Maria Dacutanan

September 02, 2020

Foundations of Programming: Python

Assignment 08

Object Oriented Programming (OOP)

# Introduction

For module 8, I venture to the wonderful and a bit confusing world of object-oriented programming, aka OOP. So far, in the previous modules, I have been writing programs in what is called **procedural programming** approach. Fundamentally, procedural code is one that directly instructs a device on how to finish a task in logical steps. This paradigm uses a linear top-down approach and treats data and procedures as two different entities. Based on the concept of a procedure call, procedural programming divides the program into procedures, which are also known as routines or functions, simply containing a series of steps to be carried out.

Simply put, procedural programming involves writing down a list of instructions to tell the computer what it should do step-by-step to finish the task at hand.[[1]](#footnote-1)

**Object-oriented programming**, on the other hand, refers to a programming methodology based on objects, instead of just functions and procedures. It is the method of structuring a program by bundling related behaviors into individual **objects.** An object is an entity that has characteristics (attributes) and behaviors (methods).

# Features of OOP

There are a few key concepts in Python to keep in mind when we talk of OOP[[2]](#footnote-2):

* **Class**

A class is a blueprint of objects that defines the common attributes and behavior.

* **Object**

Objects are an instance of the class. It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access data.

* **Method**

As mentioned, objects have attributes and behaviors. These behaviors are called methods.

* **Inheritance**

Inheritance specifies that one object acquires all the properties and behaviors of parent object. By using inheritance, we can define a new class with a little or no changes to the existing class. The new class is known as derived class or child class and from which it inherits the properties is called base class or parent class. It provides re-usability of the code.

* **Polymorphism**

It is an OOP methodology where one task can be performed in several different ways. To put into simple words, it is a property of an object which allows it to take multiple forms.

* **Data Abstraction**

We touched on this concept back in Module 6. This [website](https://stackify.com/oop-concept-abstraction/)[[3]](#footnote-3) explains it very well. Abstraction is a mechanism which represents essential features without including implementation details.

* **Encapsulation**

This is a concept we touched on back in module 6. Encapsulation is a principal of abstraction. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object’s definition.

A class is an example of encapsulation as it encapsulates all the data that is the member of functions, variables etc.

While abstraction is implementation hiding, encapsulation is information hiding.[[4]](#footnote-4)

# How To Create A Class

Now that we have pertinent OOP terminologies out of the way, time for some action.

To define a class, you use the **class** keyword, followed by the class name and a colon. In line 11 of Figure 1 - Lab08\_A.py, I created class TrackInfo.



Figure 1 - Lab08\_A.py

# Defining A Method

Defining a method is similar to defining a function. In fact, methods are functions associated with objects. In lines 15-17 of Figure 2 - Lab08\_B.py, I defined a **method** , more specifically , a **constructor method.** A constructor method is a special method that is automatically invoked right after a new object is create. It is also called **initialization method**. In line 15, I type in the keyword **def** followed by the keyword **\_\_init\_\_,** followed by the two parameters enclosed by parenthesis. The first parameter is called **self.**

The self

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name self.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name self - any other name is definitely frowned upon.[[5]](#footnote-5)

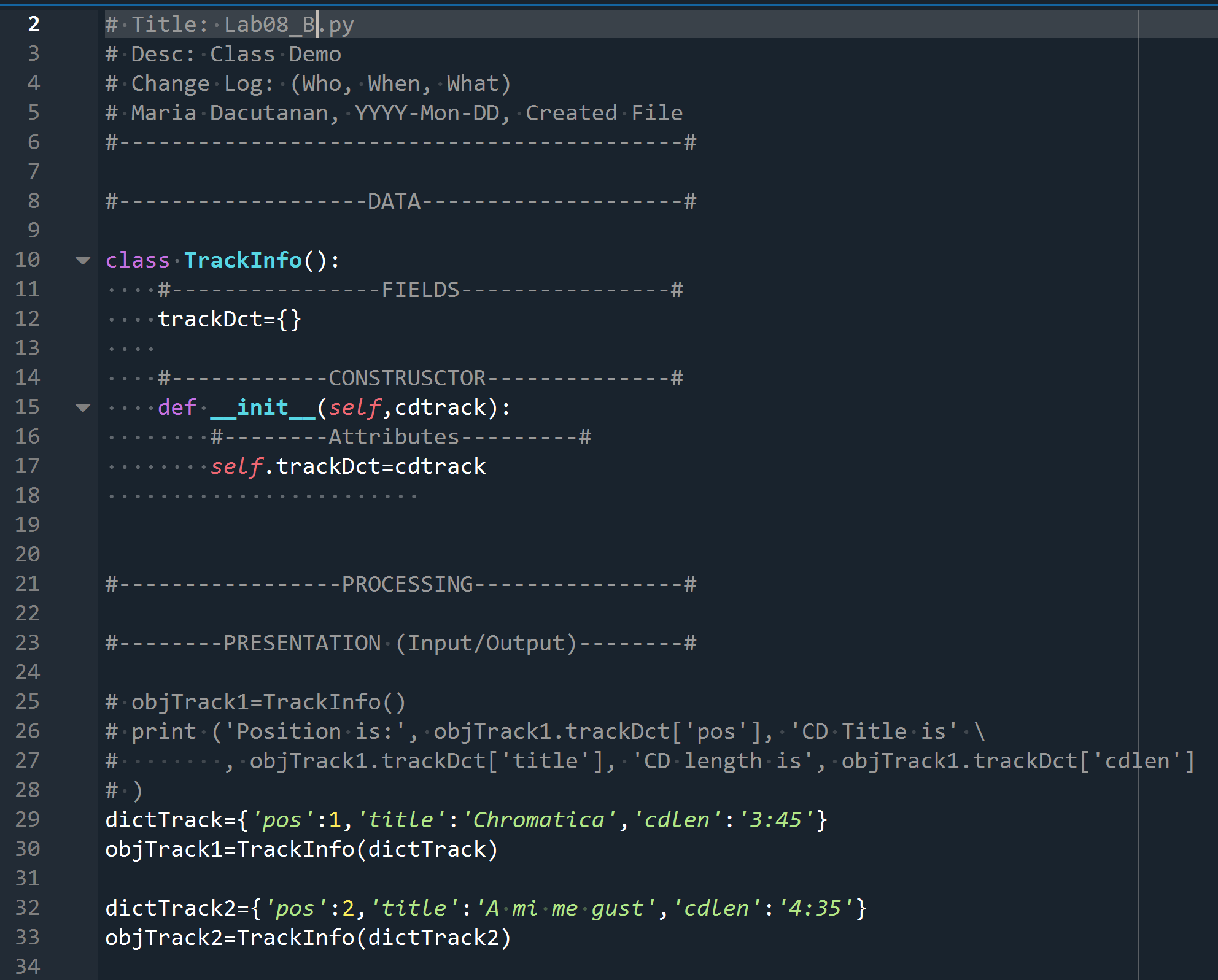


Figure 2 - Lab08\_B.py

# Instantiating Objects

Instantiating an object is creating a copy of the class which inherits all class variables and methods. To instantiate an object, we simply call the class as it were a function. The return value will be the newly created object.

In line 19 of Figure 1 - Lab08\_A.py shows how a new object is instantiated and stored into objTrack1.

After an object is instantiated, the fields defined with the TrackInfo is now made available to me via the objTrack1 object. In lines 27-28, I print out the pos, title and cdlen values of the trackDct dictionary defined in the class trackInfo. Figure 3 - Output of Lab08\_A.py shows the output of the print statement.

