Sean McElmurry

2020Mar15

Foundations of Programming, Python

Assignment08

Binary Files and Structured Errors

# Introduction

Over the eighth week of the Foundation of Programming, Python class, we started our introduction into object-oriented programming by establishing a custom object class. To apply the information, we modified our script from the prior week’s assignment. The execution of the script in a terminal window can be seen in Figure 1 below.

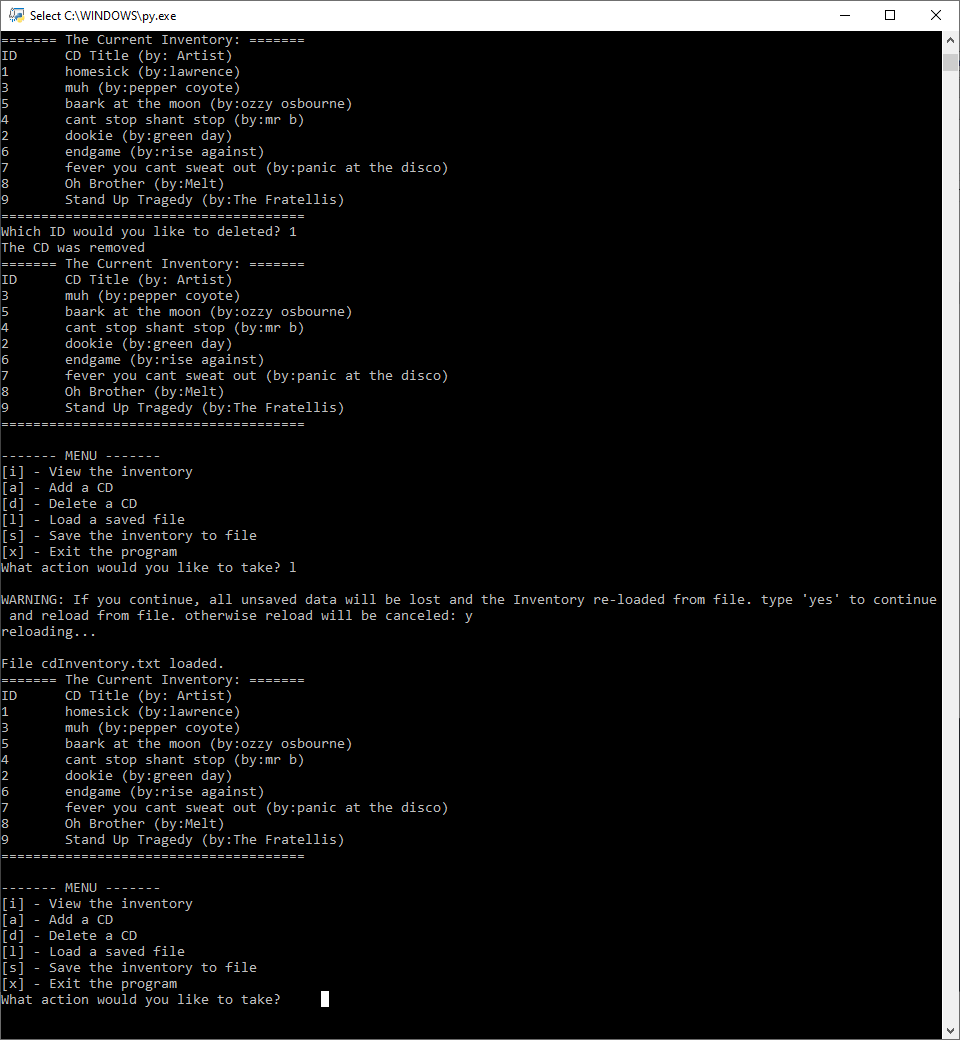
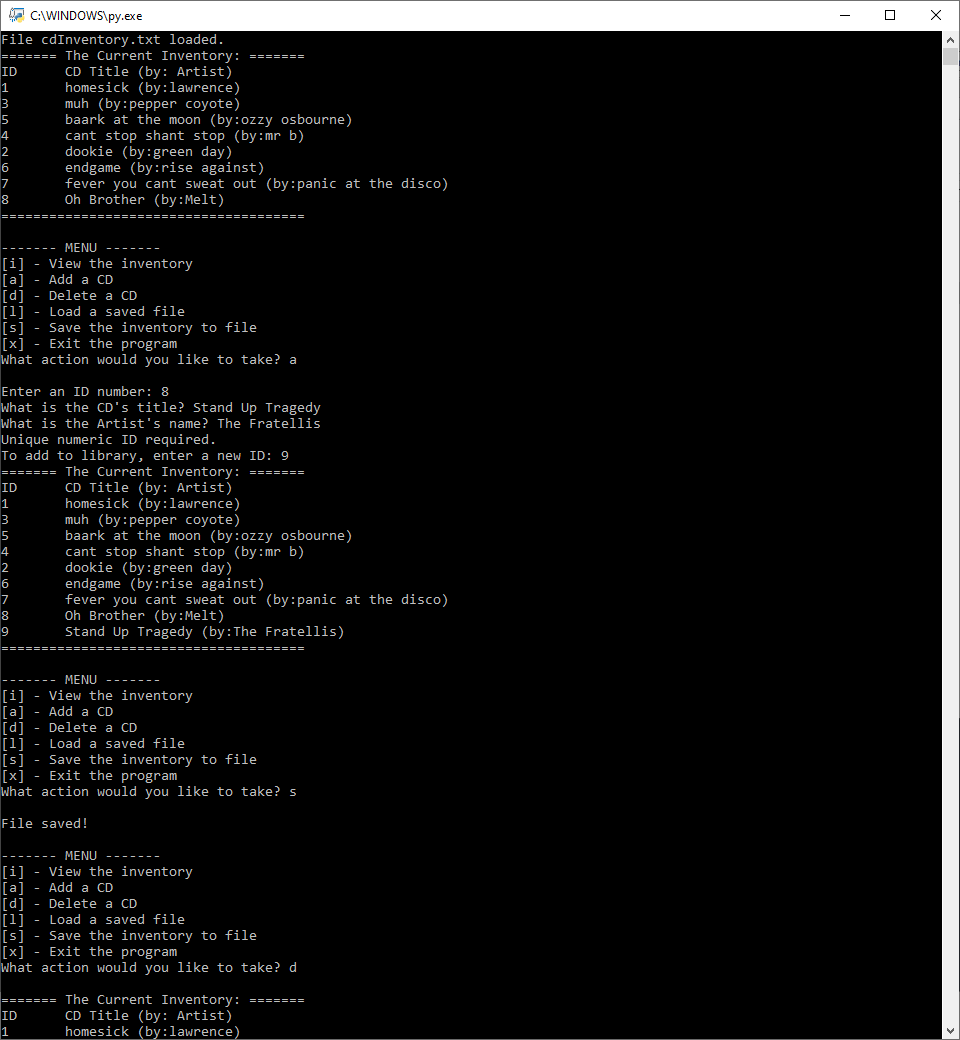


