Phillip Thompson
6/24/23
IT FDN 110 A
Assignment 8
github.com/philthom10/IntroToProg-Python-Mod08

Assignment 8 – Final – Custom Classes

1. Introduction

Module 08 this week focused on custom classes and Object-Oriented-Programming. This assignment is the Final for the course, so it incorporates the new information taught in Module 08 as well as demonstrating how that works with everything else we have learned in the course this quarter. A starter file with pseudo-code was given for this assignment, then it was completed using classes for creating objects, classes with static method functions, reading and writing to a file, and it incorporates error handling.

The program itself takes user input for products and their price, creates objects for the various products, then saves those to a text file. The program also reads data that is existing on the text file. Exception handling is used throughout the program to enhance user experience and ensure correct function.

2. Writing the Code

The starter code came with existing classes named *Product*, *FileProcessor*, and *IO.* My program completed the code in those classes and added a new class called *DataProcessor* that will be described later in this document.

2.1. *Product* Class

Knowing that the product information would be created as objects and then processed and written to a file, I decided to start writing my code in the Product class, so I would have those objects to use in completing the processing classes. From the information in the Module this week as well as the Psuedo-code To Do line, I knew I would need a constructor, a "getter" and "setter" for properties, and methods within the class.

The existing docstring for the class also listed the properties that would be needed, so that was the starting point I went with. The docstring describing the properties is shown below (Listing 1)

```
properties:
        product_name: (string) with the product's  name

        product_price: (float) with the product's standard price
```

Listing 1: Property description in docstring from starter file

2.1.1. *Constructor*

I first created the constructor function within the class, shown below (Figure 1).

```
# -- Constructor ---
def __init__(self, name, price=0.00):
    self.product_name = name  # defining the attribute, will be managed by property
    self.product_price = price
```

Figure 1: Product constructor

The *__init__* function is the constructor, and I used the parameters *name* and *price*. I set the *price* parameter equal to a float value of 0.00.

2.1.2. *Properties*

Next, I defined the attributes of *product_name* and *product_price* that are managed by the properties described next, and shown below (Figure 2).

The first property of *product_name* was defined with its "getter" and "setter". The getter @property for *product_name* simply returns a string of the product name with Title case formatting.

```
    # -- Properties --
    3 usages
    @property
    def product_name(self):
        return str(self.__product_name).title()  # private attribute of product name with formatting

    1 usage
    @product_name.setter
    def product_name(self, value):
        if not str(value).isnumeric():  # exception handling checks to make sure product name is not numeric.
            self.__product_name = value
        else:
            raise Exception("The product name should not be a number.")
```

Figure 2: Product_name getter and setter functions

The setter function then sets the *product_name* attribute to the parameter *value*. This is the first place that I used exception handling. The handling checks to ensure the *value* parameter is not numeric before assigning it to the *product_name* attribute. It if is numeric, it raises the exception saying the name should not be a number.

Both of these functions use a private attribute *__product_name* with the double-underscore preceding the attribute to make it private, so it can't be accessed directly outside of this class.

The code for the *product_price* getter and setter was similar to that of *product_name* properties, and is shown below (Figure 3).

```
    3 usages
65  @property
66  def product_price(self):
67      return self.__product_price
68
    1 usage
69  @product_price.setter
70  def product_price(self, value):
71      if str(value).replace(".", "").isnumeric():  # exception handling checks to make sure the value is a float
72          self.__product_price = round(float(value), 2)  # sets property and rounds price to two decimal places
73      else:
74          raise Exception("The product value should be a number in dollars and cents, e.g. '19.99' ")
```

Figure 3: Product_price getter and setter functions

I chose to use the *round()* function to round the float of the price to two decimal places to make it accurate as a price. The exception handling checks to see if the price value is numeric and if it is, then it sets the entry to the property. Since this is a float value, the *isnumeric()* does not work with the decimal present. Therefore, I replaced the decimal with a blank using the *replace()* function.

2.1.3. *Methods*

Finally, I completed the code for the methods. These simply edit the default *__str__()* method to return as a csv, and then use the *to_string()* method as an alias for this. See code for this below (Figure 4).

```
76      # -- Methods --
77      def to_string(self):
78          """ Returns a string with the product data """
79          return self.__str__()  # creates alias for the __str__ method
80
81      def __str__(self):
82          """ Returns a string with the product data """
83          return self.product_name + ',' + str(self.product_price)  # returns string csv line with product data
```

Figure 3: __str__ method and alias

I also added a method called *display()* that prints feedback when the object is created. See Below (Listing 2) for code.

```
def display(self):
        print("Item", self.product_name, "with price", self.product_price,
"created")
```

Listing 2 display() Method

2.2. *Processing Section*

The next section of the code is the Processing section. This section has *FileProcessor* class that was started in the starter file as well as a new class *DataProcessor* that I added to update the list of data within the program before writing to a file. The key difference between the Processing classes and the Product class is that the Product class uses Object Oriented Programming by creating objects within the class using methods that are not called outside of the class, while the Processing and IO classes use the staticmethod methods that are called in the main code.

