Patricia Coonrad

June 14, 2020

IT FDN 100 A Sp 20: Foundations Of Programming: Python

Assignment 08

GitHubURL

Working With Classes & Objects

# Introduction

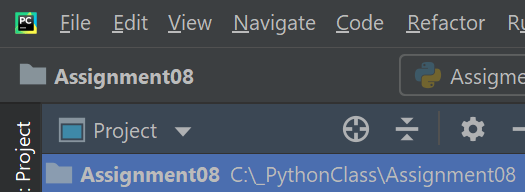
In this assignment I will describe the steps I took to apply the use of classes and objects to the Python program created for Assignment 08. I will demonstrate some of the concepts I learned during the course, such as how to make a class, use of class components: fields, constructors, attributes, properties, methods, objects, use of keyword ‘self’, static methods, private attributes, type hints, DocStrings, to name a few.

The objective of the assignment was to read and understand the pseudo-code provided in the starter file, and add code that included classes and error handling to make the application work. The script will allow the user to manage a list of product names and prices. Once the program is started, the system will load the contents of an existing ‘products.txt‘ file into memory. The program will then perform different actions depending on the menu selection made by the user. The user will be able to view the existing list of product names and prices, add new product and price to the list, save the changes to the ‘products.txt‘ file, and exit the program.

# Add Code to Assignment08\_Starter.py file

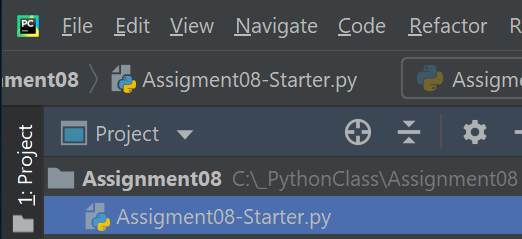
In this exercise, I will detail the steps I performed to add code to Assignment 08 Starter file. I will run the script and capture the image of the script execution using both PyCharm and the Command Window.

1. As per directions, I created a new Project in PyCharm that uses the \_PythonClass\Assignment08 folder as its location (Figure 1).



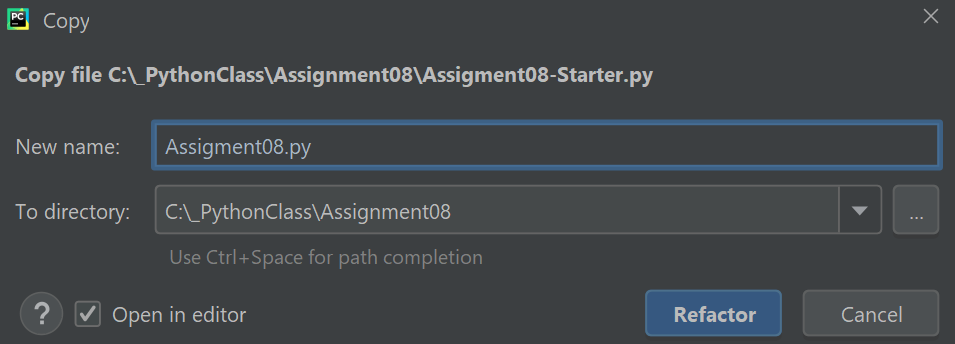
***Figure 1. Assignment08 PyCharm Project***

1. I added the starter file ‘Assignment08\_Starter.py’ to the project. This file was provided by instructor Randy Root and only included the pseudo-code (Figure 2).

****

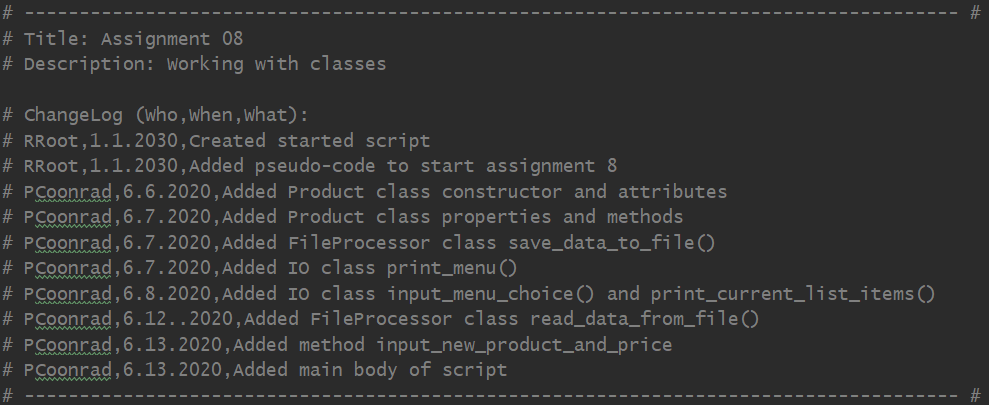
***Figure 2. Adding the ‘Assignment08\_Starter.py’ file to the project***

1. I made a copy of the template file and named it ‘Assignment08.py’ (Figure 3).



***Figure 3. Copying the starter file and naming ‘Assignment08.py’***

1. Once the ‘Assignment08.py’ file was created, I started modifying the script. I updated the script header change log to reflect the changes I made to the code. The script header is a comment at the beginning of the script. It includes information about the script such as title, brief description of the program, and change log. (Figure 4).



***Figure 4. Assignment08 Script header***

1. I first added code to the Product class under the Data layer. In this program, the Product class is used to organize Product data which are variables or constants (aka field). I updated the DocString provided in the starter file with additional information regarding the code I was adding.

The Product class will store data about the product such as name and price. It will serve as the blueprint, the template for creating the product objects (instance). It will define the characteristics (attributes) and behavior (methods) of the objects. Each object created using this class, will have the same basic structure.

I followed the standard class pattern which usually includes the following components: Fields, Constructors, Attributes, Properties, and Methods.

* Fields are used to store the data, and they are variables or constants in a class.
* Constructors are a special method usually used to set up the initial values of the class.
* Attributes are similar to fields, they are a virtual field. Once you use an attribute, you don’t use the fields anymore.
* Properties are a special type of method used for managing the data (fields and attributes) in the class.
* Methods are functions inside a class, they are used to perform a specific task.

The Constructor is a special method that is automatically executed when a new object is created. It sets (assigns) the initial values of the fields when the object is created.