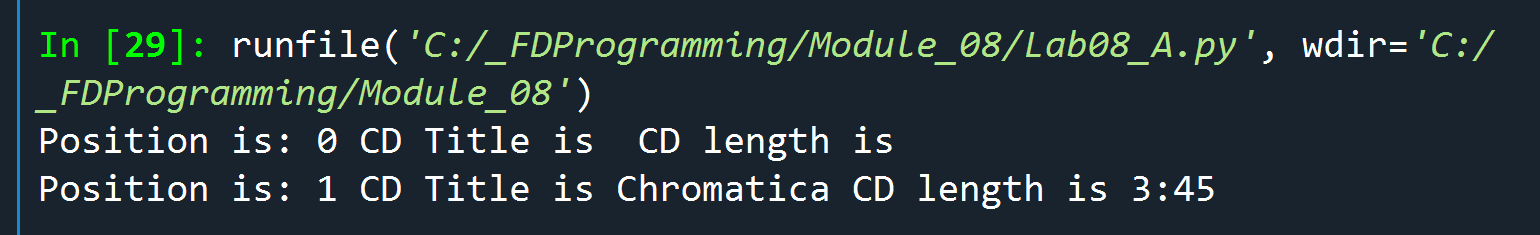


Figure 3 - Output of Lab08\_A.py

# Creating Attributes

An object’s attributes can be automatically created and initialized after its instantiated through its constructor method. In Figure 4 - Lab08\_C.py, lines 17-19 shows that the constructor method has three attributes that are automatically created after an object is instantiated. Attribute name is created by using the keyword **self** followed by dot (.) and then the attribute name. The attribute gets the value of the variable passed on to it through the constructor method.

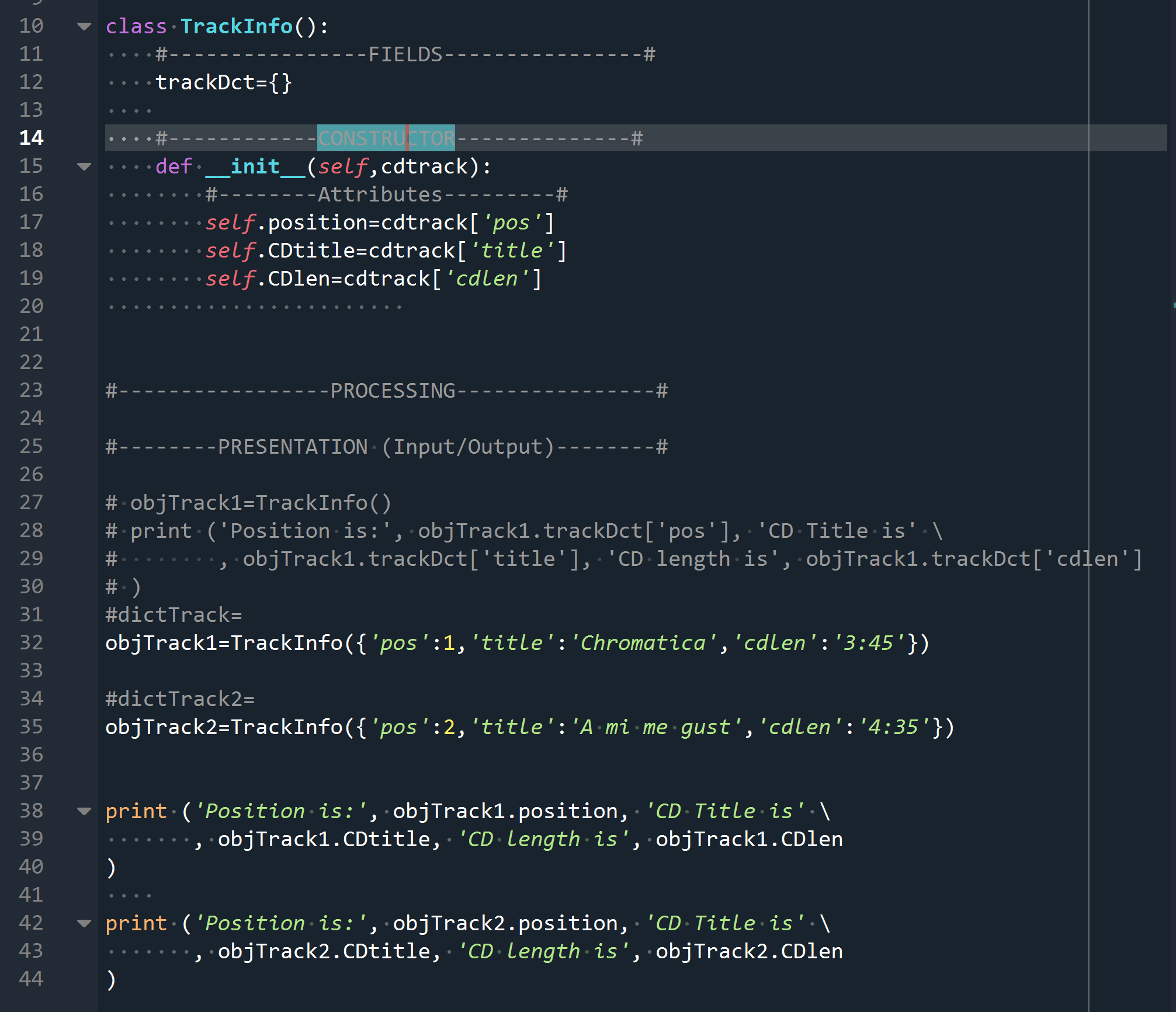


Figure 4 - Lab08\_C.py

# Accessing Attributes

By default, you can access and modify an object’s attribute outside the class. To do this, you can use the dot notation. Type the variable name, followed by the dot followed by the attribute name. Lines 38-39 in Figure 4 - Lab08\_C.py demonstrates how I invoked the three attributes (position, CDtitle, CDlen) that I have defined under the constructor method via the objTrack1 object. Figure 5 - Lab08\_C.py output is the output Lab08\_C.py

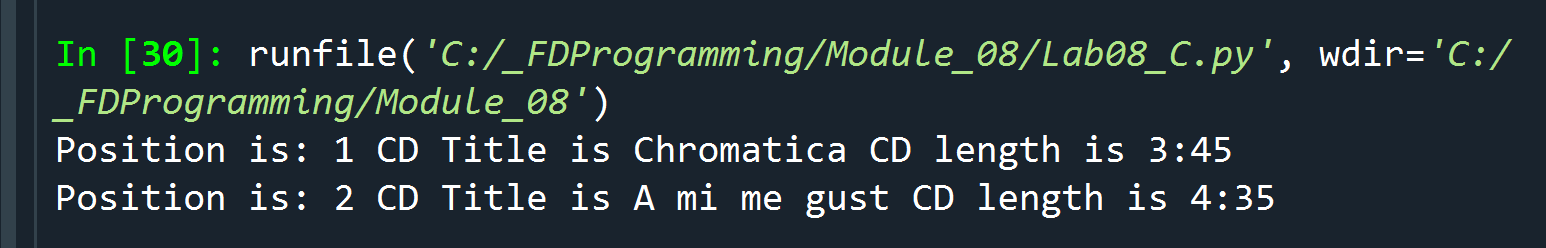


Figure 5 - Lab08\_C.py output

# Using Private Attributes and Private Methods

By default, all of an object’s attributes and methods are public, meaning that they can be directly accessed or invoked by a client. To encourage encapsulation, you can define an attribute or method as private, meaning only other methods of the object itself can easily access or invoke them.[[6]](#footnote-6)

## Creating Private Attributes

In lines 17-19 of Figure 6 - Defining Private Attributes, I have modified the attributes under the constructor method by prepending the attribute name with two underscores (\_\_).

