Name: Matthew Penny

Date: 2021 AUG 27

Course: Foundations Of Programming: Python

Assignment: Assignment 08

**Object Oriented Programming: Instances, Classes, Properties, and More**

**Introduction**

In this module, we covered the topics of OOP (Object Oriented Programming). Components of OOP we covered were: Fields, Attributes, Properties (getters), Constructors, Methods, Setters, and how to instantiate an object. Overall, this module was very dense and we were instructed that it would be a challenge. It was, but even so, the labs and final project were a very good way of introducing this complicated subject. Through this document, I’ll note some of the issues I ran into and how I was able to eventually get around them and apply my knowledge to make a class and create instantiations of it.

**Lessons Learned and Concepts of Interest**

As a requirement for this module, I reviewed all videos and content from the module 08 section here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site). I also read chapter eight of “Python Programming, Third Edition – For the Absolute Beginner”, reviewed one web page that detailed examples of content we were reviewing as well as a youtube video covering the content as well. After covering these materials, I am able to address points of interest laid out in the course materials:

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

Professor Besinger brought up a real world example of what classes were and what objects made from them are during one of his lectures. He noted that “classes” are like a set of blueprints that can be used as a set of instructions. In this case, to build a house – which would be an object. A class is a powerful set of properties that can be used over and over again to make objects; each time we’re doing this we’re creating an instantiation of a class in the form of the object. You can use this "blueprint" over and over again to make objects, these objects are separate and typically are not linked to changes – like a set of houses on a street crafted from the same set of blueprints used by a builder or contractor.

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

The components that typically make up a class are:

Fields: forms to store data in a class. They're created in the same way as variables.

A constructor (described below)

Attributes: These are internal fields or values that hold data, but they differ from fields.

Properties: Special attributes that have getter/setter abilities and decorators

Methods: These behave very similarly to functions and are code built within a class to accomplish a specific purpose. For example: list.clear() is a method belonging to the list class that empties out a list object.

**What is the purpose of a class constructor?**

Constructors are a special, dedicated method that is invoked automatically when an object is being created and are the first components of a class to . These are very useful for instantiating certain data being immediately loaded when creating an object. For example:

def \_\_init\_\_(self,msg)

self.message = msg

This method is always called first and automatically and is defined with \_\_init\_\_ and binds a passed message to the self.message field (self is described below in greater detail).

**When do you use the keyword "self"?**

This is a semi-official (not mandatory, but recommended) way to use it and it is handed in as the first parameter at any method you are invoking on an object - It gives a reference back to the object calling the class so the class is aware of where and what is being instantiated from it. As demonstrated in some of the video content I covered:

objCan1 = CanOnaString('my message')

Within the class CanOnaString:

def \_\_init\_\_(self,msg)

**objCan1** -> the (memory reference) this object holds, is passed to "self" so the copy of the class/blueprint knows with what object to do what the method is asking it to do. You're giving it a copy of reference to self so all the data that is stored with the object at the memory with that object is now available to the methods in the class. Hence, it's always required.

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

Simply put, you invoke a staticmethod (using the above keyword) for cases when you want methods to be called on a class level and not on an instance level. Static methods for example, do NOT use the “self” keyword as they don’t need to refer back to the object/instance making the request directly. It will perform some form of dedicated logic within the class, but not require or dictate direct interaction with an object that called the class (I have a good example of this below in the notes for Lab E). That way, it acts as a class level method instead of a method that requires the “self” keyword and any other direct action with the object calling the method.

**How are fields and attributes and property functions related?**

Fields and attributes are variables that exist within a class. Fields are created the same way as variables and can be interacted with directly, for example:

Class myClass()

myMsg = “”

myObj = myClass

myObj.myMsg = “Large fries”

will store the value “Large Fries” in myMsg. This is now a copy of the class, or an instance of it.

For attributes, these are typically associated with your \_\_init\_\_ method in a class and is created when an object is instantiated, but typically only when the \_\_init\_\_ constructor is called. For example, we would create an attribute like so:

Class myClass()

def \_\_init\_\_(self,msg)

self.message = msg

and call it in the same way we may in the first example. Next up, we have…

**Properties -** these are used for added control; these can also be made private so they cannot be accessible from the outside of the class (at least not directly). The special methods you use to interact with attributes are called properties. Typically, two for each attribute. One to set a value, and the other to access it.

These are also called "getter" or "accessor" for reading or getting, and "setter" or "mutator" for setting the attribute. You can use the keyword private in some languages to make it only accessible from within the class. The python way to do this is to prepend two underscores to the front of the name. A getter property is required to allow access to the private field or attribute.

