Nathaniel Smith

June 8, 2020

Foundations of Programming: Python (IT FDN 100 A Sp 20)

Assignment08

https://github.com/10neg9/IntroToProg-Python-Mod08.git

Python: Script Using Classes and Objects

Introduction

In this paper I describe the code that I wrote for the Python script file Assignment08.py. The script uses classes to create objects and to call static methods. The purpose of this script is to create a list of class objects and store their attributes to a file. There are two attributes: a product name, and its price. In general, the script provides a user with a menu and depending on the user's choice, it takes input from a user in the form of a product name and its price, adds that information to a list, prints that list when requested, and then saves that list to file when requested.

Script Header

The script header in Figure 1 shows the title, description, and the changelog. There are other changelogs embedded in the docstrings of the classes. From the changelog you can see that I updated class Product, class FileProcessor, class IO, and main.

```
# ------------------------------------------------------------------------ #
# 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
# NSmith,6/7/2020,Updated code in class Product
# NSmith,6/7/2020,Updated code in class FileProcessor
# NSmith,6/7/2020,Updated code in class IO
# NSmith,6/7/2020,Added Main body code
# NSmith,6/8/2020,Added try-except to Main - Add a Product
# ------------------------------------------------------------------------ #
```

Figure 1. Assignment 08 Script Header

Separation of Concerns

The script file is structured into 4 sections: data, processing, input/output, and main. The data section includes the variable declarations and the class Product. The processing section has a single class, FileProcessor. The input/output section also has a single class, IO. The main section calls the classes of the other sections to use their methods and to create objects from the Product class.

Data – Variables

Figure 2 shows the variable declarations.

```
strFileName = 'products.txt'  # name of file for reading and writing
lstOfProductObjects = []  # list of objects
strChoice = ''  # user menu choice
productName = ''  # name of a product
productPrice = 0.0  # price of a product
productObject = None  # object instance of class Product
```

Figure 2. Variables

Data – class Product Docstring

The data section has one class, Product. From the docstring shown in Figure 3, you can see that the Product class stores data about a product. It has two property methods: product_name and product_price. The __init__ method is used to initialize an object instance of the Product class when the object is created.

```
class Product:
    """Stores data about a product:

    properties:
        product_name: (string) with the products's  name
        product_price: (float) with the products's standard price
    methods:
        __init__: initializes product attributes
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
        NSmith,6/7/2020,Added Constructor
        NSmith,6/7/2020,Added product_name property
        NSmith,6/7/2020,Added product_price property
        NSmith,6/8/2020,Removed try-except statements--will handle errors in
IO
    """
```

Figure 3. class Product docstring

Data – class Product Constructor

Constructor is another name given to __init__ method. This method has two parameters: a string and a float. This is clearly shown in Figure 4. The constructor is executed when the Product class is called to create an object instance. It calls the property methods to create an object with two attributes: product_name and product_price.

```
# -- Constructor --
def __init__(self, name: str, price: float):
    """Sets product name and price

    :param name: (string) name of product:
    :param price: (string) price of product in float format:
    :return: nothing
    """
    # -- Attributes --
    self.product_name = name  # call product_name property method to set name
    self.product_price = price  # call product_price property method to set
price
```

Figure 4. Constructor Method of class Product

Data – class Product Properties

In Figure 5 and Figure 6 you get a more detail look at the properties of Product. Both properties have a getter (indicated by the @ property decorator) and a setter (indicated by the @*method_name*.setter decorator).

The product_name getter returns the product's name as a string. The product_name setter, takes the value passed to it and assigns it to a hidden attribute, __product_name, in title case (note the two underscores preceding the attribute name).

```
# Product Name
@property
def product_name(self):  # define getter of property product_name
    """ Gets product name

    :return: (string) title case of of product name
    """
    return str(self.__product_name).title()  # return string title case
product name

@product_name.setter
def product_name(self, value: str):  # define setter of property product_name
    """ Sets product name

    :param value: (string) the name of product to set:
    :return: nothing
    """
    self.__product_name = value  # assign value to hidden attribute
```

Figure 5. product_name Getter and Setter Properties

The product_price getter returns the product's price as a float value. The product_price setter, takes the value passed to it and assigns it to a hidden attribute, __product_price.