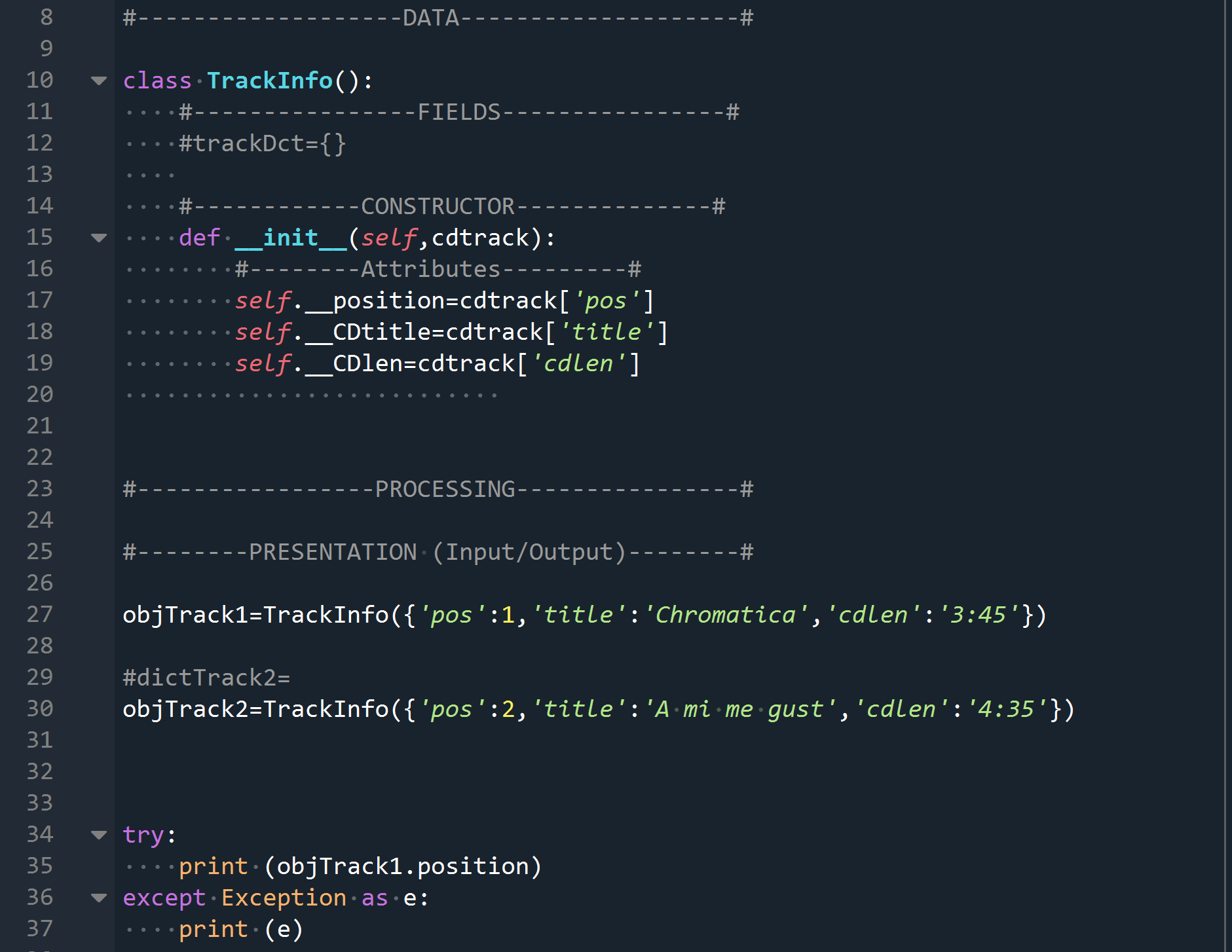


Figure 6 - Defining Private Attributes

So, what happens when I try to access a private attribute directly? I get an attribute error as shown in Figure 7 - Output of Code on Figure 4.

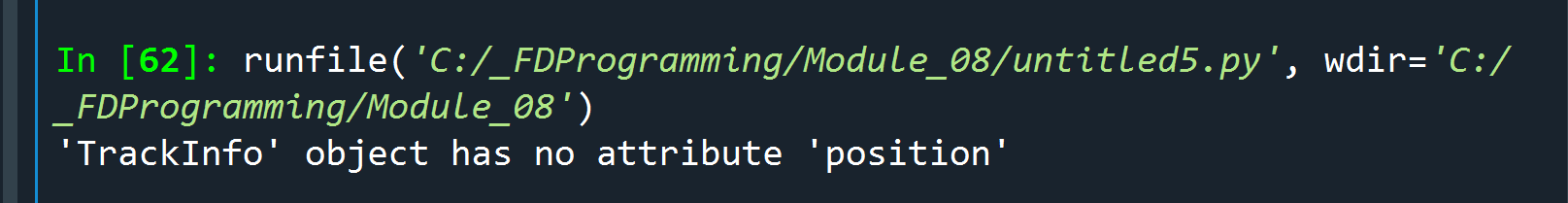


Figure 7 - Output of Code on Figure 4

Can private attributes ever be directly accessed? Technically, yes (modify line 35 in Figure 6 - Defining Private Attributes to *print (objTrack1.\_TrackInfo\_\_position)*). But we should never try to directly access the private attributes (or methods) of an object outside of its class definition.

# When to Implement Privacy & Best Practices[[7]](#footnote-7)

Implementing privacy should be only be used when necessary. Make any method you don’t want a client to invoke. If it’s critical that an attribute never be directly accessed by a client, make it private.

When you write a class:

* Create methods to reduce the need for clients to directly access an object’s attributes.
* Use privacy for those attributes and methods that are completely internal to the operation of objects.

When you use an object:

* Minimize the direct reading of an object’s attributes
* Avoid directly altering an object’s attributes
* Never attempt to directly access an object’s private attributes or methods

# Controlling Attribute Access

Creating Properties

One way to control access to a private attribute is by creating a **property**. A property is an object that allows indirect access to private attributes.

In line 21 Figure 8 - Lab08\_D.py, I start to define a property by typing in the **@property** decorator. Then right beneath the decorator, I define a method for the field that I want to share/limit access to. Lines 22-23 shows I am sharing read access to the position attribute. Now, I actually want to share more than read access but also write access to this attribute. To do this I start a new property block but this time using the **@setter** decorator. In line 25, I specify that I start the block with @position.setter and then right beneath it, lines 26-30, I impose a restriction – that the value being assigned to the private attribute must be numeric. If not, raise an exception. Otherwise, make the changes to the attribute value.

