Bramble Eagle-Jack

2021-03-01

Foundations Of Programming - Python

Assignment 8 Knowledge Document

Object Oriented Programming

# Introduction

This week we take an in-depth look at object-oriented programming with an emphasis on defining classes in order to create objects. The assignment asks us to take another look at the CD inventory program but this I am writing a majority of the code and using a list of objects as my CD inventory.

# Classes and Objects

Using Fields, Constructors, Attributes, Properties and Methods, classes create a blueprint from which each object is created or instantiated. An object is created by assigning a class to a variable and each instance inherits the characteristics of the parent class. Changes to one object of a class type do not affect the other objects of the same type nor will they affect the parent class.

Fields are data stores of a class and are placed immediately below the definition of the class name. They contain explicit data and are shared between all objects belonging to the parent class containing the field. A field can be accessed by the class or any object of that class using dot notation. They’re useful for storing data related to the instantiation of objects because the value is maintained at the class level.

1. **class** TrackInfo():
2. #Fields
3. tracks = 0
4. **def** \_\_init\_\_(self, position, title, length):
5. # Attributes
6. self.\_\_position = position
7. self.\_\_title = title
8. self.\_\_length = length
9. TrackInfo.tracks += 1

Listing 1 - demonstration of fields and attributes from trackinfo.py

Constructors are a specialized method and Python’s constructor method is the dunder init. Constructors are convenient ways to immediately load data when an object is instantiated. To instantiate an object, I assign the class to a variable and the blueprint of the class is transferred to the object. Arguments can be passed into the dunder init method to be assigned to the object’s attributes when it’s instantiated. The attributes are shared between all objects in the class but the values I assign them are unique to each object.

The constructor method is the first place where we see the self keyword. It is not an official keyword and I could use a different parameter if I wanted to but it’s become part of the community standards and best practice. It’s the first argument in every method of a class that references an object because it is the reference to the object. If I don’t use a keyword reference to an object when calling the class method then I will get an error saying that I did not enter the required amount of arguments.

There’s no intrinsic control over what gets entered into attributes during runtime so properties help define what can be assigned to them. Classes should be created in a manner that they can be used with any other script so this help take care of any issues another programmer might encounter. Typically, two are created for each attribute: one to access it (getter) and one to set it (setter). Getters can return transformed versions of attributes depending on the purpose of the script and setters are used to check the values of the attributes are data types that I want to accept. The getter property is always followed by the setter property because the getter function name is used to define the decorator of the setter. Decorators start with the @ symbol and are in the line directly above the property it decorates and changes or defines its scope. However, the naming conventions for getters and setters are not consistent in Python.

1. #Properties
2. @property
3. **def** position(self):
4. **return** self.\_\_position
6. @position.setter
7. **def** position(self, string: str) -> int:
8. **if** string.isnumeric():
9. self.\_\_ = int(string)
10. **else**:
11. **raise** Exception('Position must be an integer.')

Listing 2 - demonstration of getter and setter from trackinfo.py

Methods are like functions in a script and are called in a similar manner using dot notation with the object name. Like other functions in a class, methods also use the self keyword to create a reference to the object that’s calling it. A method within a class with just the self keyword as an argument would not require any arguments when called in the main body. Static methods use the @staticmethod decorator and are called at the class level so the self keyword is not required as an argument as they do not reference objects. They are called using dot notation on the name of the class. Even if the method is used with an object belonging to the parent class, the value is the same because the value belongs to the class and all objects inherit the same value.

The dunder string method is a default method that is called when an object is cast to a string. By default it will return the memory position of the object. I can also define it within a class to return a value of my choice. It’s one of the few private methods that is, by convention, acceptable to call directly. The double underscore is what makes the method private and can be used on my custom class methods as well as fields, attributes, and properties. Making anything private means they can’t be accessed directly and the AttributeError is raised if I try to call a dunder method. There are ways to access them but usually only to debug.

