Custom Classes

# Introduction

This assignment had us use a starter script and some pseudocode to create a custom class to store data, then work with various instances of objects created using that class to perform some tasks. The assignment really brought together several concepts we’ve learned about over the last few weeks.

# Topic 1: Custom Class

The first step was to create a custom class that stored data about a product. This class had two properties: the product name, and the product price. I created a constructor to initialize each property (Figure 1).

def \_\_init\_\_(self, product\_name, product\_price):  
 self.\_\_product\_name = product\_name  
 self.\_\_product\_price = product\_price

**Figure 1: Property constructor in the Product class**

Then, I used the property getters and setters to format each property the way I wanted, as well as to check that they always held valid values (namely non-numeric for the product name, and numeric only for the product price) (Figure 2).

@property  
def product\_name(self):  
 return str(self.\_\_product\_name).title()  
  
@product\_name.setter  
def product\_name(self, value):  
 if str(value).isnumeric() == False:  
 self.\_\_product\_name = value  
 else:  
 raise Exception("Name cannot be numbers.")  
  
@property  
def product\_price(self):  
 return self.\_\_product\_price  
  
@product\_price.setter  
def product\_price(self, value):  
 if isinstance(value, float) == True:  
 self.\_\_product\_price = value  
 else:  
 raise Exception("Prices must be numbers.")

**Figure 2: Using getters and setters to format and validate property values.**

Finally, I added a string method to overwrite the automatic string method to display the data the correct way in the context of this program (Figure 3).

def \_\_str\_\_(self):  
 return self.product\_name + ",$" + "%.2f" % self.product\_price

**Figure 3: Displaying the data in a way that makes sense for the end user**

# Topic 2: File Processor Functions – Working with lists of objects

The next step was to create some methods within the FileProcessor class that would read data from a file into a list, then write data from the list to the file.

Reading data from the file to the list of objects required the code to:

1. Read each line of data in the file
2. Remove the newline character
3. Separate the contents into two discrete elements
4. Set a new instance of the Product class using each of the above elements
5. Append that new object to the list

Once I understood the concept, this was actually easier than creating a list of lists or a list of dictionaries as I didn’t have to worry about keys and values, formatting the list, etc. For complex data especially I can see why this would be a better way to go. This portion of my script was still a decent number of lines long (Figure 4).

@staticmethod  
def read\_data\_from\_file(file\_name, list\_of\_objects):  
 *""" Reads data from .txt file and translates to list of objects  
  
 :param file\_name: name of file you want to read from  
 :param list\_of\_objects: list of objects you want to add file data to  
  
 :return: list\_of\_objects  
 """* # clear out current list if applicable  
 try:  
 list\_of\_objects.clear()  
 # open file in read mode  
 file = open(file\_name, "r")  
 # iterate over file, remove newline character  
 # split on comma and initialize each row into a new instance of Person object  
 for line in file:  
 line.strip()  
 file\_product, file\_price = line.split(",$")  
 row\_object = Product(file\_product, float(file\_price))  
 list\_of\_objects.append(row\_object)  
 file.close()  
 return list\_of\_objects  
 except FileNotFoundError:  
 print("File not found! No items loaded.")

**Figure 4: The file reading method script**

Writing to the file was even easier as well. Because my Product class included a string method that formatted the object in exactly the way it needed to be in my end file, I was able to write a simple for loop writing each item and a new line to the file (Figure 5).

def write\_data\_to\_file(file\_name, list\_of\_objects):  
 *"""Write data to file from list of objects  
 :param file\_name: name of file you want to write to  
 :param list\_of\_objects: list of objects you want to write from  
  
 :return: nothing  
  
 """* file = open(file\_name, "w")  
 for item in list\_of\_objects:  
 file.write(f"{item}\n")  
 file.close()

**Figure 5: The file writing method script – much shorter!**

# Topic 3: IO, Main Script, and Exceptions

From here, the only things left to do were create some IO methods, write the main script, and do some exception handling.

The IO section was relatively easy as these are just static methods that display things like the menu, current list, etc. The most complex portion was creating a method to get a new item from the user. For this particular method, I got the item name and price via some input statements, then returned an instance of the Product class with those two items passed in (Figure 6).

