



Hyperiondev

Workshop: Functions & Function Scope

Lecture – Housekeeping

- ❑ The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
- ❑ No question is daft or silly - **ask them!**
- ❑ There are Q/A sessions midway and at the end of the session, should you wish to ask any follow-up questions.
- ❑ You can also submit questions here:
<http://hyperiondev.com/sbc4-se-questions>
- ❑ For all non-academic questions, please submit a query:
www.hyperiondev.com/support
- ❑ Report a safeguarding incident:
<http://hyperiondev.com/safeguardreporting>
- ❑ We would love your feedback on lectures:
<https://hyperiondev.wufoo.com/forms/zsgv4m40ui4i0g/>

Objectives

1. Recap on:
 - a. Why are functions useful?
 - b. Function signature
 - c. Function body
 - d. Parameters and arguments
2. Parameters and arguments
 - a. Positional and keyword arguments
 - b. Default parameters
3. Function scope
 - a. Local and global variables

Github Repository – Lecture Examples

https://github.com/HyperionDevBootcamps/C4_SE_lecture_examples

Why are functions useful?

- **Reusable code** – Sometimes you need to do the same task over and over again.
- **Error checking/validation** – Makes this easier, as you can define all rules in one place.
- **Divide code up into manageable chunks** – Makes code easier to understand.
- **More rapid application development** – The same functionality doesn't need to be defined again.
- **Easier maintenance** – Code only needs to be changed in one place.

Function Signature

- **Function name :**

- The name that identifies the function and is used to call it later in the code. It should follow naming conventions (e.g., using *lowercase letters and underscores*)

- **Parameters:**

- The *inputs that the function expects to receive* when it is called. Each parameter consists of a name and can optionally have a default value assigned to it.

- **Return Type (optional):**

- The data type that the function is expected to return when it finishes executing. If the function doesn't return a value, the return type is usually specified as *None* or omitted altogether.

```
def my_function(a, b) -> int:
```

Function Body

The statements within the function body can perform any desired operations, such as calculations, conditionals, loops, or even calling other functions.

It is within the function body that you define the logic and actions the function should execute:

```
def my_function(a, b) -> int:
```

```
    result = a * b  
    return result
```

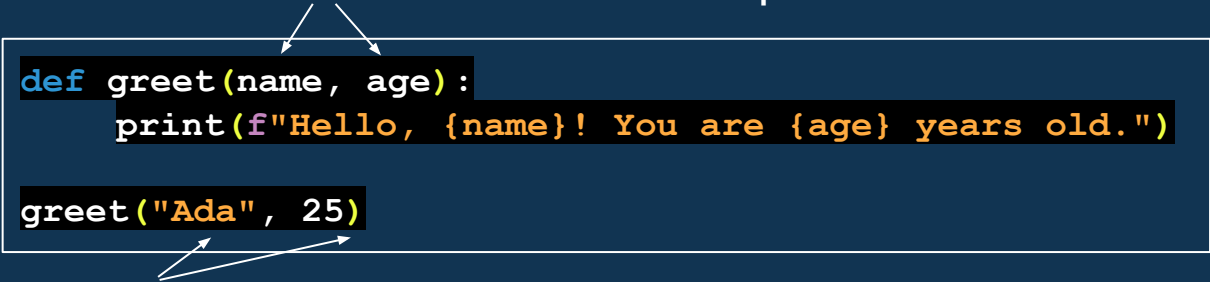
← Function Signature

← Function Body

The function signature defines the structure and expectations of the function, while the function body contains the **actual code** that is executed when the function is **called**

Parameters and arguments

- **Parameters** are the variables listed in the function's definition, representing the **inputs that the function expects** to receive when it is called. They are defined in the **function signature** and act as **placeholders** for the values that will be passed to the function.



A diagram illustrating the relationship between function parameters and arguments. A box contains the following Python code:

```
def greet(name, age):  
    print(f"Hello, {name}! You are {age} years old.")  
  
greet("Ada", 25)
```

Two arrows point from the text "Parameters" in the bullet point above to the parameters `name` and `age` in the function signature `def greet(name, age):`. Two arrows point from the text "Arguments" in the bullet point below to the arguments `"Ada"` and `25` in the function call `greet("Ada", 25)`.

- Arguments, on the other hand, are the **actual values** that are passed to the function when it is called. They correspond to the parameters defined in the function signature and provide the necessary data for the function to work with.

