Barbara Spadavecchia

March 8, 2022

IT FTN 110 A

Assignment08

Creating Scripts and Using Custom Classes with Python

Introduction

This assignment goes over creating scripts and custom classes. Classes are used to organize functions and data. When the class's code is used, it can be used directly or indirectly. An object instance of the class is created when the code is used indirectly. This assignment gave practice in writing a script in python that used both options to manage a list of products and their prices.

Classes and Objects

Classes are mainly designed to focus on data or processing. Code can be used directly but it only loads once. To use code indirectly, an object instance of the class is created. The advantage of using the code indirectly is that there can be multiple object instances. One of the considerations for the list of products and their prices was to determine which class to use.

Lab 8-1 Example of Fields

Fields are the data members of a class that are the variables and constants in a class. The value of these variables in this lab were entered as objP1. In this lab, two fields were added: strFirstName and strLastName.

```
# --- Make the class ---
class Person(object):   # object is optional
    # --Fields--
    strFirstName = ""
    strLastName = ""

    # -- Constructor --
    #         -- Attributes --
    # -- Properties --
    # -- Methods --
# --End of class--

# --- Use the class ----

objP1 = Person()

objP1.strFirstName = "Bob"
objP1.strLastName = "Smith"
print(objP1.strFirstName, objP1.strLastName)
```

Figure 1 Lab 8.1

Constructors

These are special methods/functions that automatically run when you create an object from a class. They are used to set the initial value. The "__init__"() calls and passes arguments each time a new object is created. The constructor method uses the keyword "self" to refer to data or functions in an object instance. Python requires the parameter "self" to be used from an object instance.

Lab 8-2 Example of a constructor used with an object instance

The constructor example showing the double underscore "__init__(self, first_name, last_name) and the parameter self.

```
8      # --- Make the class ---
9      class Person(object):
0          # --Fields--
1          strFirstName = ""
2          strLastName = ""
3
4          # -- Constructor --
5          def __init__(self, first_name, last_name):
6              self.strFirstName = first_name
7              self.strLastName = last_name
8
9          #       -- Attributes --
0          # -- Properties --
1          # -- Methods --
2      # --End of class--
3
4      # --- Use the class ----
5      objP1 = Person("Bob", "Smith")
6      print(objP1.strFirstName, objP1.strLastName)
```

Figure 2 Lab 8-2

Attributes and Properties

Attributes are virtual fields that are managed by properties (functions). Two properties are created for each field/attribute. One is the "getter" to get data and the other is the "setter" for validation and error handling. If a value passed into the "setter" parameter is valid, it will be assigned to the field or attribute. Lab 8-4 gives an example of a getter and setter using the hidden attributes __first_name and __last_name. It also shows how the names of the setter must match the names of the property. The setter in this example checks if the first name is numeric and passes the value through if it's validated.

```
    # -- Constructor --
    def __init__(self, first_name, last_name):
        #       -- Attributes --
        self.__first_name = first_name
        self.__last_name = last_name

    # -- Properties --
    # first_name
    @property   # DON'T USE NAME for this directive!
    def first_name(self): # (getter or accessor)
        return str(self.__first_name).title()  # Title case

    @first_name.setter  # The NAME MUST MATCH the property's!
    def first_name(self, value):  # (setter or mutator)
        if str(value).isnumeric() == False:
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")

    # last_name
    @property   # DON'T USE NAME for this directive!
    def last_name(self):  # (getter or accessor)
        return str(self.__last_name)  # Title case

    @last_name.setter  # The NAME MUST MATCH the property's!
    def last_name(self, value):  # (setter or mutator)
        self.__last_name = value
```

Figure 3 Lab 8-4 Example of getter and setter

Methods

The functions that are inside of a class are called methods. They allow you to organize processing statements into groups. Some examples of methods: "__str__()" which is a built-in method in python that return's a class's data as a string. Static methods are called directly from the class. A class that focuses on processing is usually static. Lab 8-5 gives an example of using methods.

```
    # -- Methods --
    def to_string(self):
        return self.__str__()


    def __str__(self):
        return self.first_name + ',' + self.last_name

# --End of class--
```

Figure 4 Lab 8-5 Example of using the method "__str__()"

Script for Assignment08 Product and Price List

The labs and the assignment08-starter.py provided a good start for the script. The first step was to identify the classes needed: Product, FileProcessor, and IO. The process that I followed was the same as the labs. I started with the constructor, then properties and methods.

```python
class Product:
    """Stores data about a product:

    constructor:
        __product_name:  private attribute
        __Product_price:  private attribute

    properties:
        product_name: (string) with the products's  name
        product_price: (float) with the products's standard price

    methods:
        __str__(self):  comma separated for printing or storing
        product_add_to_lst(obj, list_Of_Product_Obj): ->list_Of_Product_Obj
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
        BSpadavecchia,03.06.2022,Modified code to complete assignment 8
    """

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

Figure 5 Script Class Product Constructor

