**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.

**Scope and user-defined functions**

Wow! At this point, you know how to define your own functions - but not only that, you know how to write functions with multiple parameters and can return multiple values using tuples. Good job!

**Crash course on scope in functions**

We'll now talk about the idea of scope in the context of user-defined functions. You've been defining variables in your programs and so far, you've been using these variables without any problems. However, one thing you should know is that not all objects that you define are always accessible everywhere in a program. Enter the idea of scope, which tells you which part of a program an object or a name may be accessed. Names refer to the variables or, more generally, objects such as functions that are defined in your program, for example, a variable x has a name, as does the function sum. There are three types of scope that you should know. The first one is the idea of the global scope. A name that is in the global scope means that it is defined in the main body of a script or a Python program. The second one, is the local scope. A name that is in a local scope means that it is defined within a function. Once the execution of a function is done, any name inside the local scope ceases to exist, which means you cannot access those names anymore outside of the function definition. The third is something called the built-in scope: this consists of names in the pre-defined built-ins module Python provides, such as print and sum. You'll play around with the built-ins module in the

**Global vs. local scope (1)**

Let's look at a couple of examples to clarify these definitions. Let's check out our example function square from earlier. We define the function and then call it. If we then try to access the variable name new\_val after function execution, the name is not accessible. This is because it was defined only within the local scope of the function. The name new\_val was not defined globally.



**Global vs. local scope (2)**

Now what if we define the name globally before defining and calling the function? In short, any time we call the name in the global scope, it will access the name in the global, such as you see here. Any time we call the name in the local scope of the function, it will look first in the local scope. That's why calling square(3) results in 9 and not 10. If Python cannot find the name in the local scope, it will then and only then look in the global scope.

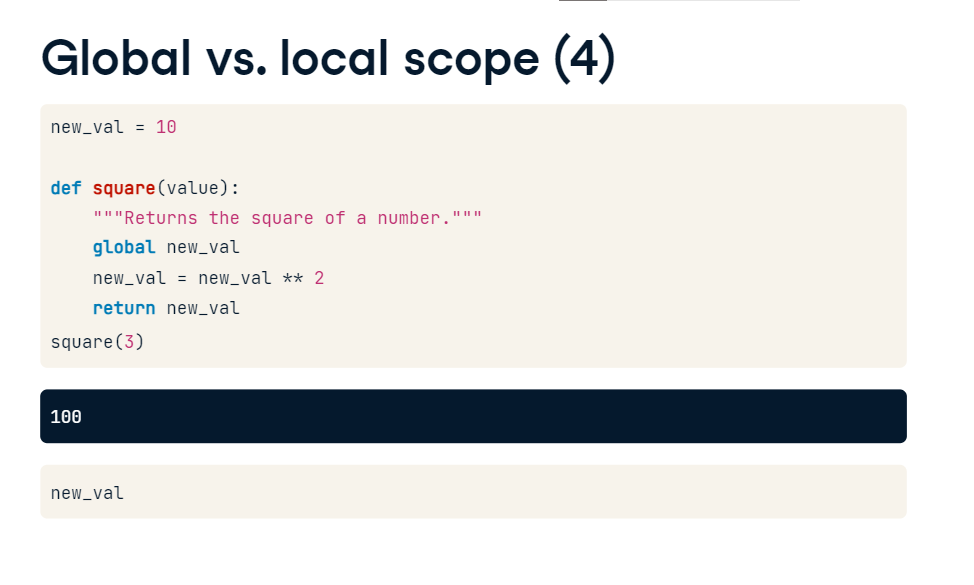


**Global vs. local scope (3)**

Here, for example, we access new\_val defined globally within the function square. Note that the global value accessed is the value at the time the function is called, not the value when the function is defined. Thus, if we re-assign new\_val and call the function square, we see that the new value of new\_val is accessed. To recap, when we reference a name, first the local scope is searched, then the global. If the name is in neither, then the built-in scope is searched.

