Hannah Clayton

March 7, 2021

Foundations of Programming: Python

Assignment 08

GitHub link

08\_Classes & Objects

I. Introduction

This document serves as a demonstration of my experience working through this week’s assignment module. In the ‘module objectives’ section, I highlight the intended goals for this week’s reading and instructional video as given by Professor Root, and in my own words communicate my understanding of the objectives. In the ‘assignment work-through’ section I show how I went about approaching the module’s assignment, and any issues or epiphanies I encountered in doing so. The overall intent of this document is to serve as both a reference for myself and others in the future and demonstrate my competency in the module to Professor Root.

II. Module Objectives

1. Objectives

• What is the difference between a class and the objects made from a class?

• What are the components that make up the standard pattern of a class?

• What is the purpose of a class constructor?

• When do you use the keyword "self?"

• When do you use the keyword "@staticmethod?"

• How are fields and attributes and property functions related?

• What is the difference between a property and a method?

• Why do you include a docstring in a class?

• What is the difference between Git and GitHub?

• What is GitHub Desktop?

1. Objective Summary

In this week’s course module content, we delved deeper into the characteristics of classes, the information they contain within them, and how to work with them. You can think of a ‘class’ as the blueprint for your home. You lay out what your home will look like, some of its functionality, etc. but you are not creating an actual house, just a representation of it. *Classes* have a standard pattern comprised of key elements and parts. These are fields, constructors, attributes, properties, and methods. *Fields* are where you create variables or constants necessary to other functions in the class. It is important to note that these objects vary from classes. Objects in classes are used to create different versions of your home blueprint (your base class). Say you wanted to add on a hot tub deck (good idea by the way), instead of completely changing your class you can do something like the following line of code:

hottubtimemachine = Home\_Blueprint(100, 400)  
hottubtimemachinev2 = Home\_Blueprint(200, 800)

In the example code above, we are assuming I have already created a base class called ‘**Home\_Blueprint**’. In it I may have functions dedicated to estimating building costs based off square footage. If I wanted to compare two different potential hot tub sizes, I create two different objects (**hottubtimemachine** and **hottubtimemachinev2**). Then I can pass in the square footage values for each version and would be able to run the base class functions according to those additional measurements.

Let us move back to class standard patterns. The second target is called ‘constructors’. *Constructors* are functions that are automatically called when doing certain things with classes. The function **‘\_\_init\_\_**’ is a constructor and is called every time a new object, regardless of whether you have explicitly called it or not. Take for example my previous lines of code for comparing two hot tub additions to my house blueprint. In my base class I would have defined the \_\_init\_\_ method and any variables associated with it, like this section of code:

class Home:  
 def \_\_init\_\_(self, w, l):   
 self.w = w  
 self.l = l  
 self.description = "This house has not been described yet"  
 self.date = "None"

In this method I identify variables I will want passed in or through the class, such as width and length (**w, l**) so I can calculate different square footages in a later class method. Note the use of ‘self’ as a function and argument. ‘*Self*’ is used to reference different object instances. This is necessary because as your program is running, you could create a bunch of different class or object instances. Take for instance my hot tub remodel, I could continue testing different lengths and widths, and each time I put in a new set of values a new class instance is created, and new instances of the class objects as well. Using ‘self’ tells the program which object or class version to pull from, that being the object within its class (itself). Alternatively, *@staticmethod* allows you to call a class’s methods/functions directly, without creating a new class object, or using self as an argument. You can do this like so:

class Home:  
 def \_\_init\_\_(self, w, l):  
 self.w = w  
 self.l = l  
 self.description = "This house has not been described yet"  
 self.date = "None"  
  
 @staticmethod  
 def sqft\_calc (width, length):  
 sqft\_tot = (width \* length\*\*3)  
 return sqft\_tot  
#Main  
gSQFT\_tot = Home.sqft\_calc(100,200)  
print(gSQFT\_tot)

I likely got the calculations for square footage wrong, but the example serves its purpose. Since I am not concerned about various hot tub configurations in this configuration, I do not need to indicate which object instance the values of width or length are referring to. Because of this nature, it is best to use static method for calculations or processing, and the ‘self’ method for storing information such as the results of the addition of a hot tub deck.

Attributes and properties are next in our chain of concern, and the two are closely related. *Attributes* are ‘fields’ created using special syntax in the constructor section of your script, and they hold data internal to the class. The ‘**sqft\_total**’ object or variable from my last bit of code is a good example of this, as it was created ‘invisibly’ and outside of the \_\_init\_\_ method, when compared to say ‘**self.w**’. Attributes, like any variable, however, are limited in the ways with which you can control the data coming in or out outside of multiple lines of complex code.

A far more efficient way to manage these things are properties. *Properties* are customized functions whose sole purpose is generally managing attribute data. Properties have *directives* that allow you to add different kinds of code to the method. For example, see the code below which uses a ‘setter’ directive, allowing me to change the parameters around when data is assigned to an attribute, and when it should be disregarded:

@house\_name.setter# (setter or mutator)  
def house\_name(self, value):  
# The name must match the attribute!  
if str(value).isnumeric() == False:  
 self.HouseName = value  
else:  
 raise Exception("Names cannot be numbers")

As the code above reads, I want to set a house name. Assuming this is a name that will be created in the main body of the script, with user input potentially, I want to ensure no numbers are included in the name. I do so with a try-except block, under the property **house\_name**.

