

## **Working with Objects**

### **Introduction**

In this week's lectures, modules and readings, we dove deeper into the concept of classes (which can contain fields, constructors/attributes, properties, and methods) and how to use them to create object instances to manage data as well as to organize program behavior.

In this week's assignment we were asked to utilize this new knowledge of classes and existing knowledge of error handling and working with files to develop an application that will work with Product data (Product Name and Product Price). The application will need to be able to load data from a file, allow the user to see the current data, allow the user to add new data and allow that data to be saved to a file.

### **Pseudocode**

As I was thinking through the assignment, I decided that my application would do the following:

- Ask the user whether they want to reload data from file (extra feature - give them a choice)
- Instruct them of required file name and content structure, waiting for them to confirm
- Handle application start with preloaded data or no data (no file loaded)
- Allow user to see current data in the list
- Allow user to add data to list
- Allow user to load/reload/reset list to original data from file (extra feature)
- Allow user to save data to a file
- Allow user to clear the list (extra feature to allow user to restart with new list)
- Allow user to exit the program
- Handle common errors that may come up – file not found, bad input, etc.

### **Creating a List (using Objects)**

To be able to have any data to work with, I needed the ability to store data in a list. With this week's focus being object classes, I started by creating a Product object class that will contain product name and product price information as attributes. I also created getter and setter properties (special functions) on this object that would require the product name to be a string (non-numeric) and for the product price to be a number (integer or float). If bad input was provided that wouldn't meet this criteria, special exception error classes would be raised.

**Code Snippet 1** below shows the constructor code and the getter/setter properties (to save space only showing the ones for product name).

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

# -- Properties
@property #getter
def product_name(self):
    return str(self.__product_name).title()

@product_name.setter #setter
def product_name(self, value):
    if str(value).isnumeric() == False:
        self.__product_name = value
    else: #is numeric
        raise ProductNameValueCannotBeNumeric
```

**Code Snippet 1** – Constructor and Properties example – Product class

## See Current Data in List

Not having coded the ability for the user to add data yet, I hardcoded the instantiation of several Product objects and loaded them to a list. I then overrode the default `__str__` function (which allowed me to alter the default behavior when the object is printed) and iterated through the list of Product objects to print them to the screen. This allowed me to validate that yes, the application was indeed storing Product objects in memory and that I could print/access their values (which will later be used to show the user the current data in the list).

**Code Snippet 2** below shows the exact code used to output current list contents (memory) to the screen.

```
@staticmethod
def output_current_data_in_list():
    """ Shows the current data in the list
    :param list: (list) of rows
    :return: nothing
    """
    print("Here's the current data in the list ...")
    print("Product Name" + " | " + "Product Price")
    for row in lstOfProductObjects:
        print(row)
```

**Code Snippet 2** – Showing current data in the list – IO class

## Adding Data to List

I was now able to move on to coding the functionality that would allow data to be added to the list by the user (so far, I hardcoded the Product object instantiation) on demand.

I started by commenting out the code that would manually instantiate the Product object and divided the task into two functions, one that would capture the user's input (product name and product price) and second that would use this input to add that data to the bottom of the list (barring any errors).

Testing out the happy path (providing valid data) in the debug mode, I was able to validate that yes, the user's input is being stored in the input variables, the input variables are being stored in a new instance of the Product object and the latest Product object is added to the list of Product objects. I then reused the code from above section 'See Current Data in List' to validate via the UI that the data was in the list.

**Code Snippet 3** below shows the exact code used to both add data to the list (user action) as well as rebuild a list based on the file (initial load or by user action to reload from file).

```
@staticmethod
def add_data_to_list(product_name, product_price, list):
    """ Adds data to a list
        :param product_name: (string) with name of product:
        :param product_price: (string) with the product's price:
        :param list: (list) you want filled with data:
        :return: (list) of rows
    """
    obj = Product(product_name, product_price)
    list.append(obj)
    return list
```

**Code Snippet 3** – Showing current data in the list – Processor class

## Save Data to File

Having confirmed the ability to create list of Product objects in memory and add new data, I was ready to move on to coding functionality to save this data to a file.

For this I was able to reuse earlier code that would create a write connection to the defined products.txt file, iterate through the list of product objects using the *write* function (which will use the redefined `__str__` formatting) and close the file.

I was then able to confirm that the data that I had in memory/list matched what was written to/saved to the products.txt file.

**Code Snippet 4** below shows the exact code used to save list data to the products.txt file.