**Global vs. local scope (4)**

Now what if we want to alter the value of a global name within a function call? This is where the keyword global comes in handy. To look at how it works, let's look at another example. Within the function definition, we use the keyword global followed by the name of the global variable that we wish to access and alter. For example, here we change new\_val to its square. The function call works as one would expect. Now calling new\_val, we see that the global value has indeed been squared by running the function square.



**Nested functions**

00:00 - 00:07

Now that you've come to grips with scope, both local and global, it's time to dive a bit deeper!

**Nested functions (1)**

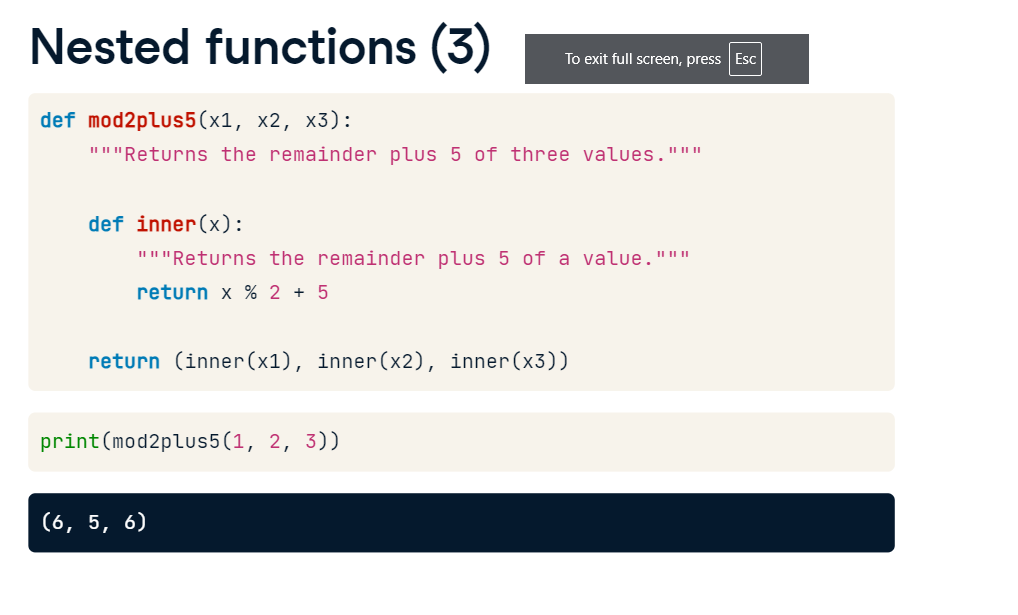
What if we have a function inner defined within another function outer and we reference a name x in the inner function? The answer is intuitive: Python searches the local scope of the function inner, then if it doesn't find x, it searches the scope of the function outer, which is called an enclosing function because it encloses the function inner. If Python can't find x in the scope of the enclosing function, it only then searches the global scope and then the built-in scope. But whoa, hold on there for a second, why are we even nesting functions?

**Nested functions (2)**

There are a number of good reasons to do so. Let's say that we want to use a process a number of times within a function. For example, we want a function that takes 3 numbers as parameters and performs the same function on each of them. One way would be to write out the computation 3 times