Properties are defined are functions that manage attributes or fields, but there are plenty of other different kinds of functions in a class that are collectively called ‘*methods*’. The ‘**sqft\_calc’** function from previous code examples is an example of this. Python also has ‘built-in’ methods such as the string method (**\_\_str\_\_()**) which can return some information about a class, like its associated doc string. *Doc strings*, by the way, are incredibly useful for conveying class information to another user or developer, such as its purpose, outputs, etc.

To wrap up this week we were introduced to GitHub Desktop, which I had already downloaded when we first started using GitHub. *GitHub Desktop* uses a program called ‘*GIT*’ to manage script versions and perform tasks such as backing up files to the cloud. GitHub desktop has remained relatively untouched since I downloaded it, as GitHub in general is taking some time to get used to, but I look forward to using it more.

III. Assignment Work-through

This week’s assignment I found far more challenging than previous assignments. Despite having access to the answer code and scouring online resources I was unable to make certain parts of the program work as intended. The answer code provided by Professor Root ran into the same problem as my program. I am hopeful that I can meet with course staff to resolve this error.

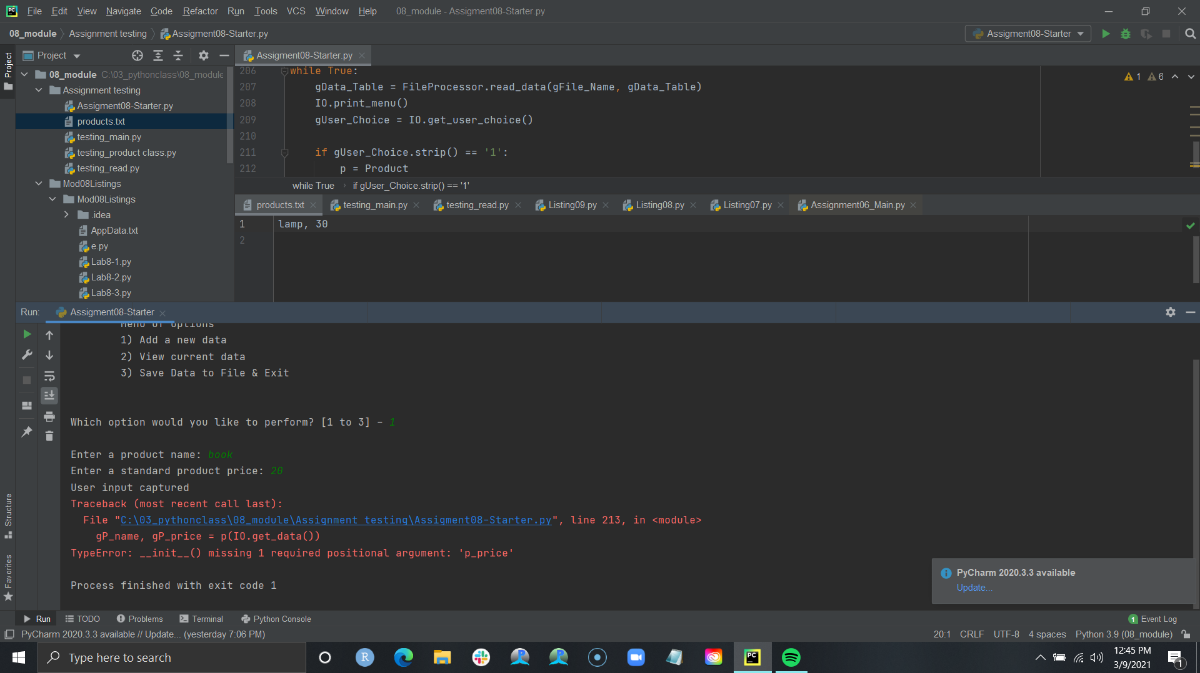
 I started this week’s assignment with the starter code provided by Professor Root. Initializing the code was simple, as for most of the needed function I transferred over code from previous assignments. I ran into quite a few errors concerning positional arguments, as seen in Image 3.1. I found that this was because the code in the main body of my script was treating name and price as one variable, instead of separate variables, therefore I was left with an unfilled argument (price). While the fix wasn’t immediately obvious to me, eventually I modified my ‘get\_data’ method to handle the variables as products instead of relying on global variables in the main body of my text, like so:

Image 3.1. Positional Error

name = str(input('Enter a product name: ').strip())  
price = float(input('Enter a standard product price: ').strip())  
print('User input captured')  
p = Product(p\_name=name, p\_price=price)  
return p

I also ran into an infinite loop while editing my ‘**read\_data**’ method, and issues with data typing which I remedied with changing from a dictionary to a list, adding the ‘**file.close()**’, and making sure to include data type in every argument I passed into each method.

