Python Fundamentals

Functions, return statement, arguments, scope

Objectives:

- Functions
- return statement
- Positional argument
- Keyword argument
- Function scope
- global statement

What is a function?

- Functions allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command.
- This avoids having to type out the same code multiple times.
- The `print` function, for example, executes several statements in order to write our values to the console.

What is a function?

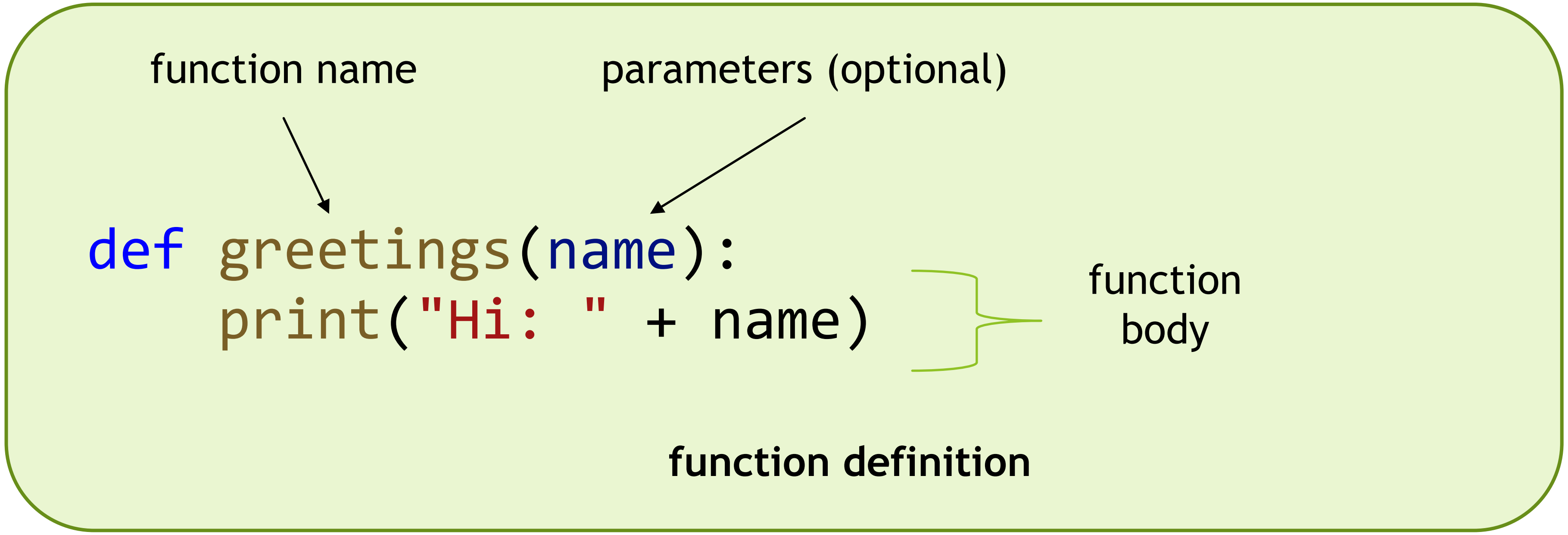
- There are generally two types of functions:
  - Functions that return a value
  - Functions that DO NOT return a value

- Every function has a *signature*, consisting of:
  - The name of the function
  - A list of parameters

- A function also has a *body*. The statements executed by the function.

Declaring a function

Syntax for declaring a function.



function name

parameters (optional)

```python
def greetings(name):
    print("Hi: " + name)
```