This marks the end of the data section.

```
# Product Price
@property
def product_price(self):  # define getter of property product_price
    """ Gets product price

    :return: (float) product price
    """
    return self.__product_price  # return product price

@product_price.setter
def product_price(self, value: float):  # define setter of property
product_price
    """ Sets product price

    :param value: (string) product price in float format:
    :return: nothing
    """
    self.__product_price = float(value)  # assign float(value) to hidden
attribute
```

Figure 6. product_price Getter and Setter Properties

Processing – class FileProcessor Docstring

The processing section contains a single class, FileProcessor. As shown in the docstring in Figure 7, FileProcessor saves data to a file with the save_data_to_file() method and copies data from a file into a list with the read_data_from_file() method. Each item it reads from the file is stored as an object in a list.

```
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
        NSmith,6/7/2020,Added read_data_from_file static method
        NSmith,6/7/2020,Added save_data_to_file static method
        NSmith,6/8/2020,Added try-except to read_data_from_file
    """
```

Figure 7. FileProcessor Docstring

Processing – Read Data from File

Figure 8 shows the code for reading data from file. The code takes the name of a file as an argument. It tries to open the file, and if the file exists it will open it in read mode. Otherwise, it will provide feedback to the user the file does not exist yet and the program is starting with an empty list. If the file does exists, each line of the file is iterated through, the lines are split into two variables, and these variables are used to create objects from the Product class. Each object is appended to a list. The file is closed after reading through all the lines and the list of objects is returned.

```
# Code to process data from a file into a list
@staticmethod
def read_data_from_file(file_name: str):
    """ Reads data from a file and stores in a list

    :param file_name: (string) with name of file:
    :return: (list) of product names and prices
    """
    lst = []  # create empty list object
    try:  # check if file can be opened first i.e. does it exist?
        f = open(file_name, "r")  # open file in read mode
    except Exception:
        print("Catalog file does not exist.")
        print("Starting with empty product list.")
    else:  # run this code if try succeeds
        for line in f:  # iterate through each line in file
            prod, price = line.split(",")  # split line with comma separator
            prodObj = Product(prod, price)  # create object from Product
class
            lst.append(prodObj)  # store object to list
        f.close()  # close the file
    return lst  # return a list of objects
```

Figure 8. read_data_from_file() Method

Processing – Save Data to File

If the user wants to save the product list to file, the code shown in Figure 9 will be executed. This method takes a filename and a list name as arguments. It opens the file in write mode, and for each element in the list, it writes a new line to the file that includes the product's name followed by a comma followed by the product's price. The file is closed after writing is complete.

This marks the end of the processing section.

```
# Code to process list data to a file
@staticmethod
def save_data_to_file(file_name: str, list_of_product_objects: list):
    """ Writes data from a list to a file

    :param file_name: (string) with name of file to write to:
    :param list_of_product_objects: (list) of data to be written to file:
    :return: nothing
    """
    f = open(file_name, "w")  # open file with write mode
    for row in list_of_product_objects:  # iterate through rows in the list
        # write each row of list to the file
        f.write(row.product_name + "," + str(row.product_price) + "\n")
    f.close()  # close file
```

Figure 9. save_data_to_file() Method

Presentation – class IO Docstring

The presentation section includes a single class, IO. The docstring for IO is shown in Figure 10. IO contains 5 static methods.

```
# Presentation (Input/Output)  -------------------------------------------- #
class IO:
    """Performs input and output tasks:

    methods:
        print_menu_Tasks():
        input_menu_choice(): -> (a string representing user's choice)
        print_current_product_list(product_list: list):
        input_product_and_price(): -> (a tuple with product name and price)
        input_press_to_continue(optional_message=''):

    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
        NSmith,6/7/2020,Added print_menu_Tasks static method
        NSmith,6/7/2020,Added input_menu_choice static method
        NSmith,6/7/2020,Added print_current_product_list static method
        NSmith,6/7/2020,Added input_product_and_price static method
        NSmith,6/7/2020,Added input_press_to_continue static method
        Nsmith,6/8/2020,Added try-except to input_product_and_price
    """
```

Figure 10. class IO Docstring

Presentation – Display Menu

Figure 11 shows the print_menu_Tasks() method. It is used to display a menu of 4 choices to the user.

```
# Static Method to show menu to user
@staticmethod
def print_menu_Tasks():
    """  Display a menu of choices to the user

    :return: nothing
    """
    # Display the menu
    print('''
    Menu of Options
    1) Print Current Products and Prices
    2) Add a Product to the Catalog
    3) Save Catalog to File
    4) Exit Program
    ''')
    print()  # Add an extra line for looks
```

