# Functions Structure and Parameters behind the scenes with diagrams!

Congratulations! You're now in week 2, diving deeper into programming and its essential components. In this case, we will talk about **FUNCTIONS**!

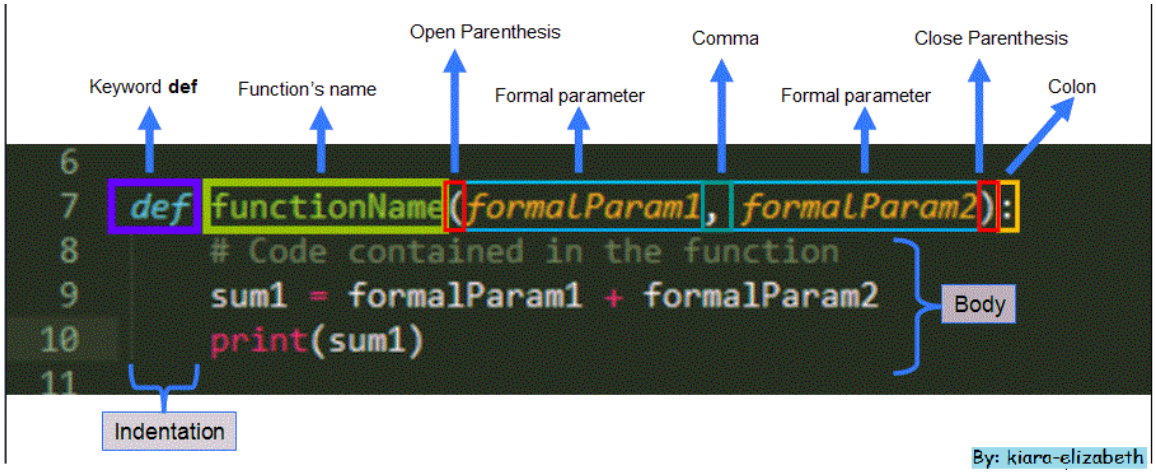
**Functions are blocks of code you can reuse or execute as many times as you'd like, by passing an input and getting an output (depending on what the function does).** They're basically a template for a procedure or algorithm you want to execute, similar to blueprints, and every time you want to execute that specific code, by "calling" the function you can execute the same "instructions" but with different values you can pass as input.

**So... Let's begin!**

- When you define a function you use the keyword **`def`** followed by its **name**. Every time you want to execute that block of code you will have to refer to it by its name, so make sure it's descriptive and related to its purpose. (i.e if it will add two numbers you may call it "add" or "sum").
- After the keyword **def** and the **name** of the function, you will **open parentheses** and add what we call **Formal Parameters** (we'll discuss them in the post) separated by commas. Then you will **close parentheses** and finally **add a colon (:)** to complete the line.

Below this line there will be indented code (it will have a space or tab to the left, like a "margin"). This is an essential characteristic when working with python. Indentation is crucial, or the code will throw an error because python won't recognize that you wanted that code to be contained inside the function.

Indentation is how python identifies at which line the function starts and where it ends, so it doesn't continue executing code below that doesn't belong to the function.




***PARAMETERS:***

We've mentioned that functions are code that can be reused and that by giving them different inputs we can get different outputs (depending on what the function does). But how does the function know where we want to use our input? **That's where FORMAL PARAMETERS come to the rescue!**

They are general representations or "placeholders" of where we want to use our input but they don't have any values assigned to them UNTIL we call the function (until we tell our code that we want to execute the function. Up until now we're just designing our blueprint; we haven't executed any lines of code)

In the example below we can see that we have **TWO** formal parameters, *formalParam1* and *formalParam2*.

