**Docstrings**

**2. A complex function**

Look at this split\_and\_stack() function. If you wanted to understand what the function does, what the arguments are supposed to be, and what it returns, you would have to spend some time deciphering the code.

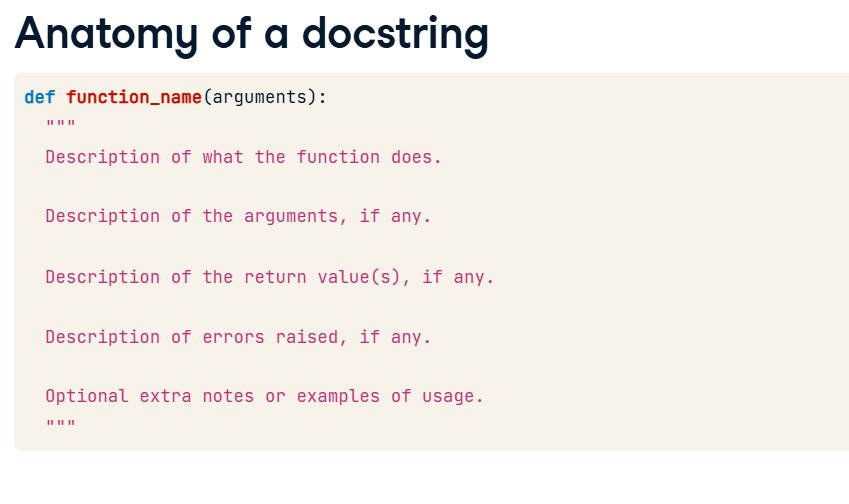
**3. A complex function with a docstring**

00:31 - 00:44

With a docstring though, it is much easier to tell what the expected inputs and outputs should be, as well as what the function does. This makes it easier for you and other engineers to use your code in the future.

**4. Anatomy of a docstring**

A docstring is a string written as the first line of a function. Because docstrings usually span multiple lines, they are enclosed in triple quotes, Python's way of writing multi-line strings. Every docstring has some (although usually not all) of these five key pieces of information: what the function does, what the arguments are, what the return value or values should be, info about any errors raised, and anything else you'd like to say about the function.



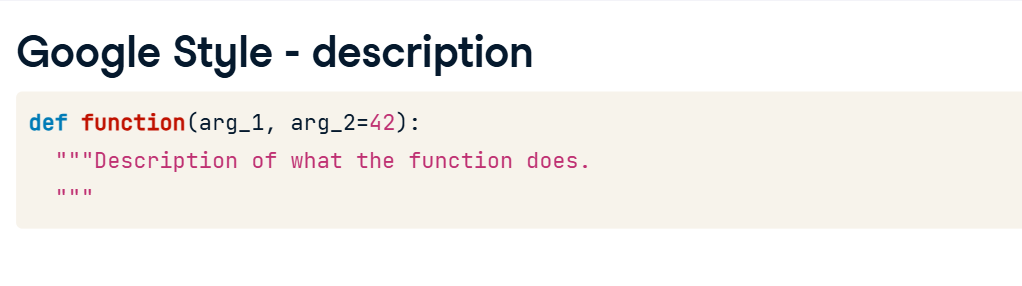
**5. Docstring formats**

01:15 - 01:30

Consistent style makes a project easier to read, and the Python community has evolved several standards for how to format your docstrings. Google-style and Numpydoc are the most popular formats, so we'll focus on those.