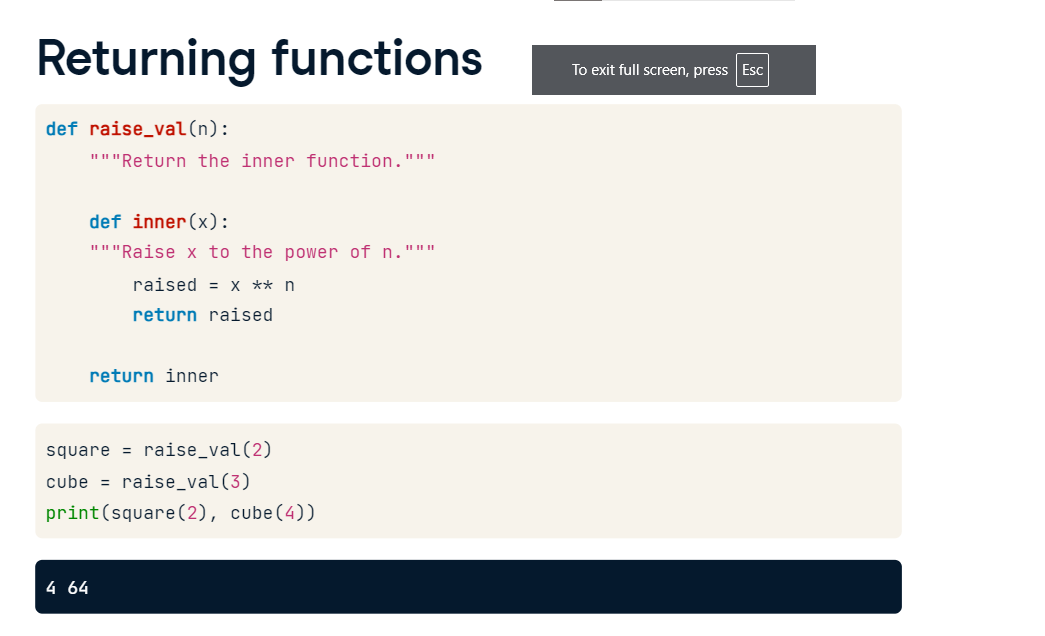
**Nested functions (3)**

but this definitely does not scale if you need to perform the computation many times. What we can do instead is define an inner function within our function definition, such as we do here, and call it where necessary. This is called a nested function. The syntax for the inner function is exactly the same as that for any other function.



**Returning functions**

Let's now look at another important use case of nested functions. In this example, we define a function raise\_vals, which contains an inner function called inner. Now look at what raise\_vals returns: it returns the inner function inner! raise\_vals takes an argument n and creates a function inner that returns the nth power of any number. That's a bit complicated and will be clearer when we use the function raise\_vals. Passing the number 2 to raise\_vals creates a function that squares any number. Similarly, passing the number 3 to raise\_vals creates a function that cubes any number. One interesting detail: when we call the function square, it remembers the value n=2, although the enclosing scope defined by raise\_val and to which n=2 is local, has finished execution. This is a subtlety referred to as a closure in Computer Science circles and shouldn't concern you too much. It is worth mentioning, however, as you may encounter it out there in the wild.

