

Classes and Objects

Introduction

In module 8 we learned about the components of classes and how to use them. We also learned how create objects using classes. In this paper I am going to discuss how I completed a script that uses three classes to manage product data.

Product Class

The Product class is used to store and manage data related to products. The constructor has two parameters, *product_name* and *product_price*. Whenever a Product object is created, these parameters are used to initialize the private attributes *__str_product_name* and *__flt_product_price* through the setter properties below.

```
# Constructor
def __init__(self, product_name, product_price):

    # Attributes (these just reference the properties)
    self.product_name = product_name
    self.product_price = product_price
```

Figure 1. Code for constructor and attributes of Product class

The Product class contains four properties: a getter and setter for *product_name* and a getter and setter for *product_price*. The getter properties are simply methods that return the name and price for the Product object. Since these variables are private, they should not be accessed directly and should be accessed through the methods. No additional formatting is included.

The setter properties assign the value passed in the method to *__str_product_name* and *__flt_product_price*. There is no validation included in the properties because this is included in the IO functions. Figure 2 below shows the code for the getter and setter properties for *product_name* and *product_price*.

```

# Properties
# Product Name
@property # getter for Product Name
def product_name(self):
    return self.__str_product_name

@product_name.setter # setter for Product Name
def product_name(self, value):
    self.__str_product_name = value

# Product Price
@property # getter for Product Price
def product_price(self):
    return self.__flt_product_price # validation occurs in IO.input_add_product_data

@product_price.setter # setter for Product Price
def product_price(self, value):
    self.__flt_product_price = value # validation occurs in IO.input_add_product_data

```

Figure 2. Properties for the Product class

I also created one method for the Product class to overwrite the `__str__` method. This returns the name and price of the Product object separated by a comma and is used to more easily write data to a file in a method in the FileProcessor class.

```

# Methods
def __str__(self):
    """ Prints product name and price when product object is printed

    :param: self:
    :return: nothing:
    """
    return self.product_name + "," + str(self.product_price)

```

Figure 3. `__str__` method for the Product class

FileProcessor Class

The FileProcessor class is used to read to and write data from a text file. It contains two static methods, `read_data_from_file` and `save_data_to_file`. Because these are static methods, they do not need to be called with an object.

The `read_data_from_file` method reads data line by line, splitting it into name and price by a comma. A Product object is created passing in the name and price values, and the Product object is appended to the list of Products. A try-except block is included in the method in case the file name that was passed in does not exist. If the file does not exist, the FileNotFound error is raised when the script tries to read the file, a simple error message is printed, and the script moves on. At the end, the `list_of_product_objects` is returned.

```

@staticmethod
def read_data_from_file(file_name):
    """ Reads data from a file into a list of dictionary rows

    :param file_name: (string) with name of file:
    :return: (list) of product objects rows:
    """
    list_of_product_objects = []
    try: # reads data if file exists
        file = open(file_name, "r")
        for line in file:
            name, price = line.split(",") #unpacks comma separated list to name and price
            product = Product(name.strip(),price.strip()) # creates new Product object with name and price
            list_of_product_objects.append(product) # adds new Product object to list of products
        file.close()
    except FileNotFoundError as e: #error handling if file does not exist
        print() # blank line for formatting
        print("File does not exist. There is no data to load.\n")
    return list_of_product_objects

```

Figure 4. Code for the `read_data_from_file` method in the `FileProcessor` class

The `save_data_to_file` method opens a file in write mode based on the file name that is passed in, then writes each object in the list of product objects to the file. Because the `__str__` method was already written to return `product_name` and `product_price` separated by a comma, no formatting is needed besides adding a new line after each object.

```

@staticmethod
def save_data_to_file(file_name, list_of_product_objects):
    """ Writes data from a list of product objects to a File

    :param file_name: (string) with name of file:
    :param list_of_product_objects: (list) you want filled with file data:
    """
    file = open(file_name, "w")
    for obj in list_of_product_objects:
        file.write(str(obj) + "\n") # writes each object in the list using __str__ method to a new line of the file
    file.close()

```

Figure 5. Code for the `save_data_to_file` method in the `FileProcessor` class

Figure 6 below shows the text file after the Product information for Apples and Bananas was saved to the file.

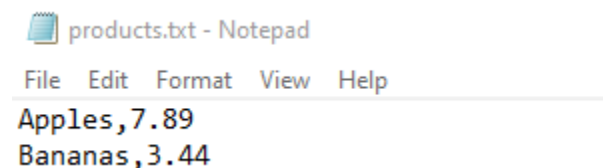


Figure 6. Text file with Product data saved

IO Class

The IO class include four static methods for presenting data to and obtaining data from the user. The `output_menu_choices` and `input_menu_choice` method prints the menu options and then gets a choice from the user, similar to scripts we have created in other modules.