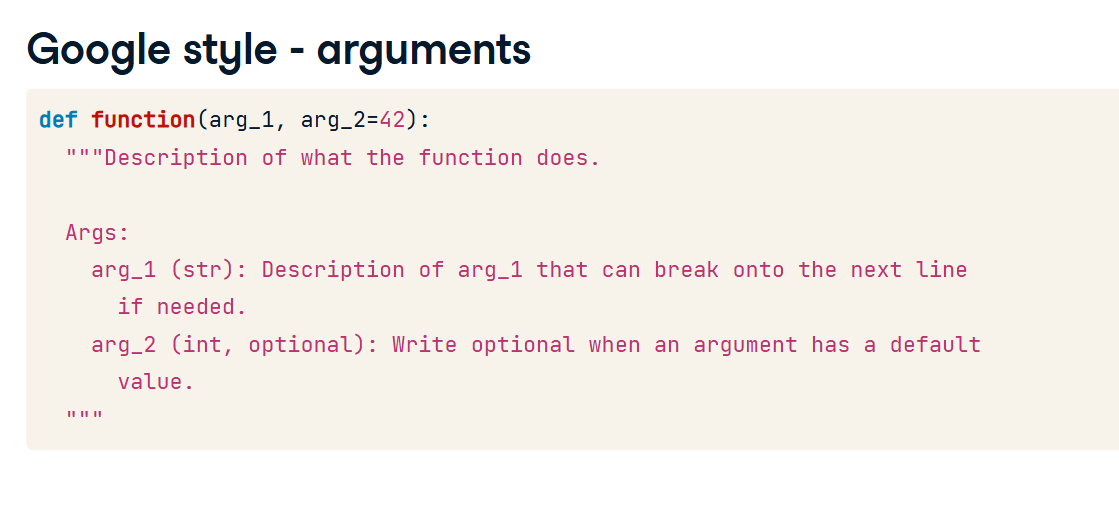
**Google Style - description**

In Google style, the docstring starts with a concise description of what the function does. This should be in imperative language. For instance: "Split the data frame and stack the columns" instead of "This function will split the data frame and stack the columns".



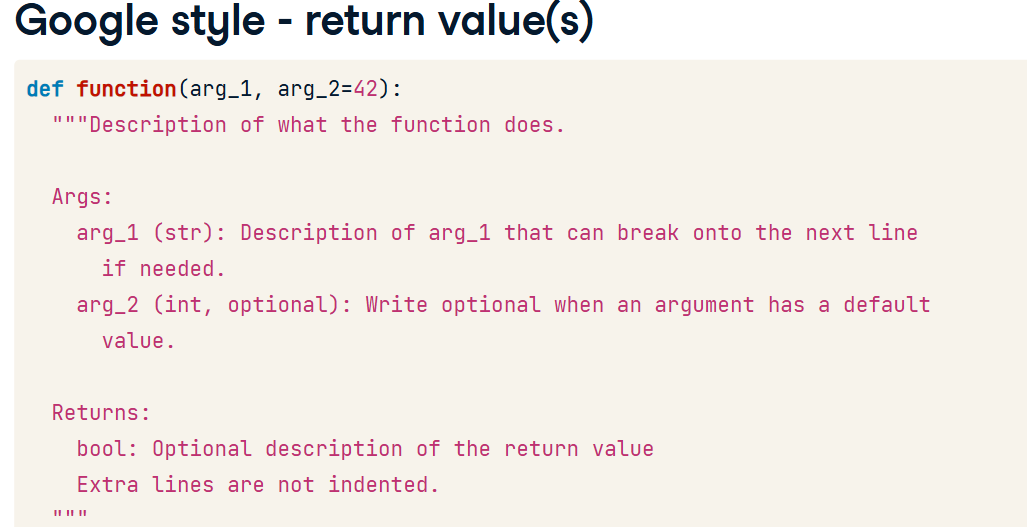
**Google style - arguments**

Next comes the "Args" section where you list each argument name, followed by its expected type in parentheses, and then what its role is in the function. If you need extra space, you can break to the next line and indent as I've done here. If an argument has a default value, mark it as "optional" when describing the type. If the function does not take any parameters, feel free to leave this section out.