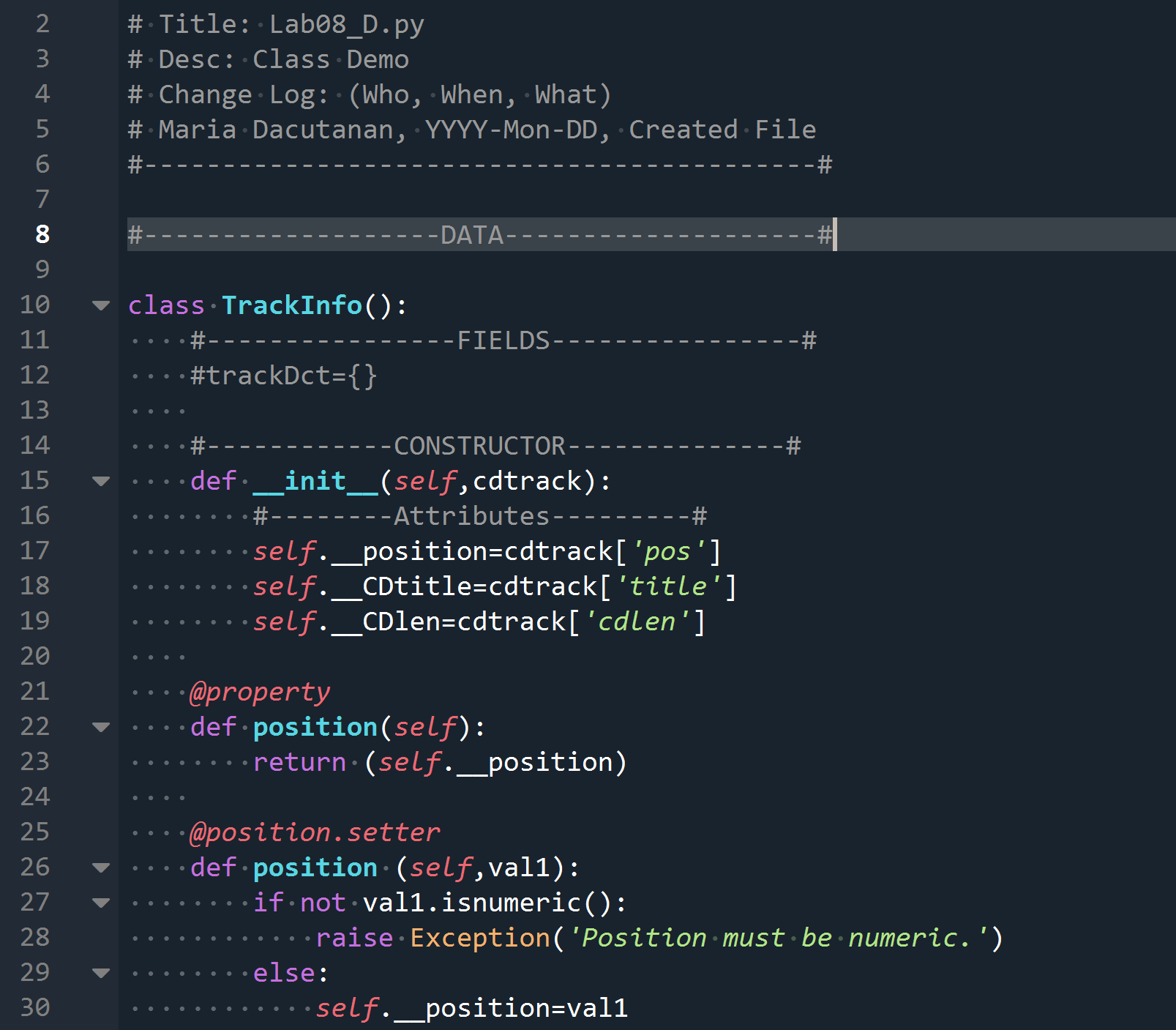


Figure 8 - Lab08\_D.py

Accessing Properties

Now that I have created properties for each private attribute in the TrackInfo class definition, I should have some level of access to the private attributes. In line 65 of Figure 9 - Lab08\_D.py cont'd, I create a new object by invoking the TrackInfo class and passing on 3 arguments. In lines 72-74, I access the private attributes by via the objTrack1 object followed by a dot (.) then the name of the attribute I want to see. Figure 10 - Lab08\_D.py output shows the printout of the attribute values that I set earlier.