@staticmethod  
def input\_new\_item():  
 *""" Get user input for new item and price  
  
 :return: string, string  
 """* try:  
 user\_item = input("Please input item: ")  
 while user\_item.isnumeric():  
 print("Please enter valid name. Must not be numbers.")  
 user\_item = input("Please input name: ")  
 user\_price = input("Please input price: ")  
 user\_price = float(user\_price)  
  
 except Exception:  
 print("Please enter valid price. Must be numbers.")  
 # user\_price = float(input("Please input price: "))  
  
 return Product(user\_item, user\_price)

**Figure 6: Getting new items from user, then returning an object instance of the Product class**

Later, in the main body of the script, I was able to assign a variable to that method, then add that the new object the method created to my list via that variable.

I also wrote a few exceptions. These included making sure the user input valid names and prices, as well as ensuring the user chose a valid choice from the menu of options (Figure 7). I’m certain there are many more ways to break the script than those I experimented with, so I also included a final except statement that would share the built in Python error for cases I didn’t anticipate.

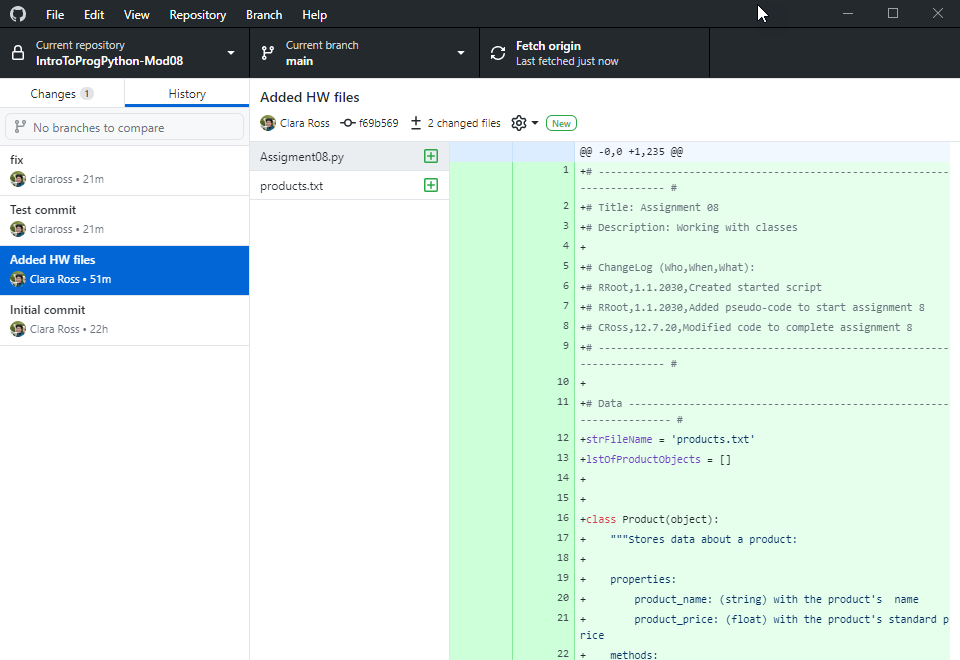
def \_\_str\_\_(self):  
 return "Please choose option 1, 2, 3, or 4"

