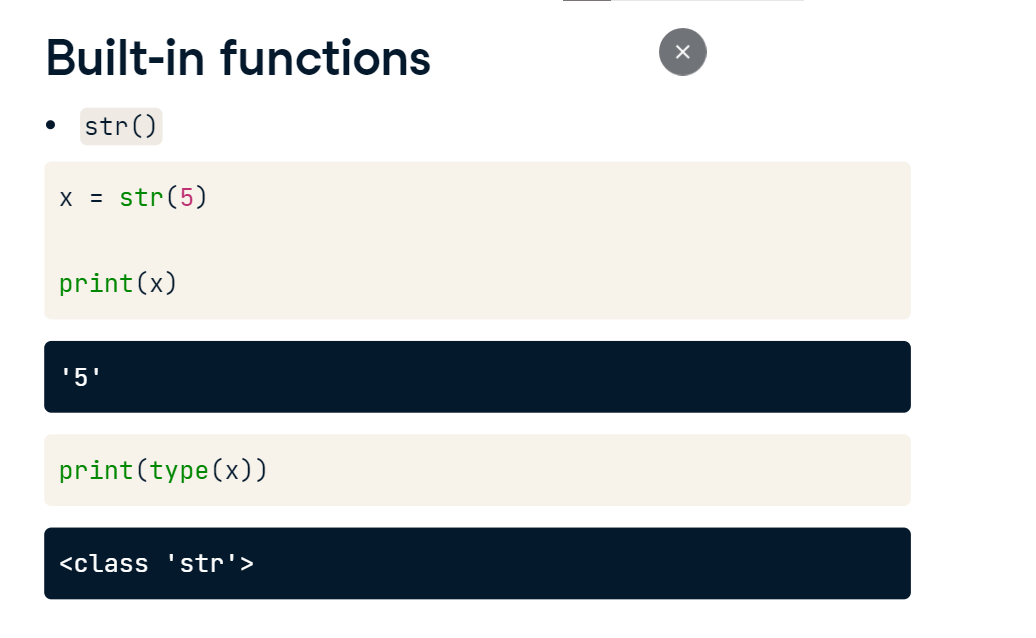
**User-defined functions**

**What you will learn:**

Specifically, in this video and in the interactive exercises that follow it, you will learn to do the following: define functions without parameters, define functions with single parameters, and define functions that return a single value. In the next section, you'll learn how to pass multiple arguments to functions, as well as return multiple values from them. Let's begin!

**Built-in functions**

Let's check out Python's built-in function str, which accepts an object such as a number and returns a string object. You can assign a call to str to a variable to store its return value. While built-in Python functions are cool, as a Data Scientist, you'll need functions that have functionality specific to your needs. Fortunately, you can define your own functions in Python!