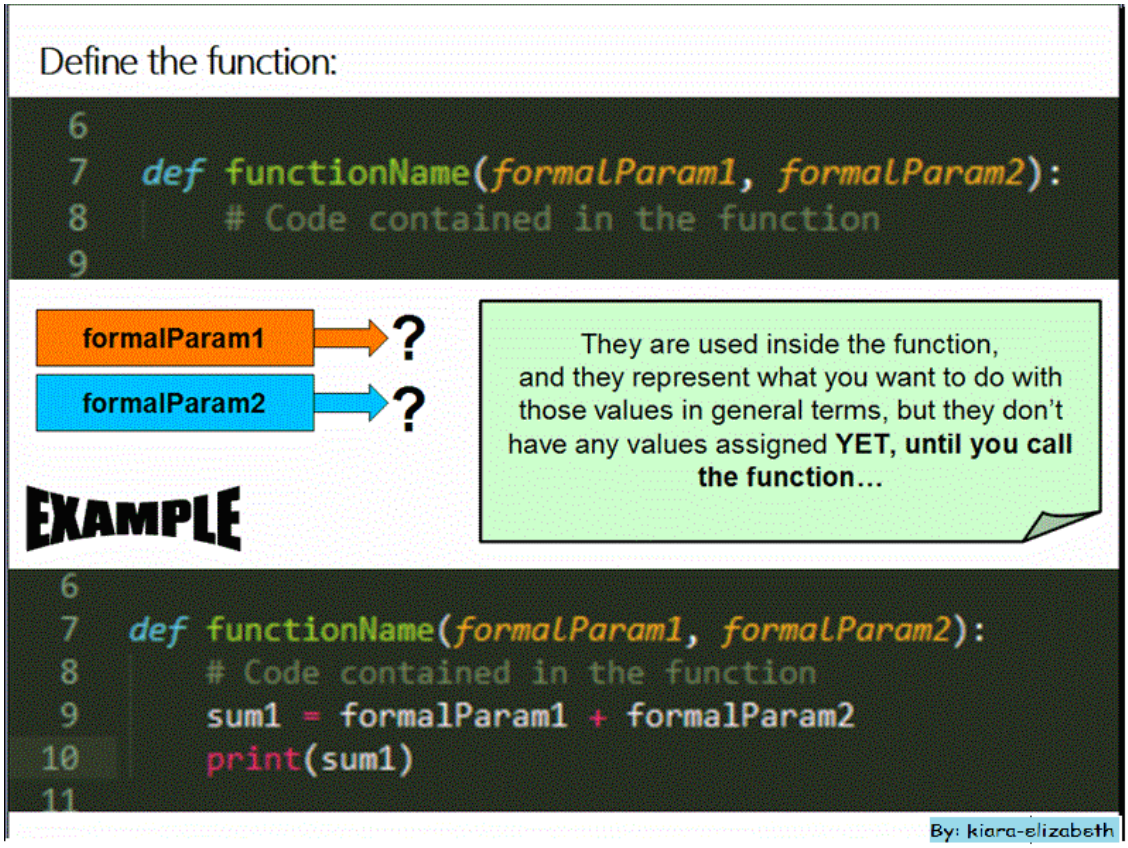
- On line 9 we find `sum1 = formalParam1 + formalParam2.`

What's happening here? We're basically telling the function that we will pass two inputs when we call it, a value for *formalParam1* and a value for *formalParam2.* But right now they are just placeholders.

We're just saying that we want the *value we pass for formalParam1* to be added to the *value we pass for formalParam2* and for that result to be assigned to a variable sum1. That's basically how formal parameters work. **They tell the function where we want to use the values we pass in.**

When we call the function, we use this syntax:

functionName(parameter1, parameter2, ... (as many parameters as needed, separated by commas))




Now, when we call the function we are going to execute the code inside the function, but this time, we will specify which values we want to operate with, these are called **ACTUAL PARAMETERS or ARGUMENTS**.