function body

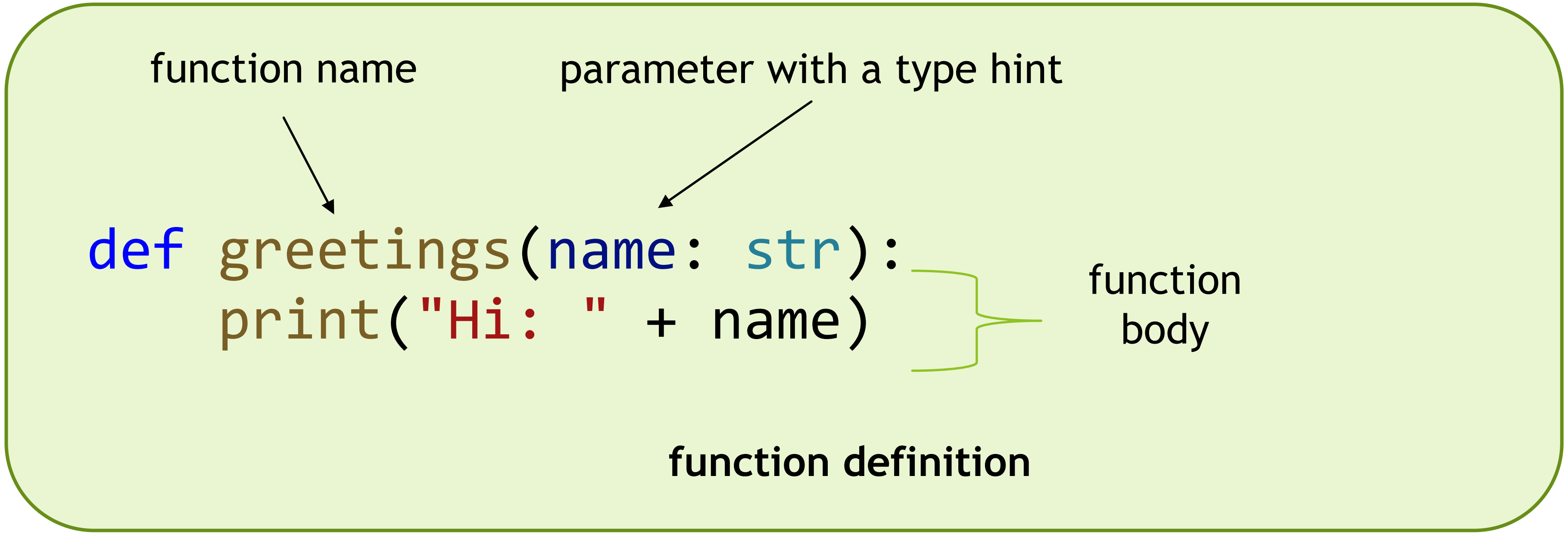
function definition

Declaring a function

- As with regular variables, parameters have no declared type.
- However, functions are often written with the expectation parameters will be of a certain type.
- In Python 3.5, type hints were introduced. This allow a coder to annotate their functions and provide clarity about expected types.

Declaring a function

Declaring a function with type hints for parameters.



function name

parameter with a type hint

```python
def greetings(name: str):
    print("Hi: " + name)
```