**8. Google style - return value(s)**

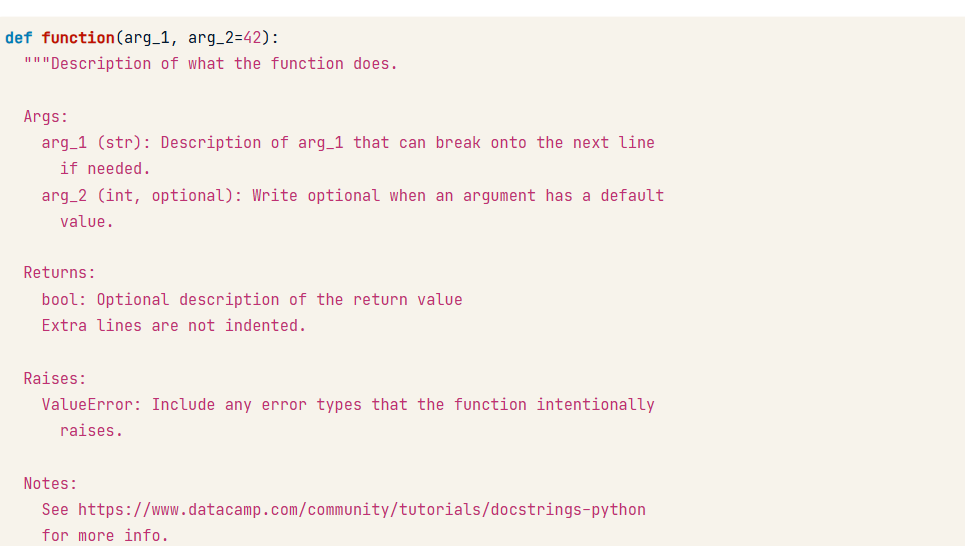
02:15 - 02:33

The next section is the "Returns" section, where you list the expected type or types of what gets returned. You can also provide some comment about what gets returned, but often the name of the function and the description will make this clear. Additional lines should not be indented.  


**9. Google-style - errors raised and extra notes**

02:33 - 02:46

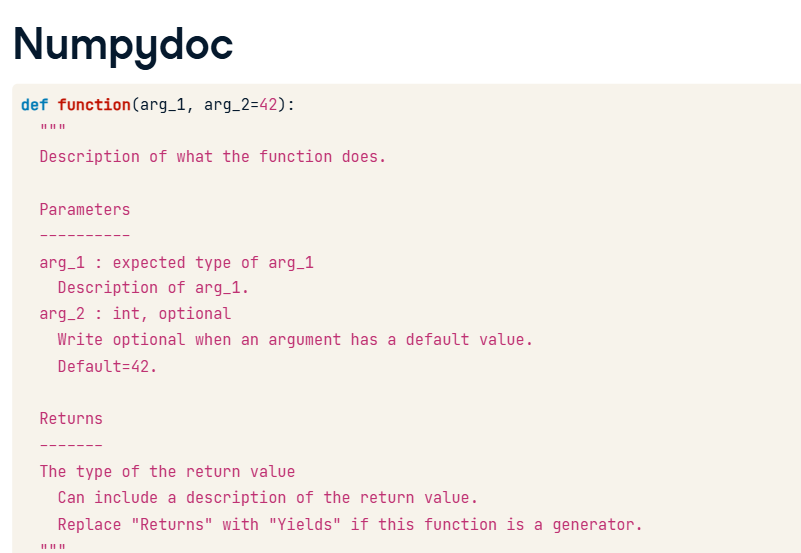
Finally, if your function intentionally raises any errors, you should add a "Raises" section. You can also include any additional notes or examples of usage in free form text at the end.