Figure - CD\_Inventory.py When Run from a Terminal Window

# Building Classes

# Object classes are a blueprint for the objects they will be creating but aren't considered the same as the object itself which holds data. To clarify, you can create an object from a class and change its data, however, that only changes the object, and not the class itself.

# Object classes are usually comprised of five parts: fields, the constructor, attributes (as part of the constructor), properties, and methods. I'm still a little confused about the difference between fields and attributes. It seems like the main difference is that attributes are meant to be private to the object class (forcing those interactions through the methods) and that attribute values are unique to each instance?

# The class constructor method is called as a object is created, and is used to help set the attributes of the object. These attributes may be made private to help limit the amount of access a user has, and force that user to access the attributes through methods. Properties may also be given to an object's attributes to fulfill a similar purpose on a smaller scale. These properties are usually divided into two types: the getter and the setter. The getter helps return the value referenced by the attribute being accessed, and the setter defines validation methods when assigning values to attributes.

# Within the constructor and some class methods, we see the 'self' keyword which is a 'pointer' that references the specific object invoking the method. Alternatively, we also have the "staticmethod" decorator. These types of methods are invoked through a class, but not an object. One way to think about the difference is that the self methods are for storing data, and static methods are for processing data.

# Docstrings should be always be included in a class for the same reason they're included in a function, to inform the user what they are, and what they contain.

# Writing the Application Script

I spent the most time this week setting up the CD Object class. It took me way too long to figure out what I needed to put in there. Trying to sort out the difference between fields and attributes, how to set up properties, and trying to figure out what we were supposed to be doing for the assignment were all very confusing. The full script can be seen by following the GitHub link in the appendix; and its execution in the Spyder terminal window can be seen in Figure 2 below.