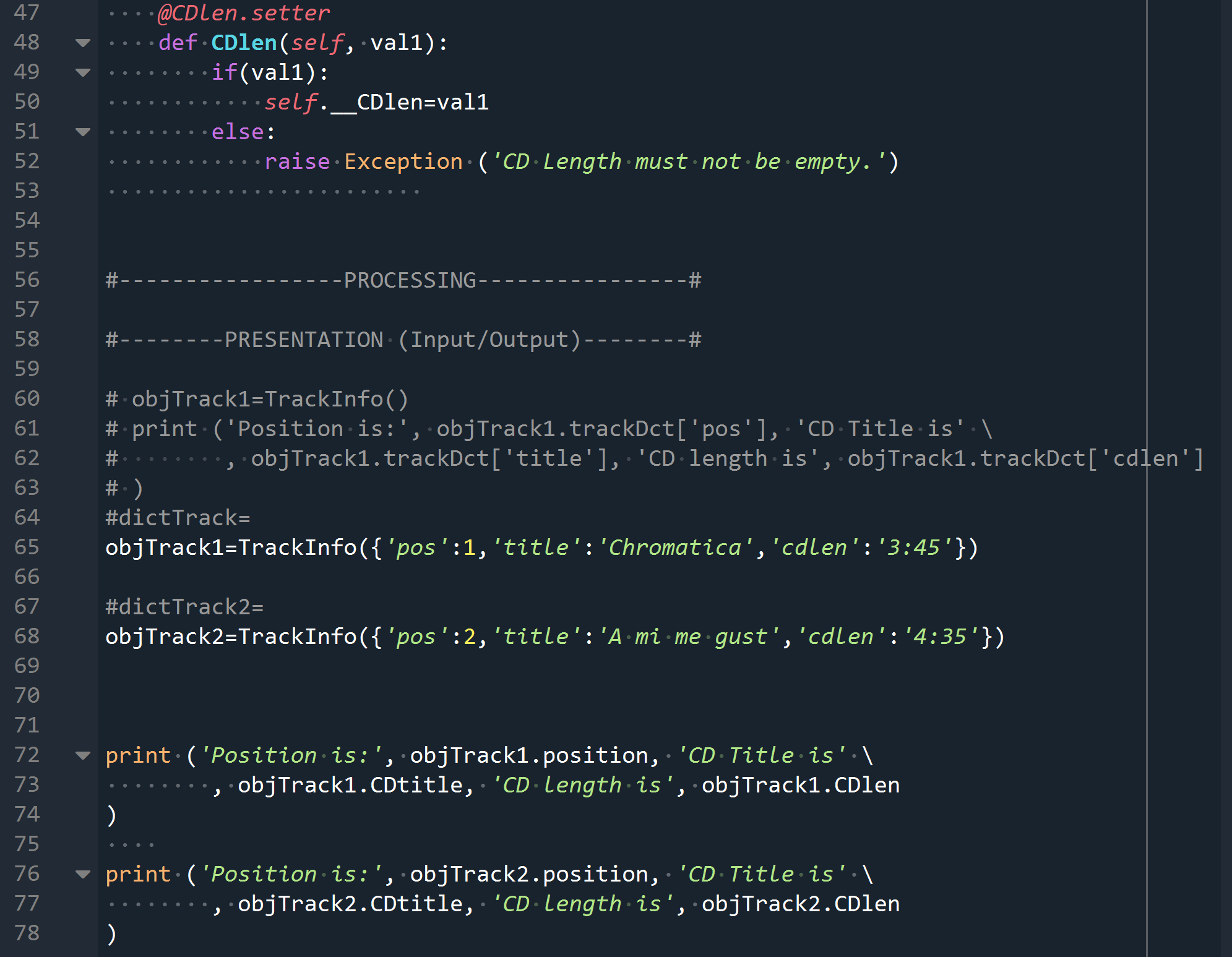


Figure 9 - Lab08\_D.py cont'd

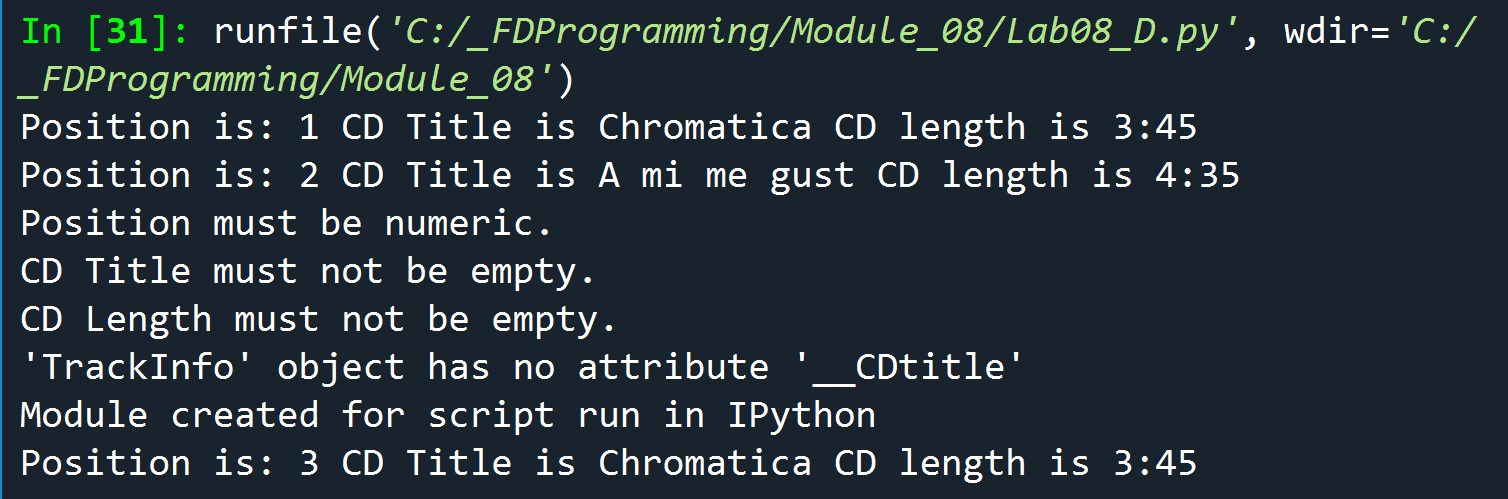


Figure 10 - Lab08\_D.py output

Now, to test the restrictions imposed on the private attributes, observe Figure 11 - Lab08\_D.py test cases. In line 81, I am attempting to assign the string ‘x’ to the private attribute postion; on line 86, I am setting the CDtitle attribute value as well as the CDlen in line 91, to an empty string. Then I try to access the private attribute CDtitle thru its dunder name and lastly, in line 102, I simple assign a string value ‘3’ to the position attribute.

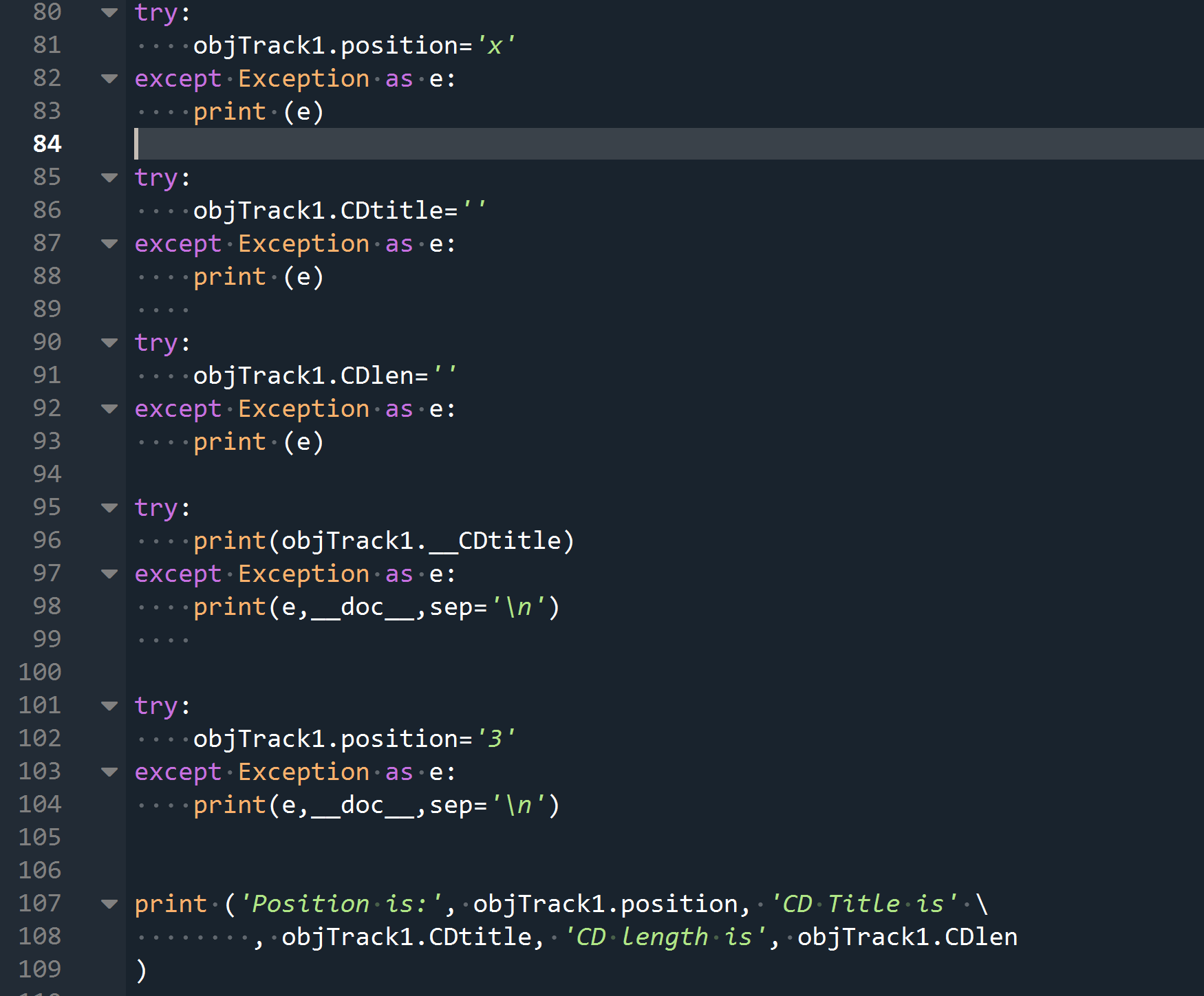


Figure 11 - Lab08\_D.py test cases

As you can see in Figure 10 - Lab08\_D.py output, the attempt to assign a string ‘x’ to private attribute position raised an exception (Position must be numeric).The attempt to assign empty strings to both CDtitle and CDlen attributes also raised an exception (CD Title must not be empty and CD Len must not be empty).When I tried to access \_\_CDtitle private attribute, I also received an exception back ('TrackInfo' object has no attribute '\_\_CDtitle'). On the other hand, assigning ‘3’ to position attribute raised no exceptions.

# \_\_string\_\_ Method

Going back to Figure 9 - Lab08\_D.py cont'd, if I try to print out the instance objTrack1 created in line 65, I get an output that is not very descriptive as shown in Figure 12 - Print Out of an Instance.