```
@staticmethod
def save_data_to_file(file_name, list):
    """ Saves data to file
    :param file_name: (string) filename to save data to (in current directory)
    :param list: (list) list to save to file
    :return: nothing
    """
    print("Option 3 selected - Save Data to File")
    objFile = open(file_name, "w")
    for row in lstOfProductObjects:
        objFile.write(str(row) + "\n")
    objFile.close()
    print("List saved to file \n")
```

**Code Snippet 4** – Showing current data in the list – File Processor class

## Reading/Loading from File

Having confirmed the ability to save to the file, I was ready to move on to coding functionality to load this data from a file.

For this I was able to reuse earlier code that would create a read connection to the defined products.txt file and for each row in the file - split each row into two data elements using the comma separator (product name and product price) and add it as a new instance of Product object to the list (was able to reuse the code from 'Adding Data to List' since once I had the results of the split function into memory, it would follow the same process to add it to the previously empty working list).

I added this code at the beginning of the main (to make sure it runs once at the start of the application) that would attempt to load data in the products.txt file (file was already available since I had saved it just before, else I would have had to stage it).

I was then able to confirm that yes, the application is able to load data from the file back into working memory/list for the user to interact with.

**Code Snippet 5** below shows the exact code used to read data from the products.txt file and load it into the list of Product objects.

```
@staticmethod
def read_data_from_file(file_name):
    """ Reads data from file
    :param file_name: (string) filename to read data from (in current directory)
    :return: nothing
    """
    try:
        file = open(file_name, 'r')
        for row in file:
            product_name, product_price = row.split(",")
            Processor.add_data_to_list(product_name.strip(), product_price.strip(), lstOfProductObjects)
            print("Preloading to list from file " + file_name)
    except Exception as e:
        #print(e)
        print("File was not found. Data not loaded. Starting application with no data..") #error handling
```

### Code Snippet 5 – Loading/Reading Data From File – File Processor class

## Error Handling

Having confirmed that all the happy paths worked for all scenarios, I was now ready to deal with the following situations:

1. File products.txt not found - unable to read it
2. User provided bad input - Product name is a number
3. User provided bad input - Product price is not a number

I handled #1 by using a try / except block in the *read\_data\_from\_file* function (shown above) within the FileProcessor class. If the attempt to read from the file returns an exception, it would gracefully print the 'File was not found' error to the screen and proceed to start the application without data being preloaded. To do this, I added an extra feature (described in the 'Extra Features' section below) that would ask whether the user wanted to do an initial load from file and wait for confirmation (to give user time to create or find the products.txt file and place it in that directory in the valid format).

I handled both #2 and #3 by using the setter properties of product name and product price within the Product class. When the product name is numeric (should be string), it will throw a custom *ProductNameValueCannotBeNumeric* error. When the product price is not numeric, it will throw a *ProductPriceValueMustBeNumeric* error. Both these errors are then caught using the try/except block within the section that handles adding data to the list. If either of these exceptions are thrown, data is not added to the list and the user is advised to try again.

**Code Snippet 1** above showed how bad input would result in an error being raised on the setter function. **Code Snippet 6** below shows the exact error handling using the try/except and the separation of the output from the main into dedicated output functions within the error class (intentionally wanted the IO class to only contain happy path).