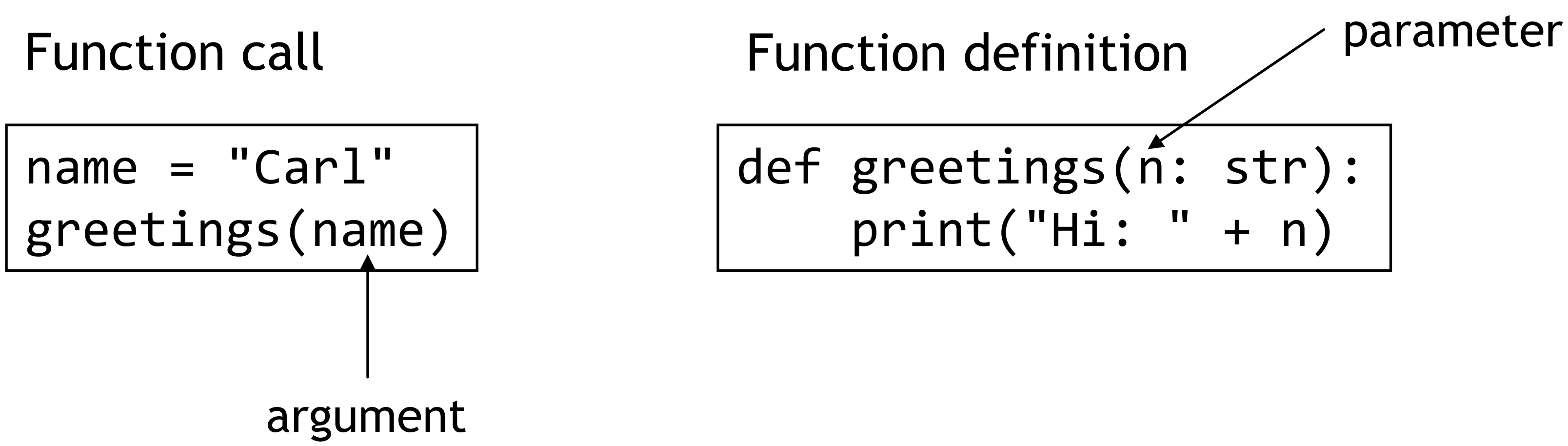
function body

function definition

Calling a function

- When we call a function we use its **name**.
- If a function has parameters, we have to provide arguments. Arguments are the information we give to a function.
- Arguments are matched to parameters by position or keyword.
- While Python does not strictly enforce type checking, the type of an argument should match the expected type of a parameter.

Calling a function



Function call

```
name = "Carl"
greetings(name)
```

argument

Function definition

parameter

```
def greetings(n: str):
    print("Hi: " + n)
```

name is the **argument**.
n is the **parameter** for our **greetings** function
Note how the argument matches the parameter by position.
In addition, because we're performing concatenation in the function, we ensure the argument is a **string**.

Calling a function

Another consideration when calling the function is the order in which your code is written. The below sample WILL work.

```
def greetings(name: str):
    print("Hello: " + name)

greetings("students")
# Execution is passed to the function
# Writes to the console: Hello: Students
# Execution returns
print("Welcome to this Python session!")
print("Bye!!")
```

Calling a function

However, if you try to call a function before it is declared, Python will raise an error at run-time.

```
greetings("students")
# Error is raised:
# NameError: name 'greetings' is not defined
print("Welcome to this Python session!")
print("Bye!!")

def greetings(name: str):
    print("Hello: " + name)
```

Returning a value from a function

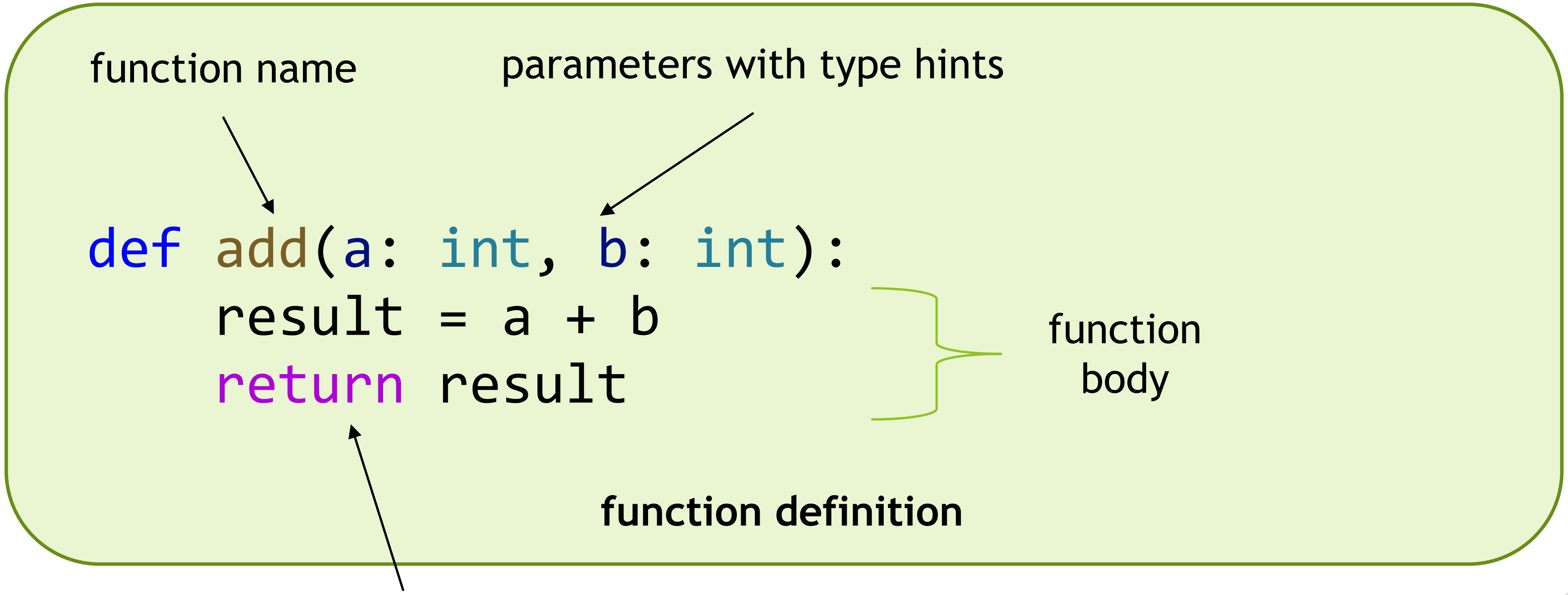
- Often you will want to have a function that returns a value. In order to do this, you have to do the following:
  - Use the return statement in the body of the function.