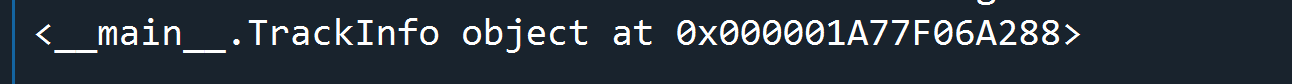


Figure 12 - Print Out of an Instance

To make the printout more meaningful, we can use the dunder method called **\_\_str\_\_** to return a string value that is much more meaningful. To do this, I simply add the \_\_str\_\_ method in my class definition.

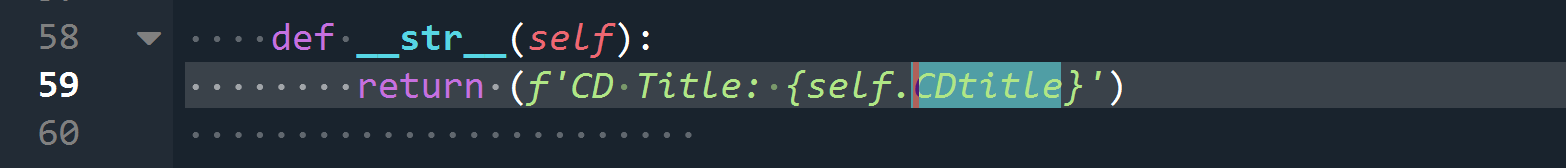


Figure 13 - \_\_str\_\_ dunder Method

After adding the \_\_string\_\_ method, printing out objTrack1 instance now looks less ambiguous as shown in Figure 14 - \_\_str\_\_ output. This [website](https://dbader.org/blog/python-repr-vs-str)[[8]](#footnote-8)has a nice writeup two dunder methods, \_\_str\_\_ and \_\_repr\_\_.

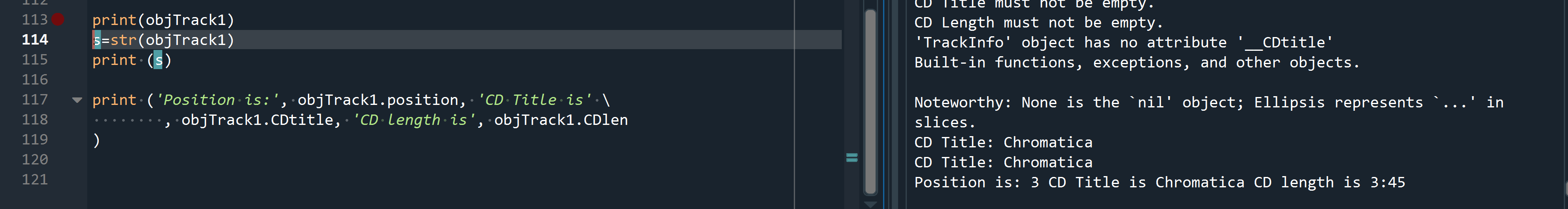


Figure 14 - \_\_str\_\_ output

# Applying OOP in CDInventory.py Program

For the CDInventory.py program, I was able to apply OOP in class CD and class FileIO. In class CD specifically, I used the \_\_init\_\_ method to create default private attributes (\_\_cd\_id, \_\_cd\_title, \_\_cd\_artist). I then used setter methods on each of these private attributes to allow setting of the attribute values but with added restrictions such us id must be numeric and cd\_id and cd\_title cannot be empty. Finally, I added a method called add\_cd to append new CD objects into existing list. Figure 15 - class CD (CDInventory.py) shows a snippe tof class CD.

In class FileIO, I added \_\_init\_\_ method to set default private attributes (\_\_filename,\_newTable,\_\_loadctr,\_\_savectr) followed by definition of properties for the getter method. I then created 2 methods: load\_inventory and save\_inventory.

Lastly, for class IO, I defined class functions (print\_menu, menu\_choice, show\_Inventory, get\_newInventory) that have been marked with @staticmethod decorator. These means that these functions can be accessed thru class even if no objects have been instantiated yet.

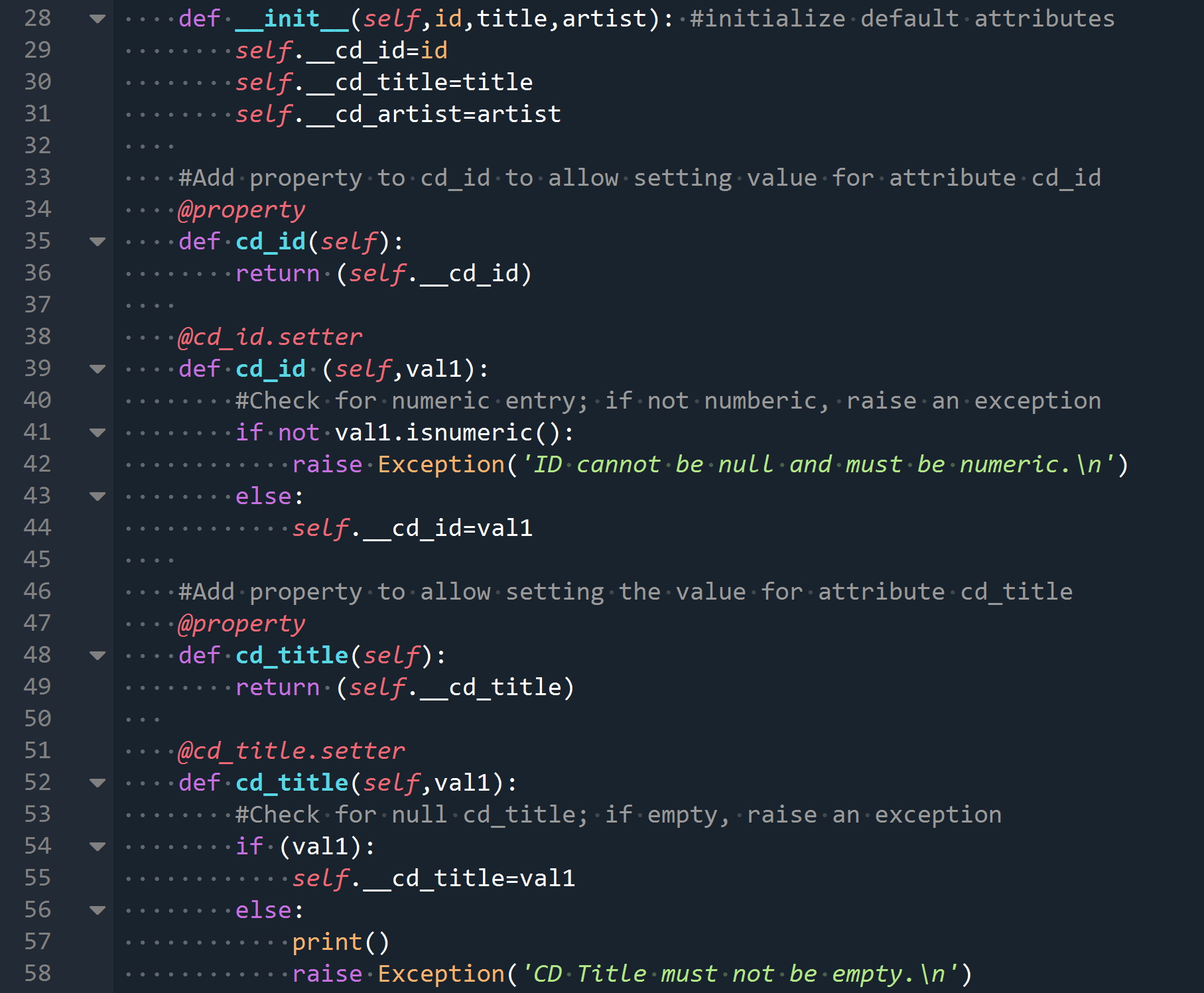


Figure 15 - class CD (CDInventory.py)

Please checkout Figure 16 - CDInventory.py in Spyder Console to view the screenshot of program in spyder console and Figure 17 - CDInventory.py in Windows Console to view it in windows console.