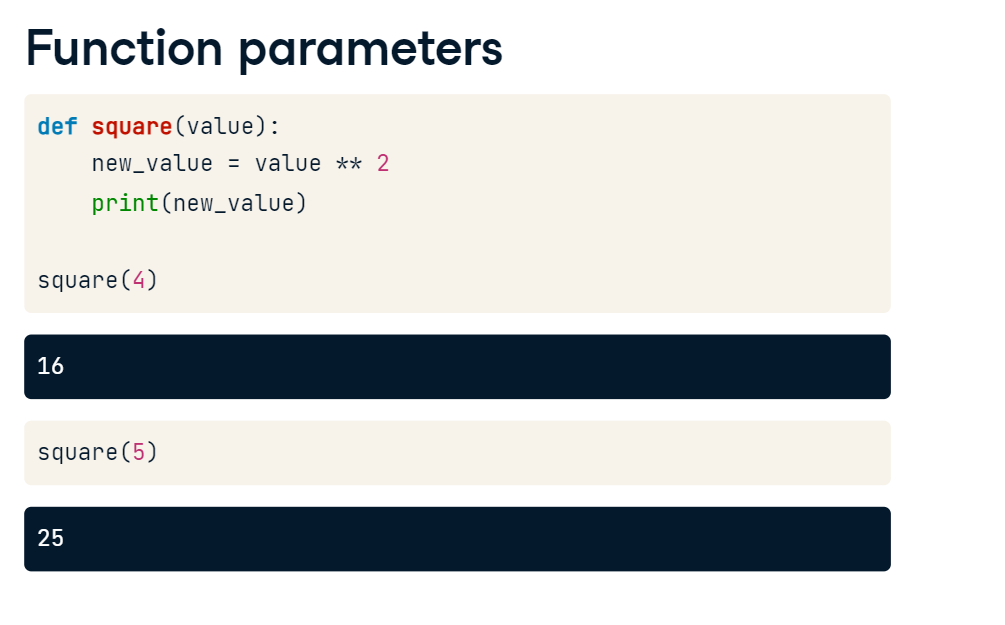


**Defining a function**

We'll now see how to define functions via an example, a function that squares a number. The function name square will be perfect for this. To define the function, We begin with the keyword def, followed by the function name square; this is then followed by a set of parentheses and a colon. This piece of code is called a function header. To complete the function definition, let's write the function body by squaring a value, say 4, and printing the output. Right now, our square function does not have any parameters within the parentheses. We will add them later. Now, whenever this function is called, the code in the function body is run. In this case, new\_value is assigned the value of 4 \*\* 2 and then printed out. You can call the function as you do with pre-built functions: square. This should yield the value, 16.

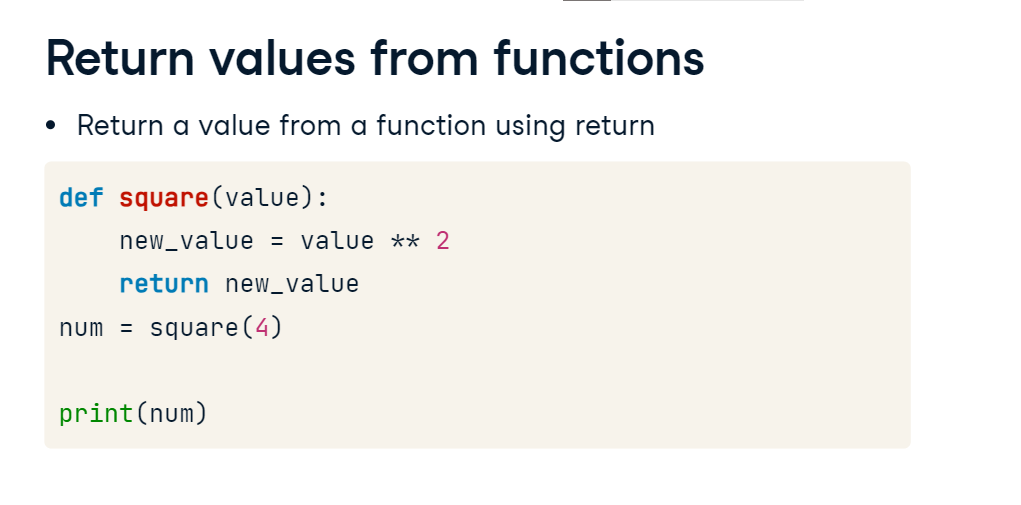
**Function parameters**

What if you wanted to square any other number besides 4, though? To add that functionality, you add a parameter to the function definition in between the parentheses. Here you see that we've added a parameter value and in the new function body, the new variable new\_value takes the square of value, which is then printed out. We can now square any number that we pass to the function square as an argument. A quick word on parameters and arguments: when you define a function, you write parameters in the function header. When you call a function, you pass arguments into the function.