**10. Numpydoc**

02:46 - 03:06

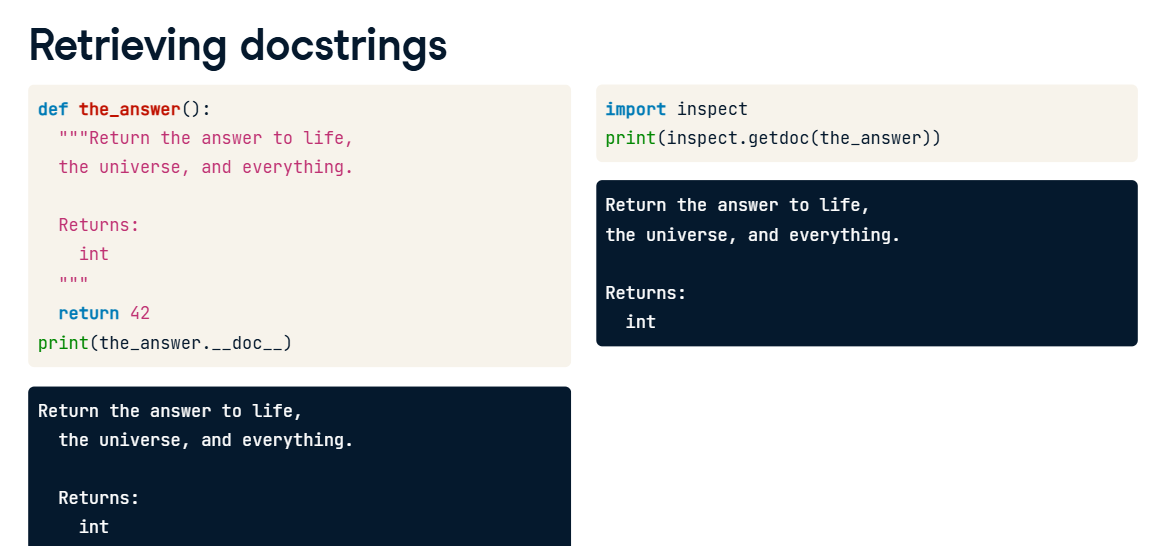
The Numpydoc format is very similar and is the most common format in the scientific Python community. Personally, I think it looks better than the Google style. It takes up more vertical space though, so this course will either use Google-style or leave out the docstrings entirely to keep the examples compact and legible.



**11. Retrieving docstrings**

03:06 - 03:46

Sometimes it is useful for your code to access the contents of your function's docstring. Every function in Python comes with a \_\_doc\_\_ attribute that holds this information. Notice that the \_\_doc\_\_ attribute contains the raw docstring, including any tabs or spaces that were added to make the words line up visually. To get a cleaner version, with those leading spaces removed, you can use the getdoc() function from the inspect module. The inspect module contains a lot of useful methods for gathering information about functions.



**DRY and "Do One Thing"**

DRY (also known as "don't repeat yourself") and the "Do One Thing" principle are good ways to ensure that your functions are well designed and easy to test. Let's see how.

**Don't repeat yourself (DRY)**

When you are writing code to look for answers to a research question, it is totally normal to copy and paste a bit of code, tweak it slightly, and re-run it. However, this kind of repeated code can lead to real problems. In this code snippet, I load my train, validation, and test data, and plot the first two principal components of each dataset. I wrote the code for the train dataset, then copied it and pasted it into the next two blocks, updating the paths and the variable names.

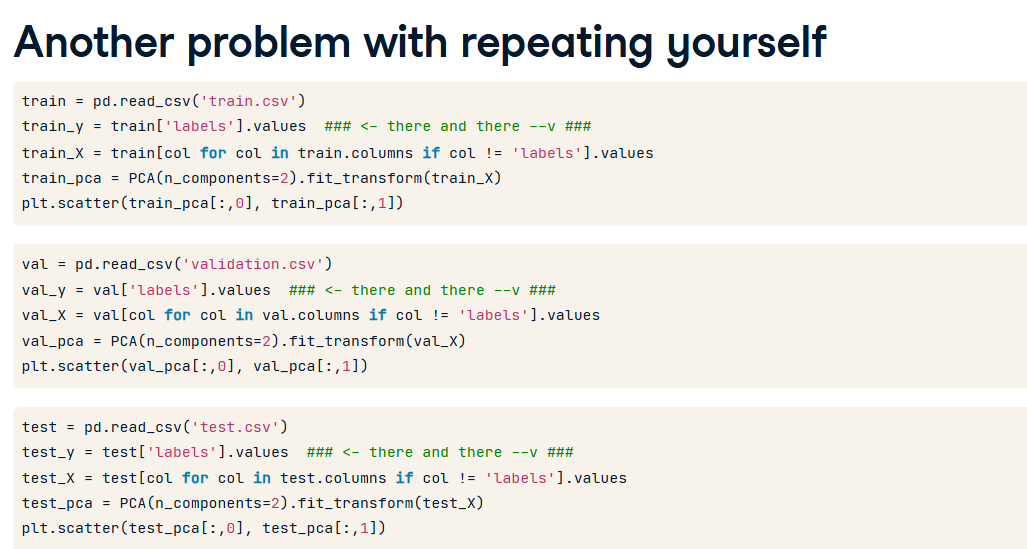
**3. The problem with repeating yourself**