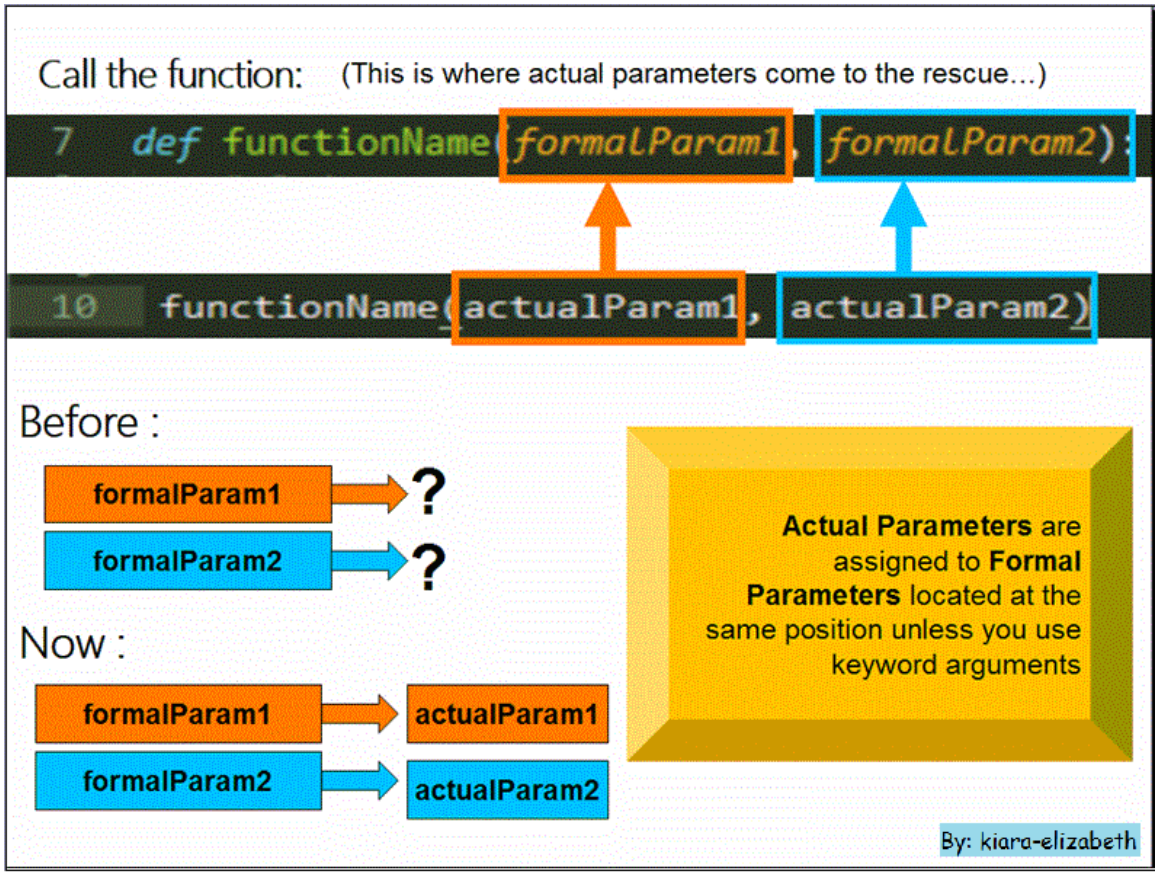
And you may ask: how will the function know which values we want to assign to which placeholder (Formal parameter)?

This is where **LOCAL SCOPES** come save the day! We will assign a value to each Formal Parameter so the function knows which values we want to operate with.

Well, if we simply give values for each formal parameter without specifying the parameter itself (we will see this case in subsequent lectures), **each value will be assigned to the formal parameter located at the same position in the sequence.**

- For example, in the diagram, *actualParam1* will be assigned to *FormalParam1* because they are located at the same position (1st in the sequence), and *actualParam2* will be assigned to *FormalParam2* because they are both 2nd in the sequence and so on for as many formal parameters as you define in the function.

Now the code will run, but these values we passed as input will fill the "placeholders" we had in the general blueprint inside the function. *formalParam1* will now hold the value *actualParam1*, and so on for every formal parameter.