In Python, the syntax of the constructor is:

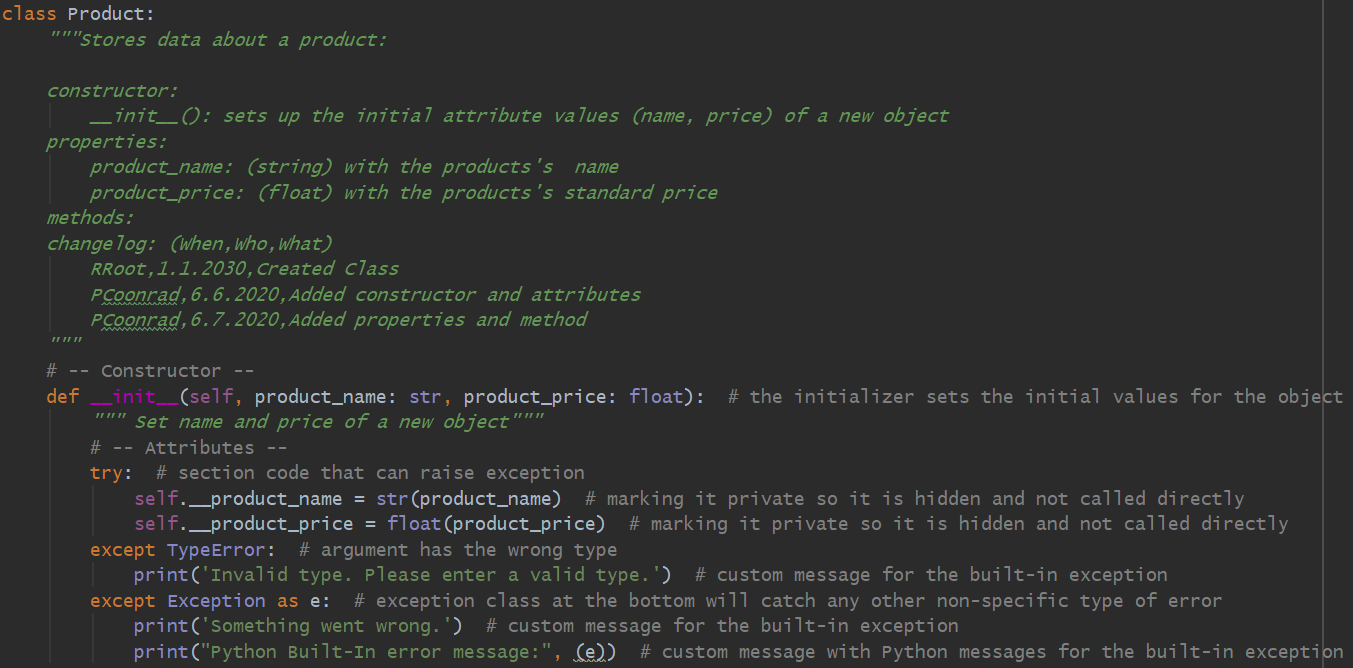
def \_\_init\_\_(self):

# body of the constructor

By convention, every instance method must have a special first parameter called ‘self’. It is used to refer to the object instance itself and not the class code loaded into memory. It allows it to access the attributes and methods of the object.

In the script, I created the constructor method for the Product class and added the additional parameters product\_name and product\_price to the method definition. I used type hints (: str and : float) to indicate what type of data I expect to see for the variables product\_name and product\_price respectively. I then used the constructor parameter values to set the product\_name and product\_price attributes, and Python automatically created the “virtual or invisible” fields to hold data. This is the reason why I didn’t set up the Fields before I set up the constructor. I created private attributes by adding double underscores at the beginning of the attributes \_\_product\_name and \_\_product\_price. This will limit direct access to the attribute from outside the Product class. By marking it private, the system understands that it is not supposed to use the attribute directly, it should use it through the properties. I also added an exception by using the try statement with an except clause. The system will raise a TypeError exception if user enters the wrong type (e.g. enter a list instead of a float)  (Figure 5).

This is type of constructor is also called parameterized constructor. “A constructor with parameters is known as parameterized constructor. The parameterized constructor take its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.” (<https://www.geeksforgeeks.org/constructors-in-python/> - external site, 2020).

***Figure 5. Constructor and Attributes***

1. I then went to set the Properties which are methods used to control access to private attribute, and to manage field and attribute data. For each attribute, I created two properties, one to ‘get’ the data and one to ‘set’ the data. They are known as ‘Getters or Accessors’ and ‘Setters or Mutators’ respectively.

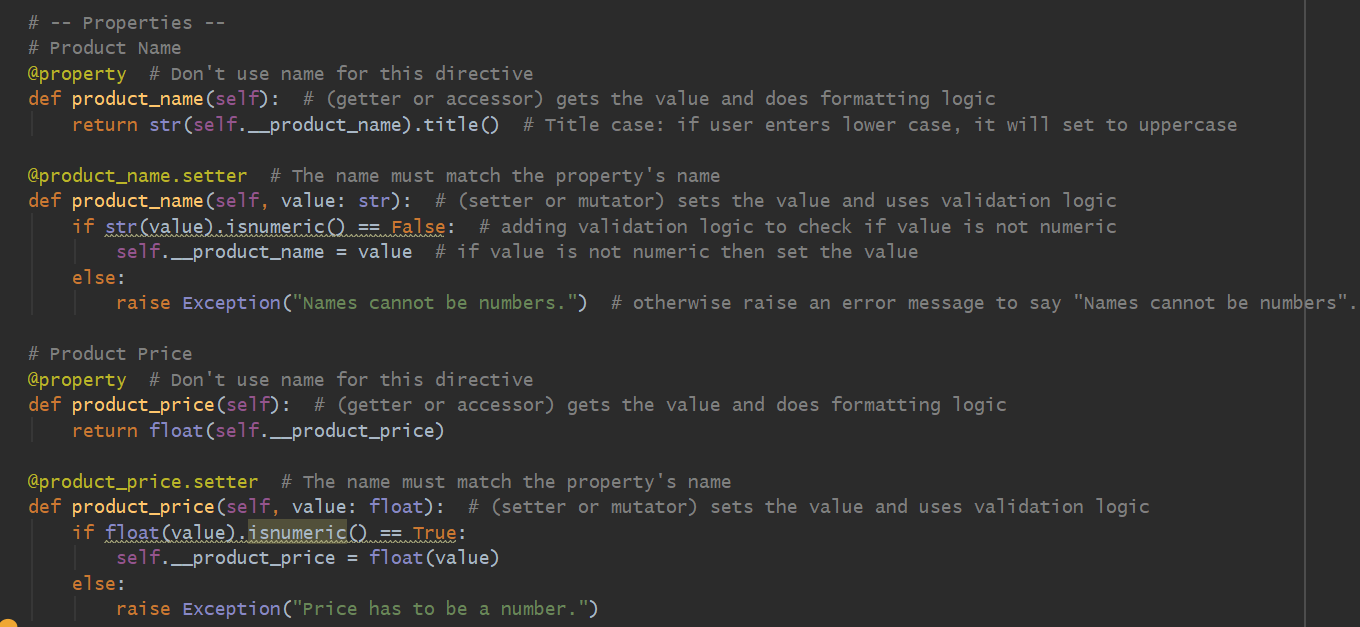
The ‘Getters or Accessors’ allow for adding formatting code. By convention, they use the @property decorator before the method definition, and have the same name as the attribute without the two underscores.