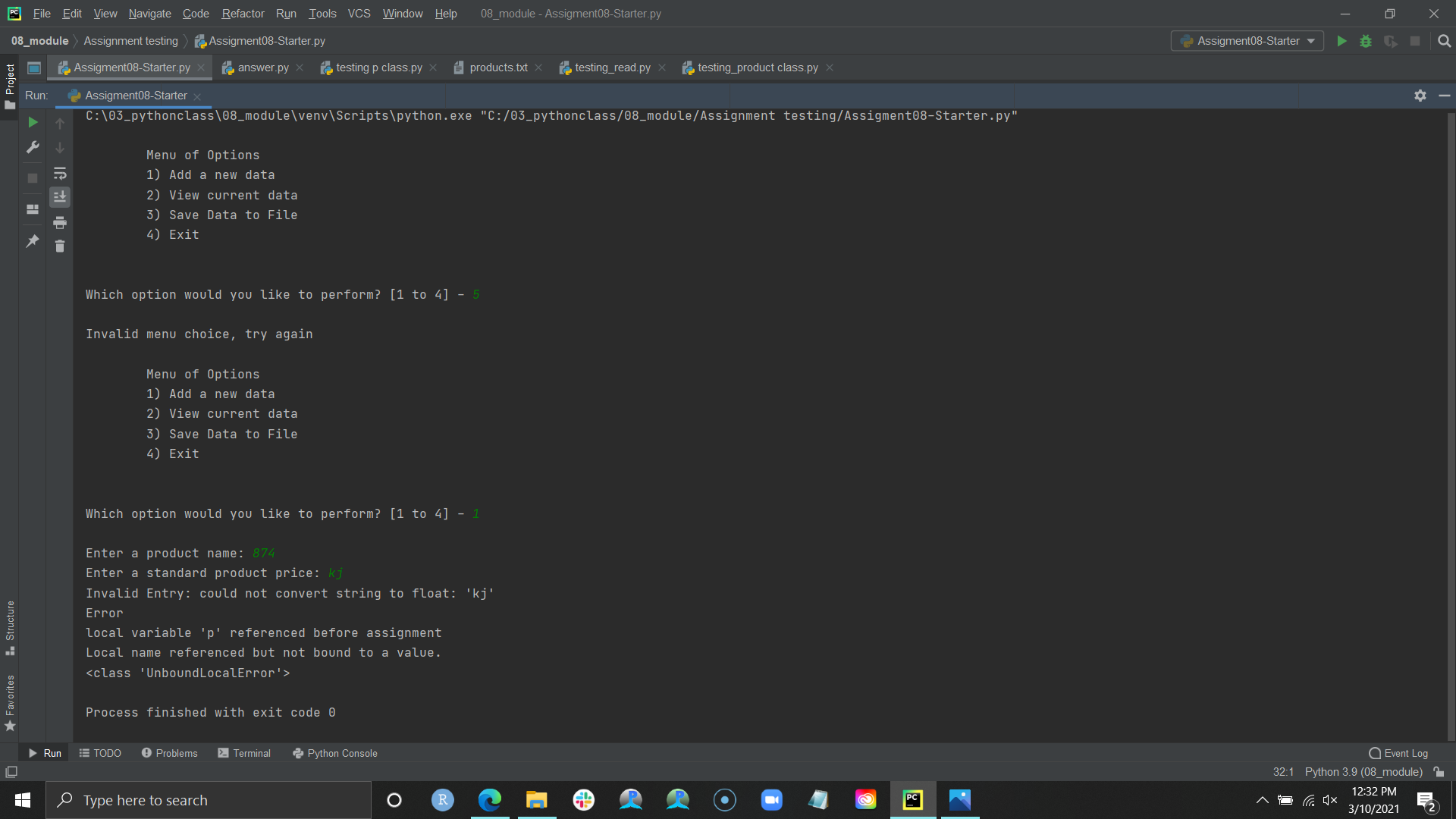
 The unresolved issue I ran into with both my script and Professor Root’s was the error handling in the setter properties not working as intended, specifically with the method intended for adding user input to the list. Despite indicating that the user input was of the Product class, if I tried to test the error handling in the setter property by inputting numerical values for the product name (or letters for the price), the program would result in a ‘**UnboundLocalError**’ and end the program, instead of the custom message indicated in the setter properties, as seen in Image 3.2. After copying and transferring Professor Root’s answer script I ran into the same error. I was ultimately unable to completely remedy this problem despite much testing, but I was at least able to catch the error in the ‘**get\_data**’ method and keep the program from exiting. However, I had to move the ‘return p’ line of code into the try block to do so, which then resulted in the data being saved to the global product list. This created another error when displaying the list data to the user, which I partially remedied with another try-except block in the ‘**show\_data**’ method. This allowed me to show a custom error message and ensure the program continued but is not a true fix for the problem which I believe is the setter properties not being called. Despite looking at many online resources I could not find a way to use that property in a way that worked for my script.

Image 3.2. ULE Error