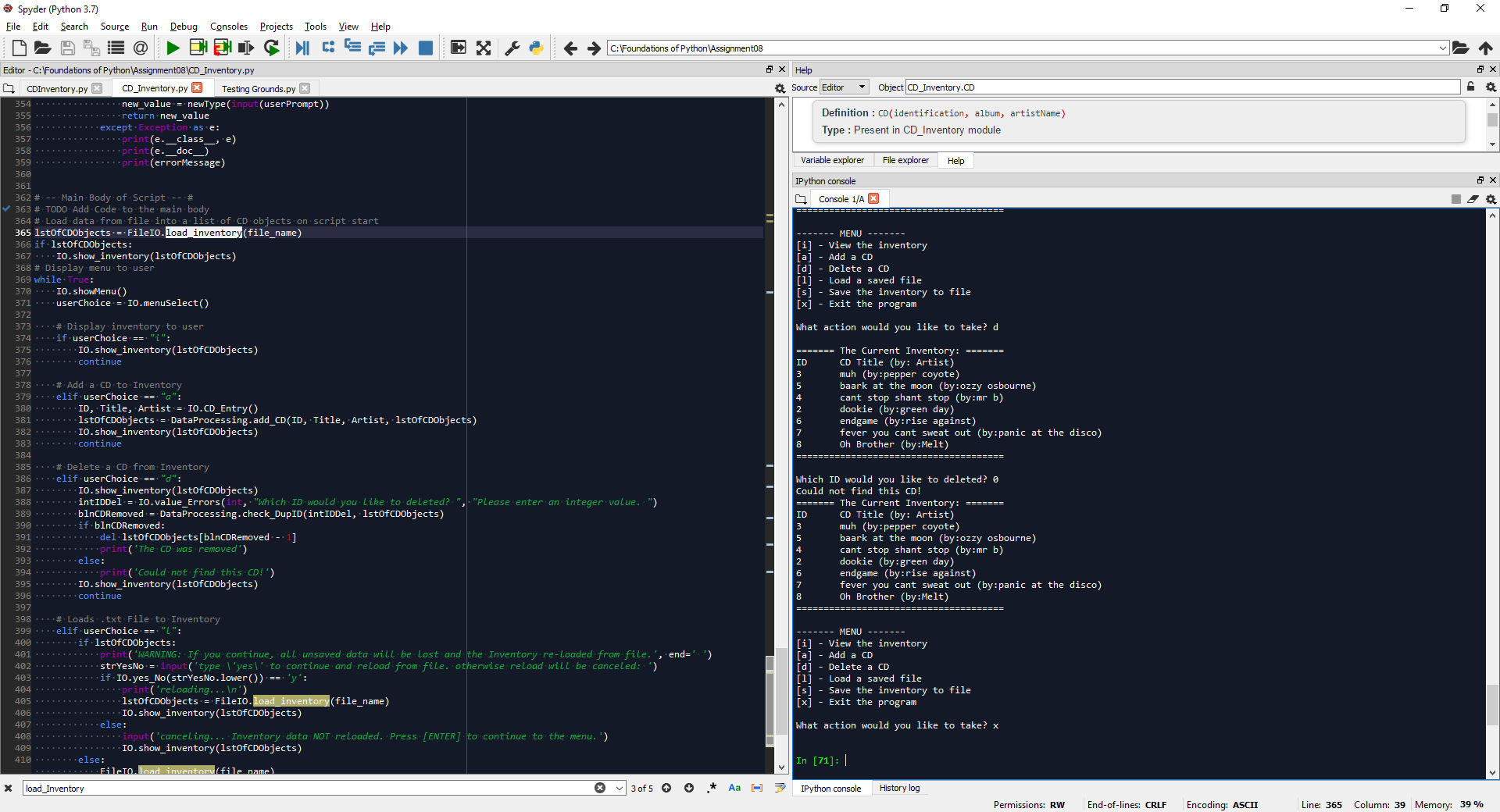
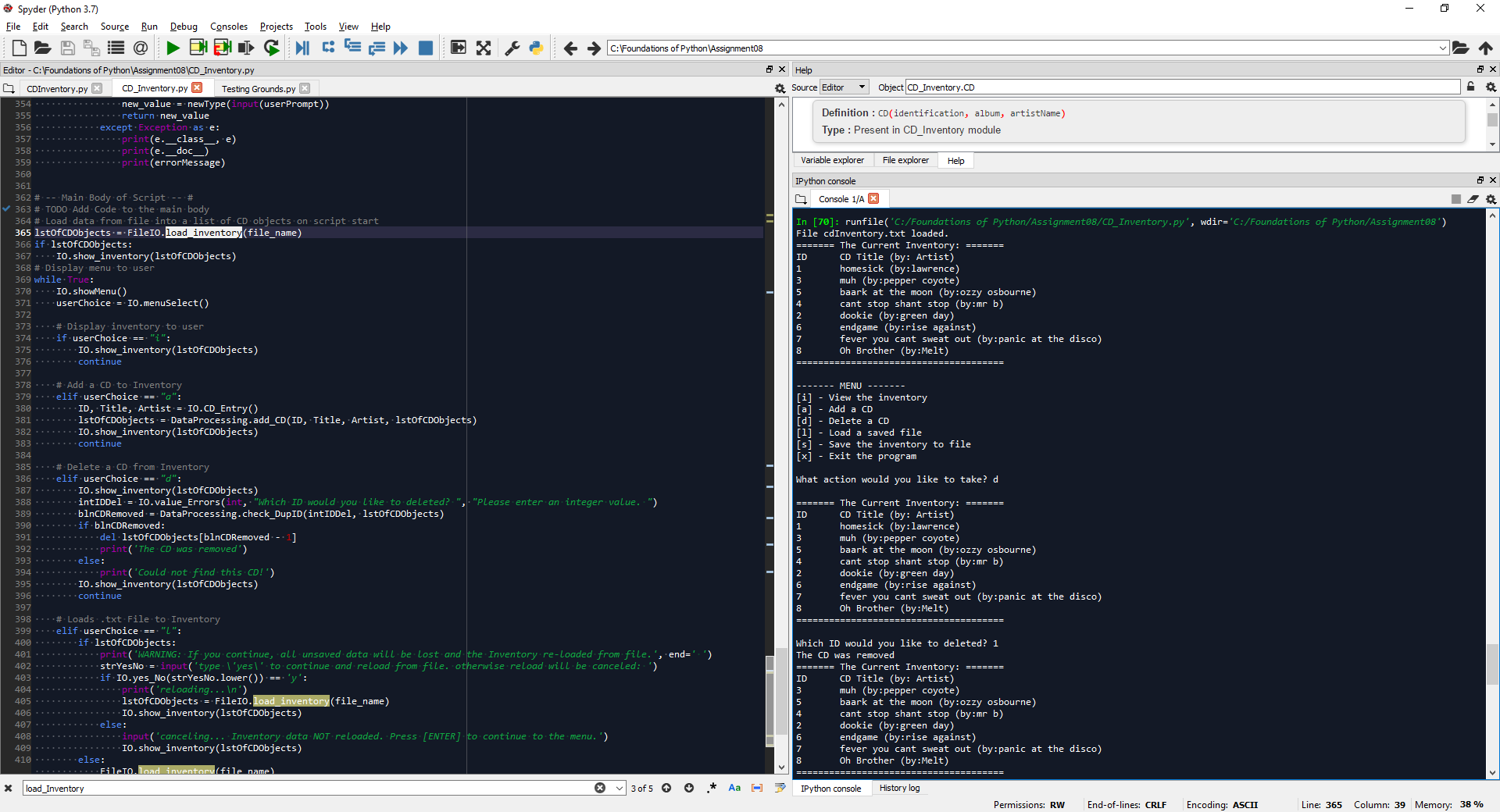