```python
    # Properties
    @property
    def product_name(self):
        return str(self.__product_name).upper()


    @product_name.setter
    def product_name(self, value):
        if str(value).isnumeric() == False:
            self.__product_name = value
        else:
            raise Exception("Product name can not contain numbers")


    @property
    def product_price(self):
        return round(self.__product_price, 2)


    @product_price.setter
    def product_price(self, value):
        if value.isalpha() == False:
            self.__product_price = float(value)
        else:
            raise Exception("Product price must be a number")
```

Figure 6 Class Product Properties with some error handling (name checked for numbers and price checked for numbers only)

The class FileProcessor is static and deals with processing data to and from a file and list of products. The processes involve reading data and saving data to a file. Class IO prints the menu, lists and captures input for product name and price.

```python
    """
    @staticmethod
    def print_main_menu_options():
        """Displays a menu of options to user
        :return nothing
        """

        print("""
        Menu of Options:
        1 - Show Current List
        2 - Add A New Product
        3 - Save Product List
        4 - Exit Program
        """)
```

Figure 7 Class IO printing menu

```python
@staticmethod
def print_list_of_products(list_of_product_objects):
    """Shows current list of products
    :param product_object: objects in list
    :param list_of_product_objects list of objects to display
    :return: nothing
    """
    print("***************PRODUCTS***************")
    if len(list_of_product_objects) > 0:
        for product_object in list_of_product_objects:
            print(product_object.product_name + " at $" + str(product_object.product_price))
    else:print("No Product Information to Display")
```

Figure 8 Class IO printing list of products with a check to see if the file has data

```python
def input_product_details():
    """Create new product object, initialize, fill with user input
    :param new_product_object: object
    :param product_name: attribute of object
    :param product_price: attribute of object
    :return: object
    """
    new_product_object = Product("", None) #Initialize new object
    new_product_object.product_name = input("What is the Product's Name?: ")
    new_product_object.product_price = input("What is the Product's Price?: ")
    return new_product_object
```

Figure 9 Class IO getting input from user on product name and price

The main body of the script shows the user the menu option and gets the user's input. Selection 1 prints the list, selection 2 allows a user to add products, selection 3 saves the product list and selection 4 exits the program and the script ends.

```python
FileProcessor.read_data_from_file(strFileName, lstOfProductObjects)
#Show User a Menu of Options
while True:
    IO.print_main_menu_options()

    # Get User's Menu Option Choice
    strChoice = IO.input_menu_choice()

    # Show User Current Data in List of Product Objects
    if strChoice.strip() =="1":
        IO.print_list_of_products(lstOfProductObjects)

    # Let User Add Data to Product Objects
    if strChoice.strip() =="2":
        try:
            objProduct = IO.input_product_details()
            Product.product_add_to_list(objProduct, lstOfProductObjects)
            print("\n\"" + objProduct.product_name + "\"" +
            "has been added to the List of Products for $" +
            str(objProduct.product_price))
        except Exception as e: #Error handling for attribute
            print(e)
    # Let User Save Data
    if strChoice.strip() == "3":
        FileProcessor.save_data_to_file(strFileName, lstOfProductObjects)
        print("The List of Products has been saved")
```

Figure 10 Main body of script (selection 4 just exits)

Assignment08.py loading the product.txt file at the start of the script, creating the text file and adding a product with a price.

```
C:\Users\bspad>python "C:\_PythonClass\Assignment08\Assignment08.py"
File not found

File loaded successfully


        Menu of Options:
        1 - Show Current List
        2 - Add A New Product
        3 - Save Product List
        4 - Exit Program

Please choose an option [1-4]: 1

***************PRODUCTS***************
No Product Information to Display

        Menu of Options:
        1 - Show Current List
        2 - Add A New Product
        3 - Save Product List
        4 - Exit Program

Please choose an option [1-4]: 2

What is the Product's Name?: soup
What is the Product's Price?: 4.99

"SOUP"has been added to the List of Products for $4.99
```
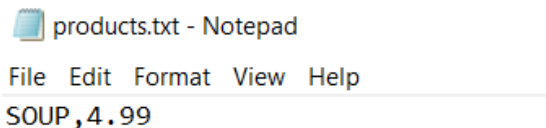
Figure 11 Assignment08.py running from the command prompt

products.txt - Notepad

File Edit Format View Help

SOUP,4.99

Figure 12 products.txt output

GitHub Desktop and "Git"

Developers often work with GitHub using a desktop application. The communication between the GitHub website and a local computer is handled by a program called "Git". Git manages different versions of files. It allows for the original version to be maintained while changes are made for a new version. Git also uses GitHub to store backup files on the "Cloud". Below are examples of making a comment change to assignment06 to verify that a change on the desktop could also be changed on GitHub.