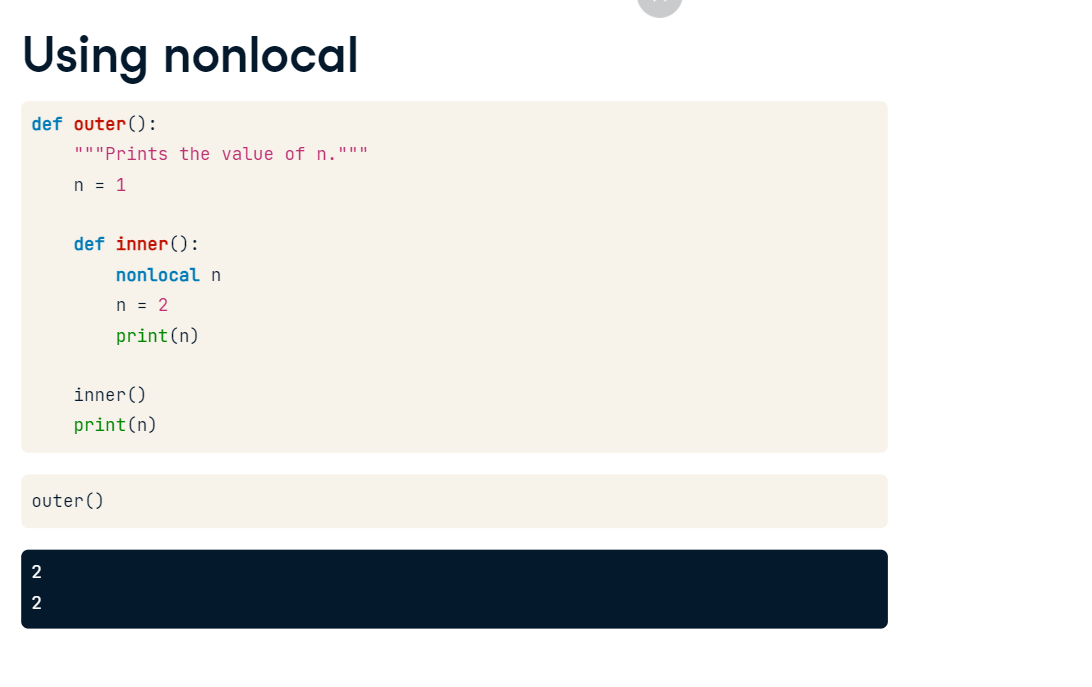


**Using nonlocal**

Recall from our discussion of scope that you can use the keyword global in function definitions to create and change global names; similarly, in a nested function, you can use the keyword nonlocal to create and change names in an enclosing scope. In this example, we alter the value of n in the inner function; because we used the keyword nonlocal, it also alter the value of n in the enclosing scope. This is why calling the function outer prints the value of n as determined within the function inner. You'll have practice using the keyword nonlocal in the interactive exercises.

**Scopes searched**

To summarize: name references search at most four scopes, the local scope, then those of enclosing functions, if there are any; then global, then built-in. This is known as the LEGB rule, where L is for local, E for enclosing, G for global and B for built-ins! Also, remember that assigning names will only create or change local names, unless they are declared in global or nonlocal statements using the keyword global or the keyword nonlocal, respectively.



**Default and flexible arguments**

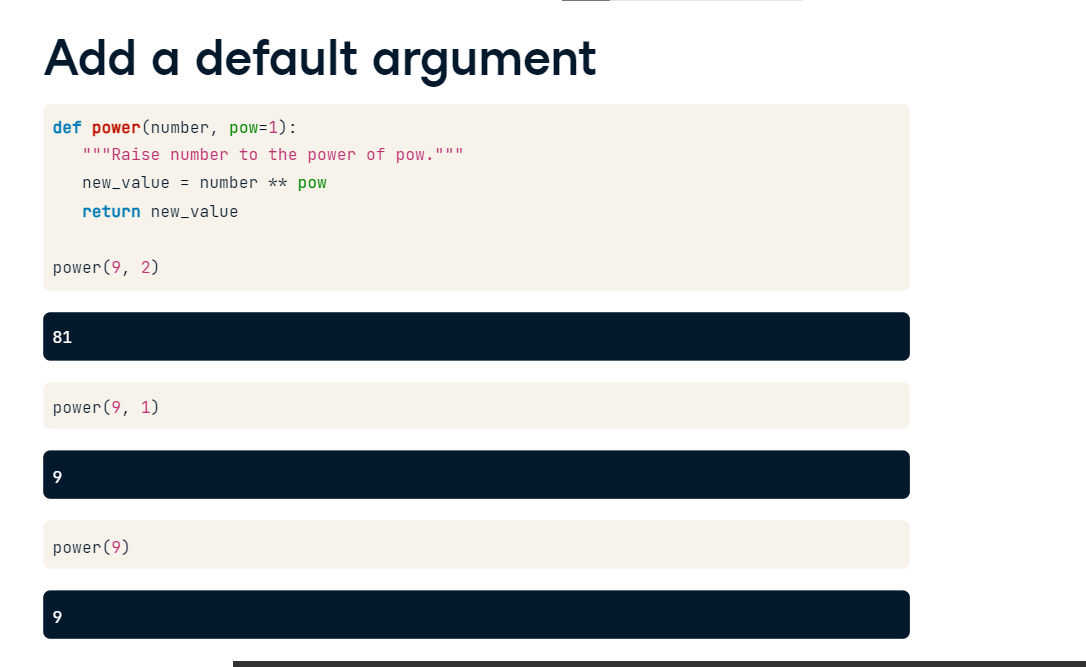
Let's say that you're writing a function that takes multiple parameters and that there is often a common value for some of these parameters. In this case, you would like to be able to call the function without explicitly specifying every parameter. In other words, you would like some parameters to have default arguments that are used when it is not specified otherwise!

