Erin Leggett
December 7, 2021
IT FDN 110A
Module 08 - Assignment 08

CREATING AND EMPLOYING CLASSES AND METHODS

OBJECTIVE

Describe the steps taken to create the Python script attached to this assignment.

INTRODUCTION

The primary focus of this week's course content was how to employ abstraction in Python by defining custom classes and methods to process and handle data. As with other languages, Python has a number of ways to handle data without directly affecting or altering its source. This also allows the developer to utilize and manipulate multiple instances of a piece of data across the program to suit their needs.

RESEARCH

In addition to watching the course video, I did some independent research to answer lingering questions about properties and attributes - primarily their use cases. I also spent some time running through and even modifying the scripts provided with the module to better understand how they worked before tacking the assignment itself.

CODE PRODUCTION

I ended up reusing some elements of Module06's "ToDo.py" script, though, having now learned how to navigate try-except blocks and custom error handling, I was able to simplify many of those functions and features. As there were a few errors that I anticipated needing to handle upfront - and my custom classes established some restrictions, such as preventing the user from writing numeric characters into a string field - I took the opportunity to set my error handling up first. Remembering that Python is read top to bottom, this allowed me to easily employ the same error messaging throughout the entire script:

```
class AlphaError(Exception):
    def __str__(self):
        return ("Invalid entry: only alphabetic characters in Product 
Name field.")

class NumError(Exception):
    def __str__(self):
        return ("Ivalid entry: only numeric characters in Product Price 
field.")

class MenuError(Exception):
    def __str__(self):
        return ("Invalid entry: please select a menu option 1-3.")

class GenError(Exception):
    def __str__(self):
        return ("Unspecified error: check your inputs and try again.")
```

From there, I went on to define my custom class Products(), which I ultimately populated with two values: product_name and product_price.

```
def __init__(self, product_name, product_price):
    self.__product_name = product_name
    self.__product_price = product_price
```

With a much better understanding of functions and newfound custom classes to leverage, the functions I defined to carry out the tasks in the assignment were relatively simple. For example, my entire product input function ended up being just three lines of code:

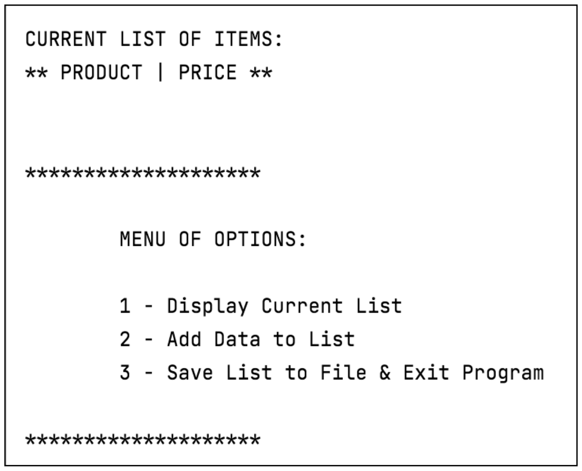
```
product_name = input("\nInput an item to catalog: ")
product_price = input("Input an approximate price: ")
return product_name, product_price
```

My functions thus defined and data organized, the main body of my script became a matter of calling out each function at the right time and adding a few try/catch blocks to handle some basic input limitations:

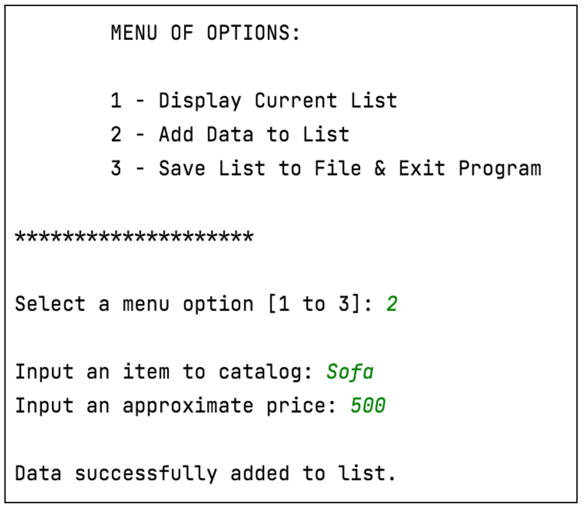
```
FileProcessor.read_data(strFileName, lstOfProductObjects)

while (True):
    IO.output_menu()
    menu_choice = IO.menu_choice()

    if menu_choice == "1":
        IO.print_list(lstOfProductObjects)

    elif menu_choice == "2":
        try:
            (product_name, product_price) = IO.input_product()
            if product_name.isnumeric():
                raise AlphaError()
            elif product_price.isalpha():
                raise NumError()
            else:
                FileProcessor.add_data(product_name, product_price,
lstOfProductObjects)
                print("\nData successfully added to list.")
        except:
            raise GenError()

    elif menu_choice == "3":
        try:
            FileProcessor.save_data(strFileName, lstOfProductObjects)
            print("\nData successfully written to file. Exiting
program. Goodbye!")
        except:
            raise GenError()
        break

    else:
        raise MenuError()
```

CODE VALIDATION

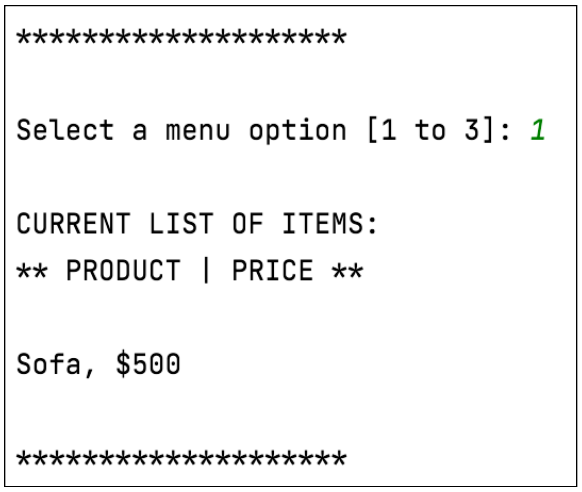
Testing, as usual, came in the form of validating my code against the IDE and then again in Terminal, thus ensuring the companion file is successfully being both written to *and* read. Here is a basic run-through of my script in PyCharm (note: I pre-populated my list with some data to test the `FileProcessor.read_data` function in PyCharm as well):



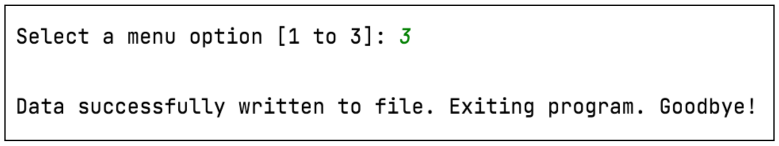
```
CURRENT LIST OF ITEMS:
** PRODUCT | PRICE **


*******************

        MENU OF OPTIONS:

        1 - Display Current List
        2 - Add Data to List
        3 - Save List to File & Exit Program

*******************
```



```
        MENU OF OPTIONS:

        1 - Display Current List
        2 - Add Data to List
        3 - Save List to File & Exit Program

*******************

Select a menu option [1 to 3]: 2

Input an item to catalog: Sofa
Input an approximate price: 500

Data successfully added to list.
```

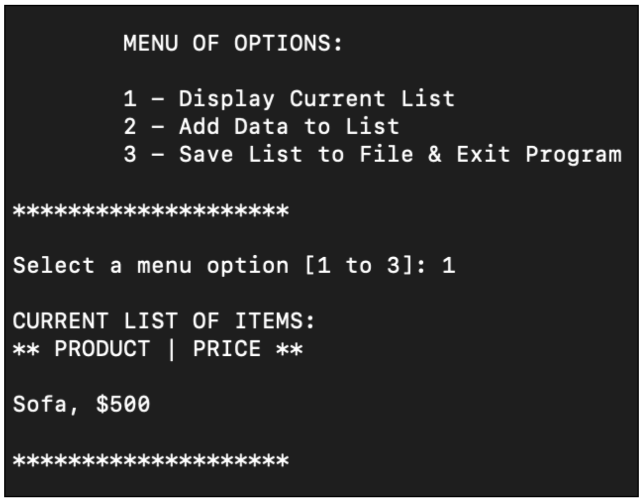


```
*******************


Select a menu option [1 to 3]: 1


CURRENT LIST OF ITEMS:
** PRODUCT | PRICE **


Sofa, $500


*******************
```