Figure 11. Display Menu Method

Presentation – Get User's Menu Choice

Figure 12 shows the input_menu_choice() method. This method gets the menu choice from the user, strips it of whitespace, stores its value in the variable choice, and returns choice.

```
# Static method to get user's choice
@staticmethod
def input_menu_choice():
    """ Gets the menu choice from a user

    :return: (string) user's choice
    """
    # Get user's choice, strip whitespace, and store in string variable
choice
    choice = str(input("Which option would you like to perform? [1 to 4] -
")).strip()
    print()  # Add an extra line for looks
    return choice  # return choice
```

Figure 12. Menu Choice Method

Presentation – Print the Product List

Figure 13 shows the print_current_product_list() method. This takes a single argument of list type. The list contains the objects created from the Product class. A header is printed and below it each product and price, and below that a footer separator is printed. Note that the print statement is calling the Product getter properties to get the product name and product price of each object.

```
# Static Method to show the current data in product list
@staticmethod
def print_current_product_list(product_list: list):
    """ Shows the current product list saved to file

    :param file_name: (string) name of file containing product info
    :return: nothing
    """
    # Print header
    print("******** Catalog Items ********")
    print("________Product | Price________")
    # iterate through the list of items in the file
    for item in product_list:
        # print the product name and price
        print(f"\t{item.product_name} | {item.product_price}")
    # print a footer
    print("*******************************")
    print()  # Add an extra line for looks
```

Figure 13. Method to Print Product List

Presentation – Request Product and Price from User

Figure 14 shows the code for the input_product_and_price() method. This method asks the user for a product name and checks if the name is valid. Then it asks for a price and checks if the price is valid. If it discovers invalid input exceptions are raised to provide feedback to the user that an invalid entry was made. If everything is on the up-and-up, the method returns a tuple containing the product's name and price.

```
# Static Method to get product data from user
@staticmethod
def input_product_and_price():
    """ Gets user to input a product and price

    :param: nothing:
    :return: (tuple) with value of product and price
    """
    # get product name from user
    str_prod = str(input("Enter a product name: ").strip())
    # check if input is valid product name
    try:
        if not(not str_prod.isnumeric() and "." not in str_prod):
            # raise exception if number in name
            raise Exception("You did not enter a valid product name.\n")
    except Exception as e:
        print(e)
    else:  # if user entered valid product name, now ask for price
        str_price = str(input("Enter the price: ").strip())
        try:
            #  try to convert string to float
            flt_price = float(str_price)
        except Exception:
            print("Prices must be numbers.\n")
        # Execute the else statement if try successful
        else:
            print()  # Add an extra line for looks
            return str_prod, flt_price  # return the product name and its
price
```

Figure 14. Method to Get Product and Price Input from User

Presentation – Pause Program

The code in Figure 15 is used to pause the program at certain locations. It has one optional parameter, and if an argument is passed to this parameter, an optional message will be printed to screen. Otherwise, it prints a message asking the user to press the enter key to continue.

```
@staticmethod
def input_press_to_continue(optional_message=''):
    """ Pause program and show a message before continuing

    :param optional_message:  An optional message you want to display
    :return: nothing
    """
    print(optional_message)
    input('Press the [Enter] key to continue...')
```

Figure 15. Method to Pause Program

Main – Startup

When the script starts up, it checks for the file, reads the file if it exists, copies the data from the file (again only if the file already exists) to a list, prints the menu, and asks the user to make a choice from the four menu options. The code is shown in Figure 16.

```
# Main Body of Script  ---------------------------------------------------- #

# Load data from file into a list of product objects when script starts
lstOfProductObjects = (FileProcessor.read_data_from_file(strFileName))

while True:
    # Show user a menu of options
    IO.print_menu_Tasks()
    # Get user's menu option choice and store in strChoice
    strChoice = IO.input_menu_choice()
```

Figure 16. Main Body Code Executed at Program Start

Main – Menu Option 1 Chosen

The user's menu option is checked against the 4 possible menu options using if-elif statements. If the user entered '1' the product list will be printed, see Figure 17.

```
# Process users input choice
if strChoice == '1':  # Print current products
    # Show user current data in the list of product objects
    IO.print_current_product_list(lstOfProductObjects)
    IO.input_press_to_continue()  # wait for user to continue
```

Figure 17. Print Current Product List

Main – Menu Option 2 Chosen

If the user chooses option 2, the user will be asked to provide a product name and price. The inputs will be checked for validity, and if valid, an object will be created from the Product class. The object is appended to the list containing current list of products. See Figure 18.