**You'll learn:**

00:22 - 00:33In this video, you'll discover how to write function with default arguments, along with using flexible arguments, which allows you to pass any number of arguments to a function, as we'll soon see.

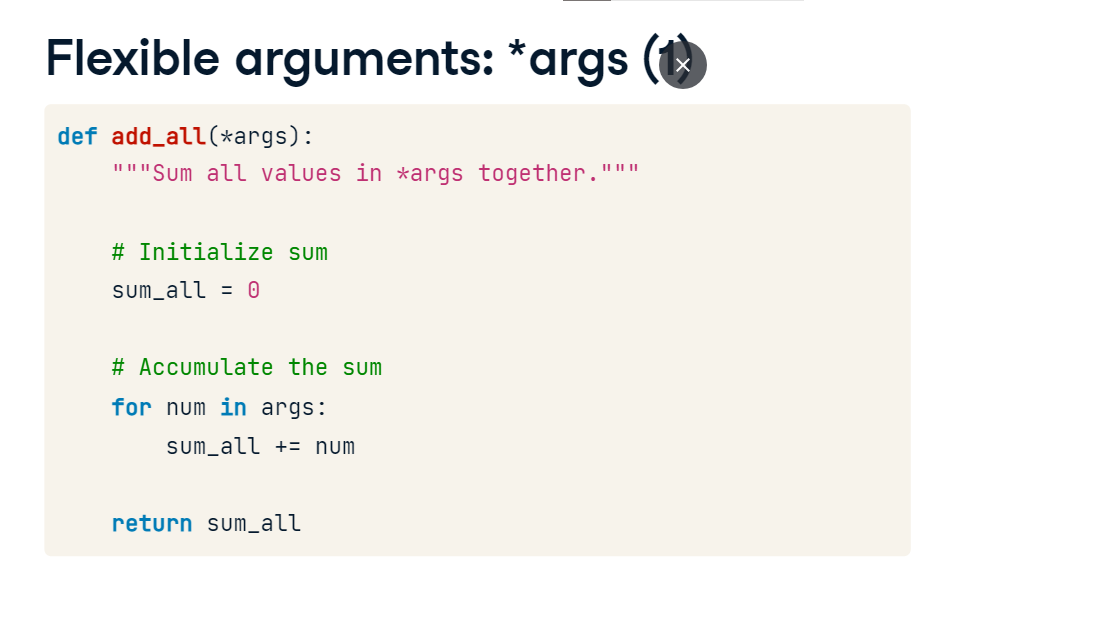
**Add a default argument**

First up, to define a function with a default argument value, in the function header we follow the parameter of interest with an equals sign and the default argument value. Notice that this function raises the first argument to the power of the second argument and the default 2nd argument value is 1. So we can call the function with two arguments as you would expect, however, if you only use one argument, the function call will use the default argument of 1 for the second parameter! Neat, huh? In the interactive exercises that follows, you'll gain expertise in writing functions with both single and multiple default arguments.