1. #Methods
2. @staticmethod
3. **def** num\_tracks():
4. **print**('There are', TrackInfo.tracks, 'tracks')
6. **def** \_\_str\_\_(self):
7. **print**('This class tracks tracks')

Listing 3 - demonstration of methods from trackinfo.py

# Assignment Eight

I first worked on the FileIO class in this week’s assignment because the pickle module was still fresh in my mind and I knew I’d need to use it in order to read and write a list of objects to a file. The load\_inventory method relies on one argument that is the file to be read. I open file as an object and assign the value of the pickle loaded file to a variable that is returned, to be assigned to the list of objects in the main program loop. The save\_inventory method relies on two arguments: the name of the file to be opened and the list of objects to be written to it. I open the file as an object and pickle dump the contents of the list into the file. Both methods rely on the with statement for opening files so they are implicitly closed.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure 1 - CDInventory.py running in Spyder

Next, I worked on the IO class methods which are also familiar and quick to code. The show\_menu method prints all the user options for the program and runs in the main program loop so that the user sees the menu repeatedly. The menu\_choice method captures the input of the user and returns the value to the main loop where it triggers conditions of the menu options based on the value input by the user. The display\_data method accepts the list of objects as an argument and iterates through them with a for loop in order to print each object’s attributes which contain the CD ID, Artist and Title. The get\_input method captures three inputs from the user into three variables that represent the CD ID, Artist and Title that are all returned as a collection where they can later be used as arguments to instantiate an object.

The CD class is used as the blueprint for creating CD object instances. I use the dunder init method to define three attributes for each object instance, cd\_id, cd\_title, and cd\_artist. The three attributes are passed into the dunder init as arguments in addition to the self keyword that references the object instance. All three attributes are private as marked with the double underscore so I have getter functions with with @property decorators for each that just return the value of the attribute as defined by the argument in the instantiation. The cd\_id attribute is the only one that requires a setter decorator because I need to check that its data type is an integer. I use a string method and if statement to check if the cd\_id is made of all numbers and if it is then I use the integer function to change its type. Otherwise, I raise a custom exception telling the user that the ID must be an integer.

The exception is used in the main loop after the user chooses to add CD to inventory. The three inputs from the user are assigned to the cd\_values as a tuple and then I use index positions of the tuple values to instantiate cd\_object with the CD class. I then check the cd\_id by trying to assign it the string input by the user for the cd\_id. If it succeeds then the cd\_object is appended to the list, having successfully cast the cd\_id as an integer. Otherwise, the exception raised in the setter decorator for cd\_id is printed and the main loop continues.

I added two more try statements to the main body, each time when the load\_inventory method from the FileIO class is called in the event that the file does not exist. I also wrote the choice of saving into an if statement so that the list of objects is only saved if it has data instead of allowing the user to save an empty list to a file.

Text

Description automatically generated

Figure 2 - CDInventory.py running in terminal

# Summary

I have previously seen classes as a way to organize the separation of concerns and house functions but they also serve to hold data and functionality of objects. They’re powerful tools for creating code abstraction much like functions but even more complex as they feel like their own little programs. This week’s introduction to classes and objects felt familiar to me as if I were learning many of the same concepts we have covered but in a different context and with different vocabulary. However, I also feel like this is just the tip of the iceberg and I’m only just beginning to grasp how useful they can be. I’m inspired to use what I’ve learned this week in a game script as I think object instances would be useful for simple objects like power-ups or more complex objects like character and enemy creation.

# Appendix

## Listing trackinfo.py

1. **class** TrackInfo():
2. #Fields
3. tracks = 0
4. **def** \_\_init\_\_(self, position, title, length):
5. # Attributes
6. self.\_\_position = position
7. self.\_\_title = title
8. self.\_\_length = length
9. TrackInfo.tracks += 1
11. #Properties
12. @property
13. **def** position(self):
14. **return** self.\_\_position
16. @position.setter
17. **def** position(self, string: str) -> int:
18. **if** string.isnumeric():
19. self.\_\_ = int(string)
20. **else**:
21. **raise** Exception('Position must be an integer.')
23. @property
24. **def** title(self):
25. **return** self.\_\_title.title()
27. @property
28. **def** length(self):
29. **return** self.\_\_length
31. #Methods
32. @staticmethod
33. **def** num\_tracks():
34. **print**('There are', TrackInfo.tracks, 'tracks')
36. **def** \_\_str\_\_(self):
37. **print**('This class tracks tracks')