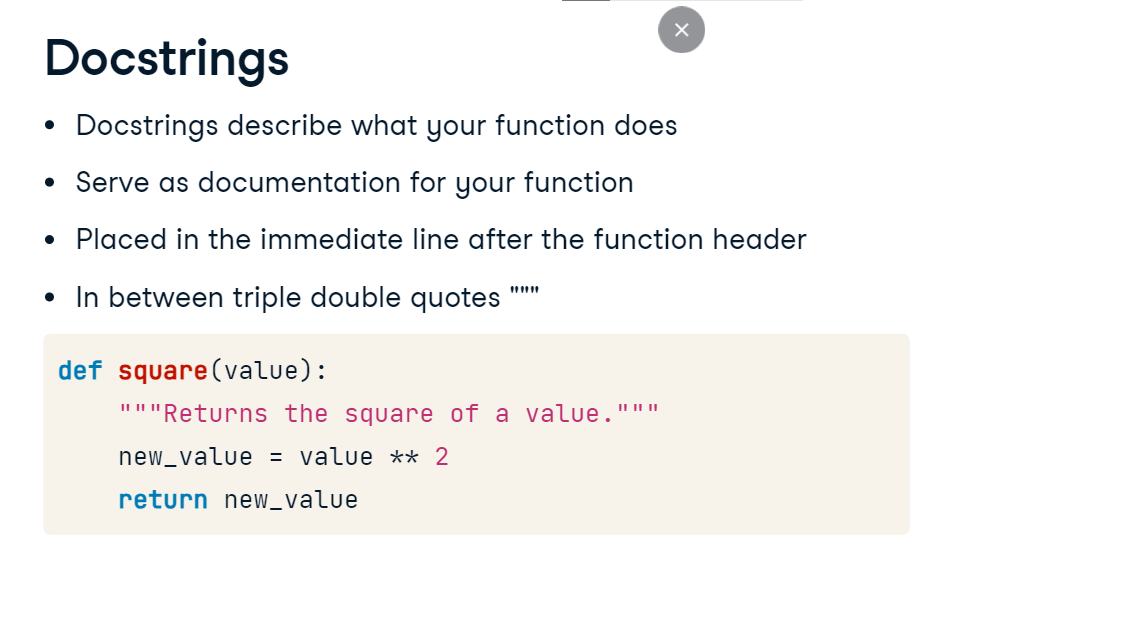
**Return values from functions**

The function square now accepts a single parameter and prints out its squared value. But what if we don't want to print that value directly and instead we want to return the squared value and assign it to some variable? You can have your function return the new value by adding the return keyword, followed by the value to return. Now we can assign to a variable num the result of the function call as you see here.



**Docstrings**

There's another essential aspect of writing functions in Python: docstrings. Docstrings are used to describe what your function does, such as the computations it performs or its return values. These descriptions serve as documentation for your function so that anyone who reads your function's docstring understands what your function does, without having to trace through all the code in the function definition. Function docstrings are placed in the immediate line after the function header and are placed in between triple quotation marks. An appropriate Docstring for our function square is 'Returns the square of a value'.

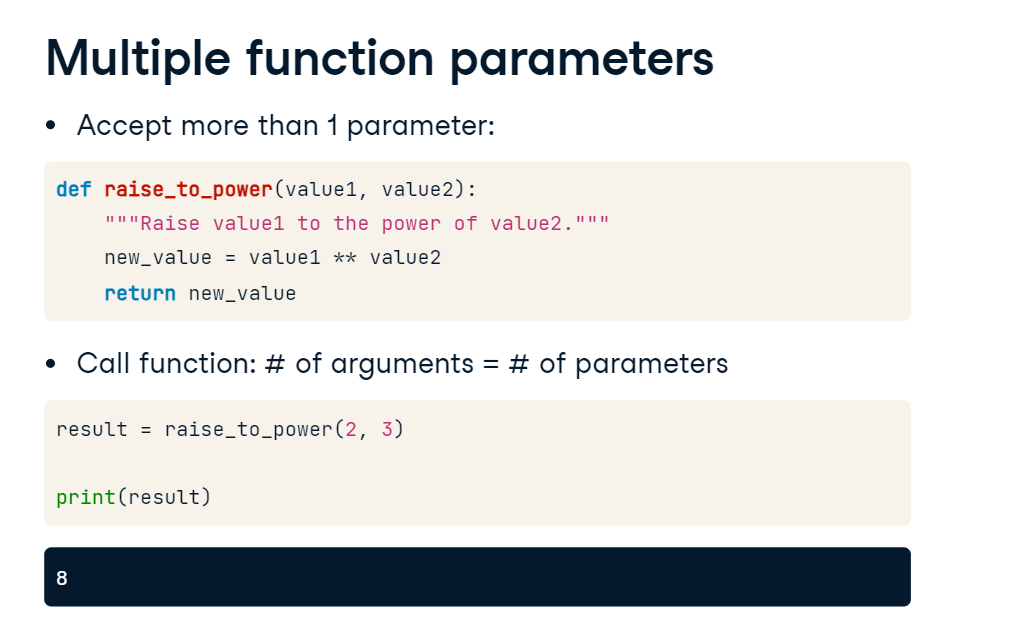


**Multiple Parameters and Return Values**

Welcome back! You're doing great at defining your own functions. Good job! At this point, you already know how to define your own functions and even return values from them. What you'll learn next is how to pass multiple arguments to functions, as well as return not just one, but multiple values from them.