The ‘Setters or Mutators’ allow for setting the value, and for adding validation and error handling code. So only valid values that passed the Setter Property parameter get assigned to the field or attribute. I want to take the value and assign it to the field.

For the Product Name property, I created the ‘Setter’ directive @product\_name.setter. The name on the directive matches the name of the method and the attribute without the double underscore. This is to indicate that I wanted to use the product\_name method as a ‘Setter’ for the product\_name property. I added validation logic to ensure that the value entered is indeed an alphanumeric value. If value entered is numeric, the system will throw an exception.

I then created the ‘Getter’ method to read the data. I used the directive syntax @property to indicate the ‘Getter’ method. For the method name, I used the same name as the attribute. Usually the getter method is used for formatting, and I added the .title() function to convert the first character in the word to Uppercase and remaining characters to Lowercase.

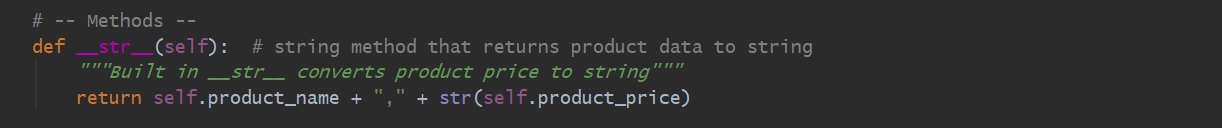
I followed the same pattern for the Product Price property, and just changed the variables name, type and the error handling. Note that I used type hints (: str and : float) to indicate what type of data I expect to see for the variable value for the Product Name and Product Price respectively. (Figure 6).



***Figure 6. ‘Getters or Accessors’ and ‘Setters or Mutators’***

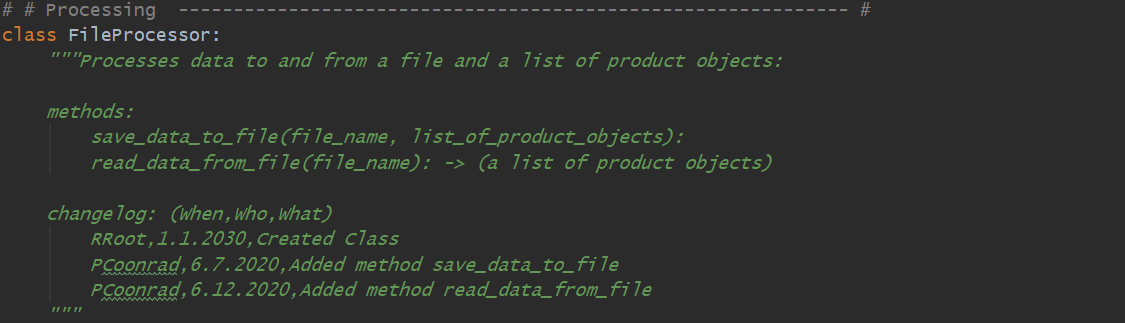
1. I created a method to override the ‘\_\_str\_\_()’ built-in Python method so the program would return the product name and price as a string with comma separated values instead of returning the name of the class and address identifier. This method converts the data from inside the class into a string.

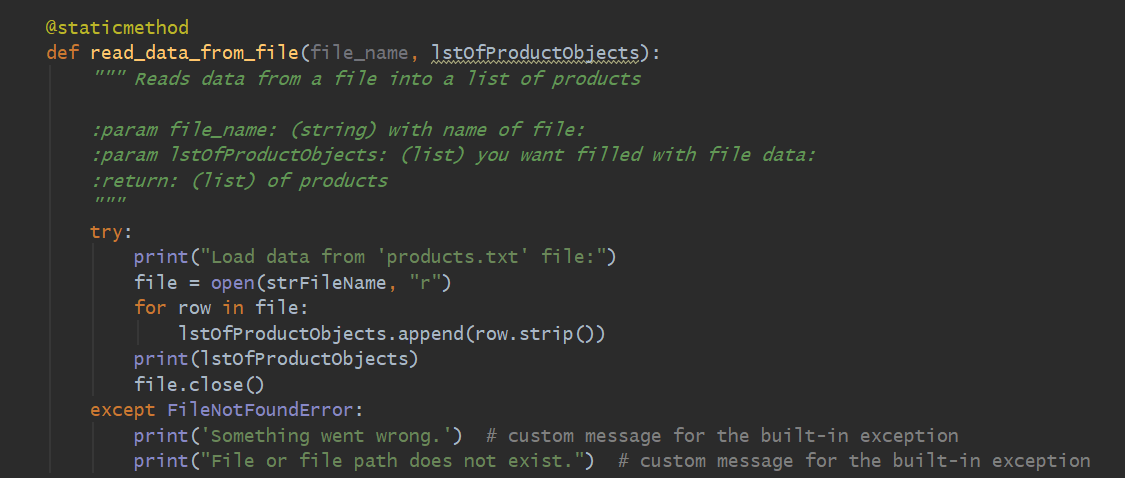
(Figure 7).

***Figure 7. Method***

1. Under the Processing layer, I have created the class FileProcessor with two static methods, one to save data to a file, and the other to read data from the file. Static methods are methods that are associated with a class. They are designed to be called through the class and therefore do not have the keyword ‘self’ as a parameter. By convention, the @staticmethod decorator, followed by the class method definition, will create a static method.