```
# User Option 2- Add Data to the list
elif menu_choice_str.strip() == '2':
    print("Option 2 selected - Add Data to List")
    product_name, product_price = IO.input_data_to_add_to_list() #capture input
    try:
        lstOfProductObjects = Processor.add_data_to_list(product_name, product_price, lstOfProductObjects)
        print("Item added \n")
        IO.output_current_data_in_list() #show updated list
        continue
    except ProductNameValueCannotBeNumeric:
        ProductNameValueCannotBeNumeric.output_product_name_value_cannot_be_numeric_error() #error output
    except ProductPriceValueMustBeNumeric:
        ProductPriceValueMustBeNumeric.output_product_price_value_must_be_numeric_error() #error output
```

### Code Snippet 6 – Error Handling of bad input – Main

Spending significant time entering bad values for one or both of the fields and then confirming the happy paths once again, I was satisfied that the application behavior and its error handling was working as expected.

### Extra Features

I wanted to add 3 extra features to the application beyond the minimum requirements:

1. Give user ability to decide at application start whether they want to preload data from file
2. Allow user to clear the list
3. Allow user to reload/reset the data in memory to what's in the file at any time

I handled #1 by giving the user a yes/no (y/n) input prompt and based on that prompt execute *read\_data\_from\_file* function or simply print the user choice (the decline of the data preload) and start the application.

#2 was handled by simply clearing out the list of Product objects in memory and then showing the now empty list to the user.

I handled #3 very similarly. When user enters '4' in the Menu of Options, the application would first clear out the list of Product objects in memory and then execute *read\_data\_from\_file* function.

### Bringing It All Together

Using the various sections above that I had individually tested one by one, it was time to bring the flow of the application together. The flow is as follows:

1. At the beginning of the application, the user will be given a choice whether to load data from a file. Based on this choice, data will either be loaded from the file or the user will start an empty list. User is shown the current state of the list after this choice.
2. The user is then shown list of potential Menu options. The Menu Options include the ability to Clear the list, See current data in the list, Add data to the list, Save data to a file, Reload data from a file or Exit the program. All Menu options but Exit will result in action being performed and Menu options shown again after.

### 3. Error Handling:

- In the case of the two errors concerning bad input for product name or product price, the application gracefully advises the user to try again with proper input.
- In the case of File Not Found error, the application starts without data being preloaded (this was intentional) – user can then load the data using the reload function once presented with the Menu options.

After being satisfied that the entire application flow and error handling was working as expected, my last step was to ensure the application and its code was well documented. I added/clarified doc strings and other comments for the various classes and methods to ensure a future developer would be able to understand the code and reuse it. At this point, I deemed the assignment complete

## Demo

So as not to duplicate everything in the demo, I will share some screenshots from PyCharm and some from Command Prompt as proof that the application is working in both. I will also show the contents of the products.txt file to show that the file writes and reads are happening successfully.

In **UI Figure 1** below, we see an example of user selecting to reload data from file, the application successfully loading that data into the list, showing user the data currently in the list (same as option 1) presenting user with menu of options and awaiting input.

```
Would you like to reload data from file ? (y/n) y

Please enter data into file products.txt in the current directory. Each product name and product price pair must be on its own row.
Please separate the product name and product price with a comma. No spaces please.
When ready, press <Enter> to import data from the file. You will later have the option to clear the list to start with a new list.

Preloading to list from file products.txt
Here's the current data in the list ...
Product Name | Product Price
Lamp,10.5
Table,202.3
Shade,17.0

Please select from Menu of Options
0) Clear List
1) See Current Data in List
2) Add Data to List
3) Save Data to File
4) Reload Data from File
5) Exit the program

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

**UI Figure 1** – File Data Loaded at Initial Load (UI) - PyCharm

In **UI Figure 2** below, we see result of user selecting option 2 (Add Data), entering 'Light' for product name and '7' for price. The user is advised that the item was added to the list and the updated list is shown.

```
Option 2 selected - Add Data to List
What is the Product you want to add? - Light
What is Products's price? (Number only, no $) - 7
```

```
Item added
```

```
Here's the current data in the list ...
```

```
Product Name | Product Price
```

```
Lamp,10.5
```

```
Table,202.3
```

```
Shade,17.0
```

```
Light,7.0
```

**UI Figure 2** – Adding new Product (UI) - PyCharm

In **UI Figure 3** below, we see the result of user selecting option 3 (Save data to File). The application saves the data to file products.txt in the background, indicates to the user that the data/list was saved to file and shows again the current data in the list (same as option 1). In this figure we also confirm that the product.txt file is indeed saved with the same data.

```
Option 3 selected - Save Data to File
```

```
List saved to file
```

```
Here's the current data in the list ...
```