**Figure 7: A custom exception class to check for invalid menu choices.**

# Topic 4: Github Desktop

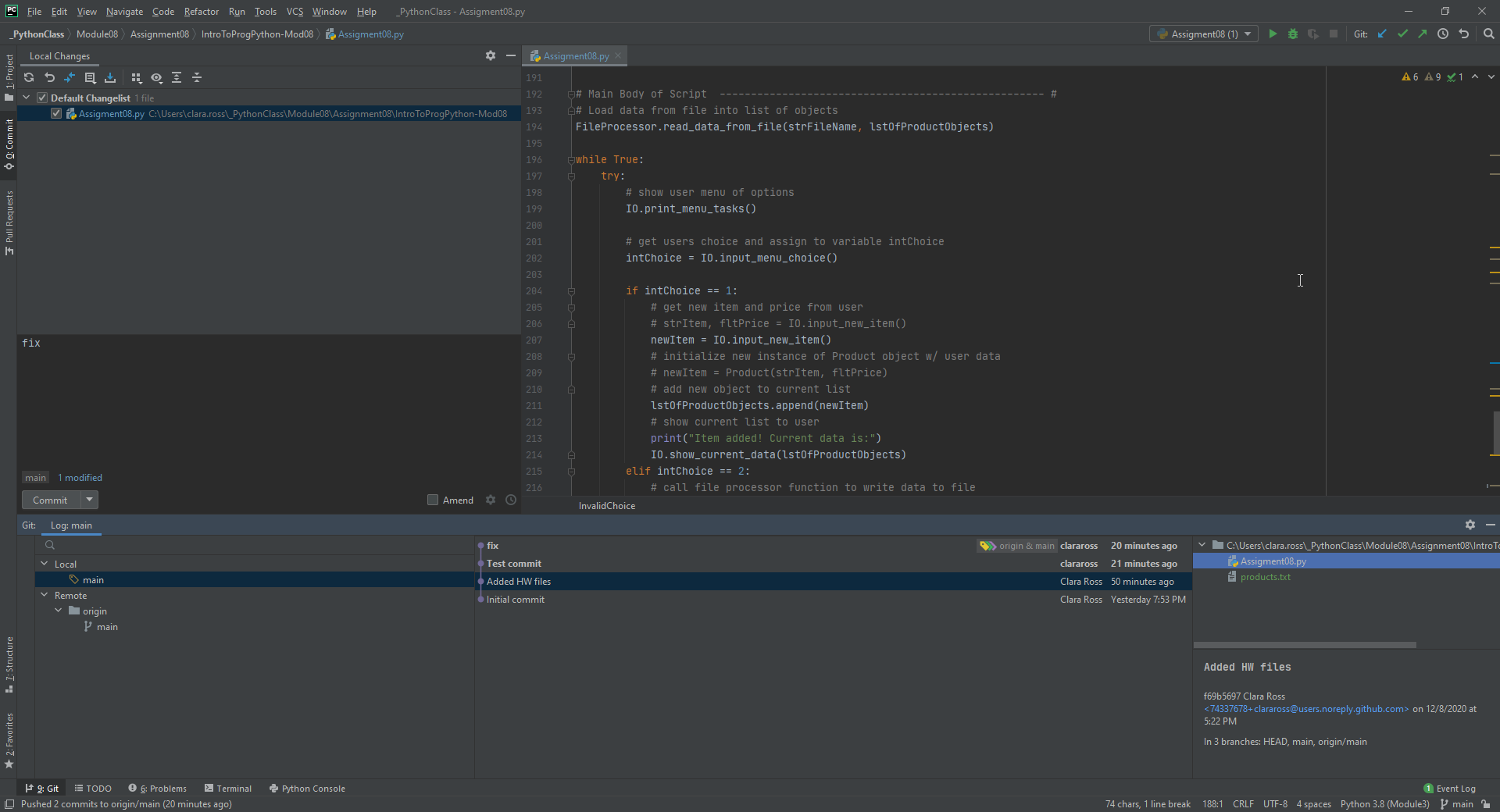
Finally, we worked with Github desktop to create a repository, commit the script files, then push up to Github when ready.

I downloaded Github desktop and authenticated my login. From there, I was able to create a repository locally, then later push that up to Github itself (Figure 8).



**Figure 8: Working with Github desktop to create a repo then push changes**

I also was able to set up Github on PyCharm and commit and push changes directly from there. This allows for a very seamless experience when working with code and wanting to push changes so that others may use them (Figure 9).



**Figure 9: Using version control on PyCharm. A seamless experience (unless you’re trying to fork).**

# Conclusion

Working with custom classes was certainly more challenging, but with a little trial and error I feel more confident in my understanding now. There were still a few areas where things seemed to work or not work without any immediately obvious reason why, but I suspect as I begin to apply these concepts to other projects patterns will become clear. Exception handling has also proven to be tricky – I created some exceptions for a few of the possible errors that could come up, but I am certain there are more and do not envy those who are creating software accessible by thousands, if not millions of people. Finally, I am glad to be working with a version control system more closely as I like to try out new things to see what does and doesn’t work, so this will keep previous versions accessible and easy to use.