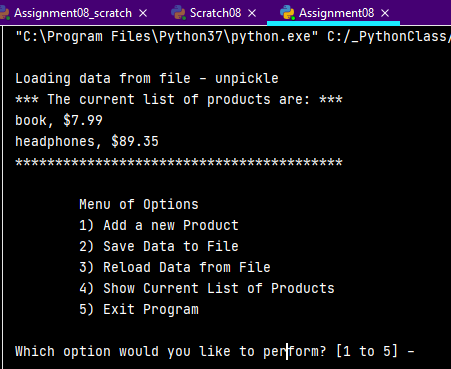


Figure 16: Unpickling data from file and printing it to screen.

I then test option 3, reloading data from file (Figure 17) by first adding another product object to the list. I select option 3 and confirm the current list of product objects is overwritten with the list from the file.

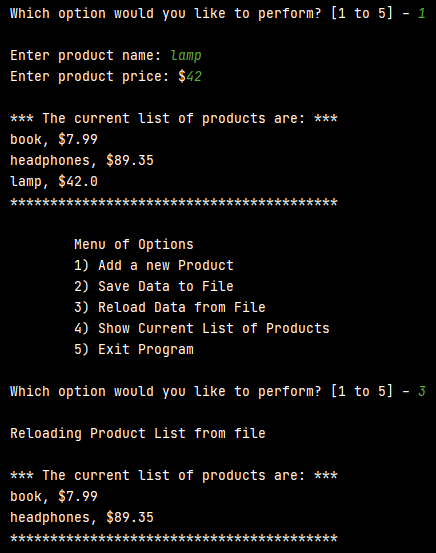


Figure 17: Reloading data from file

Lastly, I test option 4, show current list of products, confirming it successfully displays the list of product objects (Figure 18).

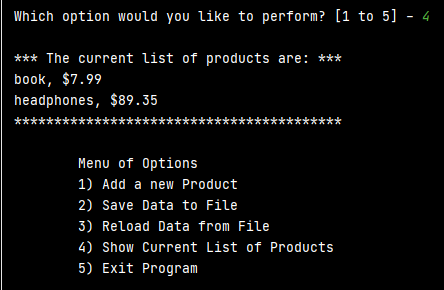


Figure 18: Show current list of products

Figure 19 shows the script running in Windows Command line and the updated products.dat file after having added additional product objects to the list and saving it to the file.

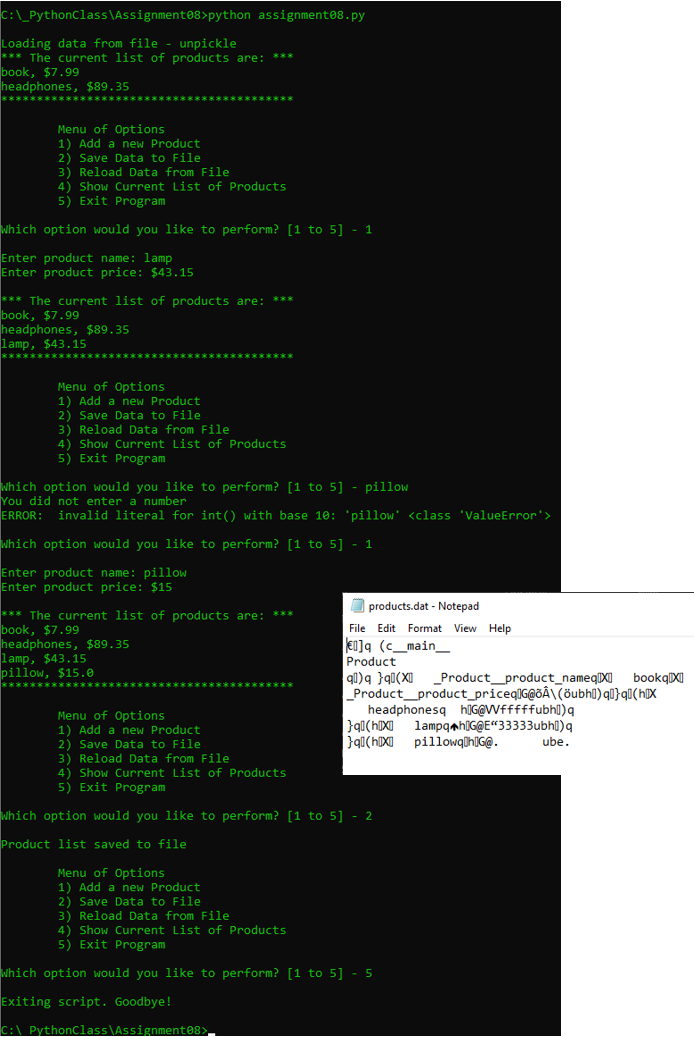


Figure 19: Script running in Windows command line

# Summary

In this week’s assignment we were provided with a starter script template consisting of 3 classes that we had to complete. One class is used to create Product objects that have a name and price property. One class handles reading a list of product objects from a file and writing them to a file. The final class handles any input required by the user as well as output to the screen. These classes give us a means of separating the script’s concerns: data, processing, and presentation. The remainder of the script reads in a list of product objects from a file, gives the user options to create new product objects and add them to the list, reload the list from the file overwriting anything they’ve added but not saved, and save the list of product objects back to a file. The two primary concepts from last week’s assignment, pickling and exception handling, are also used in this assignment in order to serialize and de-serialize the list of objects when reading from and writing to a file and to catch any exceptions that may occur with incorrect input from the user.