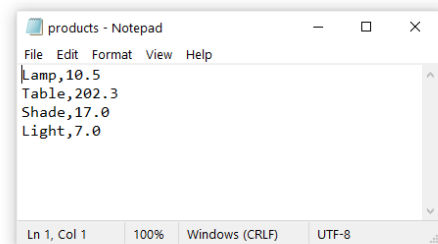
```
Product Name | Product Price
```

```
Lamp,10.5
```

```
Table,202.3
```

```
Shade,17.0
```

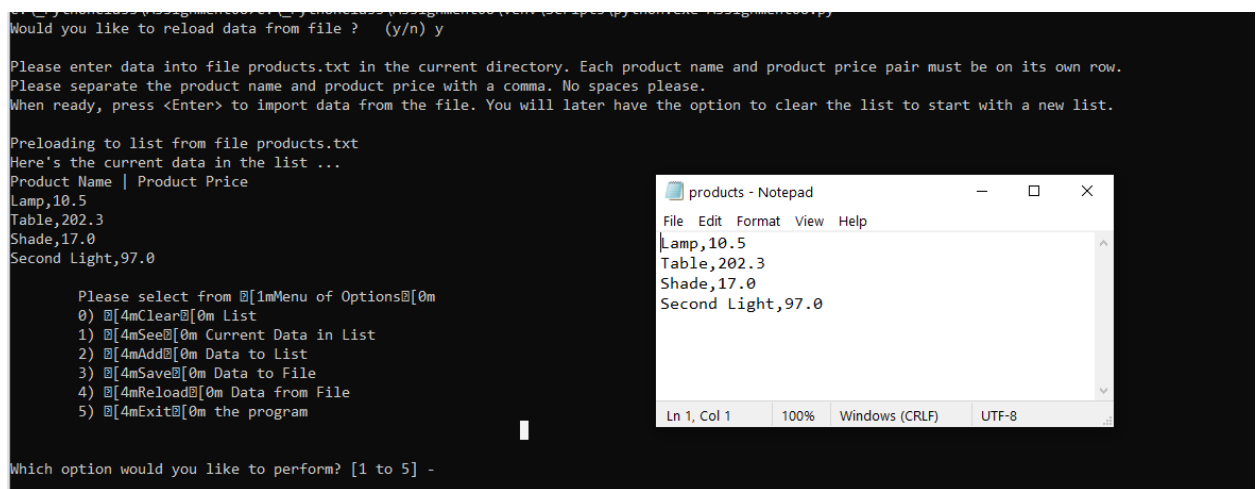
```
Light,7.0
```



**UI Figure 3** – Data Saved to File (UI) – PyCharm/Txt file contents

In **UI Figure 4** below, we see the result of the user making a change to the file (last item in the list). After running the application and the user confirming that they want to reload the current data, we now see in Command Prompt that the latest data has been read into the application (the same behavior/code runs when user selects option 4 to reload data from file).





```

C:\Python39\Scripts>python assignment08.py
Would you like to reload data from file ? (y/n) y

Please enter data into file products.txt in the current directory. Each product name and product price pair must be on its own row.
Please separate the product name and product price with a comma. No spaces please.
When ready, press <Enter> to import data from the file. You will later have the option to clear the list to start with a new list.

Preloading to list from file products.txt
Here's the current data in the list ...
Product Name | Product Price
Lamp,10.5
Table,202.3
Shade,17.0
Second Light,97.0

Please select from Menu of Options
0) Clear List
1) See Current Data in List
2) Add Data to List
3) Save Data to File
4) Reload Data from File
5) Exit the program

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

products - Notepad

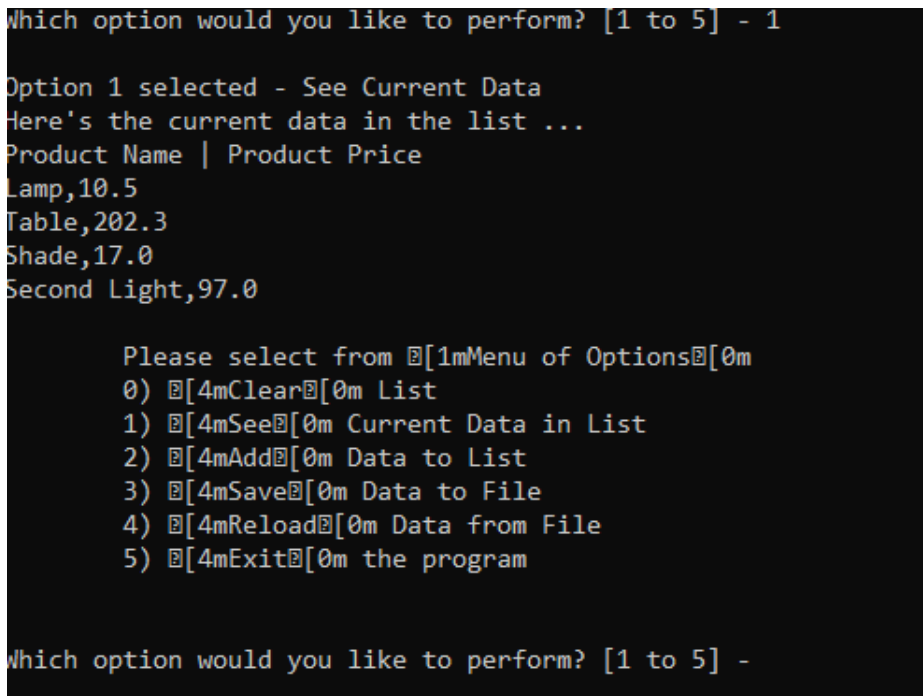
```

File Edit Format View Help
Lamp,10.5
Table,202.3
Shade,17.0
Second Light,97.0
  
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

**UI Figure 4** – Data Loaded from File (UI) –Command Prompt/Txt file contents

To provide another screenshot from the Command Prompt, **UI Figure 5** validates once more the functionality of seeing the current data in the list.



```

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

Option 1 selected - See Current Data
Here's the current data in the list ...
Product Name | Product Price
Lamp,10.5
Table,202.3
Shade,17.0
Second Light,97.0

Please select from Menu of Options