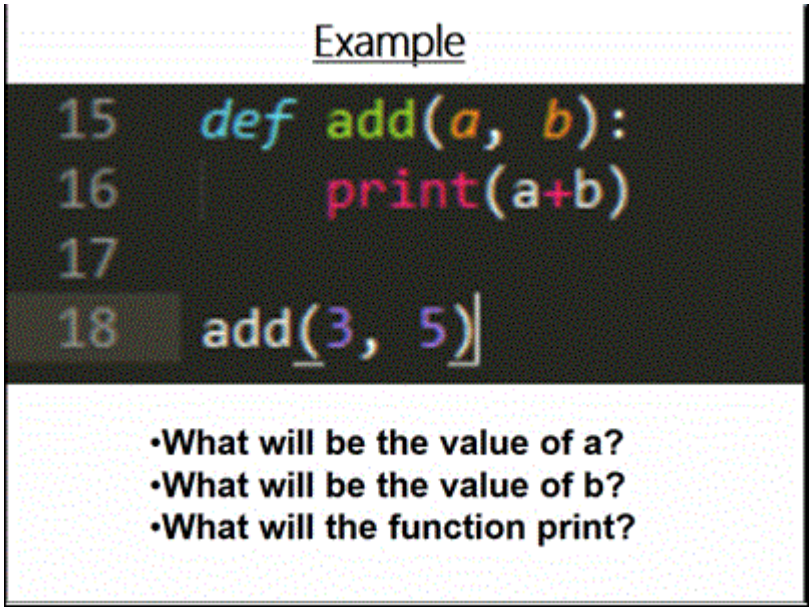
**As an example of these incredibly important building blocks of functions we have this function:**

First, we use the keyword **def** followed by the function's name, **add**. Then we open parentheses and give as many **FORMAL parameters** as we need. Since we will add TWO numbers, we will need two placeholders for those numbers we will pass as input. We give them a "name", in this case a and b that we will use throught the function to tell it where we want to use the values passed as inputs for each one.

**REMEMBER:** a and b are just placeholders, they tell the function where to use the values we pass in, and won't have a value until we call the function with actual parameters.

When we call the function add, we use its name **add** followed by parentheses and as many actual parameters as we initially defined in the function, in this case two numbers.

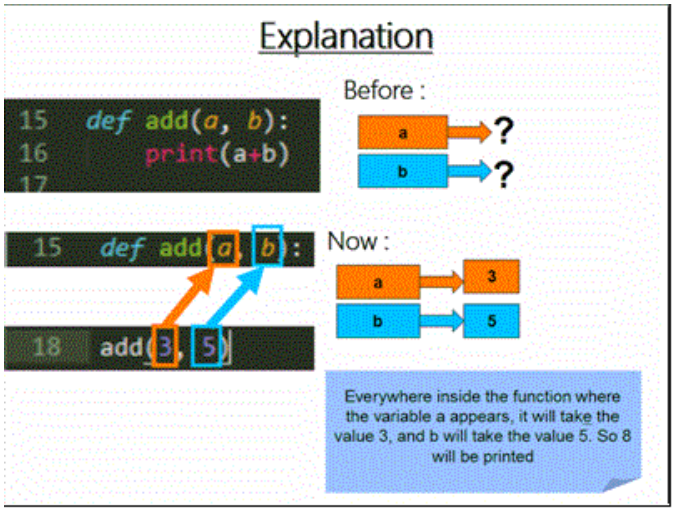
If you'd like, try to determine what will be the value of a, b and what will the function print before moving on : )




We're back! Since 3 is first in the sequence, it will be assigned to the first formal parameter in the function definition (in this case, a) so when you see an a in the function definition, that a will now hold the value 3.

Since 5 is the second element in the sequence, it will be assigned to the second formal parameter in the function definition (in this case b, so when you see b in the function, that b will now hold the value 5)

Since print() will print the sum of the values for a and b we passed as inputs, this function call will print 8.




Here you can see this example in the shell. I've added two print statement to print the values of a and b.

```
>>> def add(a, b):
        print(a+b)
        print(a)
        print(b)


>>> add(3, 5)
8
3
5
```

Hope it helps! If you have any questions, please post them on the forums! : )

Estefania (kiara-elizabeth).