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

“@property” is a special decorator used before a function that makes it specifically a property function – namely a “getter” function, and in the cases we used it for was to allow access to a private field or attribute. If not specified, the function that would ordinarily be defined wouldn’t really be used for this purpose. The alternating identifier for a function is a “setter” function, designed to set the attribute itself after a property decorated function (getter) obtains access to a private field/attribute.

**Why do you include a docstring in a class?**

For similar reasons compared to the previous uses we’ve covered; namely to inform others what the details of the class are. For example, we may have a docstring with the following:

* A basic description of what the class does; such as ‘’’This class is used to assemble furniture objects passed to it by a user””
* Args, such as what is mandatory, optional, or default and why they’re needed
* Methods included in a class, and what purposes they accomplish in relation to furniture assembly
* Return values, noting what will be returned from the class itself and/or it’s methods

These details can also be used in the \_\_doc\_\_ method to provide said details to a user if they require any of the above details and provide an added layer of support to users in order to lesson confusion and ensure more reliable usage of your class with less issues.

**Labs and Examples Covered**

As noted prior, we were given several labs to cover from module 08, here: https://saravji.github.io/saravjis\_hut/FDN\_Prog/Modules.html (external site)

I will discuss in detail these labs below and show output from them.

First, we covered lab A:

A picture containing text, screenshot, monitor, screen

Description automatically generated

***Fig. 01 – A simple class that has three fields defined that are populated and displayed using print***

This lab familiarized us with fields and how to set them directly. In this case, I just have three separate lines that store input in the Position, Title, and Length fields within the class before printing them. This is done using the Classname.Fieldname notation, before we do our printing. In this case, we’re interacting and assigning directly. Pretty straightforward and simple as far as the lab material goes so far. Next, we made it a bit more complex in Lab B:

A picture containing text, monitor, screenshot, indoor

Description automatically generated

**Fig. 02 – Updated code using an \_\_init\_\_ definition with three attributes: Position, Title, Length**

In this case, we’re defining a constructor this time with the \_\_init\_\_ definition and initializing our variables as attributes before we access them the same as last time, via the Classname.attributeName format. I did realize at this point that the code may function better by calling the function this way: TrackInfo(Position,Title,Length) – having assumed those to variables beforehand and passing them in. This change is reflected in Lab E below. In this case, we did successfully use the constructor and attributes to store and print out data. Next, we moved on to lab C:

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig. 02 – Updated code using an \_\_init\_\_ definition with three attributes: Position, Title, Length with fields commented out**

This code is only slightly different than the previous lab and I’m still using a constructor with three attributes: self.position, self.Title, and self.length. The rest of the logic is the same and as noted before, it would have been good for me to call the method one time on a single line while passing all three variables. The code however, still did work as expected. Next, we performed lab D:

A picture containing text, screenshot, monitor

Description automatically generated

**Fig. 03 – Upper section of code with the \_\_init\_\_ block, property declarations, and two setter methods**

A picture containing text, screenshot, monitor, indoor

Description automatically generated

**Fig. 04 – Lower section of the code with last setter method, error handling, and main body**

For this lab, we needed to introduce getter (using @property) methods and setter (using @field.setter) methods. First, I created three corresponding getter methods using @property and have them return our private variables from the constructor method. Once they have been retrieved, we then have three corresponding setter methods that “set” the values via return statements as well as running error handling. However, as noted below, the error handling would not work and I was able to fix it in Lab E (see further below).

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig. 04 – Running the code successfully. Error handling steps aren’t working for the three cases tried.**

I was able to run the code successfully with getter and setter attributes set, but was unsure how to get error handling to work after many attempts. Thankfully, I was able to get assistance from Professor Besigner and Laura during the Friday office hours session in which they noted what I was doing wrong. This leads in to lab E:

A screenshot of a computer

Description automatically generated with medium confidence

**Fig. 05 – Upper section of final code - includes constructor, three methods, and three properties**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig. 06 – Setter functions, and two of the new functions I made that are called in the constructor for error handling**

A screenshot of a computer

Description automatically generated with medium confidence

**Fig. 06 – Final error handling method, \_\_str\_\_ method used for returning a string, main body code, and multiple outputs**

In lab E, I tried to fix the error handling code initially and was unsuccessful. I was able to however, join a Friday office hours session where both Laura and Professor Besinger pointed out the flaws in my code. The fault lied in the \_\_init\_\_ block as I was only creating attributes and not doing any validation during that point. So, the code I wrote in the setter methods was not doing anything. To get around this, they recommended I create static methods to run in the \_\_init\_\_ block to test out user input. I crafted three and had them all immediately run before setting the initial attributes to be used below in the getter/setter methods. They also noted that the .isNumeric() function wasn’t working on line 42 because I was casting the “usrPos” variable as int before passing it in.