If the user does not provide valid input for either the product name or price, friendly messages will be provided. These messages are handled through the try-except statement.

```
elif strChoice == '2':  # Add a product
    # Let user add data to the list of product objects
    try:  # try to run code, basically check if inputs are valid
        productName, productPrice = IO.input_product_and_price()  # unpack
tuple
    # if try fails run this exception
    except Exception:
        print("Product not added, let's try again.")
    else:  # if try succeeds run this code
        productObject = Product(productName, productPrice)  # create object
instance
        lstOfProductObjects.append(productObject)  # append object to list
        IO.input_press_to_continue("Product Added!")  # provide feedback
```

Figure 18. Add a Product to the List

Main – Menu Option 3 Chosen

If the user chooses option 3, the list of products will be saved to a file. This code is shown in Figure 19.

```
elif strChoice == '3':  # Save catalog to file
    # let user save current data to file and exit program
    FileProcessor.save_data_to_file(strFileName, lstOfProductObjects)
    IO.input_press_to_continue("File Saved!")   # provide feedback
```

Figure 19. Save Product List to File

Main – Menu Option 4 Chosen

Figure 20 shows the code to exit the program if the user chooses option 4.

```
elif strChoice == '4':  # Exit Program
    print("Goodbye!")  # provide feedback
    break  # and Exit
```

Figure 20. End the Program

Main – Menu Option Invalid

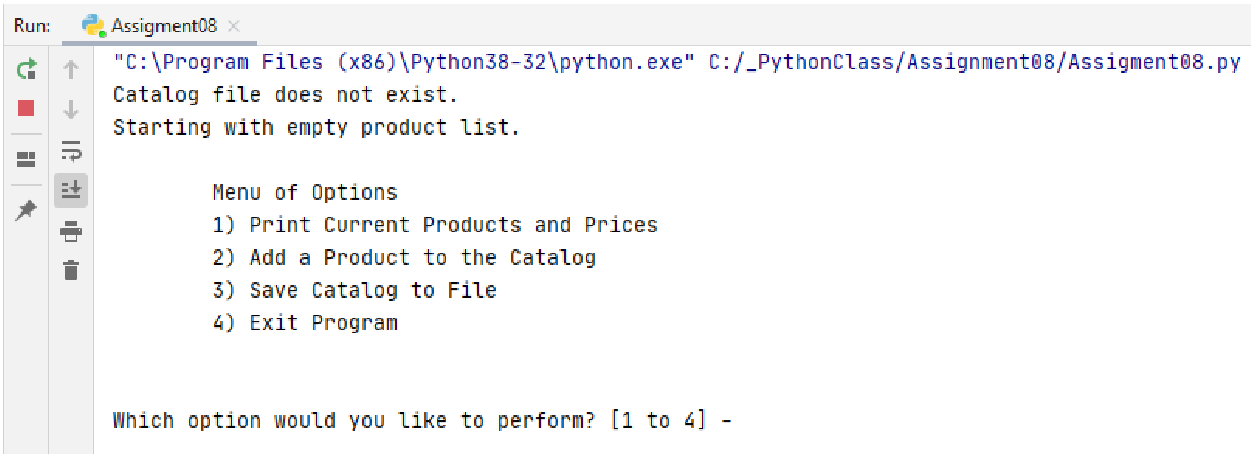
The last statement in the script, shown in Figure 21, executes if the user did not choose a number from 1 to 4. A friendly message is displayed and the user has to press enter to continue at which point the menu is printed again and the user is asked to choose an option from the menu.

```
else:  # user did not make a valid selection from the menu
    IO.input_press_to_continue(f"You entered '{strChoice}', "
                               f"but this is not a valid option.\n")   #
provide feedback
```