But one of the problems with copying and pasting is that it is easy to accidentally introduce errors that are hard to spot. If you'll notice in the last block, I accidentally took the principal components of the train data instead of the test data. Yikes!

**4. Another problem with repeating yourself**

00:57 - 01:20

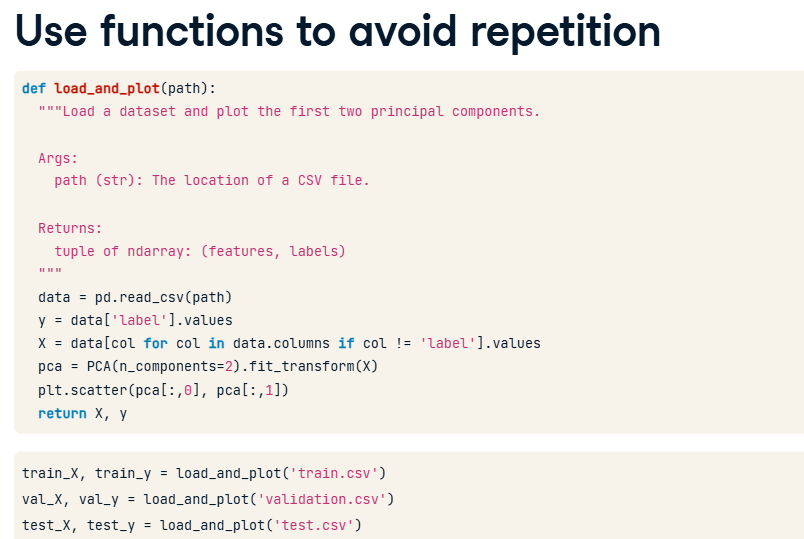
Another problem with repeated code is that if you want to change something, you often have to do it in multiple places. For instance, if we realized that our CSVs used the column name "label" instead of "labels", we would have to change our code in six places. Repeated code like this is a good sign that you should write a function. So let's do that.



**5. Use functions to avoid repetition**

01:20 - 01:44

Wrapping the repeated logic in a function and then calling that function several times makes it much easier to avoid the kind of errors introduced by copying and pasting. And if you ever need to change the column "label" back to "labels", or you want to swap out PCA for some other dimensionality reduction technique, you only have to do it in one or two places.



**6. Problem: it does multiple things**

However, there is still a big problem with this function.

First, it loads the data.

Then it plots the data.

And then it returns the loaded data. This function violates another software engineering principle: Do One Thing. Every function should have a single responsibility. Let's look at how we could split this one up.

**Do One Thing**

Instead of one big function, we could have a more nimble function that just loads the data and a second one for plotting. We get several advantages from splitting the load\_and\_plot() function into two smaller functions. First of all, our code has become more flexible. Imagine that later on in your script, you just want to load the data and not plot it. That's easy now with the load\_data() function. Likewise, if you wanted to do some transformation to the data before plotting, you can do the transformation and then call the plot\_data() function. We have decoupled the loading functionality from the plotting functionality.



**Advantages of doing one thing**

02:46 - 03:07

The code will also be easier for other developers to understand, and it will be more pleasant to test and debug. Finally, if you ever need to update your code, functions that each have a single responsibility make it easier to predict how changes in one place will affect the rest of the code.