2.2.1. *FileProcessor Class*

From the docstring in the starter file, I knew the methods within the class that I needed to create were the *save_data_to_file* and *read_data_from_file*. As the data was to be saved to a .txt file, I knew that the program should use the code similar to assignment 06, rather than the pickling methods used in Assignment 07. The code for the two methods are shown below (Figures 4 and 5).

```
1 usage
@staticmethod
def read_data_from_file(file_name, list_of_rows):
    """ Reads data from a file into a list of dictionary rows

    :param file_name: (string) with name of file:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    list_of_rows.clear()  # clear current data
    try:
        file = open(file_name, "r")  # if file exists, read it and do the following
        for line in file:
            product, price = line.split(",")
            row = {"Product": product.strip(),
                   "Price": price.strip()}  # creates dictionary row from each line in file
            list_of_rows.append(row)  # adds each row to the list
        file.close()
    except FileNotFoundError:  # if file doesn't exist, it creates the file
        file = open(file_name, 'w')
        file.close()
    return list_of_rows

```

Figure 4: read_data_from_file Method

While the code reading and writing data as lists of dictionary rows is something that I have demonstrated in previous assignments, this was the first time I used exception handling within the file processing code. The Try-Except code opens and reads the file if one is existing, but if there is no existing file, a blank one is created and closed. I knew going into this assignment that this is a common use of exception handling, but the usefulness of it became quickly apparent when I first tried running the program without this exception handling the first time – it stopped and caused an error since there was no file to read from.

```
2 usages
@staticmethod
def write_data_to_file(file_name, list_of_rows):
    """ Writes data from a list of dictionary rows to a File

    :param file_name: (string) with name of file:
    :param list_of_rows: (list) you want filled with file data:
    :return: (list) of dictionary rows
    """
    file = open(file_name, "w")  # PT opens file with passed in file_name parameter
    for row in list_of_rows:
        file.write(str(row["Product"]) + "," + str(row["Price"]) + "\n")  # saves each row to file
    file.close()
    return list_of_rows
```

Figure 5: save_data_to_file Method

2.2.2. *DataProcessor Class*

The *DataProcessor* class that I added just has one method to save the objects that have been created from user input into the list of rows prior to writing to a file. The code is shown below (Figure 6).

```
1 usage
@staticmethod
def add_data_to_list(list_of_rows, new_product_row):
    """ Adds new data row to existing list of data

    :param list_of_rows: (list) with rows of products and prices:
    :param new_product_row: (dictionary) dictionary row with new product and price to add:
    :return: (list) of dictionary rows
    """
    row = new_product_row
    list_of_rows.append(row)  # appending table with new entries as new dictionary row
    return list_of_rows
```

Figure 6: add_data_to_list Method

The code for this method is similar to what I have shown in a few of the previous assignments, appending the list with the new product row.

2.3. *Presentation Section*

The presentation section has one class *IO* that was started in the starter file. The docstring for this class named three methods to be used: *print_menu_items(), print_current_list_items(), and input_product_data().* I created the additional methods *menu_choice()* to capture the user menu choice and *input_yes_or_no()* for asking the user if they want to save unsaved data or not before exiting.

2.3.1. *menu_choice() Method*

The *menu_choice()* method captures the user input selection of which menu option they would like to perform. See below (Figure 7) for an image of the code. This code is similar to similar to what I did in Assignment 7 for capturing input, so I used that as a framework. The code compares to the list of valid

choices inside of a while loop, so it keeps asking for a valid choice until one is selected. I tested this this block of code separately, but it did not work at first and kept repeating the loop. Upon investigating the issue, I found that the reason it kept repeating is that I typed the list as integers 1,2,3,4 not strings "1","2","3","4", and since the user input is captured as a string, it neve matched with a valid choice in the list. This is a good reminder that even with these more complex programs there can be simple errors on the basics like this, which is I why I tested this block on its own before I ran it as part of the larger program.

```
202     @staticmethod
203     def menu_choice():
204         """ Gets the menu choice from a user
205
206         :return: choice (string) menu selection
207         """
208         choice = None
209         valid_choices = ["1", "2", "3", "4"]  # created a list of valid choices for error handling
210         while choice not in valid_choices:  # repeats loop until valid option is chosen, else kicks exception
211             try:
212                 choice = input("Choose a menu option: ")
213                 if choice not in valid_choices:
214                     raise InvalidChoiceException
215             except InvalidChoiceException:
216                 print("Invalid option. Please select an option 1 through 4 from the menu.")
217         return choice
```

Figure 7: menu_choice() Method

I used exception handling to ensure the user selected one of the acceptable numbers. A custom exception class is raised if the selection is not in the list of number 1 through 4. The code for this custom exception is shown below in Listing 2.

```
class InvalidChoiceException(Exception):  # custom exception to be used later
    """Raised when input is invalid."""
    Pass
```

Listing 3: Custom exception class

2.3.2. *input_new_product() Method*

The next noteworthy method in the *IO* class is the *input_new_product_info()* method. This captures the user input for product name and price, which are later used to create the objects for products using the *Product* class described in section 2.1 of this document. The code for this is shown below (Figure 8).