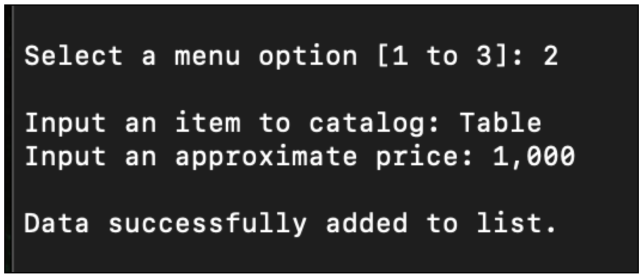


```
Select a menu option [1 to 3]: 3

Data successfully written to file. Exiting program. Goodbye!
```

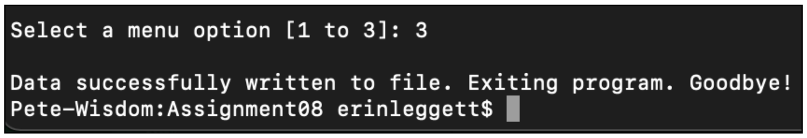
As the script was built to read the products.txt file upon running, I was able to confirm in Terminal whether my data saved by immediately attempting to read my upload list back to myself using option 1:



```
        MENU OF OPTIONS:

        1 - Display Current List
        2 - Add Data to List
        3 - Save List to File & Exit Program

********************

Select a menu option [1 to 3]: 1

CURRENT LIST OF ITEMS:
** PRODUCT | PRICE **

Sofa, $500

********************
```

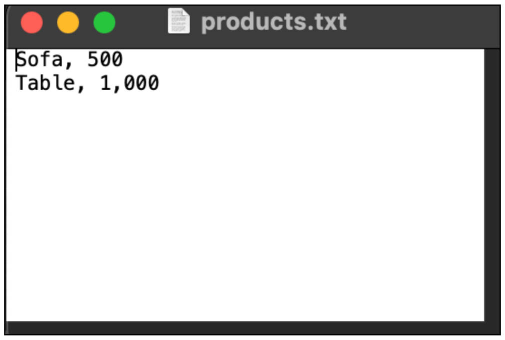


```
Select a menu option [1 to 3]: 2

Input an item to catalog: Table
Input an approximate price: 1,000

Data successfully added to list.
```



```
Select a menu option [1 to 3]: 3

Data successfully written to file. Exiting program. Goodbye!
Pete-Wisdom:Assignment08 erinleggett$
```