**12. Code smells and refactoring**

03:07 - 03:26

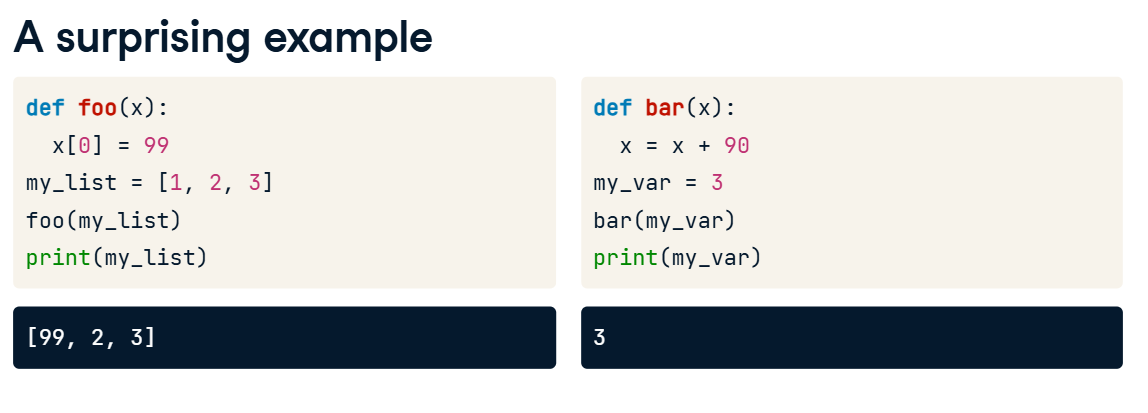
Repeated code and functions that do more than one thing are examples of "code smells", which are indications that you may need to refactor. Refactoring is the process of improving code by changing it a little bit at a time. This process is well described in Martin Fowler's book, "Refactoring", which is a good read for any aspiring software engineer.

**Pass by assignment**

The way that Python passes information to functions is different from many other languages. It is referred to as "pass by assignment", which I will explain in this lesson.

**2. A surprising example**

Let's say we have a function foo() that takes a list and sets the first value of the list to 99. Then we set "my\_list" to the value [1, 2, 3] and pass it to foo(). What do you expect the value of "my\_list" to be after calling foo()? If you said "[99, 2, 3]", then you are right**. Lists in Python are mutable objects, meaning that they can be changed**. Now let's say we have another function bar() that takes an argument and adds ninety to it. Then we assign the value 3 to the variable "my\_var" and call bar() with "my\_var" as the argument. What do you expect the value of "my\_var" to be after we've called bar()? If you said "3", you're right. In **Python, integers are immutable, meaning they can't be changed.**



**3. Digging deeper**

01:10 - 01:18

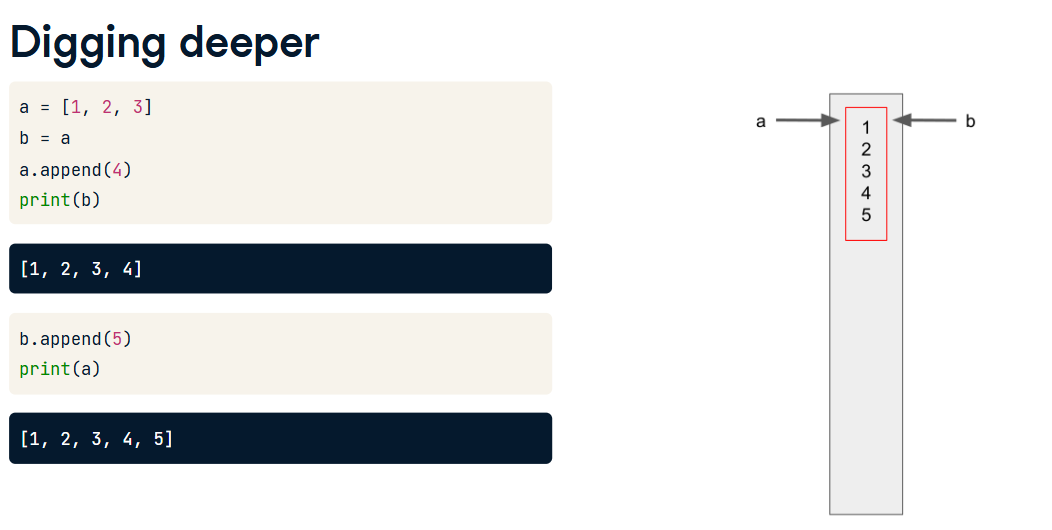
Let's look at another example to understand what's going on. Imagine that this gray bar is your computer's memory.