Figure - CD\_Inventory.py Run in Spyder to test delete functionality

## CD Object Class

After wrestling with whether or not to use fields in my CD class, I decided against it based on the example from the class book that used a field (total) to track the entire number of Critter objects they had created (Foundations of Programming, Dawson). After this example they seemed out of place, even contrary to other examples online that treated fields and attributes almost identically.

Setting up the constructor seemed simple enough. I couldn’t tell much difference when setting up the attributes as private or public, or how important it was to this assignment, but it seemed that good practices dictate making attributes private and establishing the properties so that they could be accessed in a controlled manner.

I wanted to make sure that I could “view” an object’s contents easily, and decided the best way to do that would be to set up a dunder-string function that would return all attributes (separated by a comma) as a string. This also set the code up for saving the CD objects to a .txt file later.

Part of my confusion stemmed from an issue I had testing the setter functions where upon initialization, an attribute could be anything. When trying to set the attribute (cd\_id) again as a string, the class would function as it should (with the exception of the initial set-up), but when trying to set the attribute as an accepted int type, absolutely nothing would happen as displayed in Figures 3 and 4 below.

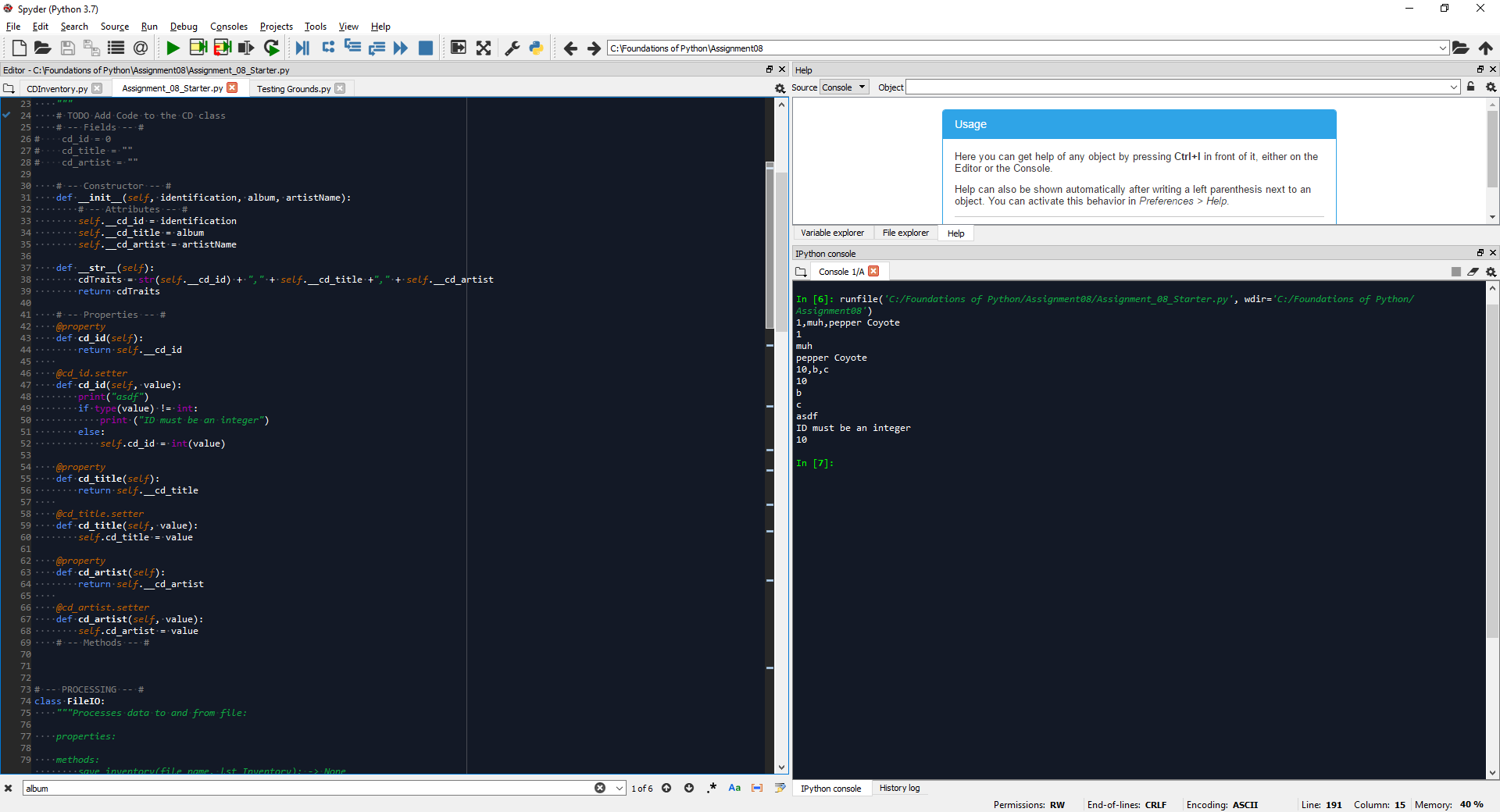
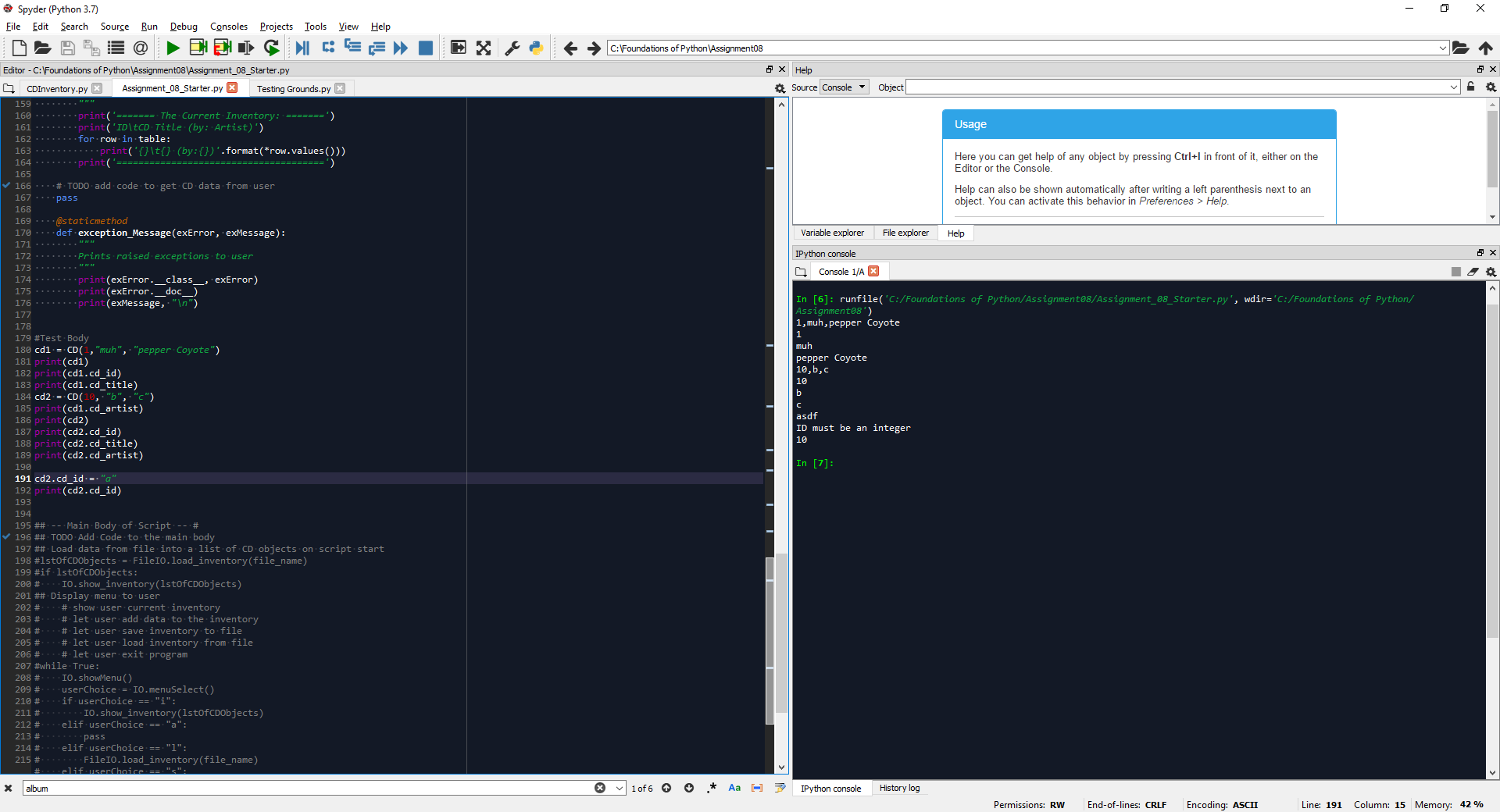
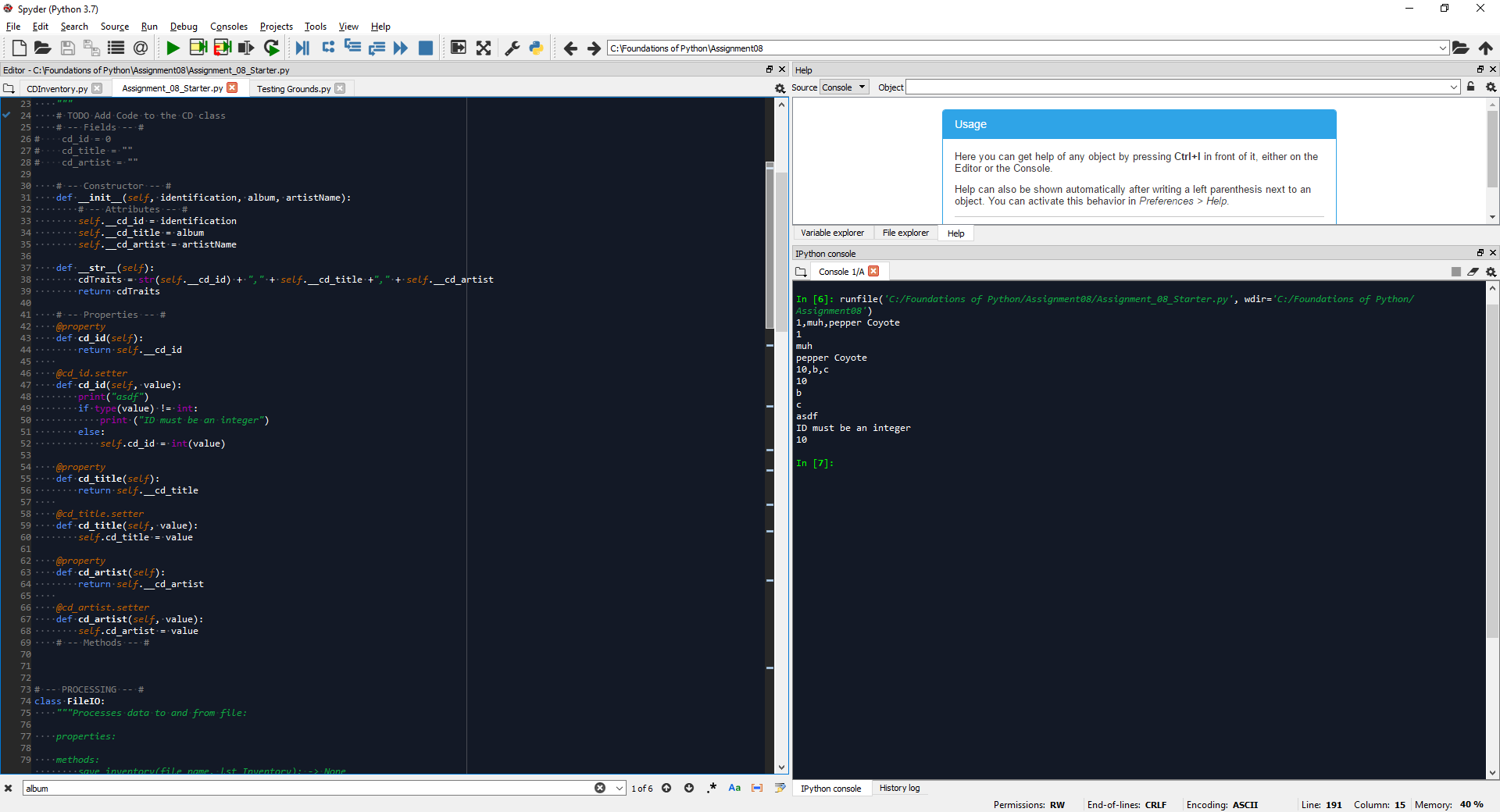


