

Classes & Objects

Introduction

This module and assignment cover how to use classes, both to create data objects and to manage functions for use in the main body of the code. It also introduces GitHub Desktop as a way of tracking files locally and connecting with the GitHub website.

Working With Classes

Classes

Classes group data and functions with a similar purpose together. A class used to manage data acts like a blueprint for creating data objects and working with them. These objects have the same fields and properties, since these come from the class, but their actual data attributes can be different. A standard class is made up of sections for fields (variables), constructors, attributes, properties, and methods (functions). Depending on the purpose of the class, some or all of these components will be used. Classes have built-in methods, and can also contain custom methods. A constructor is a built-in method that runs when an object is created using the class blueprint. It can set initial field values or use values you pass in to create a new object. Attributes are virtual fields, internal to the class, that can be used in conjunction with properties to control the data in the field. Properties are methods that allow you to manage how data in a given field is displayed (the getter property) or set (the setter property). The setter property can be used to raise an error if an update to a field is not allowed.

Classes that are used to make objects are set up differently than classes used only for processing. The functions defined in processing classes are usually used outside of the class, in the main body of the code. These functions should be preceded with the keyword “@staticmethod.” In contrast, classes which deal with creating and storing data use the “self” keyword in their methods because they are called from within the object instance itself. Each data object created from the class has the same “self” methods that can run with that object’s unique set of data values.

Docstrings

Even a simple processing script such as the one in this assignment has multiple classes and functions within those classes. It can be easy to lose track of what the purpose of a function is, what the expected variables are, and what a class contains. Docstrings are code blocks that contain this type of metadata. They follow a specific format so that they can be displayed using tooltips in IDEs or called within the code.

Git

Git is a file versioning and management software. Git communicates with the GitHub website to make backups and allow for sharing. GitHub Desktop provides a GUI to view what Git is tracking on your computer, as well as an alternate way to connect to the GitHub website.

Product Script

This script uses a class to create Product objects from name and price data read in from a text file. The user has the option to display the data, add a product, or save and exit, which writes the data from the full list of product objects back to the text file. A starter file was provided for the script.

Class Setup

After reviewing the pseudocode outline, I began with the product class. I based this section on the work from the module listings and labs. I tested creating objects from the class before moving on to the next sections. In the Processing and Presentation sections of the script, the methods all use the @staticmethod keyword because they are called outside the classes the main body of the script. For the FileProcessor class, I added and tested the methods for reading from and writing to the text file. The IO class repurposed a good number of methods from past assignments. The main exception is the method to return the list of products: in this script, I used the string method from the Product class as shown in Listing 1 below:

Listing 1: Code for displaying the list of products using string method (Class: IO)

```
@staticmethod
def output_current_products(list_of_objects):
    """ Shows the current products in the list of product objects

    :param list_of_objects: (list) of objects you want to display
    :return: nothing
    """

    print('Current product names and standard prices:')
    for objProduct in list_of_objects: # Use string method from Product class
        print(objProduct)
```

Error Handling

Thanks to testing during setting up classes, assembling the main body of the code was relatively straightforward. However, I ran into trouble when I tried to use the exception defined in the Product class's property for product_price. I was trying to disallow values that could not be converted to a floating point value, and wanted to practice using the property instead of catching it when entered by the user. First, my logic for testing in the if statement was wrong (not boolean), so I changed it to the try/except block shown in Listing 2 below.

Listing 2: Code for the product_price setter attribute (Class: Product)

```
@product_price.setter # Setter for price
def product_price(self, new_value):
    try: # Check that the price can be converted to a floating point value
        self.__product_price = float(new_value)
        # Store the price as a floating point value
    except ValueError:
        print('Product price can only contain numbers. Please try again')
        # ValueError exception
```

However, I was still not able to display the error when I expected it. After using the debug tools, testing things out in one of the simple listings, and reading, I realized that I needed to work with already-initialized default values and try to change them to the user inputs, rather than creating a new product object from scratch. With a little additional tweaking to prevent objects with the default values from being added to the list, this track ultimately worked. The final code for this section in the main body of the script is in Listing 3 below.

Listing 3: Code for the user selection to add a product to the list (Main Body)

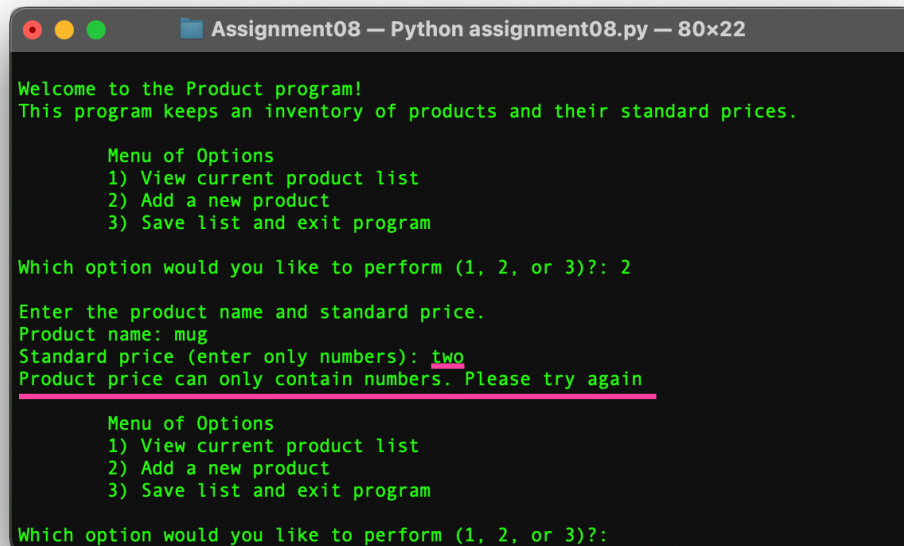
```
elif userChoice == '2': # User selection 2: Add a new product
    name_add, price_add = IO.input_new_product()
    # Capture user-entered name and price
    objProductAdd = Product()
    # Initialize new product object with the default values
    objProductAdd.product_name = name_add
    # Try setting the name to the user-entered value
    try:
        objProductAdd.product_price = price_add
        # Try setting the name to the user-entered value
    except ValueError as e: # Raise exception if the price is non-numeric
        print(e)
    if objProductAdd.product_price != 0: # Check if product_price is not 0
        lstOfProductObjects.append(objProductAdd)
        # Append new object to list of product objects
    else:
        continue # If 0, move on without appending the partially
        updated object to the list
```