```
Assignment06.py
   ↑      @@ -8,7 +8,7 @@
   8    8    # RRoot,1.1.2030,Created started script
   9    9    # BSpadavecchia,02.21.2022,Modified code to complete assignment 06
  10   10    # ------------------------------------------------------------------- #
  11         -
       11   +# BSpadavecchia,03.08.2022,Used to test Git management of file
  12   12    # Data ----------------------------------------------------------- #
  13   13    # Declare variables and constants
  14   14    file_name_str = "ToDoList.txt"  # The name of the data file
```

Figure 13 Use of Changing a line in a Git managed file

```
   8       8      # RRoot,1.1.2030,Created started script
   9       9      # BSpadavecchia,02.21.2022,Modified code to complete assignment 06
  10      10  +   # -------------------------------------------------------------
  11          -
          11  +   # BSpadavecchia,03.08.2022,Used to test Git management of file
  12      12      # Data ---------------------------------------------------------
  13      13      # Declare variables and constants
  14      14      file_name_str = "ToDoList.txt"  # The name of the data file
```

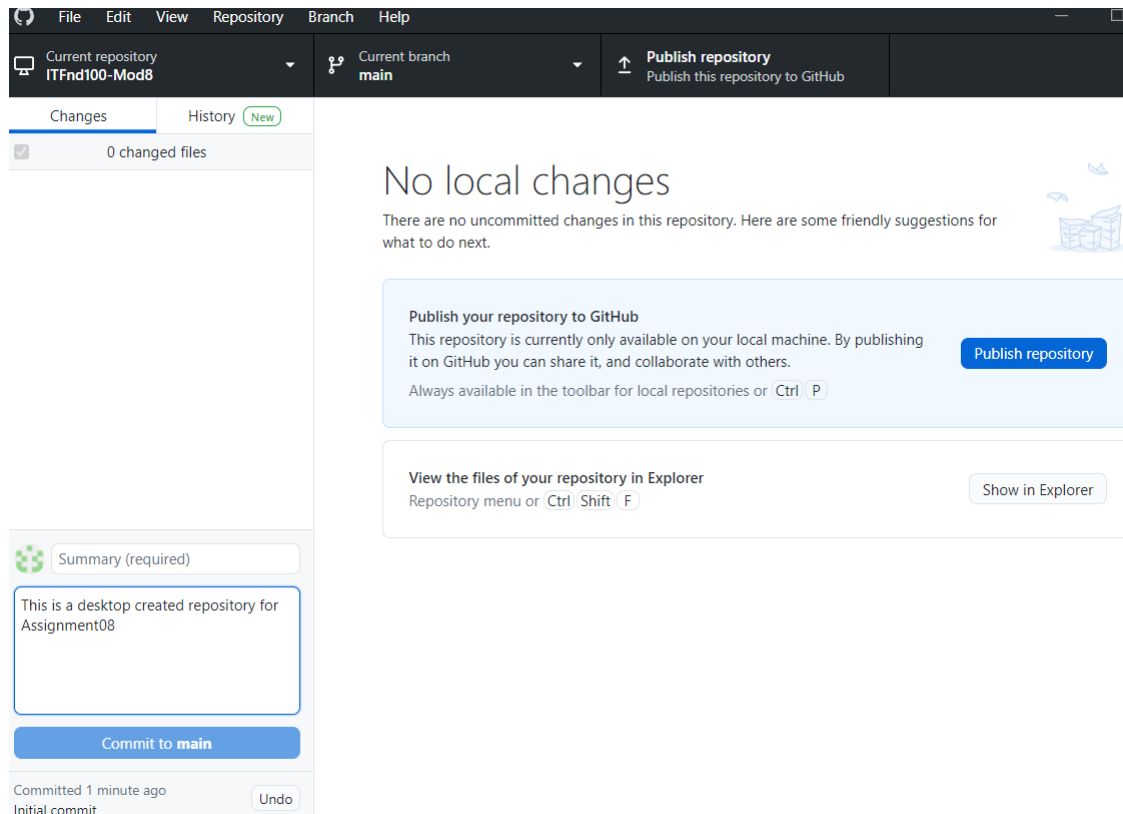*Figure 14 Commit to Main and Uploading the changes to GitHub by using "push origin"*

Figure 15 created repository from desktop.

Conclusion

This assignment gave a lot of practice thinking about the functions and how classes are used to organize them. On this assignment I spent more time going over different code examples from the listings, exercises, websites, videos, and the book. I started with the first part of the book and then watched the module video. This was an easier approach for me. The installation of "Git" was straightforward and appears that it will be a valuable tool. Creating the repository was simpler through the desktop.