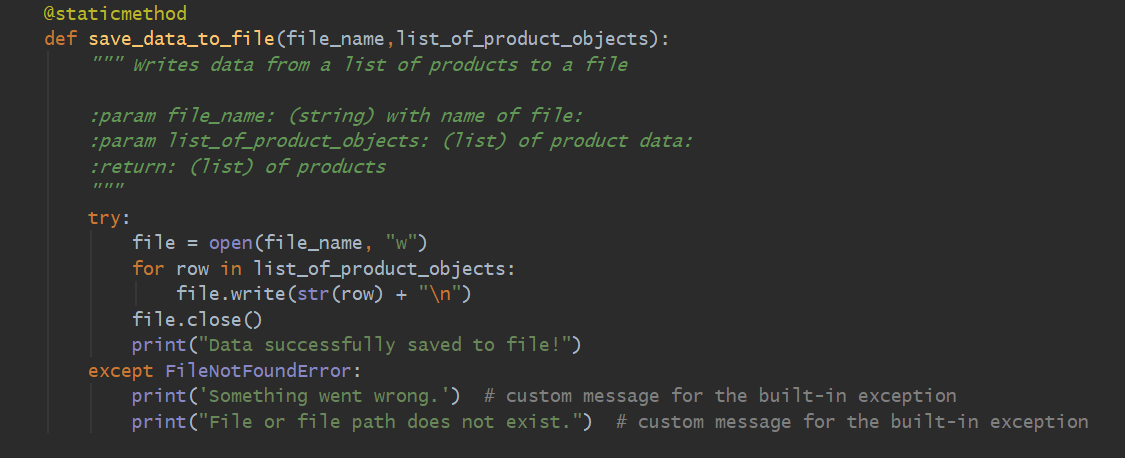
The read\_data\_from\_file() method will access the information from the existing .txt file. In this method, I used the open() function, provided the file name and the action I wanted it to perform once the file was opened (option “r” ). The file ‘products.txt.’ has two columns of data, ‘Product Name’ and ‘Product Price’. I used the for loop to loop through all rows in the file, read the data from those columns into a list and displayed to the user. Once the action was complete, I used the close() function to close the ‘products.txt’ file. If the file of file path is not found, the system will throw an exception (Figure 8).



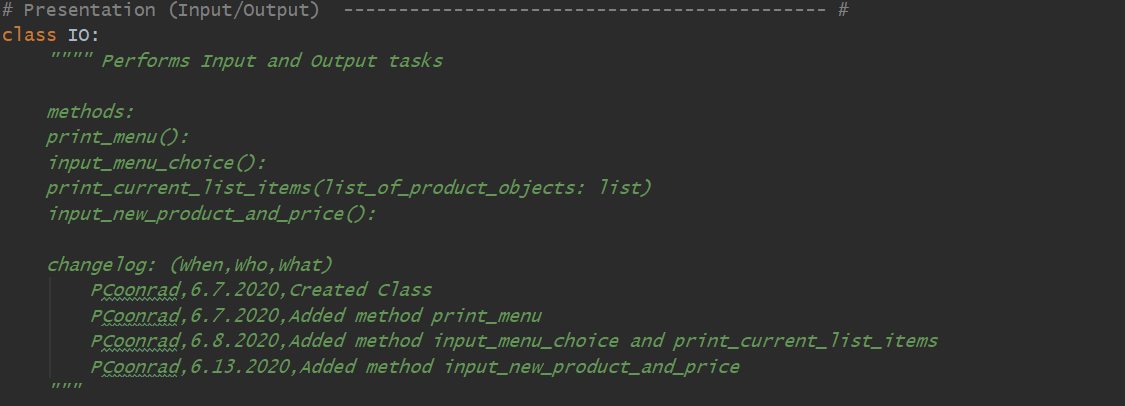


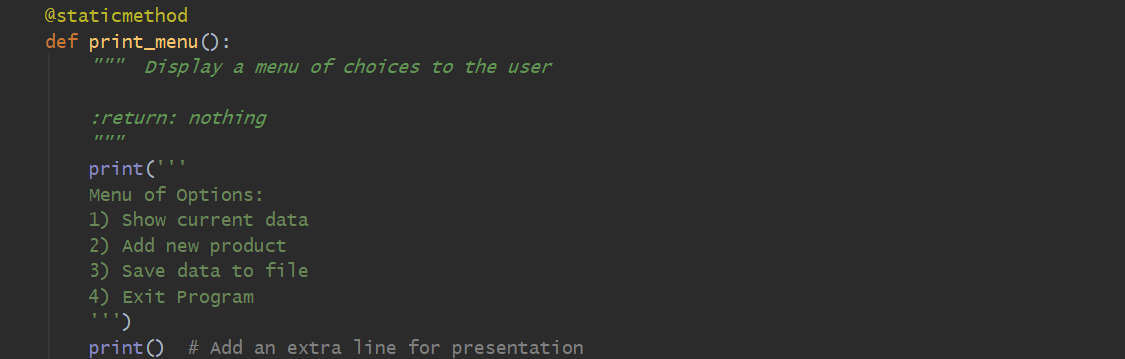
***Figure 8. Code to process data from file***

1. I order to save information to the file, I used the open() function and I provided the file name and the action I wanted it to perform once the file was opened (option “w” ). I used the for loop to loop through the rows in the list of product objects, and write it to the file. Once the action was complete, I used the close() function to close the ‘products.txt’ file. If the file of file path is not found, the system will throw an exception (Figure 9).

***Figure 9. Code to process data to file***

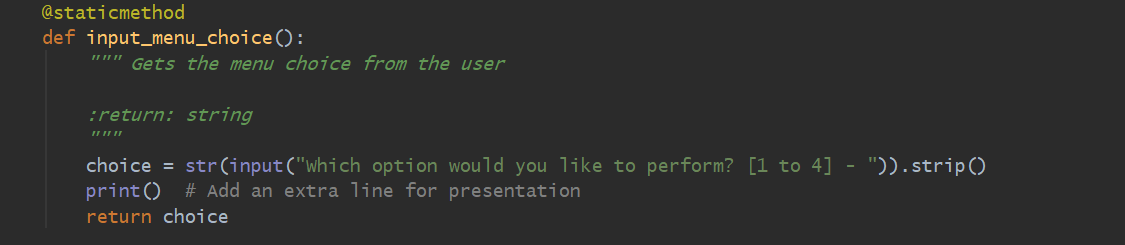
1. Under the Presentation layer, I added the class IO with four methods. The print\_menu() method displays the menu of options to the user (Figure 10).