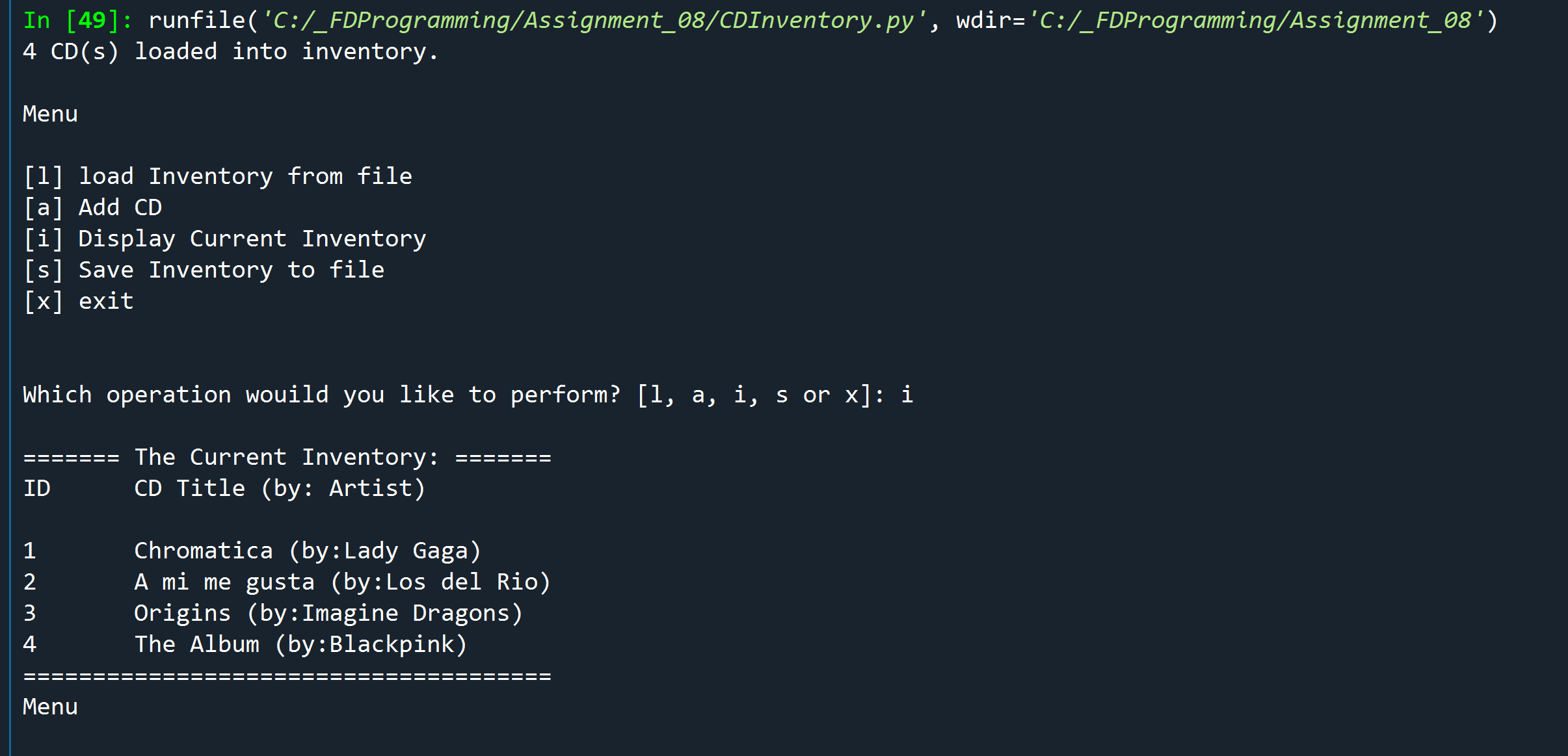


Figure 16 - CDInventory.py in Spyder Console

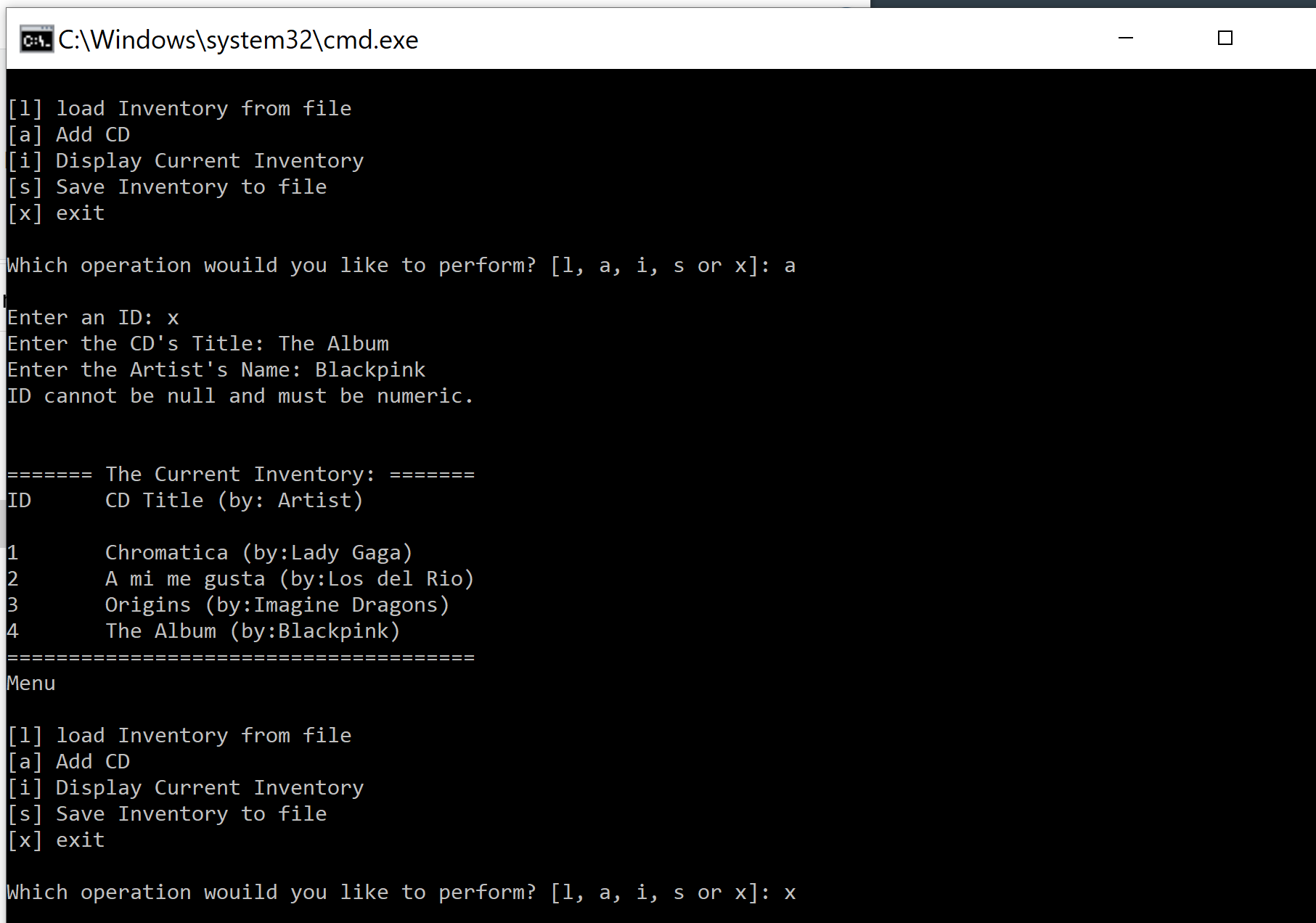


Figure 17 - CDInventory.py in Windows Console

# Summary

For module 8, I took a plunge in the world of objected oriented (or as my classmate from class said, disoriented) programming OOP. I learned how to define a class which is the blueprint of the object. I learned that I can default attribute values through the \_\_init\_\_ method (aka as constructor method) which is automatically called every time a new object is instantiated. Furthermore, I learned how to make these attributes private so that it cannot be directly accessed outside of the class. However, I can give a client indirect access to these attributes when necessary by creating getter or setter properties. I also learned how o create methods, which are essentially functions, that can be used on the objects created in the same class. I learned that I can create an attribute that is associated with the class itself called class attribute. A class attribute cannot be accessed thru an object but thru the class itself. This also means that a class attribute can be invoked even before an object is created. In the same vein, the @static method is a method defined for the class itself which means it can’t be invoked thru an object but only through its class.

I also picked up a few hints of when to implement privacy on both methods and attributes. When defining classes, create methods to reduce direct access to an object’s attributes. Implement privacy on attributes and methods that are internal to the operation of the objects. Furthermore, when using objects, one should refrain from accessing an object’s attributes as much as possible and to avoid directly modifying it as well. Lastly, a client should never attempt to directly access private attributes and methods.

Indeed, I gained a new knowledge about object-oriented programming in this module but I must admit, I am still not 100% sure when I should be using OOP when programming in python. I found this [article](https://hub.packtpub.com/python-3-when-use-object-oriented-programming/)[[9]](#footnote-9) that might give some helpful hints. It mentioned that “*Objects are things with data and behavior. If we are working with only data, we are often better off storing it in a list, set, dictionary, or some other*[*Python*](https://subscribe.packtpub.com/learn-python/)*data structure. On the other hand, if we are working with only behavior, with no stored data, a simple function is more suitable.*” But to novices like myself, it is not as straightforward as it seems so I think in the end, the ability to decipher when to implement OOP comes down, as with everything else, building experience over time.

# Appendix

1. Lab08\_E.py
2. #------------ ------------------------------#
3. # Title: Lab08\_E.py
4. # Desc: Class Demo
5. # Change Log: (Who, When, What)
6. # Maria Dacutanan, YYYY-Mon-DD, Created File
7. #-------------------------------------------#
9. #-------------------DATA--------------------#
11. **class** TrackInfo():
12. #----------------FIELDS----------------#
13. #trackDct={}
15. #------------CONSTRUCTOR--------------#
16. **def** \_\_init\_\_(self,cdtrack):
17. #--------Attributes---------#
18. self.\_\_position=cdtrack['pos']
19. self.\_\_CDtitle=cdtrack['title']
20. self.\_\_CDlen=cdtrack['cdlen']
22. @property
23. **def** position(self):
24. **return** (self.\_\_position)