Figure 21. Give User Feedback About Invalid Selection

Testing Script in PyCharm

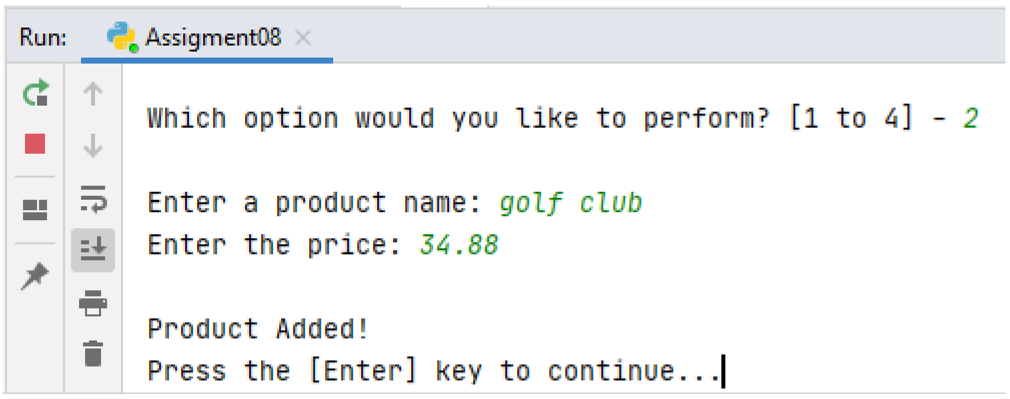
I first tested the code in PyCharm. The startup is shown in Figure 22.



```
"C:\Program Files (x86)\Python38-32\python.exe" C:/_PythonClass/Assignment08/Assigment08.py
Catalog file does not exist.
Starting with empty product list.

        Menu of Options
        1) Print Current Products and Prices
        2) Add a Product to the Catalog
        3) Save Catalog to File
        4) Exit Program


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

Figure 22. Assignment08.py Startup in PyCharm

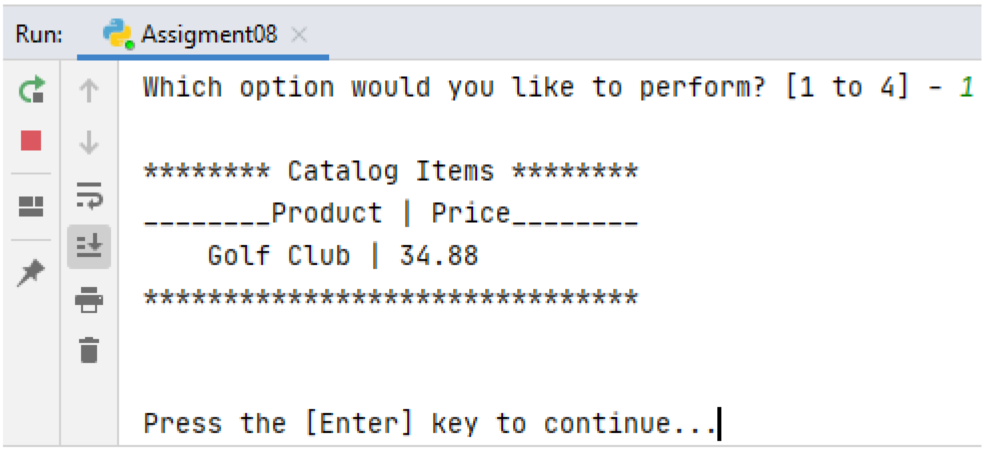
Because the text file did not yet exist, the list started empty. I added an item to the list as shown in Figure 23.



```
Which option would you like to perform? [1 to 4] - 2

Enter a product name: golf club
Enter the price: 34.88

Product Added!
Press the [Enter] key to continue...
```

Figure 23. Added a Product

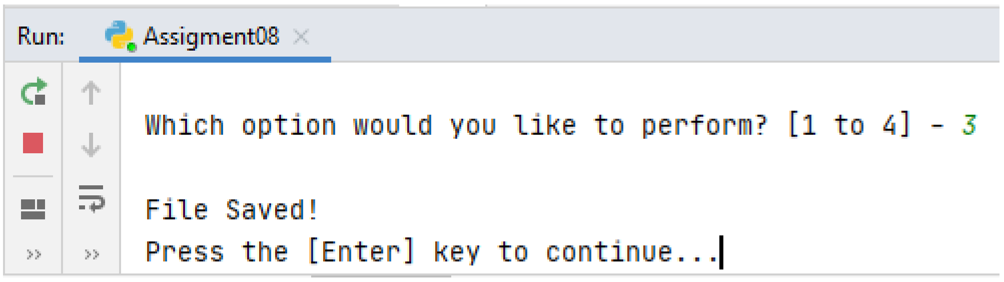
Then I printed the list with my newly added item, see Figure 24.



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

******** Catalog Items ********
________Product | Price________
    Golf Club | 34.88
*******************************


Press the [Enter] key to continue...
```