0) Clear List
1) See Current Data in List
2) Add Data to List
3) Save Data to File
4) Reload Data from File
5) Exit the program

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

**UI Figure 5** – Current Data in the list (UI) –Command Prompt

Now back to PyCharm and to showcase the error handling.

**UI Figure 6** shows the error received when the user chooses to preload the data from the file and the products.txt file is not in the directory.

```
File was not found. Data not loaded. Starting application with no data..  
Here's the current data in the list ...  
Product Name | Product Price
```

**UI Figure 6** – Error Message: No file found to load from – UI- PyCharm

**UI Figure 7** shows the error received when the enters a non-numeric value for the product price.

```
Option 2 selected - Add Data to List  
What is the Product you want to add? - ValidName  
What is Products's price? (Number only, no $) - InvalidPrice  
  
Error - You must enter a product price value that is numeric  
Item not added. Try again.
```

**UI Figure 7** – Error Message: Invalid input for product price – UI - PyCharm

**UI Figure 8** shows the error received when the enters a numeric value for the product name.

```
Option 2 selected - Add Data to List  
What is the Product you want to add? - 000  
What is Products's price? (Number only, no $) - 000  
  
Error - You must enter a product name value that is not numeric  
Item not added. Try again.
```

**UI Figure 8** – Error Message: Invalid input for product name – UI - PyCharm

## Summary

This week, we dove deeper into our knowledge of classes by working with different types of classes (data/object classes vs processing classes). We used object classes by instantiating instances of that class (using constructors/attributes) to manage data and processing classes to manage general application behavior.

We were also introduced to properties (getters and setters) and different types of methods (static vs dynamic) to organize code.

This was a fun albeit challenging assignment. Looking forward to learning new concepts in the coming weeks that will allow me to build more complex and useful applications.