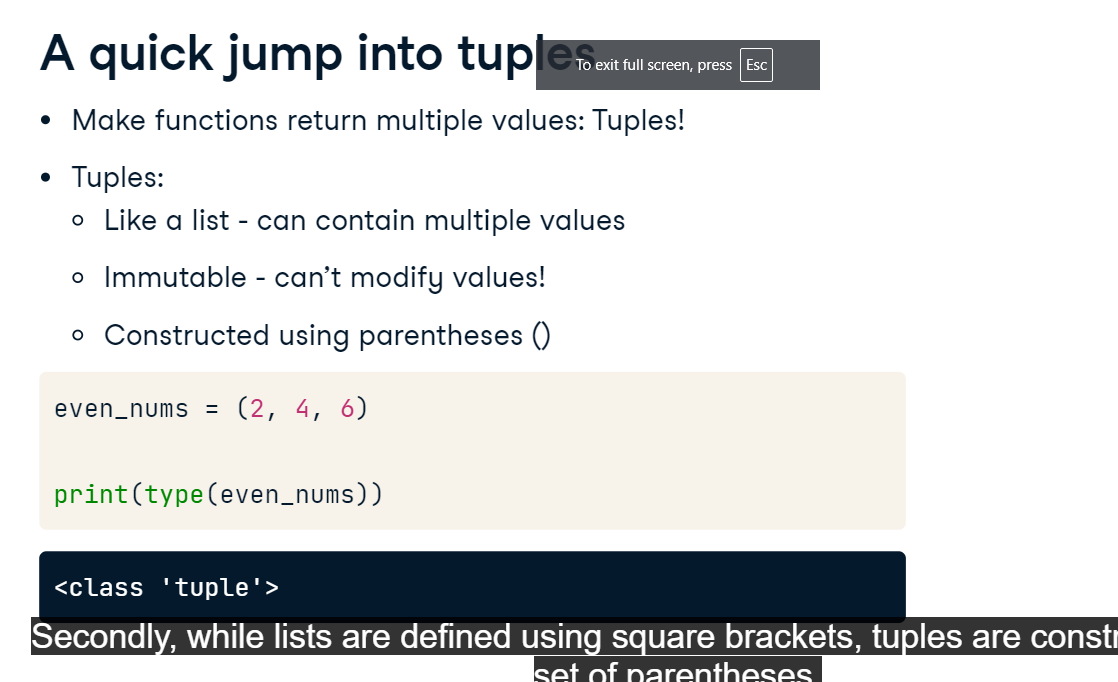
**Multiple function parameters**

Let's tweak the square function we've been working on a little bit more. Suppose that, instead of simply squaring a value, we'd like to raise a value to the power of another value that's also passed to the function. We can do this by having our function accept two parameters instead of just one. You should also change your function name AND docstrings to reflect this new behavior. raise\_to\_power is an appropriate function name. Notice that there are now two parameters in the function header instead of one, value1 and value2. In the lines after that, the behavior of the overall function was also changed by raising value1 to the power of value2. You can call the function by passing in two arguments because the function has two parameters, as declared in the function header. The order in which the arguments are passed correspond to the order of the parameters in the function header. This means that when we call raise\_to\_power(2, 3), when the function is executed, 2 would be assigned to value1 and 3 to value2. Looking at the function body, this means that the computation value1 to the power of value2 translates to 2 to the power of 3. This function call then returns the value 8.



**A quick jump into tuples**

You can also make your function return multiple values. You can do that by constructing objects known as tuples in your functions. A tuple is like a list, in that it can contain multiple values. There are some differences, however. Firstly, unlike a list, a tuple is immutable, that is, you cannot modify the values in a tuple once it has been constructed. Secondly, while lists are defined using square brackets, tuples are constructed using a set of parentheses.