Figure 24. Print Product List

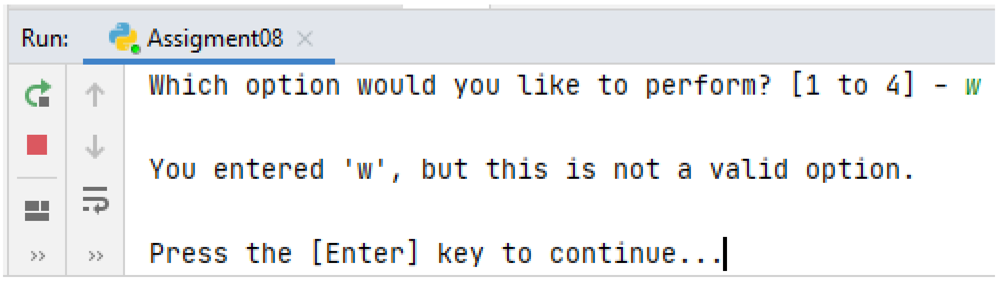
The list was saved to file, see Figure 25.



```
Which option would you like to perform? [1 to 4] - 3

File Saved!
Press the [Enter] key to continue...
```

Figure 25. File Saved

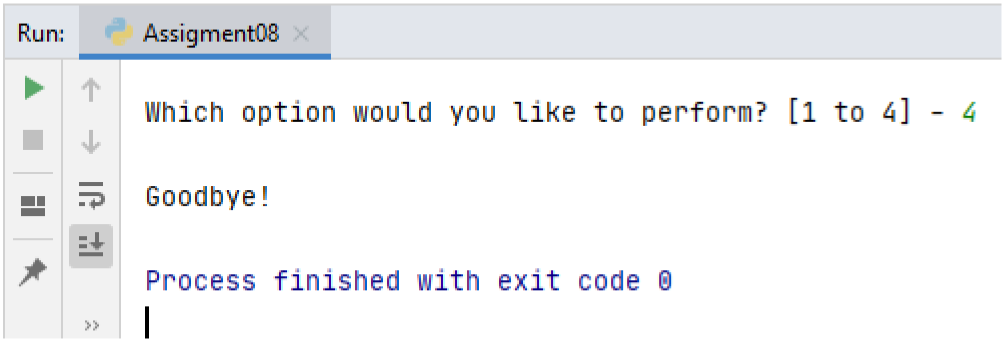
I made an invalid menu selection, see Figure 26.



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

You entered 'w', but this is not a valid option.

Press the [Enter] key to continue...
```

Figure 26. Invalid Selection

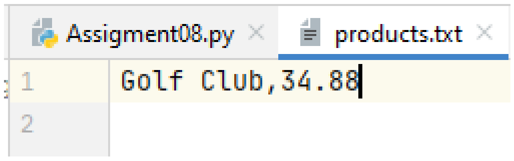
Time to exit the program, see Figure 27.



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

Goodbye!

Process finished with exit code 0
```

Figure 27. Exit Program

And finally I checked that the file was created with my product list as shown in Figure 28.