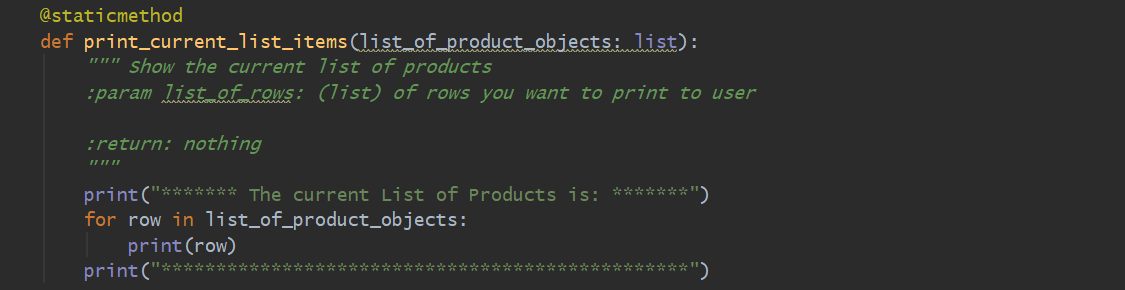
***Figure 10. Code to display menu of options to the user***

1. The input\_menu\_choice() method will request the user to enter a selection from the menu (Figure 11).



***Figure 11. Code to display menu of options to the user***

1. The print\_current\_list\_items() method displays the current list of products to the user (Figure 12).



***Figure 12. Code to display current list of products to the user***

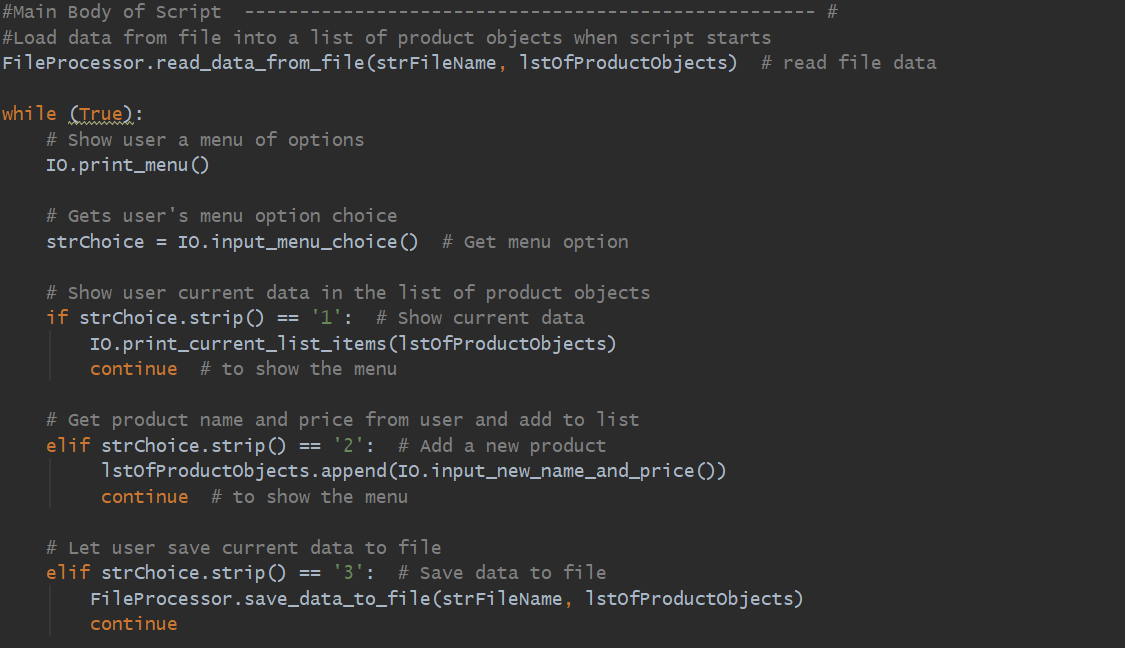
1. The input\_new\_name\_and\_price() method will prompt the user to enter the new product name and price (figure 13).

@staticmethod  
def input\_new\_name\_and\_price():  
 *""" Ask user to input new product and price  
 :return: np  
 """* name = str(input("Enter a New Product Name: "))  
 price = input("Enter a New Product Price: ")  
 np = Product(product\_name=name, product\_price=price)  
 except Exception as e:  
 print(e)  
 return np

***Figure 13. Code to prompt user to enter new product and price***

1. Lastly, under the Main Body of Script, I first called the FileProcessor class and the read\_data\_from\_file() method to load into memory any data stored in the file ‘products.txt’ when the program starts.

Then, using a while loop, it will call the different classes and methods to execute the different tasks depending on the selection the user makes from the menu of options displayed (figure 14).





***Figure 14. Main Body of Script calling the classes and methods to execute specific tasks***

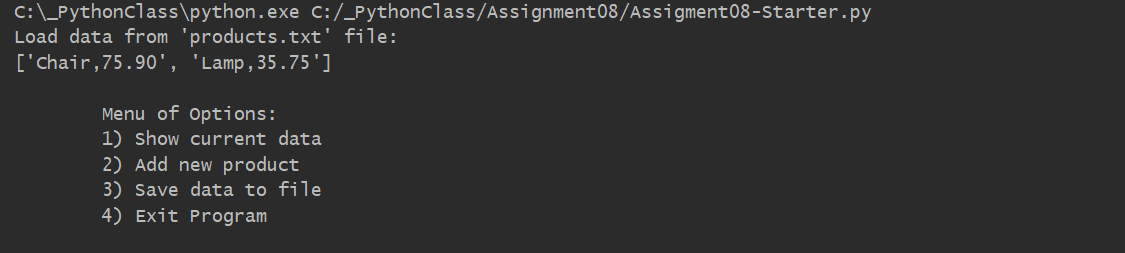
1. Below is a copy of the entire script in PyCharm (Figure 15).

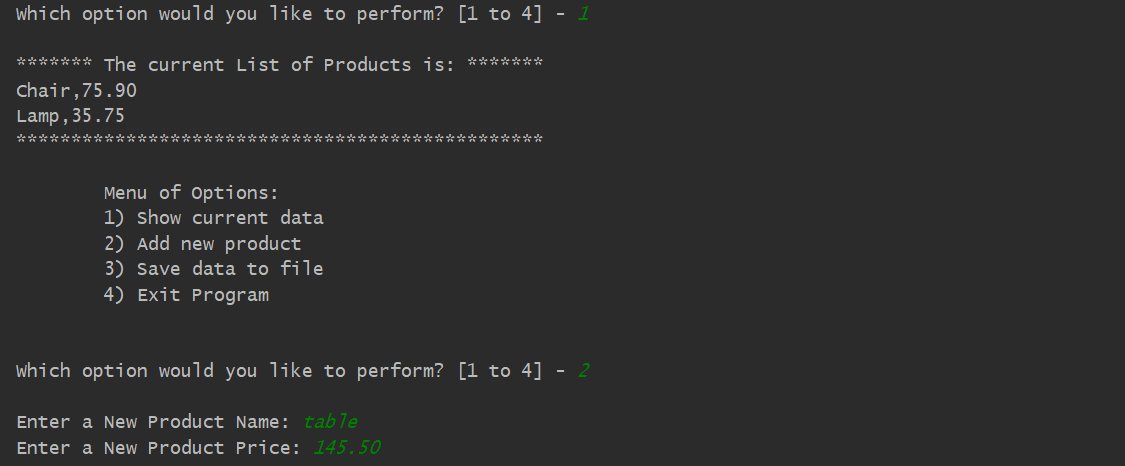
# ------------------------------------------------------------------------------------- #  
# Title: Assignment 08  
# Description: Working with classes  
  