The `output_product_data` method takes the `list_of_product_objects` argument and prints each Product item in the list. This method uses the Product getter properties to get the object's name and price and is printed with some additional text from formatting. If `list_of_product_objects` is empty, "There are no Products to show" is printed.

```
@staticmethod
def output_product_data(list_of_product_objects):
    """ Shows the current product names and prices

    :param list_of_product_objects: (list) of objects that was read from file
    :return: nothing
    """
    if list_of_product_objects == []: # handles cases when products have not yet been added
        print('There are no Products to show.')
    else:
        for object in list_of_product_objects:
            # loops through each Product object in list and uses product_name and product_price properties
            # to get private variables
            print('Product Name: ' + object.product_name.title() + '\nProduct Price: $'
                  + str(object.product_price) + '\n')
```

Figure 7. Code for the `output_product_data` method in the IO class

When the script was first run without a `products.txt` file in place, the message that a file does not exist from the FileProcessor `read_data_from_file` method is printed. After selecting menu item 1, the message is printed that there are no Products to show.

```
C:\Users\User>python.exe C:\Python\PythonClass\Assignments\Assignment08\Assignment08.py
File does not exist. There is no data to load.

    Menu of Options
    1) Show Current Products
    2) Add a New Product
    3) Save Data to File
    4) Exit Program

Which option would you like to perform? [1 to 4] - 1
There are no Products to show.
```

Figure 8. Running script in Command Prompt without an existing `products.txt` file

When selecting option 1 after Product items have been added, the Product names and prices are printed, as seen in Figure 9.

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

Product Name: Apples
Product Price: $7.89

Product Name: Bananas
Product Price: $3.44
```

Figure 9. Selecting menu item 1 after Products have been added

The `input_add_product_data` method loops through asking for user input until a valid value has been added for both `product_name` and `product_price`. If the `product_name` is numeric, an error is raised indicating that the Product Name cannot be a number. If the `product_price` is not a number, the `ValueError` is raised when trying to convert the input to a floating-point value. After the loop ends, `product_name` and `product_price` is returned.

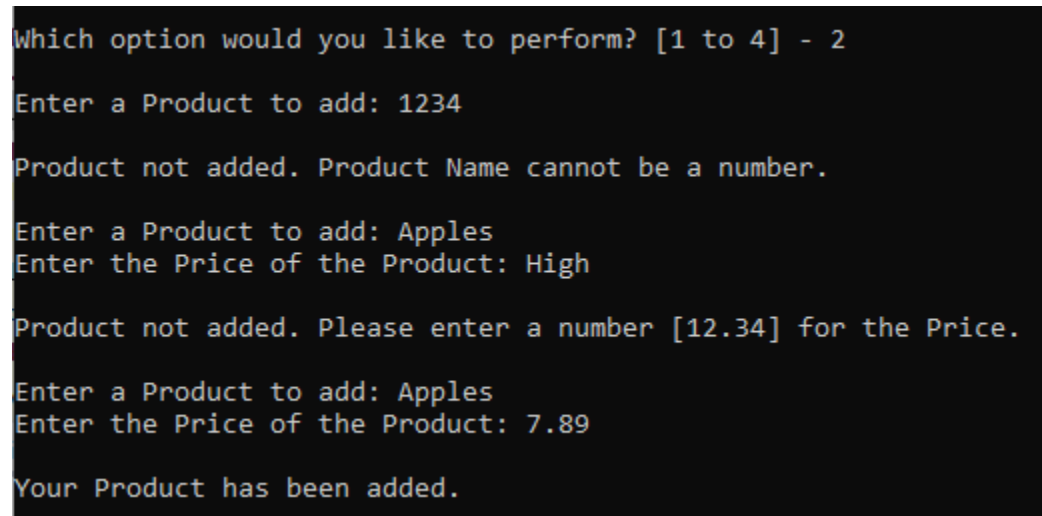
```
@staticmethod
def input_add_product_data():
    """ Gets name and price values to be added to the list

    :return: (string, float) with product name and price
    """
    product_name = None
    product_price = None
    # continues looping until there is a valid entry for both name and price
    while product_name == None or product_price == None:
        try:
            product_name = input("Enter a Product to add: ")
            if product_name.isnumeric() == True: # checks to see if name is a number and raises error if true
                raise Exception('Product not added. Product Name cannot be a number.')
            product_price = float(input("Enter the Price of the Product: "))

            # ValueError will be raised when trying to convert input to float if product_price is not a number
        except ValueError as e:
            print()
            print('Product not added. Please enter a number [12.34] for the Price.')
            print()
        except: # this message will be used if error is not ValueError
            print()
            print('Product not added. Product Name cannot be a number.')
            print()
    print()
    return product_name, product_price
```

Figure 10. Code for the `input_add_product_data` method in the IO class

Figure 11 below shows the outcome in the Command Prompt when a Product with a name “1234” and a Product with a price of “high” is added. Both attempts cause an error to be raised. Finally, a Product with the name “Apple” and price “7.89” is successfully added.