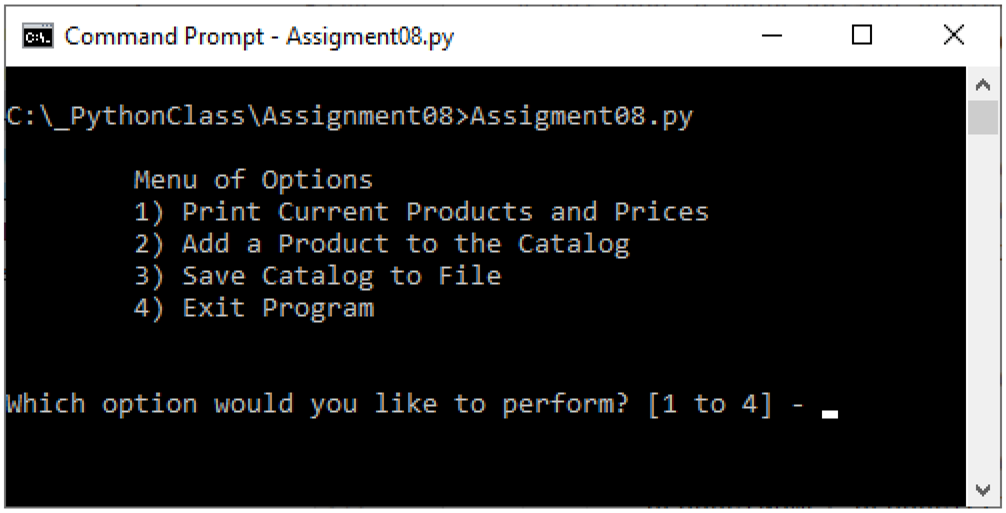


```
Golf Club,34.88
```

Figure 28. products.txt

Testing Script in Command Prompt

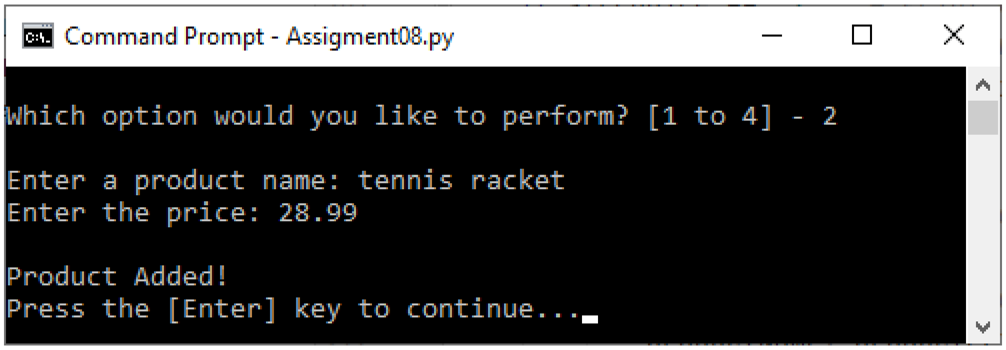
After testing in PyCharm, I tested the code from Windows Command Prompt. The startup screen is shown in Figure 29.



```
Command Prompt - Assigment08.py

C:\_PythonClass\Assignment08>Assigment08.py

        Menu of Options
        1) Print Current Products and Prices
        2) Add a Product to the Catalog
        3) Save Catalog to File
        4) Exit Program


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

Figure 29. Assignment08.py Startup in Command Prompt

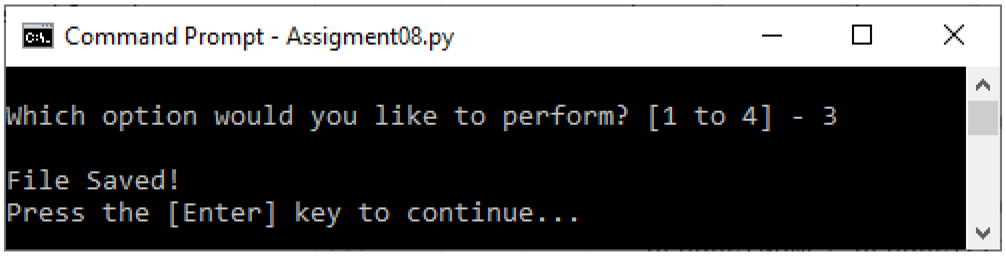
I added an item with option 2, see Figure 30.



```
Command Prompt - Assigment08.py

Which option would you like to perform? [1 to 4] - 2

Enter a product name: tennis racket
Enter the price: 28.99

Product Added!
Press the [Enter] key to continue...
```

Figure 30. Added a Product

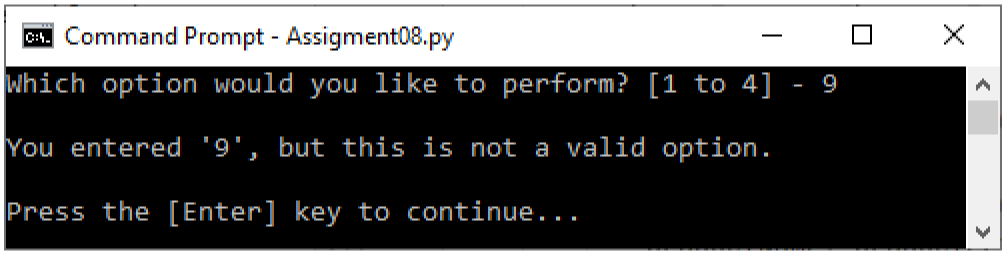
I saved the new list to file with option 3, see Figure 31.