# ChangeLog (Who,When,What):  
# RRoot,1.1.2030,Created started script  
# RRoot,1.1.2030,Added pseudo-code to start assignment 8  
# PCoonrad,6.6.2020,Added Product class constructor and attributes  
# PCoonrad,6.7.2020,Added Product class properties and methods  
# PCoonrad,6.7.2020,Added FileProcessor class save\_data\_to\_file()  
# PCoonrad,6.7.2020,Added IO class print\_menu()  
# PCoonrad,6.8.2020,Added IO class input\_menu\_choice() and print\_current\_list\_items()  
# PCoonrad,6.12..2020,Added FileProcessor class read\_data\_from\_file()  
# PCoonrad,6.13.2020,Added method input\_new\_product\_and\_price  
# PCoonrad,6.13.2020,Added main body of script  
# ------------------------------------------------------------------------------------- #  
# Data -------------------------------------------------------------------------------- #  
# Declare variables and constants  
strFileName = 'products.txt' # The name of the data file  
lstOfProductObjects = [] # List of Product Objects  
  
class Product:  
 *"""Stores data about a product:  
  
 constructor:  
 \_\_init\_\_(): sets up the initial attribute values (name, price) of a new object  
 properties:  
 product\_name: (string) with the products's name  
 product\_price: (float) with the products's standard price  
 methods:  
 changelog: (When,Who,What)  
 RRoot,1.1.2030,Created Class  
 PCoonrad,6.6.2020,Added constructor and attributes  
 PCoonrad,6.7.2020,Added properties and method  
 """* # -- Constructor --  
 def \_\_init\_\_(self, product\_name: str, product\_price: float): # the initializer sets the initial values for the object  
 *""" Set name and price of a new object"""* # -- Attributes --  
 try: # section code that can raise exception  
 self.\_\_product\_name = str(product\_name) # marking it private so it is hidden and not called directly  
 self.\_\_product\_price = float(product\_price) # marking it private so it is hidden and not called directly  
 except TypeError: # argument has the wrong type  
 print('Invalid type. Please enter a valid type.') # custom message for the built-in exception  
 except Exception as e: # exception class at the bottom will catch any other non-specific type of error  
 print('Something went wrong.') # custom message for the built-in exception  
 print("Python Built-In error message:", (e)) # custom message with Python messages for the built-in exception  
  
 # -- Properties --  
 # Product Name  
 @property # Don't use name for this directive  
 def product\_name(self): # (getter or accessor) gets the value and does formatting logic  
 return str(self.\_\_product\_name).title() # Title case: if user enters lower case, it will set to uppercase  
  
 @product\_name.setter # The name must match the property's name  
 def product\_name(self, value: str): # (setter or mutator) sets the value and uses validation logic  
 if str(value).isnumeric() == False: # adding validation logic to check if value is not numeric  
 self.\_\_product\_name = value # if value is not numeric then set the value  
 else:  
 raise Exception("Names cannot be numbers.") # otherwise raise an error message to say "Names cannot be numbers".  
  
 # Product Price  
 @property # Don't use name for this directive  
 def product\_price(self): # (getter or accessor) gets the value and does formatting logic  
 return float(self.\_\_product\_price)  
  
 @product\_price.setter # The name must match the property's name  
 def product\_price(self, value: float): # (setter or mutator) sets the value and uses validation logic  
 if float(value).isnumeric() == True:  
 self.\_\_product\_price = float(value)  
 else:  
 raise Exception("Price has to be a number.")  
  
 # -- Methods --  
 def \_\_str\_\_(self): # string method that returns product data to string  
 *"""Built in \_\_str\_\_ converts product price to string"""* return self.product\_name + "," + str(self.product\_price)  
  
# -- End of class --  
  
# # Processing ------------------------------------------------------------- #  
class FileProcessor:  
 *"""Processes data to and from a file and a list of product objects:  
  
 methods:  
 save\_data\_to\_file(file\_name, list\_of\_product\_objects):  
 read\_data\_from\_file(file\_name): -> (a list of product objects)  
  
 changelog: (When,Who,What)  
 RRoot,1.1.2030,Created Class  
 PCoonrad,6.7.2020,Added method save\_data\_to\_file  
 PCoonrad,6.12.2020,Added method read\_data\_from\_file  
 """* @staticmethod  
 def read\_data\_from\_file(file\_name, lstOfProductObjects):  
 *""" Reads data from a file into a list of products  
  
 :param file\_name: (string) with name of file:  
 :param lstOfProductObjects: (list) you want filled with file data:  
 :return: (list) of products  
 """* try:  
 print("Load data from 'products.txt' file:")  
 file = open(strFileName, "r")  
 for row in file:  
 lstOfProductObjects.append(row.strip())  
 print(lstOfProductObjects)  
 file.close()  
 except FileNotFoundError:  
 print('Something went wrong.') # custom message for the built-in exception  
 print("File or file path does not exist.") # custom message for the built-in exception  
  
 @staticmethod  
 def save\_data\_to\_file(file\_name,list\_of\_product\_objects):  
 *""" Writes data from a list of products to a file  
  
 :param file\_name: (string) with name of file:  
 :param list\_of\_product\_objects: (list) of product data:  
 :return: (list) of products  
 """* try:  
 file = open(file\_name, "w")  
 for row in list\_of\_product\_objects:  
 file.write(str(row) + "\n")  
 file.close()  
 print("Data successfully saved to file!")  
 except FileNotFoundError:  
 print('Something went wrong.') # custom message for the built-in exception  
 print("File or file path does not exist.") # custom message for the built-in exception  
  