The exception handling raises the custom InvalidEntryFormat exception if either the product name is numeric or the price is not numeric. While this covers the most likely cases and shows the use of exception handling, the main limitation of how I wrote it is if there is a product that is intended to be entirely numeric, it would still prompt the user for a non-numeric entry.

```python
    @staticmethod
    def input_new_product_info():
        """  Gets product and prices values to be added to the list

        :return: (string, float) with product and price
        """
        while (True):
            try:
                print()  # Add an extra line for looks
                product = input("Product name: ").title().strip()  # captures user input and uses strip method
                if str(product).replace(".", "").isnumeric() == True:  # includes replacing . in case there is a price entered
                    raise InvalidEntryFormat
                else:
                    break  # and captures product input
            except InvalidEntryFormat:
                print(" **  A numeric value is not expected for a product name. Please enter again. ** ")

        while (True):
            try:
                price = input("Standard price: (00.00): ").strip()  # captures price and converts to float
                if str(price).replace(".", "").isnumeric() == False:  # exception handling checks to make sure the value is a float
                    raise InvalidEntryFormat
                else:
                    break  # and captures product input
            except InvalidEntryFormat:
                print(" **  A numeric price value in the format e.g. 19.99 is required ** ")
        print()  #  Add an extra line for looks
        return product, price
```

Figure 8: input_new_product_info() Method

The code for the custom exception class is shown below in Listing 3.

```python
class InvalidEntryFormat(Exception):  # custom exception to be used later
    """Raised when input format type is invalid"""
    Pass
```

Listing 4: Custom exception class

2.4. *Main*

I wrote the main body this program as a *main()* function similar to the Chapter 8 textbook "Class Critter" example **(DAWSON 245).** As an attempt to start using some of the information I was reading about in Module 09, I called this *main()* function using the if statement as shown below (Figure 9), like it is described in Chapter 9 of the class textbook **(DAWSON 269).** Module 09 is outside of the scope of this program, so this was just for my own learning. I inherited the global variables into the main class as parameters named the same as the global variables so that I could easily copy from a code that was first written just in the main body of the script into this this *main()*class instead.

```python
if __name__ == "__main__":  # if condition statement to run directly
    main(lstOfProductObjects, strFileName, save_status)  # runs main code with arguments of variables
    input("\n\nPress the enter key to exit.")
```

Figure 9: Calling main() function with if statement

2.4.1. *Option 2: Adding a new Item*

The majority of the *main()* section is similar to previous assignments, so I will not go into detail of each line, but the section worth noting is the block for the user selects menu item 2 to add a new item. This section is what is called to create the new objects using the *Product* class. The code for this is shown below (Figure 10).

```
308         elif menu_choice_str == '2':  # add new item
309             new_product_name, new_product_price = IO.input_new_product_info()  # defining new variables with
310             obj_new_product = Product(name=new_product_name, price=new_product_price)  # creates new object product
311             obj_new_product.display() # prints feedback to user
312             # print("New product:", obj_new_product, "added")  # displays feedback to user of product creation (not using)
313             dic_new_product_row = {"Product": obj_new_product.product_name,
314                                    "Price": obj_new_product.product_price}  # creating new dictionary row from object attributes
315             lstOfProductObjects = DataProcessor.add_data_to_list(list_of_rows=lstOfProductObjects,
316                                                                  new_product_row=dic_new_product_row)
317             save_status = False  # sets flag to false upon addition of new data
318             continue
```

Figure 10: Creating Object for New Product

The name and price are arguments created from the *IO* class and passed to the *Product* class for creation of the object. Line 311 calls the *display()* method of the new product to print back a line saying it was created. I originally provided this feedback as written in line 312, which uses the string created in the *to_string()* method to print feedback. I commented it out since I chose to use the *display()* method, but either option works – I wanted to try both of them as it helped me with my understanding of how object methods work.

3. Running the Program

An image of the program running in PyCharm is shown below (Figure 11).