```
Command Prompt - Assigment08.py

Which option would you like to perform? [1 to 4] - 3

File Saved!
Press the [Enter] key to continue...
```

Figure 31. File Saved

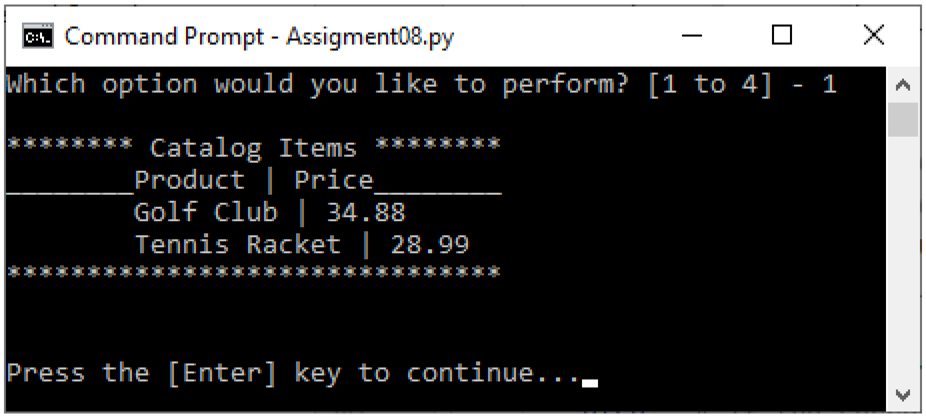
I tested an invalid option, see Figure 32.



```
Which option would you like to perform? [1 to 4] - 9

You entered '9', but this is not a valid option.

Press the [Enter] key to continue...
```

Figure 32. Invalid Selection

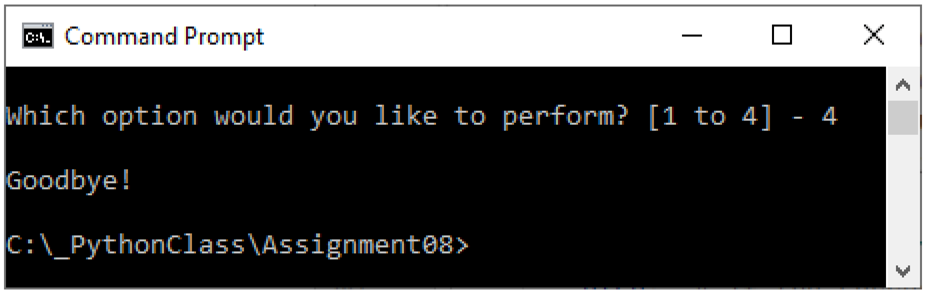
I printed the product list with option 1, see Figure 33.



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

******** Catalog Items ********
________Product | Price________
        Golf Club | 34.88
        Tennis Racket | 28.99
*******************************


Press the [Enter] key to continue...
```

Figure 33. Print Product List

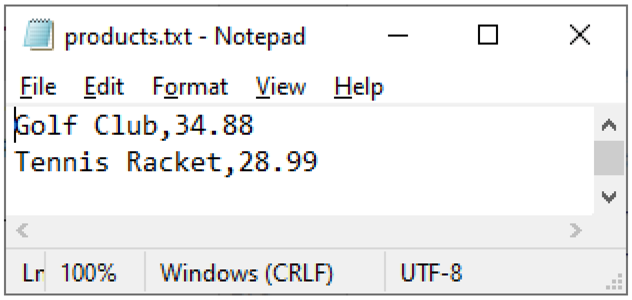
Time to exit the program, see Figure 34.



```

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

Goodbye!

C:\_PythonClass\Assignment08>
```

Figure 34. Exit Program

And finally, I checked that the code worked by opening the text file in Notepad. See Figure 35.



```
Golf Club,34.88
Tennis Racket,28.99
```

Figure 35. product.txt Text File

Summary

In summary, I successfully wrote a program that creates object instances from classes. The objects were stored in a list until it was time to save the object attributes to a file. And when the file was opened, the file’s contents were used to create objects that were stored to a list. This code could be improved by adding an option to remove objects from the list.