When we set the variable "a" equal to the list [1, 2, 3], the Python interpreter says, "Okay, now 'a' points to this location in memory."

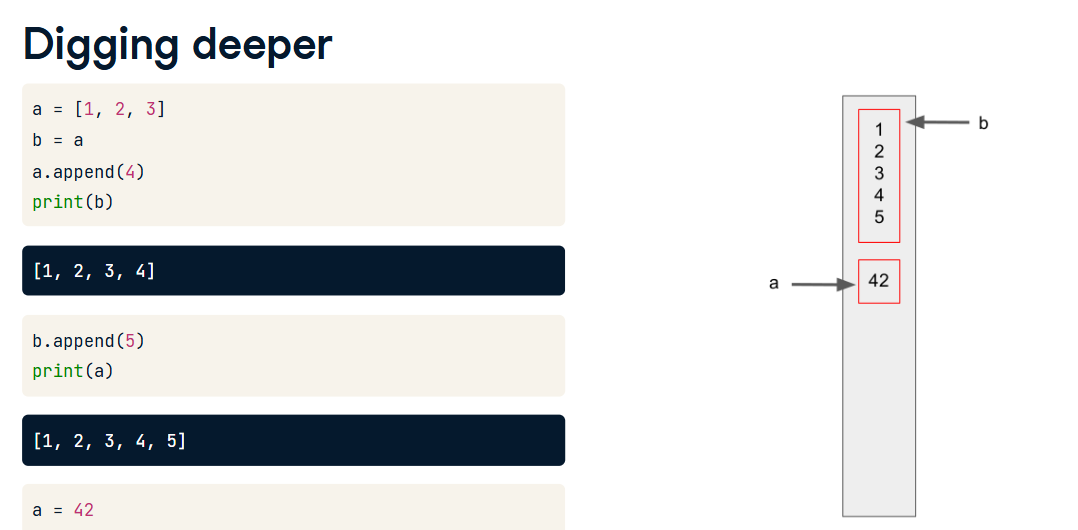
Then if we type "b = a", the interpreter says, "Okay, now 'b' points to whatever 'a' is pointing to."

So if we were to append 4 to the end of "a", both variables get it because there is only one list.

Likewise, if we append 5 to "b", both variables get it.



However, if we assign "a" to a different object in memory, that does not change where "b" is pointing. Now, things that happen to "a" are no longer happening to "b", and vice versa.



**9. Pass by assignment**

02:06 - 02:11

How does this relate to the example functions we saw earlier?

When we assign a list to the variable "my\_list", it sets up a location in memory for it.

Then, when we pass "my\_list" to the function foo(), the parameter "x" gets assigned to that same location.

So when the function modifies the thing that "x" points to, it is also modifying the thing that "my\_list" points to.