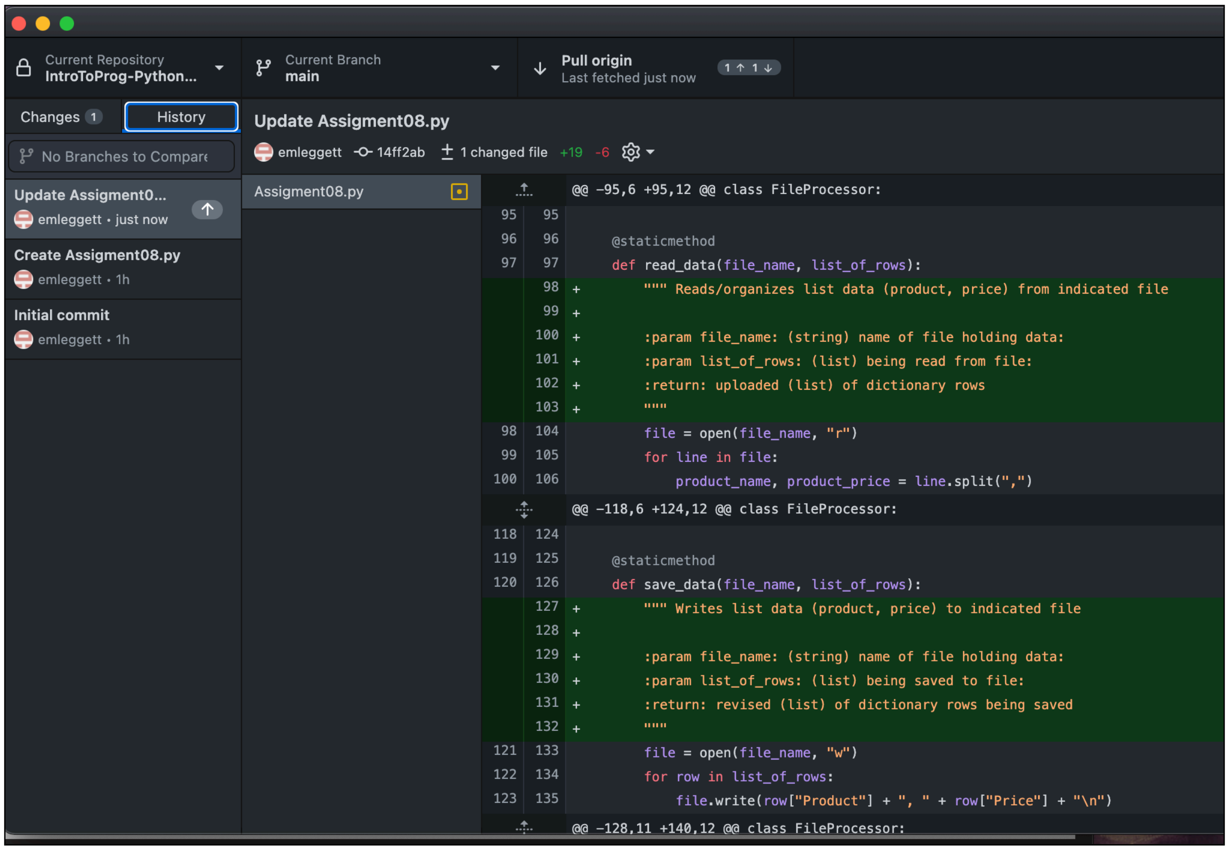
Finally, I am able to see my Sofa and Table entries in my products.txt file with the correct prices listed next to them:



```
Sofa, 500
Table, 1,000
```

PUBLICATION

Lastly, we were tasked with creating a repository in GitHub Desktop rather than online. Though I haven't yet uploaded this file, I was able to successfully configure my IntroToProg-Python-Mod08 repository and upload it via the Documents folder:



```
@@ -95,6 +95,12 @@ class FileProcessor:

    @staticmethod
    def read_data(file_name, list_of_rows):
+       """ Reads/organizes list data (product, price) from indicated file
+
+       :param file_name: (string) name of file holding data:
+       :param list_of_rows: (list) being read from file:
+       :return: uploaded (list) of dictionary rows
+       """
        file = open(file_name, "r")
        for line in file:
            product_name, product_price = line.split(",")
@@ -118,6 +124,12 @@ class FileProcessor:

    @staticmethod
    def save_data(file_name, list_of_rows):
+       """ Writes list data (product, price) to indicated file
+
+       :param file_name: (string) name of file holding data:
+       :param list_of_rows: (list) being saved to file:
+       :return: revised (list) of dictionary rows being saved
+       """
        file = open(file_name, "w")
        for row in list_of_rows:
            file.write(row["Product"] + ", " + row["Price"] + "\n")
@@ -128,11 +140,12 @@ class FileProcessor:
```

LESSONS LEARNED

My main takeaway from this week's module and assignment was how versatile and valuable classes and methods are. Based on our earlier use of loops and conditionals, I expected that any amount of processing would involve a lot of code; however, by breaking the script into reusable parts and recycling certain functionalities and features, I was able to a relatively quick and lean script file.

One aspect of Python that I am still struggling a bit with is how to loop back through a script after raising an error message. In Assignment 06, I was able to achieve that through loops, conditionals, and boolean flags; in Python, as far as I have observed, the user exits the program when an error is raised. I haven't been able to work my way around that, as yet, but think it would be nice to be able to add that feature.

GITHUB LINKS

https://github.com/emleggett/IntroToProg-Python-Mod08
https://emleggett.github.io/IntroToProg-Python-Mod08/