Figure - Initial Testing of Assigning a String to Int Attribute

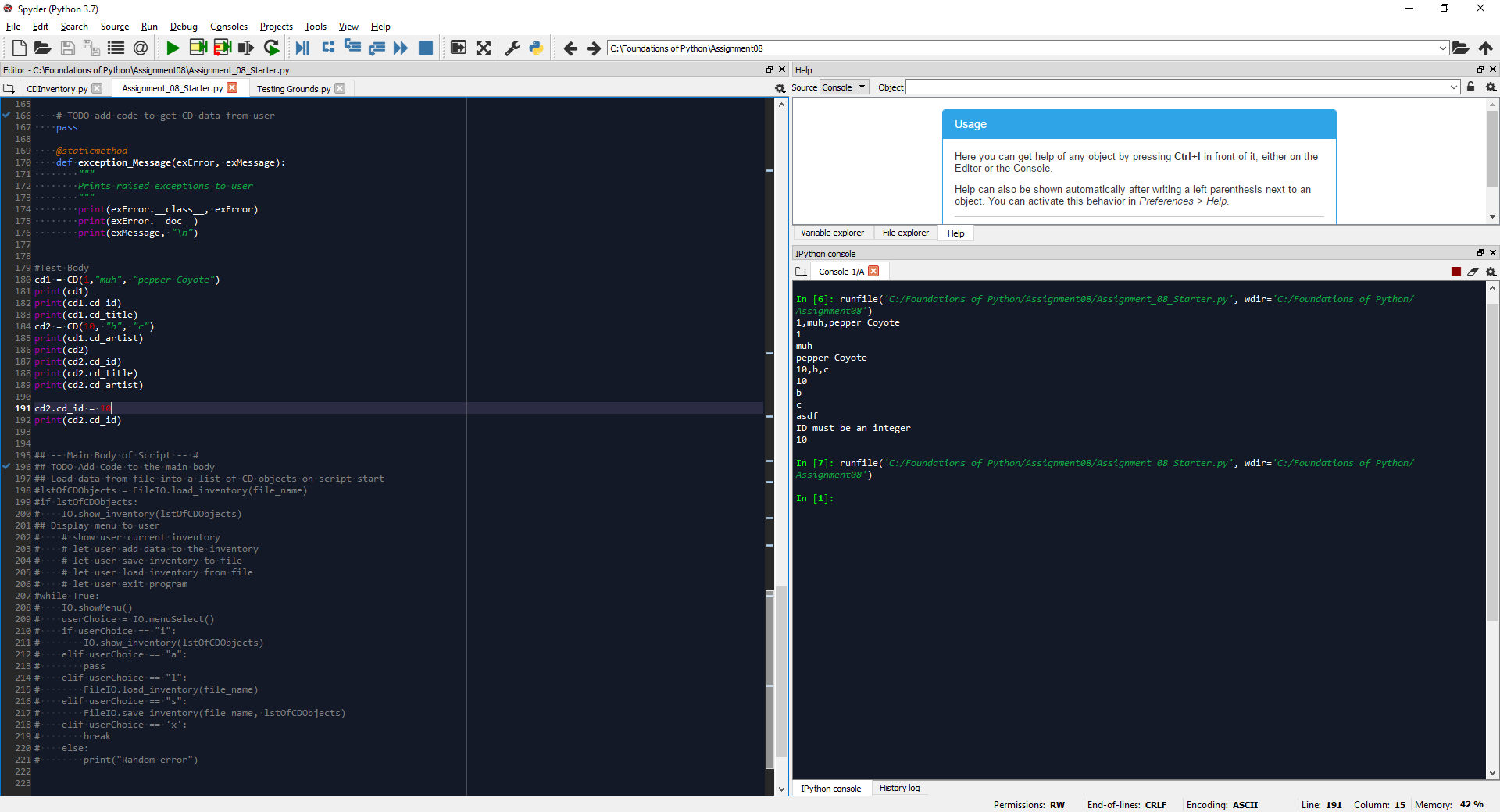
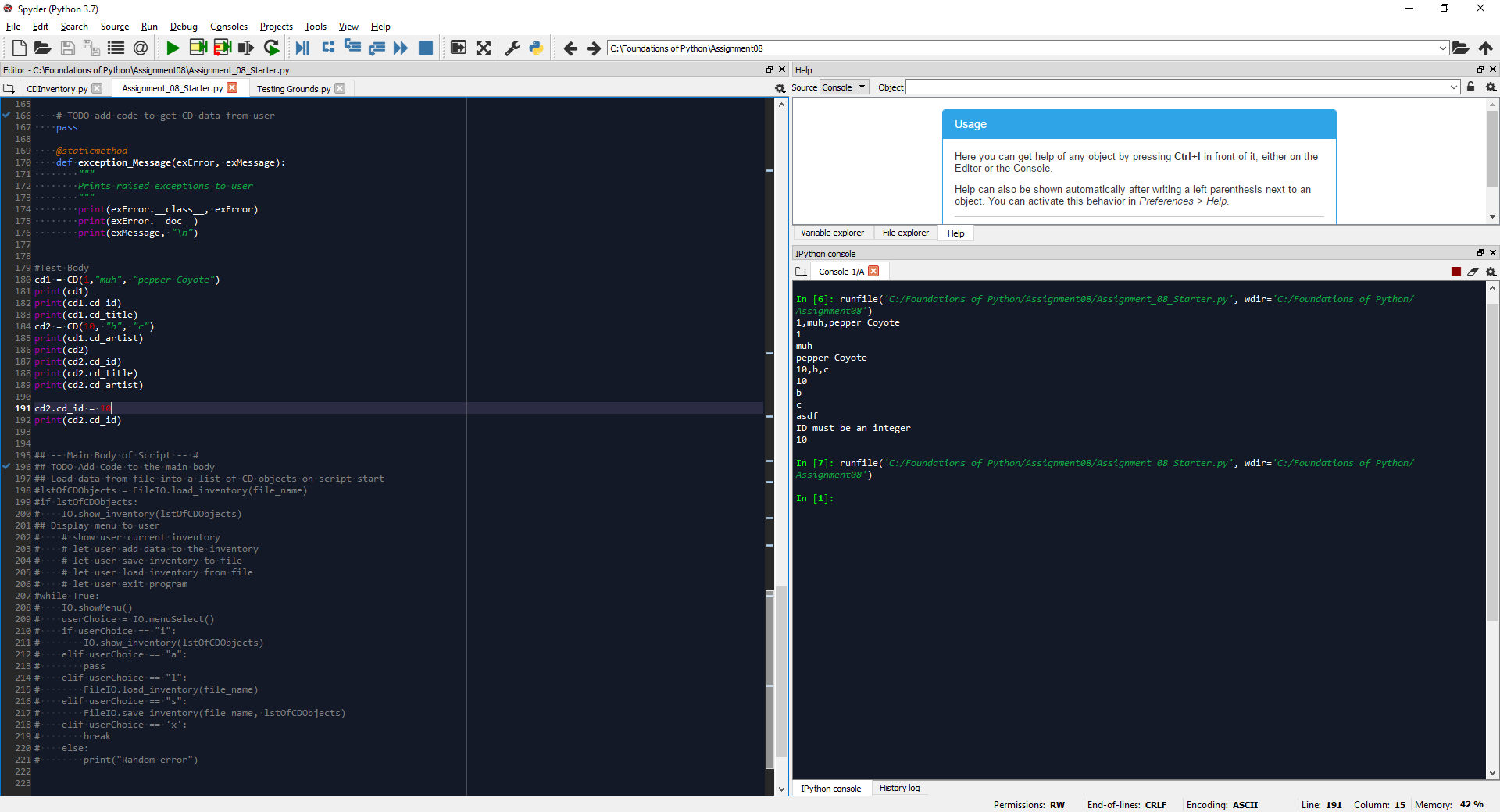