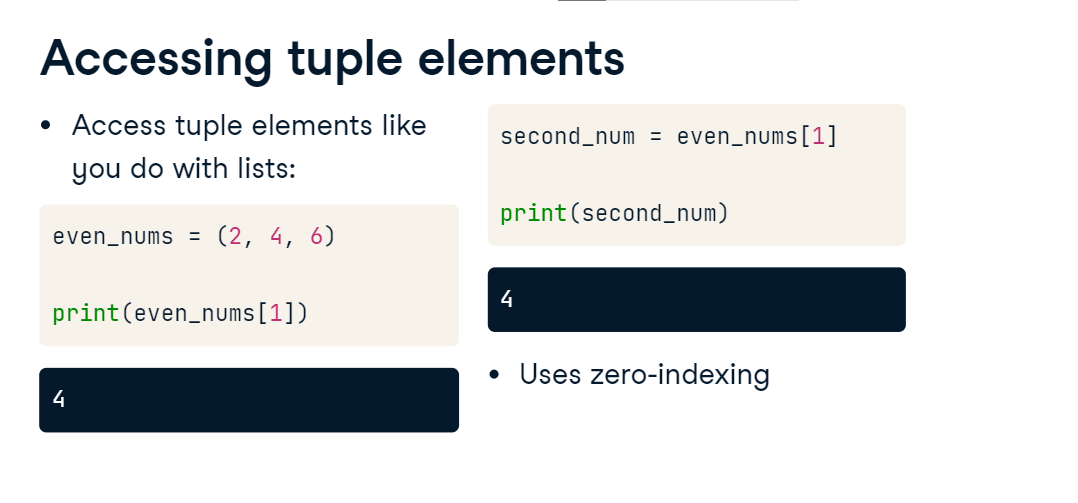
**Unpacking tuples**

Here we construct a tuple containing 3 elements. You can also unpack a tuple into several variables in one line. Doing so means that you assign to the variables a, b, and c the tuple values, in the order that they appear in the tuple.



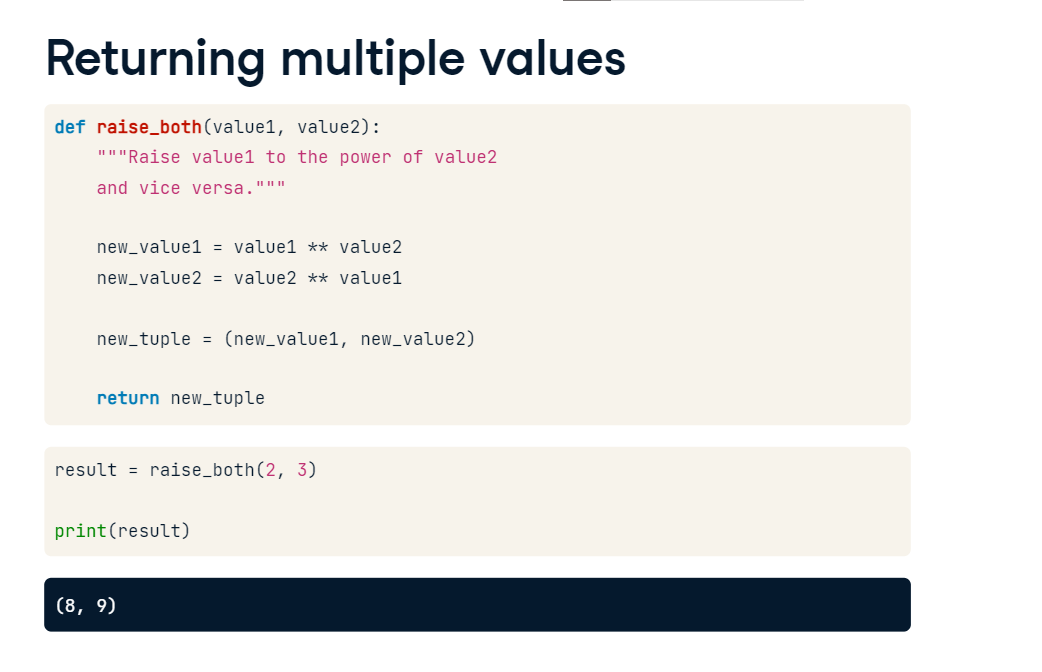
**Accessing tuple elements**

Additionally, you can also access individual tuple elements like you do with lists. Doing this here accesses the second element of the tuple. Why is that? Recall that with lists, you can use zero-indexing to access list elements. You can do the same thing with tuples! Pretty cool, right?



**Returning multiple values**

Let's now modify the behavior of your raise function. Instead of returning just the value of value1 raised to the power of value2, let's also return the value of value2 raised to the power of value1. You thus need to make raise return two values instead of one. We can use what we now know of tuples to do this! We first change the name of our function and the docstring to reflect the new behavior of our function. We then, in the function body, construct a tuple consisting of the values we want the function to return and, also in the function body, we return the tuple! Calling the function constructed demonstrates that it does exactly what we want!



**What you have learned:**

that accept multiple parameters and return multiple values, it is time to see how these learnt skills can be valuable in a Data Science context. In the following exercises, you'll write a function that analyzes a DataFrame of twitter data. The function that you'll write will return a dictionary containing data of how many times each language was used across all the tweets in the DataFrame. We'll see later in this course that we can generalize such a function to count occurrences of any items in a DataFrame column. Let's recap the basic ingredients of a function.

**Basic ingredients of a function**

We have a function header which begins with the keyword def. This is followed by the function name, parameters in parentheses and a colon. We then have the function body, which contains docstrings enclosed in triple quotation marks; docstrings describe what the function does; the rest of the function body performs the computation that the function does; the function body closes with the keyword return, followed by the value or values returned by the function. That's it from me.