- The return statement not only returns a value, but immediately leaves the execution of the function.

- When calling a function with a return value, we can assign the result of that function call to a variable

The return statement

Declaring a function that returns a value



function name

parameters with type hints

```
def add(a: int, b: int):
    result = a + b
    return result
```

function body

function definition

The return statement terminates execution of the function and specifies the value that the function returns.

The return statement

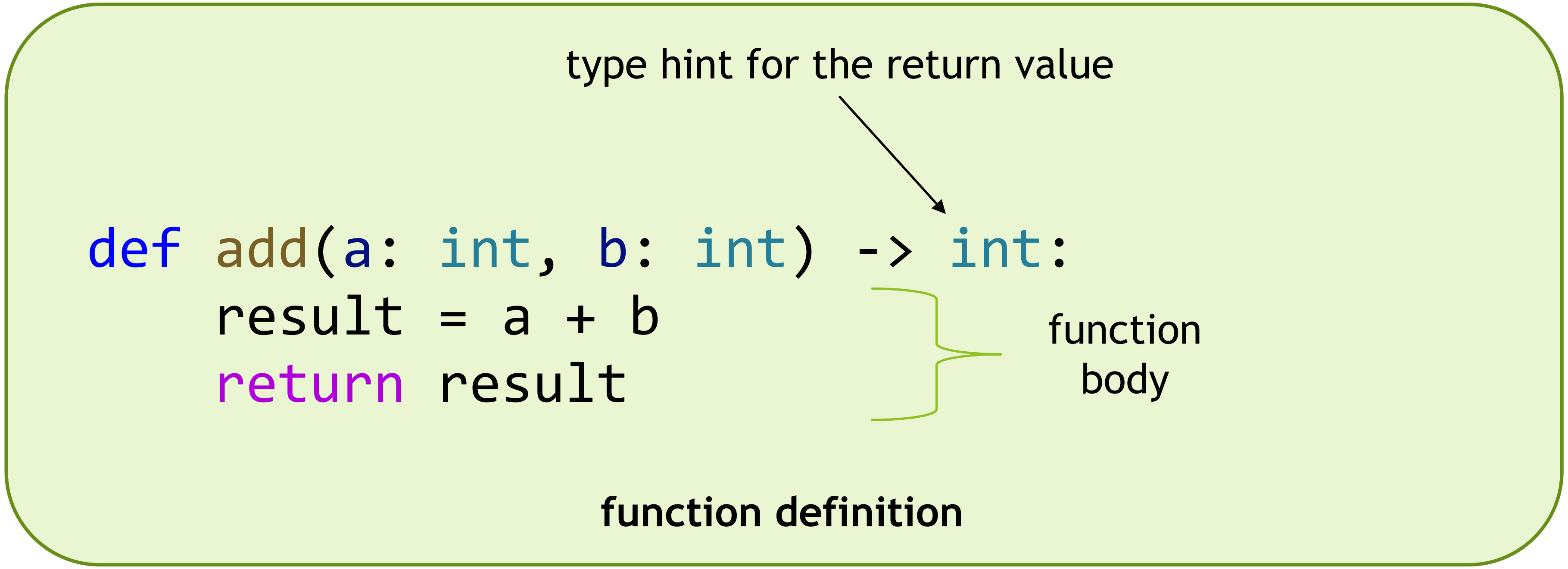
- In the example we saw not only how to return a value, but how to declare multiple parameters. Each parameter was separated by a comma.