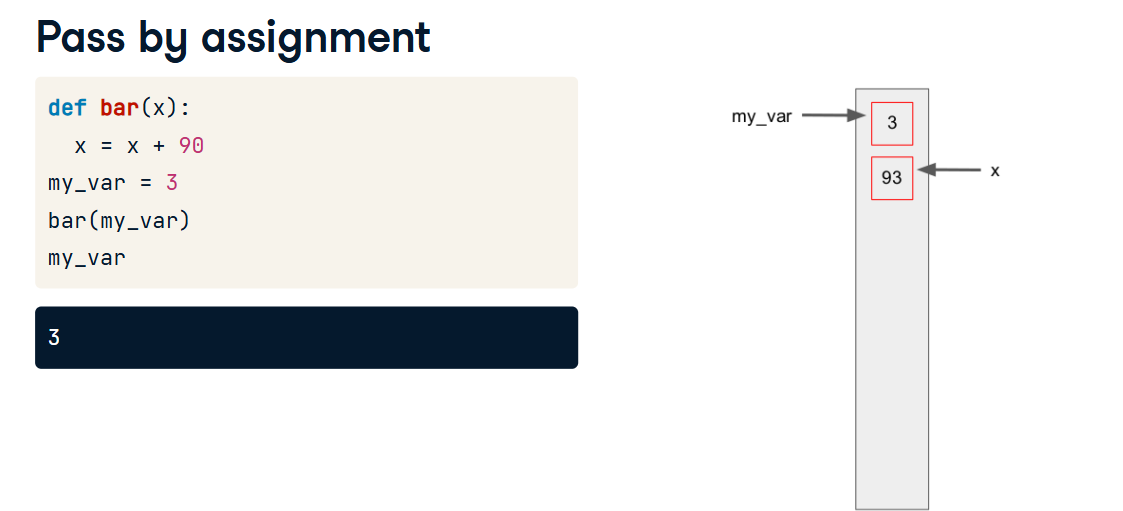
**13. Pass by assignment**

In the other example, we created a variable "my\_var" and assigned it the value 3.

Then we passed it to the function bar(), which caused the argument "x" to point to the same place "my\_var" is pointing.

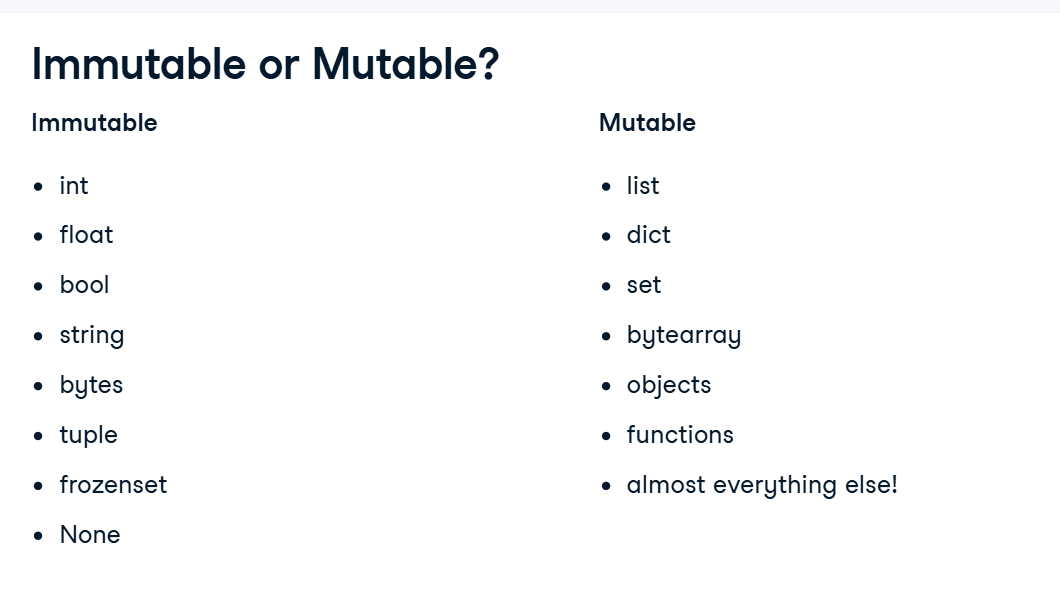
**15. Pass by assignment**

But the bar() function assigns "x" to a new value, so the "my\_var" variable isn't touched. In fact, there is no way in Python to have changed "x" or "my\_var" directly, because integers are immutable variables.



**16. Immutable or Mutable?**

There are only a few immutable data types in Python because almost everything is represented as an object. The only way to tell if something is mutable is to see if there is a function or method that will change the object without assigning it to a new variable.



**17. Mutable default arguments are dangerous!**

Finally, here is a thing that can get you into trouble. foo() is a function that appends the value 1 to the end of a list. But, whoever wrote this function gave the argument an empty list as a default value. When we call foo() the first time, we get what you would expect, a list with one entry. But, when we call foo() again, the default value has already been modified! If you really want a mutable variable as a default value, consider defaulting to None and setting the argument in the function.