I also updated the main body code to have the input assigned to three variables, and then passed these all in on one line to TrackInfo() on line 96, assigning the output to myData.

I then print out the data in three ways:

* One by directly printing the variable myData
* Second, by casting it to string, assigning it to a variable, and printing the variable
* And lastly, using the new \_\_str\_\_() method I defined in the class that holds all the data in the same format

After that lesson-heavy project, I moved on to assignment 08:

For this assignment, empty starter code and pseudocode (along with TODO’s) noted what we had to get done. I actually recycled a lot of code from previous assignments – but even so, it had to be altered to accommodate the brand new CD() class that was written from which I was instantiating objects to be used. We’ll go over that code first:

Text

Description automatically generated

**Fig. 07 – The new CD() class with a docstring, a constructor (that includes error checking methods) and two getter segments**

In this first bit of code, we have a docstring detailing what the class does as well as the properties and methods defined. I start by setting the cd\_id, cd\_title, and cd\_artist attributes to self.\_\_ private variables. I then define three associated getters, and then three setters:

Text

Description automatically generated

**Fig. 08 – The setter properties and our two class functions**

After the setters have all been defined, I created a \_\_str\_\_ method to return all values back as a formatted string that can be used by the rest of the program. The final staticmethod “strCheck” is used from my Lab assignments to check in the \_\_init\_\_ block if blank fields are passed and will raise an exception if found. Next, we enter the FileIO class with addressed TODO items for writing/reading files:

Text

Description automatically generated

**Fig. 09 – The read\_file method that reads a text file from disk and imports it in our global list**

This code has been re-used, but adjusted for the list usage and also combines newly created error checking to throw an error if the file is not found. The Write\_File method below this one does the same, but using ‘w’ instead and throws a general exception if an error is encountered.

Text

Description automatically generated

**Fig. 09 – Static methods from class IO with updated text and values**

This was code from a previous lab, but I made some changes to exclude the deletion option from the menu. Since these just need to do the same thing regardless of when they’re instantiated, they’re static methods.

Text

Description automatically generated

**Fig. 10 – Invoking the CD class to make an instance (usrCD) that will be appended to the main list**

For the add\_CD() method, we invoke the CD class I wrote way up top. We collect data from the user as before, then pass all three values to the CD class and point it at out instance. We also call the \_\_str\_\_ method written in the class to tell the user what we added (which would be a simple comma separated set of three values as dictated in the method) before we take that series of comma separated values and append it to the list. This section also includes error handling and will invoke the error handling within the CD class as well if blank values are passed.

Text

Description automatically generated

**Fig. 11 – The updated main loop from previous examples**

Lastly, we have the main loop – which is an updated version from past assignments that has new variable names and some added tweaks. Note that it’s also been updated to exclude the ‘delete” function that was removed. I tested out the code and the results are below in the appendix – this just includes successful workings and not all the error handling I coded in.

**Summary**

The concept of object oriented programming was honestly indeed hard to grasp. Thankfully, through watching the instructional videos a few times, checking examples online, and receiving some help during the office hours session, I was able to work it successfully into code and get it to work smoothly. I will definitely need to repeat the practice and build more involved classes in the future and am looking forward to learning about inheriting from classes besides just object, as was done in this case.

**Appendix**

Below are working examples of my code from Spyder and the console:

A picture containing text, screenshot, monitor, black

Description automatically generated

**Fig. 12 – The add function being used in Spyder successfully**

A picture containing text, screenshot, monitor, screen

Description automatically generated **Fig. 13 – The display function being demonstrated in Spyder successfully**

A picture containing text, screenshot, monitor, screen

Description automatically generated

**Fig. 14 – The load from file option being demonstrated in Spyder successfully**

A screenshot of a computer

Description automatically generated with medium confidence

**Fig. 15 – The save to file function being run successfully, with exported values shown**

**Text

Description automatically generated**

**Fig. 16 – The add and display methods being run successfully via the console**

**Text

Description automatically generated**

**Fig. 17 – The load from file method being demonstrated successfully via the console**

**Text

Description automatically generated**

**Fig. 18 – The save function being demonstrated successfully via console with output.**