# -- End of class --  
  
# Presentation (Input/Output) -------------------------------------------- #  
class IO:  
 *"""" Performs Input and Output tasks  
  
 methods:  
 print\_menu():  
 input\_menu\_choice():  
 print\_current\_list\_items(list\_of\_product\_objects: list)  
 input\_new\_product\_and\_price():  
  
 changelog: (When,Who,What)  
 PCoonrad,6.7.2020,Created Class  
 PCoonrad,6.7.2020,Added method print\_menu  
 PCoonrad,6.8.2020,Added method input\_menu\_choice and print\_current\_list\_items  
 PCoonrad,6.13.2020,Added method input\_new\_product\_and\_price  
 """* @staticmethod  
 def print\_menu():  
 *""" Display a menu of choices to the user  
  
 :return: nothing  
 """* print('''  
 Menu of Options:  
 1) Show current data  
 2) Add new product  
 3) Save data to file  
 4) Exit Program  
 ''')  
 print() # Add an extra line for presentation  
  
 @staticmethod  
 def input\_menu\_choice():  
 *""" Gets the menu choice from the user  
  
 :return: string  
 """* choice = str(input("Which option would you like to perform? [1 to 4] - ")).strip()  
 print() # Add an extra line for presentation  
 return choice  
  
 @staticmethod  
 def print\_current\_list\_items(list\_of\_product\_objects: list):  
 *""" Show the current list of products  
 :param list\_of\_rows: (list) of rows you want to print to user  
  
 :return: nothing  
 """* print("\*\*\*\*\*\*\* The current List of Products is: \*\*\*\*\*\*\*")  
 for row in list\_of\_product\_objects:  
 print(row)  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
  
 # *TODO: Add code to get product data from user* @staticmethod  
 def input\_new\_name\_and\_price():  
 *""" Ask user to input new product and price  
  
 :return: np  
 """* try:  
 name = str(input("Enter a New Product Name: "))  
 price = input("Enter a New Product Price: ")  
 np = Product(product\_name=name, product\_price=price)  
 except Exception as e:  
 print(e)  
 return np  
  
#Main Body of Script ---------------------------------------------------- #  
#Load data from file into a list of product objects when script starts  
FileProcessor.read\_data\_from\_file(strFileName, lstOfProductObjects) # read file data  
  
while (True):  
 # Show user a menu of options  
 IO.print\_menu()  
  
 # Gets user's menu option choice  
 strChoice = IO.input\_menu\_choice() # Get menu option  
  
 # Show user current data in the list of product objects  
 if strChoice.strip() == '1': # Show current data  
 IO.print\_current\_list\_items(lstOfProductObjects)  
 continue # to show the menu  
  
 # Get product name and price from user and add to list  
 elif strChoice.strip() == '2': # Add a new product  
 lstOfProductObjects.append(IO.input\_new\_name\_and\_price())  
 continue # to show the menu  
  
 # Let user save current data to file  
 elif strChoice.strip() == '3': # Save data to file  
 FileProcessor.save\_data\_to\_file(strFileName, lstOfProductObjects)  
 continue  
  
 # Let user Exit Program  
 elif strChoice.strip() == '4':  
 print("Goodbye!")  
 break # and Exit

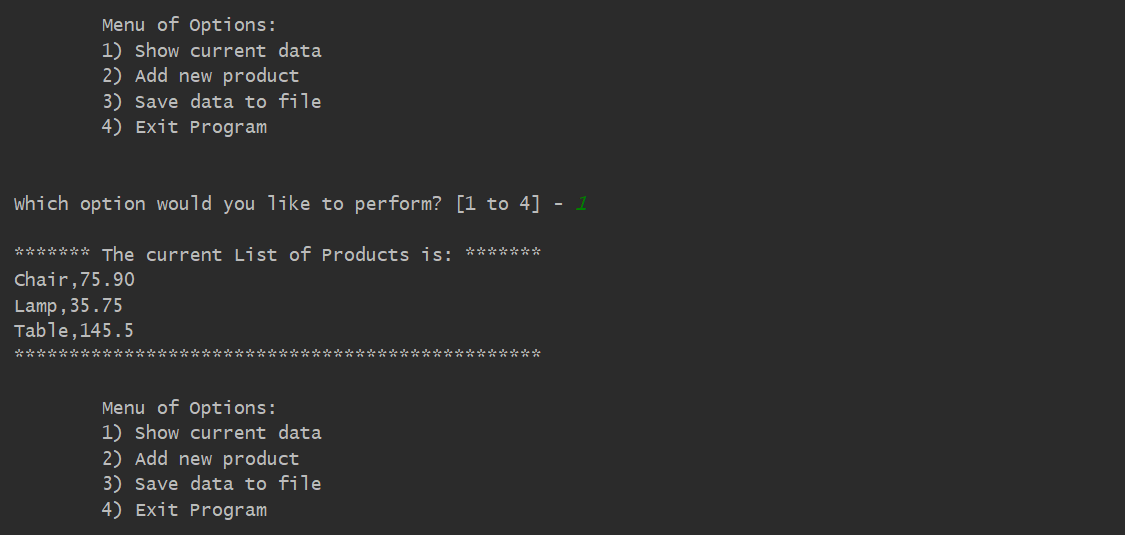
***Figure 15. Copy of the script in PyCharm***

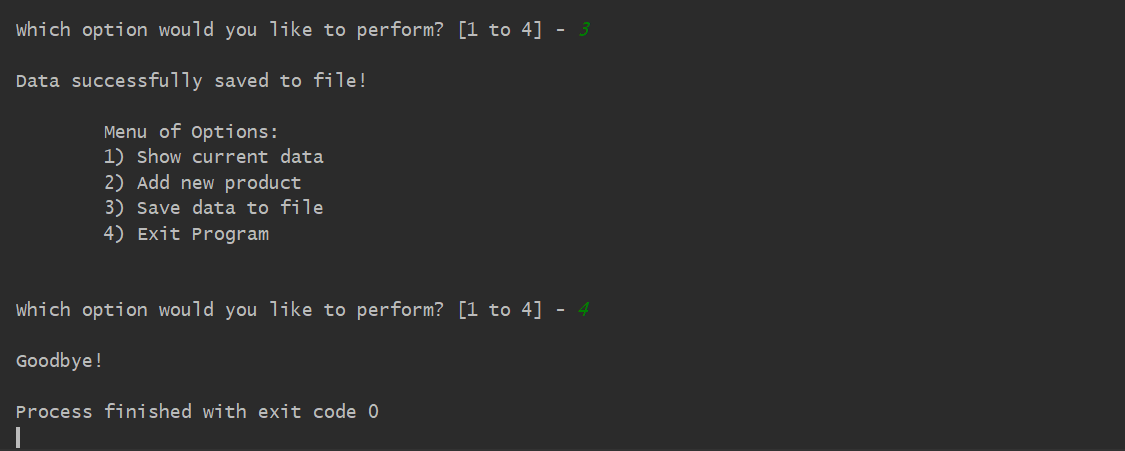
# Running the Program Using PyCharm

After the code was complete, I successfully ran the program using PyCharm (Figure 16).