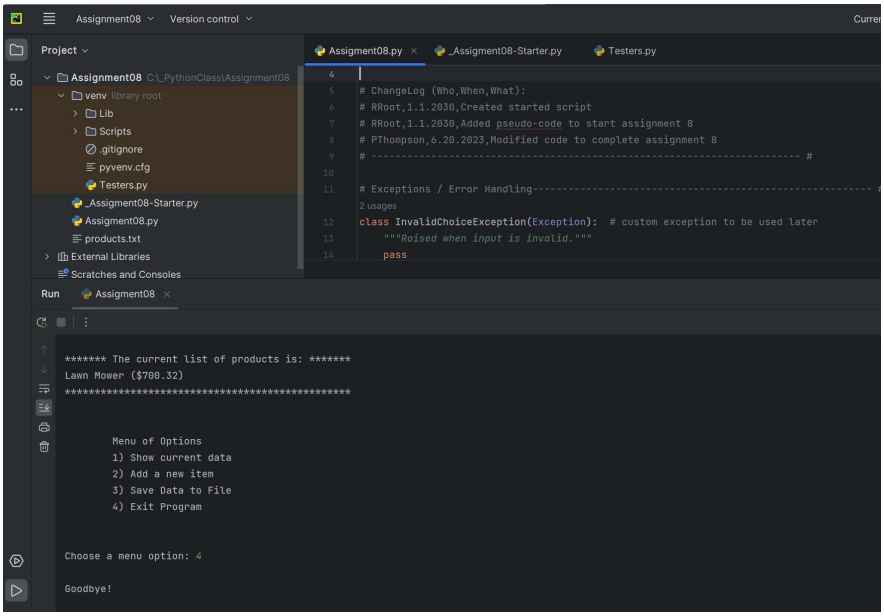

Figure 11: Program Running in PyCharm

An image of the program running in the Windows Command Window is shown below (Figure 12).

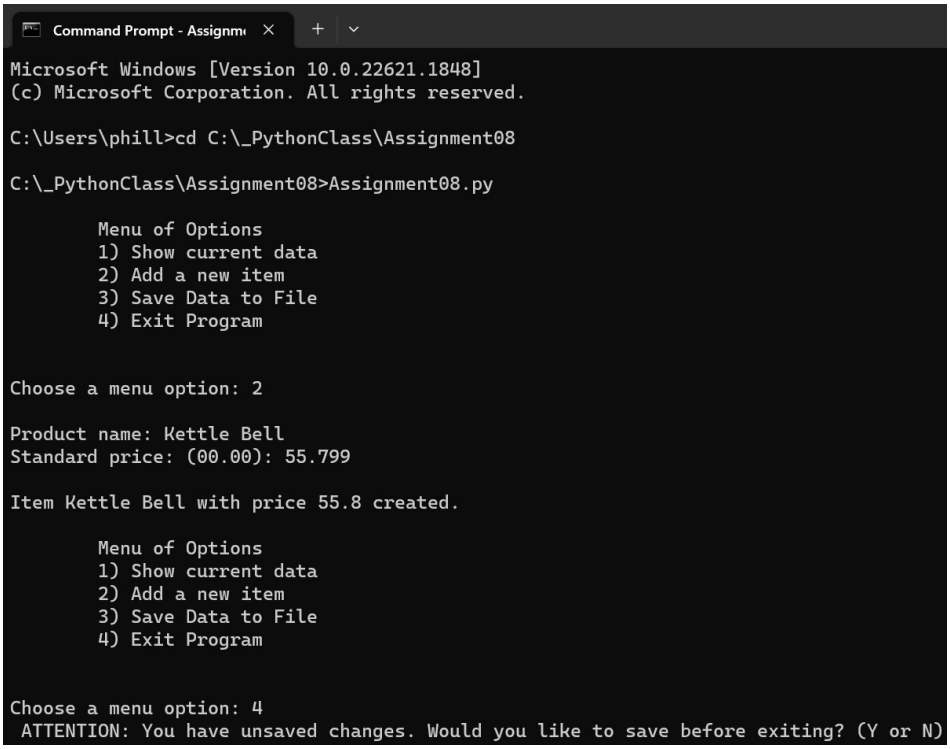

Figure 12: Program Running in Windows Command Window

4. Summary

This program for recording items and their prices was a good opportunity to tie together everything we have learned in the course this quarter. While the use of Object-Oriented Programming is not necessary to accomplish a similar end result in the program, the program shows how to simply create objects using a constructor and properties within a class. This lays the foundation for making more interesting programs that allow for more robust use of objects, such as updating properties as part of a more interactive program. Overall, writing this program was straight-forward, but I did get more experience with troubleshooting some of the lines that did not work. I found it helpful to step through certain sections to make sure they were working, which saved some troubleshooting down the line.

Works Cited

Dawson, Michael. *Python Programming for the Absolute Beginner*, Third Edition, Cengage Learning, 2010