Figure - Initial Testing of Assigning a Int to Int Attribute

The only notable difference between the class script above when I initially started and what I settled on was moving the *\_\_str\_\_(*self*,)* function from between the constructor and properties to the methods section. Does this mean that for the future, nothing should be placed between the constructor and those properties? And if so, why did the script still work when triggering an exception, but appeared to run nothing when assigning a correct integer value to cd2.cd\_id?

Once that issue was resolved, there wasn’t much left to do to set up the class. Really, the largest thing was to update references to objects within the governing list from dictionary keys and values, to accessing specific attributes in CDs.

## Other Edits

Once I had the class interactions established, I could start working on improvements from the week prior. This mostly meant two things, removing the cross-class interactions, and removing global variable calls from wherever possible. I tried to keep everything within the already established classes, but it became clear that many functions I had written in the IO class, didn’t fit there, and didn’t even have a presentation portion. So I created the DataProcessing class to host these functions. It did feel like writing extra lines at time, making code where there didn’t need to be. But In the end, it did end up making the script as a whole much easier to read since it was organized according to SoC. As for removing global variables, I didn’t have to add extra lines so much as make sure that the table was being passed as a parameter into the function when it was called.

The last big edit I had made dawned on me as I was writing the docstrings for everything: that *find\_Row()­* and *check\_DupID()* were the same function. They both returned a Boolean, and *check\_DupID()* even called *find\_Row()*. The only real difference was that *find\_Row()* also contained three lines to delete a function based on the menu choice. Once I combined the two functions, I wanted to remove the last call to the global list that I could, so I changed the return type to an int-type I could overload as a Boolean in other functions. That way, I could still use the function as a Boolean when called in *add\_CD()* and as a Boolean/Int when deleting.

# Summary

By far this was the most challenging module to me. I think it would be helpful to see more examples of setting up different object classes, as most of the examples we referred to either in videos or the class book had different structures to their classes, and because of that created a muddled explanation of the differences between object class components.

# Appendix

## Sources

“Foundations of Programming (Python) Module 03 Part 4.” YouTube, 2020Jan07, youtu.be/Nc4spVYdai8. Accessed 2020Feb07.

## GitHub Repository

<https://github.com/SMcElmo/Assignment_08>