

08: Objects & Classes

Introduction

This week revolved around Objects and Classes. The attached, culminating assignment follows the same premise as the previous ones: a menu that allows the user to add entries to a list and to then save the list to a file that can be retrieved later. What made this distinct from the others was that the majority of the heavy lifting would be done in classes that had to be fully implemented.

Planning

The script was laid out into four parts: Data, Processing, Presentation and the main body. Data had a Product() class which would make up the objects acting as entries for the list. Processing was performed with FileProcessor() whose sole purpose was to read and write to file unlike previous versions. Presentation had the IO() class for gathering user input and outputting the menu and list of objects. Lastly was the menu which was relatively unchanged from recent assignments.

Product Class

Product was the easiest of the classes to implement. Product consisted of two fields: string product_name to represent the product name & float product_price for the product price. Besides the setters, getters, and the initialization function I implemented the to-string (__str__) function for Product so that when a Product was printed it would print a string in a readable format. (Figure 1).

```
class Product:
    """Stores data about a product:..."""
    product_name = ""
    product_price = 0.0

    def __init__(self, product_name, product_price):
        self.product_name = str(product_name)
        self.product_price = float(product_price)

    def __str__(self):
        string_representation = "(" + self.product_name + ", $" + str(self.product_price) + ")"
        return string_representation

    def getName(self):
        return self.product_name

    def getPrice(self):
        return self.product_price

    def setName(self, product_name):
        self.product_name = str(product_name)

    def setPrice(self, product_price):
        self.product_price = str(product_price)
```

Figure 1: Product Class

Menu

The menu wasn't different from previous assignments; it mainly consisted of a while-loop that presented a list of options with the bulk of the greater work done by the IO Class. The only exception was try-catch block to handle an `UnboundLocalError` exception for wrong input when attempting to add a new Product to the list (Figure 2).

```
elif choice == '2':
    print("Adding...")
    try:
        product_name, product_price = myIO.new_data()
        newProduct = Product(product_name, product_price)
        lstOfProductObjects.append(newProduct)
    except UnboundLocalError as e:
        print(e.__str__())
        print("Bad Entry. Canceling...")
    continue
```

Figure 2: Handling Bad Input

IO Class

The second class to be implemented was the IO class. In this class were functions to print the menu, asking for user input, the printing of data, and asking the user to input a new product (Figure 3).

```
"""
...
"""
menu = "MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|"
# TODO: Add code to show menu to user
def print_menu(self):
    print(self.menu)
# TODO: Add code to get user's choice
def get_choice(self):
    user_choice = input("INPUT: ")
    return user_choice
# TODO: Add code to show the current data from the file to user
def print_data(self, lstData):
    lstData = list(lstData)
    for product in lstData:
        print(product)
# TODO: Add code to get product data from user
def new_data(self):
    try:
        new_product_name = str(input("Product Name: "))
        new_product_price = float(input("Product Price: "))
    except ValueError as e:
        print(e.__str__())
        print("String for Name & Number for Price!!!")
    return new_product_name, new_product_price
```

Figure 3: IO Class

The latter two required the most effort within `IO()`. Rather than explicitly verifying each entry in the list of Product objects was in fact a Product I chose for `print_data()` to have a for-loop iterate through the list, using

the overwritten to-string function I had implemented in the Product Class to implicitly print a readable series of entries. new_data() meanwhile asked for the product name and price and casted them to a string and float in-order; to handle ValueError the input & casting was placed in a try-catch block.

FileProcessor

The last class to be implemented was the most challenging. I adapted code from Assignments 6 to make the read and write functions. Assignment 6's code involved iterating through a list, writing to a file by breaking down a dictionary into a string, and reading from a file by breaking a string up to make a dictionary to add to a list. For Assignment 8 it was mostly unchanged except rather working with dictionaries taken from and/or added into a list, read_data_from_file() & save_data_to_file() used the Product objects instead of dictionaries (Figures 4 & 5).

```
def read_data_from_file(file_name, list_of_product_objects):
    """Reads data from a file into a list of dictionary rows.."""
    list_of_product_objects.clear() # clear current data
    try:
        file = open(file_name, "r")
        for line in file:
            line.strip("\n")
            product_name, product_price = line.split(",")
            newProduct = Product(product_name, product_price)
            list_of_product_objects.append(newProduct)
        file.close()
    except FileNotFoundError as e:
        print(e.__str__())
        print(file_name + " doesn't exist!")
    except BaseException:
        print("Unknown Error!")
    return list_of_product_objects

# TODO: Now code to process data to a file
def save_data_to_file(file_name, list_of_product_objects):
    try:
        file = open(file_name, "w")
        for product in list_of_product_objects:
            line = product.getName() + "," + str(product.getPrice()) + "\n"
            file.write(line)
        file.close()
    except FileNotFoundError as e:
        print(e.__str__())
        print(file_name + " doesn't exist!")
    except BaseException:
        print("Unknown Error!")
    return list_of_product_objects
```