Positional arguments vs keyword arguments

- Positional arguments are passed to a function based on their **position** or **order** in the function call. The **order** in which the arguments are passed **must match** the order of the parameters in the function signature.
- Keyword arguments are passed to a function using the **names** of the parameters in the function signature. They allow you to specify arguments by name, **regardless of their position**.

```
def greet(name, age):  
    print(f"Hello, {name}! You are {age} years old.")  
  
greet(age=25, name="Ada")
```

Default parameters

Default parameter values are assigned to parameters in the **function signature**, providing a default value that is used when no argument is provided for that parameter.

```
def greet(name, age=30):  
    print(f"Hello, {name}! You are {age} years old.")  
  
greet("Ada")
```

Note: If arguments are provided, then the default values are overwritten.

Function Scope

- Function scope refers to the **visibility** and **accessibility** of **variables** within a function.
- Variables defined within a function have **local scope**, meaning they are **only** accessible within that specific function.

The rule of thumb is that a function is covered in one-way glass: it can see out, but no one can see in.

Error message when trying to access local variable outside a function

- Generally, whenever code is executed, variables become accessible across the entire script.
- Functions are different, however. Variables declared within functions are not accessible outside the function.
 - This avoids variable names being overwritten.

```
def multiply(x,y):  
    product = x * y  
    return product  
  
answer1 = multiply(2,3)  
  
print(f"{x} times {y} is {answer1}")
```

```
print(f"{x} times {y} is {answer1}")  
NameError: name 'x' is not defined
```

Local vs global variables

- Local variables are variables that are defined within a function and are only accessible within that function's scope. They are created when the function is called and are destroyed when the function finishes executing. Local variables are useful for storing **temporary data** that is only needed within the function.

```
def my_function():  
    x = 10  
    print(x)
```

- Global variables are variables that are **defined outside** of any function or in the global scope. They can be accessed and modified from anywhere within the code, **including inside functions**.

```
x = 10  
def my_function():  
    print(x)
```

Accessing and modifying global variables inside a function

- Inside a function, you can access the value of a global variable by referencing its name directly. Python will look for the variable in the local scope first and, if not found, will then look in the global scope.
- By default, if you try to modify a global variable inside a function, Python will create a new local variable with the same name, instead of modifying the global variable. To modify the actual global variable from within a function, you need to use the **global keyword**.

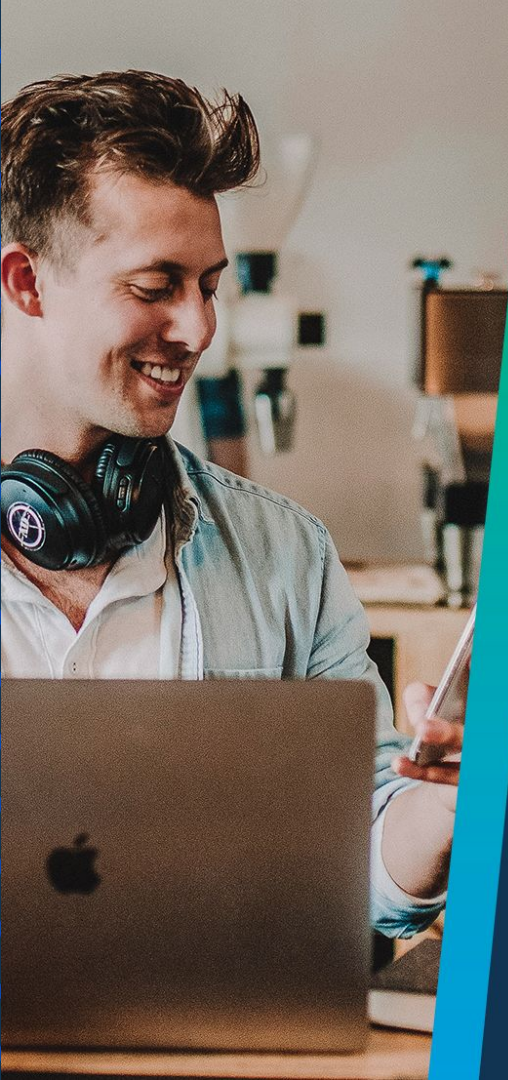
```
x = 10
def my_function():
    global x
    x = 20
```

**Let's have a look at some
more examples in VS
code :)**

Hyperiondev

Q & A Section

Please use this time to ask any questions relating to the topic explained, should you have any



Hyperiondev

Thank you for joining us

Stay hydrated
Avoid prolonged screen time
Take regular breaks
Have fun :)