```
Which option would you like to perform? [1 to 4] - 2
Enter a Product to add: 1234
Product not added. Product Name cannot be a number.

Enter a Product to add: Apples
Enter the Price of the Product: High
Product not added. Please enter a number [12.34] for the Price.

Enter a Product to add: Apples
Enter the Price of the Product: 7.89
Your Product has been added.
```

Figure 11. Adding new Products in the Command Prompt

Main Body of the Script

The main body of the script starts by calling the *FileProcessor.read_data_from_file* method and assigning it to *lstOfProductObjects*. Then the script moves to a while loop that continues until the break point is hit when the user enters option 4. The menu options are presented with the *IO.output_menu_choices* method, then the user input is obtained with *IO.input_menu_choice* and assigned to *choice_str*.

The loop contains a series of if statements dependent on the value assigned to *choice_str*. If 1 is entered, *IO.output_product_data* is called to print the current list of Product objects in *lstOfProductObjects*.

```
# Loops until break is hit when 4 is entered
while (True):

    # Show user a menu of options
    IO.output_menu_choices()

    # Get user's menu option choice
    choice_str = IO.input_menu_choice()

    # Show user current data in the list of product objects
    if choice_str.strip() == '1':
        IO.output_product_data(list_of_product_objects = lstOfProductObjects)
        continue
```

Figure 12. Beginning of the while loop in the main body of the script

If the user enters 2, *IO.input_add_product_data* is called, and the *product_name* and *product_price* that is returned by the method is unpacked into the *product_name* and *product_price* variables. These variables are passed as arguments to the Constructor method when creating a new Product, which is assigned to the variable *product_object*. Then the new *product_object* is appended to the list of Products, and a statement is printed confirming that the Product has been added.

```
# Let user add data to the list of product objects
elif choice_str.strip() == '2':
    # unpacks product_name and product_price to be used below
    product_name, product_price = IO.input_add_product_data()
    # creates new Product object from name and price returned from IO.input_add_product_data()
    product_object = Product(product_name, product_price)
    # adds new Product object to list of objects
    lstOfProductObjects.append(product_object)
    print('Your Product has been added.')
    continue
```

Figure 13. Code in the main body of the script to add a new Product

When 3 is entered, *FileProcessor.save_data_to_file* is called, and the Product objects in *lstOfProductObjects* are saved to the file assigned to the variable *strFileName*, *products.txt*. If 4 is entered, script exists the loop. There is a final else statement to handle other menu entries. In this case, a message to make a selection 1 - 4 is printed and the loop restarts.

```

# let user save current data to file and exit program
elif choice_str.strip() == '3': # saves data
    FileProcessor.save_data_to_file(strFileName, lstOfProductObjects)
    print('Your data has been saved.')
    continue

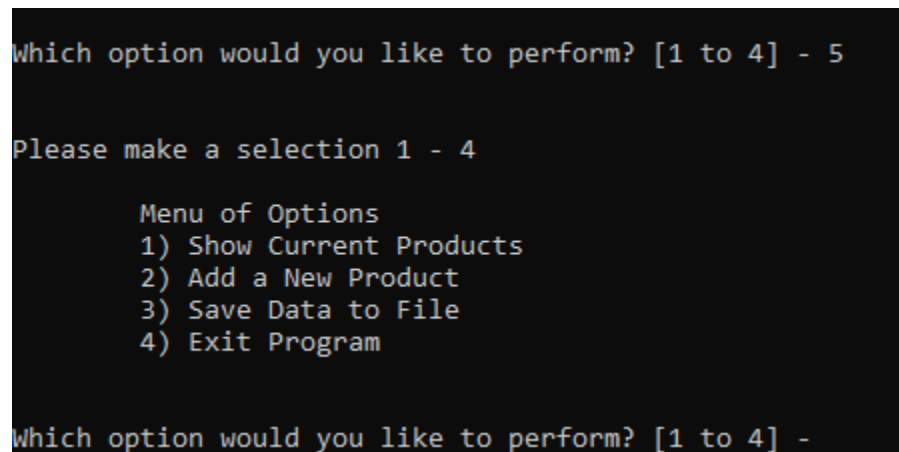
elif choice_str.strip() == '4': # exits program
    print('Goodbye!')
    break # exits loop

else:
    print('\nPlease make a selection 1 - 4') # handles situations when users give invalid input

```

Figure 14. Code for menu selections 3, 4, and other in the main body of the script

Figure 15 below shows the outcome when a user enters the number 5 for the menu selection.



```

Which option would you like to perform? [1 to 4] - 5

Please make a selection 1 - 4

    Menu of Options
    1) Show Current Products
    2) Add a New Product
    3) Save Data to File
    4) Exit Program

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

```

Figure 15. Menu selection other than 1 – 4 in Command Prompt

Summary

In this paper I discussed how I completed a script uses a Product class, FileProcessor class, and IO class to manage product data.