Figures 4 & 5: Read & Write functions

Testing

Testing was done in PyCharm and standard Windows Command Prompt shell. Products.txt was pre-written with three products labeled 'a', 'b', & 'c' priced \$1, \$2, & \$3 respectively. The file was successfully read with each entry converted into a Product object that could be printed. (Figure 6).

```
C:\_PythonClass\Assignment08>python.exe "C:\_PythonClass\Assignment08\Assignment08-Starter.py"
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 1
Printing...
(a, $1.0)
(b, $2.0)
(c, $3.0)
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT:
```

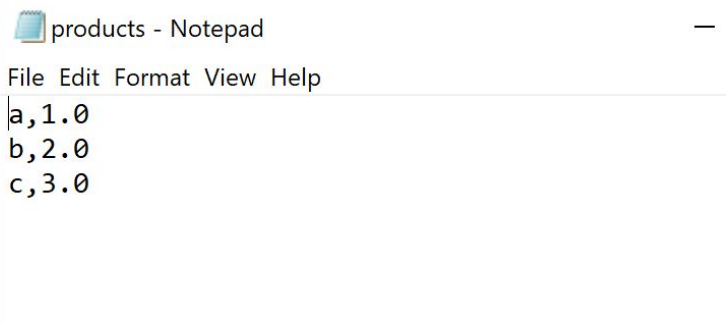


Figure 6: Reading from File and Printing.

A new entry was added to the list, (d, \$4.00), and saved. Opening products.txt again shows that the new object was successfully created and saved (Figure 7).

```
C:\_PythonClass\Assignment08>python.exe "C:\_PythonClass\Assignment08\Assignment08-Starter.py"
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 1
Printing...
(a, $1.0)
(b, $2.0)
(c, $3.0)
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 2
Adding...
Product Name: d
Product Price: 4
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 3
Saving...
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 1
Printing...
(a, $1.0)
(b, $2.0)
(c, $3.0)
(d, $4.0)
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT:
```

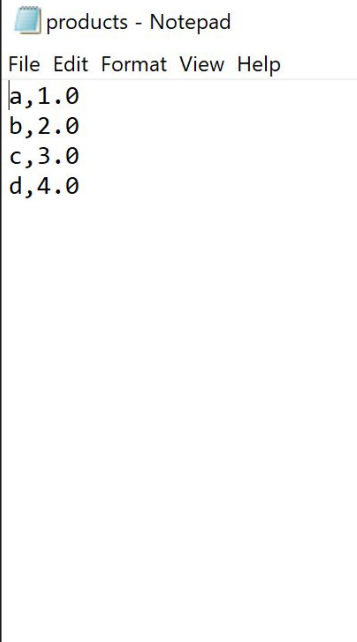


Figure 7: New Product object added and successfully saved.

Two examples of exception handling were performed by attempting to add a new Product with values 'e' & 'fake'. As the program expects a string THEN a float the 'fake' input will cause the Try-Catch block to throw a ValueError error within the new_data() function for wrong data type, a second Try-Catch block from the greater menu will also throw an UnboundLocalError error as it does not have a string and float to create a new product. This will then cancel the 'add' action and send the user back to the menu (Figure 8).

```
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 1
Printing...
(a, $1.0)
(b, $2.0)
(c, $3.0)
(d, $4.0)
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT: 2
Adding...
Product Name: e
Product Price: fake
could not convert string to float: 'fake'
String for Name & Number for Price!!!
local variable 'new_product_price' referenced before assignment
Bad Entry. Canceling...
MAIN MENU: |1-Print |2-Add |3-Save |4 - Quit|
INPUT:
```

Figure 8: Exception Handling with two Try-Catch block throws.

Summary

The implementation of the code was not the difficult part of this assignment. What was important was to plan out how each class would interact with each other and the greater whole. Starting with bare code stubs to point to where led to what the rest of the implementation was filling in the blanks so that expected inputs would lead to expected outputs. The lesson I have learned is that what was between two 'Point A' & 'Point B' matter less than the fact the program could get from A to B.