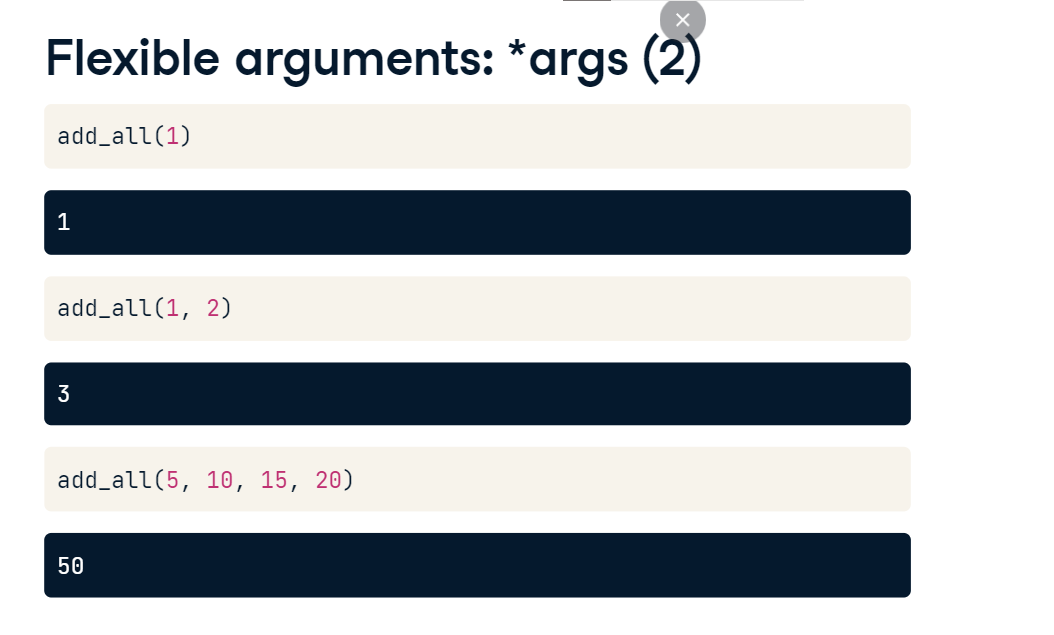
**Flexible arguments: \*args (1)**

Lets now look at flexible arguments: let's say that you want to write a function but aren't sure how many arguments a user will want to pass it; for example, a function that takes floats or ints and adds them all up, irrespective of how many there are. Enter flexible arguments! In this example, we write the function that sums up all the arguments passed to it. In the function definition, we use the parameter star followed by args: this then turns all the arguments passed to a function call into a tuple called args in the function body; then, in the function body, to write our desired function, we initialize our sum sum\_all to 0, loop over the tuple args and add each element of it successively to sum\_all and then return it.



**Flexible arguments: \*args (2)**

We can now call our function add\_all with any number of arguments to add them all up!



**Flexible arguments: \*\*kwargs**

You can also use a double star to pass an arbitrary number of keyword arguments, also called kwargs, that is, arguments preceded by identifiers. We'll write such a function called print\_all that prints out the identifiers and the parameters passed to them as you see here.

**Flexible arguments: \*\*kwargs**

Now to write such a function, we use the parameter kwargs preceded by a double star. This turns the identifier-keyword pairs into a dictionary within the function body. Then, in the function body all we need to do is to print all the key-value pairs stored in the dictionary kwargs. Note that it is NOT the names args and kwargs that are important when using flexible arguments, but rather that they're preceded by a single and double star, respectively.



## Bringing it all together

It's now time to get you hands dirty using your newly acquired skills. Recall that, in the previous chapter, you wrote a function that would perform the following: it would take a DataFrame of twitter data and return a dictionary containing languages as keys and the number of times a tweet was written in a given language as values.

**Next exercises:**

In the following exercises, you will write a function that processes a DataFrame and returns a dictionary with counts of occurrences in any column at all! By default, however, it will process a column called lang. This generalizes the previous function that you wrote. You will then generalize this further so that you can pass the function a DataFrame and any number of column names to perform the computation on an arbitrary number of columns.

**Add a default argument**

Let's remind ourselves of the techniques necessary: recall that to define a function with a default argument, all that you need to do is provide that argument in the function header as you can see here.

**Flexible arguments: \*args (1)**

Next up, to write a function that you can pass an arbitrary number of arguments to, that is, flexible arguments, we use the arguments args as here, and then we can loop over all elements of args in the function body. Now that we've had that quick refresher, let's write some more functions!