- As with parameters, Python provides type hinting for the return value of a function as well.

- Again, this requires use of Python 3.5 or higher.

The return statement

Declaring a function with a type hint for the return value



type hint for the return value

```
def add(a: int, b: int) -> int:
    result = a + b
    return result
```

function body

function definition

Function call and definition (return)

Function call

```
x, y = 3, 8
z = 0.0
z = average(x, y)
```

Function definition

```
def average(a: int, b: int) -> float:
    c = 0.0
    c = (a + b) / 2.0
    return c
```

x and **y** are the arguments.
a and **b** are parameters for our **average** function
Note that the arguments match the parameters both in position and type.
In this case the type of the arguments and parameters are **int**
The value of **c** will be returned from the function and assigned to **z**

Positional vs Keyword arguments

- In Python there are two methods for passing arguments to a function, and having those arguments assigned to a parameter.

- Arguments can be passed either by position, or keyword.

- Positional we've already seen. Keyword is when you specify the name of the parameter in your list of arguments.

Positional vs Keyword arguments

Below illustrates the difference between passing arguments by position, or by keyword.

```python
def add(a: int, b: int) -> float:
    result = a + b
    return result

# Passing arguments by position
result = add(2, 4)

# Passing arguments by keyword
result = add(b=4, a=2)
```

Positional vs Keyword arguments

- The sample showed how we can call a function either with positional arguments, or keyword arguments. In fact, you can even use a mix of both!

- In addition, keyword arguments allow you to specify arguments in an order different from the parameters.

- However, there are a number of pitfalls to be aware of.

Positional vs Keyword arguments

Below we can see some issues that can arise when mixing positional and keyword arguments.

```
# First a correct mix of positional and keyword
result = add(2, b=5)

# Error: Positional argument after a keyword argument
result = add(b=2, 2)

# Error: Argument assigned to the same parameter twice
# Error: No value for parameter 'b'
result = add(2, a=5)
```

Function Scope

When dealing with functions we also have to be aware of the concept of **scope**. What is **scope?**

It's how visible variables and functions are to other parts of our code. Or to put it another way, what is accessible to other parts of our code.

Anything we declare **inside** of a function is not visible **outside** of that function.

Function Scope

```
def add_numbers(x, y):
    total = x + y
    return total

print(total)

def multiply_numbers(x, y):
    total = x + y
    return total
```

will throw an error.
variable out of scope!

different function, so
variable name can be
used again

Function Scope

Proper use of scope allows us to reuse variable names, and ensure that different parts of our code don’t conflict.

It also gives us a means to control the accessibility of our variables. Understanding scoping is an important part of many of Python’s more advanced design patterns.

global statement

- There may, however, be situations where you want to access a variable outside of a function. Assign a value, for example.
- An issue arises with assignment, however. Python will interpret the line `total = x + y` as an instruction to create a new variable scoped to the function, instead of modifying the global variable.
- This is where the `global` statement comes in.

global statement

Below demonstrates the usage of the global statement.

```
total = 0

def add_numbers(x, y):
    # Use the global variable 'total' in this function
    global total
    total = x + y

add_numbers(x=4, y=2)

print(total)
# Prints the value 6
```

References

Python tutorial on functions

https://docs.python.org/tutorial/controlflow.html#defining-functions

Demonstration:

- Functions
- return statement
- Positional argument
- Keyword argument
- Function scope
- global statement