26. @position.setter
27. **def** position (self,val1):
28. **if** **not** val1.isnumeric():
29. **raise** Exception('Position must be numeric.')
30. **else**:
31. self.\_\_position=val1
33. @property
34. **def** CDtitle(self):
35. **return** (self.\_\_CDtitle)
37. @CDtitle.setter
38. **def** CDtitle(self,val1):
39. **if** (val1):
40. self.\_\_CDtitle=val1
41. **else**:
42. **raise** Exception('CD Title must not be empty.')
44. @property
45. **def** CDlen(self):
46. **return** (self.\_\_CDlen)
48. @CDlen.setter
49. **def** CDlen(self, val1):
50. **if**(val1):
51. self.\_\_CDlen=val1
52. **else**:
53. **raise** Exception ('CD Length must not be empty.')
55. #------------METHODS--------------#
56. #def noAnswer(self):
57. #   return('I am an object of class TrackInfo')
59. **def** \_\_str\_\_(self):
60. **return** (f'CD Title: {self.CDtitle}')


64. #-----------------PROCESSING----------------#
66. #--------PRESENTATION (Input/Output)--------#
68. # objTrack1=TrackInfo()
69. # print ('Position is:', objTrack1.trackDct['pos'], 'CD Title is' \
70. #        , objTrack1.trackDct['title'], 'CD length is', objTrack1.trackDct['cdlen']
71. # )
72. #dictTrack=
73. objTrack1=TrackInfo({'pos':1,'title':'Chromatica','cdlen':'3:45'})
75. #dictTrack2=
76. objTrack2=TrackInfo({'pos':2,'title':'A mi me gust','cdlen':'4:35'})


80. **print** ('Position is:', objTrack1.position, 'CD Title is' \
81. , objTrack1.CDtitle, 'CD length is', objTrack1.CDlen
82. )
84. **print** ('Position is:', objTrack2.position, 'CD Title is' \
85. , objTrack2.CDtitle, 'CD length is', objTrack2.CDlen
86. )
88. **try**:
89. objTrack1.position='x'
90. **except** Exception as e:
91. **print** (e)
93. **try**:
94. objTrack1.CDtitle=''
95. **except** Exception as e:
96. **print** (e)
98. **try**:
99. objTrack1.CDlen=''
100. **except** Exception as e:
101. **print** (e)
103. **try**:
104. **print**(objTrack1.\_\_CDtitle)
105. **except** Exception as e:
106. **print**(e,\_\_doc\_\_,sep='\n')

109. **try**:
110. objTrack1.position='3'
111. **except** Exception as e:
112. **print**(e,\_\_doc\_\_,sep='\n')
114. **print**(objTrack1)
115. s=str(objTrack1)
116. **print** (s)
118. **print** ('Position is:', objTrack1.position, 'CD Title is' \
119. , objTrack1.CDtitle, 'CD length is', objTrack1.CDlen
120. )

1. Definition taken from this [website](https://hackr.io/blog/procedural-programming). Last retrieved 31-Aug-20 [↑](#footnote-ref-1)
2. List taken from this [website](https://www.edureka.co/blog/object-oriented-programming-python/#:~:text=Major%20OOP%20(object%2Doriented%20programming,idea%20of%20classes%20and%20objects.). Last retrieved 01-Sept-20 [↑](#footnote-ref-2)
3. Last retrieved 01-Sept-20 [↑](#footnote-ref-3)
4. Excerpt from this [website](https://medium.com/@manjuladube/encapsulation-abstraction-35999b0a3911#:~:text=Encapsulation%20means%20that%20the%20internal,is%20member%20functions%2Cvariables%20etc.). Last retrieved 1-Sept-20 [↑](#footnote-ref-4)
5. Snippet taken from this [website](https://python.swaroopch.com/oop.html). Last retrieved 01-Sept-20 [↑](#footnote-ref-5)
6. Quoted from page 232 of Python Programming for the Absolute Beginner, 3rd Edition textbook by Michael Dawson. [↑](#footnote-ref-6)
7. Take from page 237 of Python Programming for the Absolute Beginner, 3rd edition textbook by Michael Dawson [↑](#footnote-ref-7)
8. Last retrieved 02-Sept-2020 [↑](#footnote-ref-8)
9. Last retrieved 02-Sept-2020 [↑](#footnote-ref-9)