I decided to enforce only the floating point value error and not check that the name was a string, since string was used by default and a product name could possibly be a number.

Running the Script

The final script was able to successfully read and display the text file contents, add new product objects from user input, and save the list back to the text file in both PyCharm and Terminal. While it does not have robust error handling, it does enforce the price entries as floating point values using the Product class's setter property.

Figure 4: Error handling for setting the product price in Terminal



```
Assignment08 — Python assignment08.py — 80x22

Welcome to the Product program!
This program keeps an inventory of products and their standard prices.

Menu of Options
1) View current product list
2) Add a new product
3) Save list and exit program

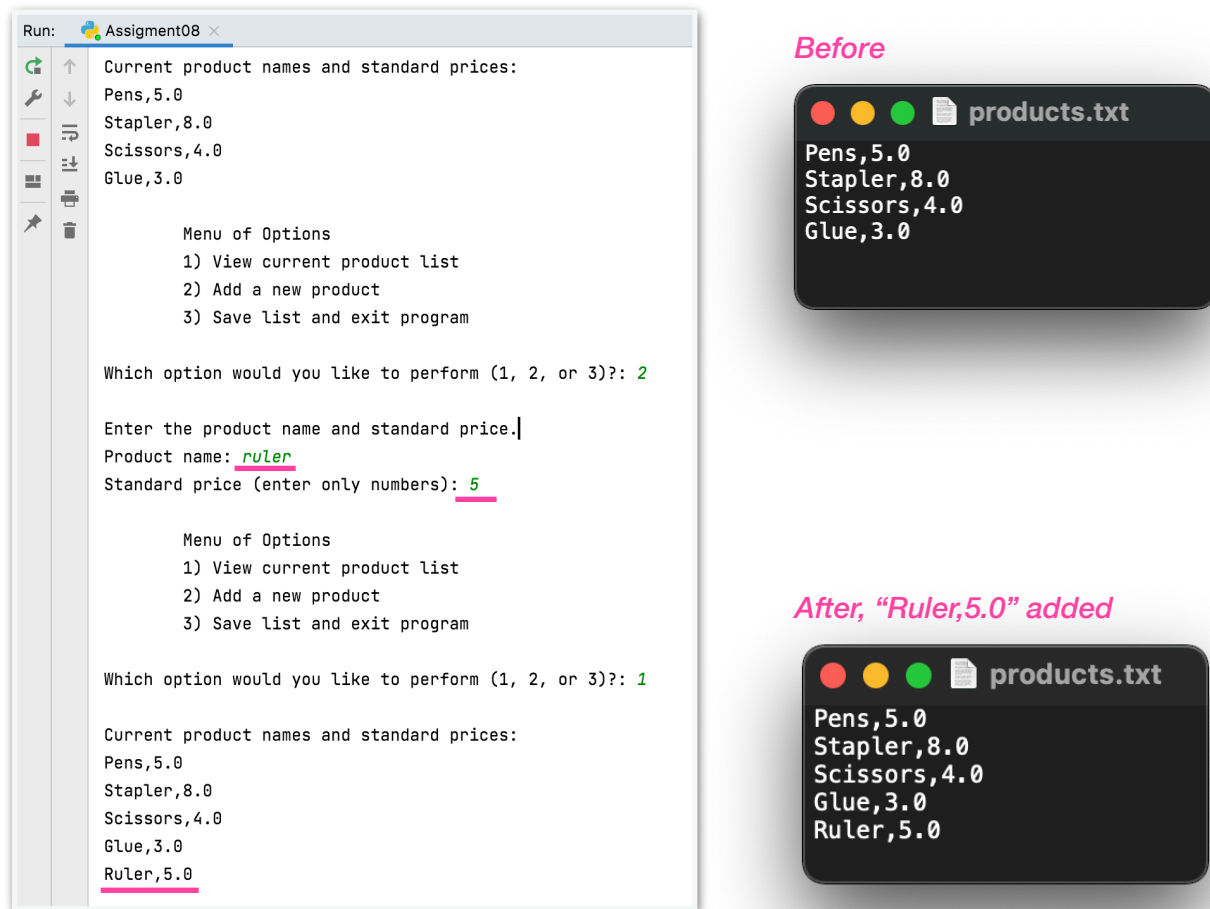
Which option would you like to perform (1, 2, or 3)? 2

Enter the product name and standard price.
Product name: mug
Standard price (enter only numbers): two
Product price can only contain numbers. Please try again

Menu of Options
1) View current product list
2) Add a new product
3) Save list and exit program

Which option would you like to perform (1, 2, or 3)?
```

Figure 5: Adding a new product to the list in PyCharm, with before/after text files



Conclusion

This assignment taught me about using classes to create and work with data objects, as well as to organize scripts and functions. I practiced using the debug tools and troubleshooting, particularly for error handling in attributes. This process helped me better understand what was happening inside the Product data class, as well as how the functions in the main body interact with the class objects. Finally, on the presentation side, I used GitHub Desktop and became more familiar with Git.