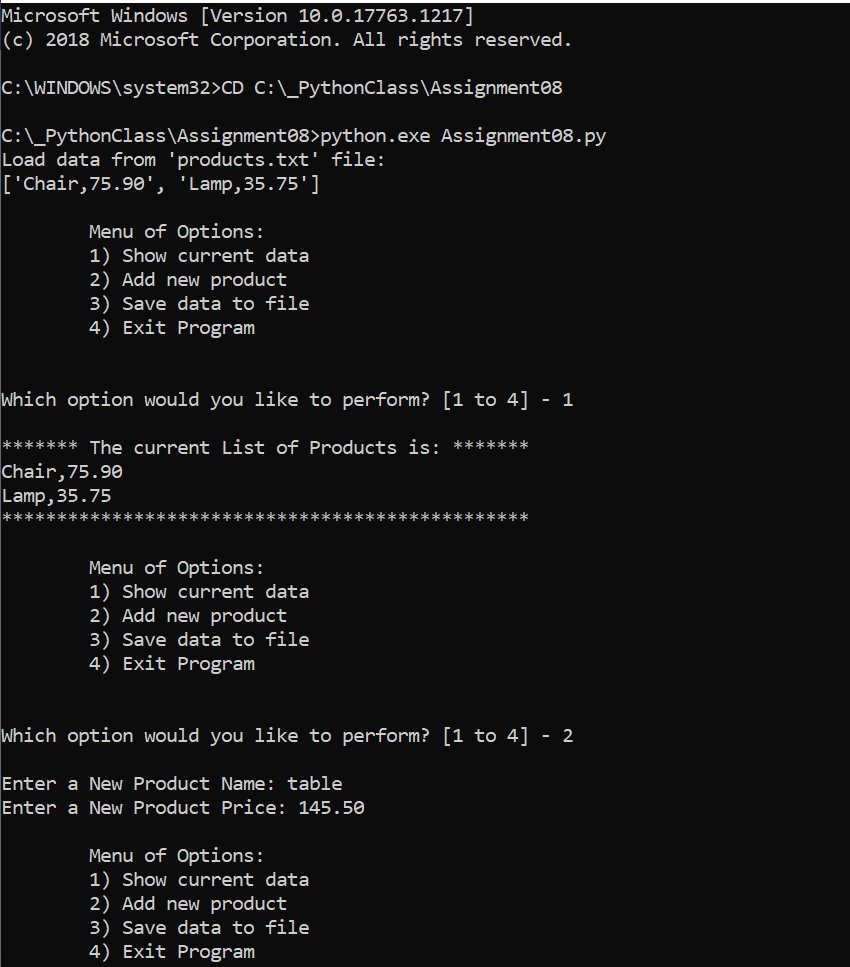


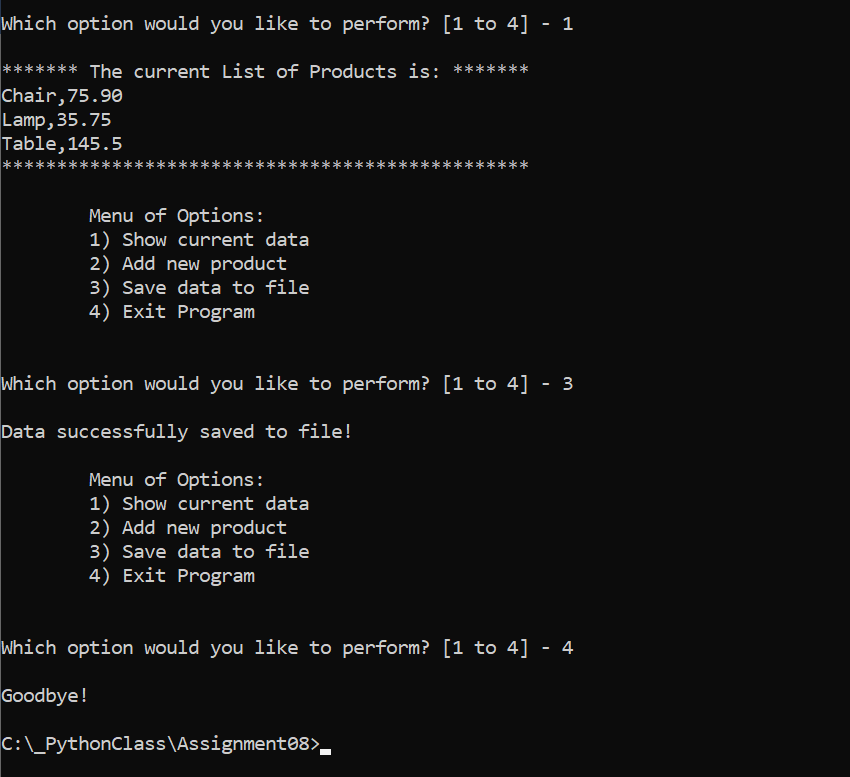


***Figure 16. Screenshot of the script running in PyCharm***

# Running the Program Using the Command Window

And I also successfully ran the program using the Command Window (Figure 17).

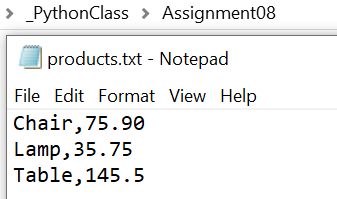




***Figure 17. Screenshot of the script running in a Command Window***

# Verify that it Worked

After I ran the program, I located the text file and opened it in a text editor to verify the file had data and it was correct (Figure 18).



***Figure 18. Verifying that the file has the correct data***

Summary

In the Assignment 08, I have modified the starter file script template and have demonstrated some of the concepts I learned during the course, such as how to make a class, use of class components: fields, constructors, attributes, properties, methods, objects, use of keyword ‘self’, static methods, private attributes, type hints, DocStrings, to name a few. By watching the instructional videos, reading the textbook and consulting additional documentation, I was able to successfully modify the script, and run the program using PyCharm and Command Window.