ScopeTreasure Courses

PYTHON COURSES · LESSON PY.8 · CODE SCOPE SERIES

Functions in Python

A function is a named block of code you can run again and again — learn def, parameters, return values, and default arguments.

Introduction

Imagine you need to greet the user in five different places in your program. Without functions, you would copy the same greeting lines five times. If the greeting ever changes, you must fix it in five places — and you will probably miss one. A **function** fixes this: you write the code once, give it a name, and then **call** it whenever you need it.

A function can also take **inputs**, called parameters, and give back an **output**, called a return value. For example, a function `add(a, b)` takes two numbers and returns their sum. You can call it with different numbers each time, and the same code handles all of them.

Functions are the first big step from small scripts to real programs. They make code shorter, easier to read, easier to test, and easier to reuse — in the same file, or in another file entirely, as you will see in PY.9.

(Roman Urdu: Function aik naam wala code ka tukra hai — aik dafa likho, phir jahan chaho us naam se chala lo, baghair dobara likhay.)

The biggest function mistake is not syntax — it is design. A function that prints its answer feels finished, but the answer is then trapped on the screen: no other code can use it. Ask, before you write a function, "will anyone need this value later?" If yes, `return` it; if the job is only to show something, printing is fine. That one question decides most function designs.

Key points

- **Define with `def`, call with the name plus brackets.** `def greet():` defines it; `greet()` runs it.
- **Parameters are inputs listed in the definition.** `def greet(name):` takes one input, called `name`.
- **Arguments are the actual values you pass when calling.** In `greet("Ali")`, `"Ali"` is the argument for the parameter `name`.
- **`return` sends a value back to the caller.** Without `return`, a function gives back `None`.
- **Printing is not returning.** A function that only prints shows output but gives nothing back for later use. If you need the value, return it.
- **Default arguments give a fallback value.** `def greet(name="Guest"):` lets you call `greet()` with no argument.
- **A function runs only when called.** Defining it does nothing on its own — the body waits until the first call.
- **Use clear function names that say what they do.** `calculate_total()` is better than `ct()` or `func1()`.

- **One function, one job.** A function that greets, calculates and saves is three functions hiding in one — split them.
- **A function can return more than one value.** `return total, average` returns a pair, and the caller unpacks it: `total, average = ...`.
- **`*args` accepts any number of arguments.** `def add_all(*numbers):` gathers all extra arguments into a tuple called `numbers`.

Examples

Example 1 — A simple function, defined and called

```
def greet():
    print("Assalam o Alaikum")
    print("Welcome to Python!")

greet()
greet()
```

What you will get:

```
Assalam o Alaikum
Welcome to Python!
Assalam o Alaikum
Welcome to Python!
```

The definition alone prints nothing. Each call runs the whole body again — that is the reuse that makes functions powerful.

Example 2 — Parameters and return values

```
def add(a, b):
    return a + b

result = add(5, 7)
print(result)
print(add(10, 20))

def greet(name):
    return f"Hello {name}, welcome!"

message = greet("Ayesha")
print(message)
```

What you will get:

```
12
30
Hello Ayesha, welcome!
```

`add(5, 7)` returns 12, and the program stores it in `result`. The returned value can be printed, stored, or used in another calculation — unlike a printed value, which is gone as soon as it appears.

Example 3 — Default arguments

```
def greet(name="Guest", greeting="Hello"):
    return f"{greeting} {name}!"

print(greet())
print(greet("Ali"))
print(greet("Ayesha", "Assalam o Alaikum"))
```

What you will get:

```
Hello Guest!
Hello Ali!
Assalam o Alaikum Ayesha!
```

Calling with no arguments uses both defaults. Calling with one argument replaces only the first default. Calling with two replaces both. Defaults must come after any parameters without defaults in the definition.

Example 4 — One function returning two values, used by another

A total and an average belong together, so one function can return both. Another function can then reuse it instead of repeating the maths:

```
def total_and_average(marks):
    total = sum(marks)
    return total, total / len(marks)

total, average = total_and_average([80, 90, 70])
print(total)
print(average)

def is_pass(marks):
    _, average = total_and_average(marks)
    return average >= 50

print(is_pass([80, 90, 70]))
```

What you will get:

```
240
80.0
True
```

`total_and_average([80, 90, 70])` returns the pair (240, 80.0). The underscore `_` in `is_pass` is a polite way of saying "I need the second value, but not the first". Reusing one function inside another is how programs stay short.

Example 5 — Any number of arguments with *args

Sometimes you do not know how many values will arrive — two names today, five tomorrow. `*args` gathers them all:

```
def greet_all(*names):
    for name in names:
        print(f"Hello {name}!")

greet_all("Ali", "Ayesha")

def add_all(*numbers):
    return sum(numbers)

print(add_all(1, 2, 3, 4))
```

What you will get:

```
Hello Ali!
Hello Ayesha!
10
```

Inside `greet_all`, `names` is a tuple of whatever was passed. `*args` must come after any normal parameters, and a function can only have one of them.

Example 6 — Real-world case: a shop bill with an optional discount

A shop's billing function takes a price and a quantity, and sometimes a discount. The discount has a default of 0, so ordinary bills need no third argument:

```
def bill(price, quantity, discount=0):
    total = price * quantity
    return total - total * discount / 100

print(bill(200, 3))
print(bill(200, 3, discount=10))
```

What you will get:

```
600.0
540.0
```

Without a discount, 3 items at 200 cost 600. With 10% off, the saving is 60 and the bill is 540. The answers print as decimals because dividing by 100 in the discount maths turns the result into a float. Naming the discount at the call (`discount=10`) is called a keyword argument — it makes the call read like a sentence.

Quick recap

- `def` defines a function; calling its name runs it.
- Parameters are inputs in the definition; arguments are values at the call.
- `return` sends a value back; without it, the function returns `None`.
- Printing shows output but does not give a value back.
- Default arguments let a call skip values that have a sensible fallback.
- Returning a pair lets the caller unpack two values at once.
- `*args` gathers any number of extra arguments into a tuple.

Common mistakes

1. **Defining a function but never calling it** — nothing happens, and beginners think the code is broken. Fix: after the definition, call the function by name with brackets.
2. **Forgetting `return` and then using the result** — `total = add(5, 7)` gives `None` if `add` only prints. Fix: use `return` when the caller needs the value.
3. **Mixing up parameter order** — `subtract(10, 3)` and `subtract(3, 10)` give different answers. Fix: pass arguments in the same order as the parameters.
4. **Putting a default parameter before a normal one** — `def f(a="x", b):` is an error. Fix: parameters without defaults come first, defaults come last.
5. **Forgetting the colon or the indent in the definition** — `def greet()` without `:` or without an indented body fails. Fix: end the `def` line with a colon and indent the body by 4 spaces.

1. **Returning inside a loop too early** — a `return` in the first round ends the whole function, so the loop never reaches later items. Fix: collect results in a list inside the loop, and return the list after the loop.

2. **Calling a function before defining it** — Python runs top to bottom, so a call above the `def` gives a NameError. Fix: define all functions near the top, and call them afterwards.

Practice

1. Write a function `say_hello(name)` that returns a greeting sentence. Call it with your name and with a friend's name, and print both results.
2. Write a function `area(length, width)` that returns the area of a rectangle. Test it with (5, 4) and (10, 3) and check the answers by hand.
3. Write a function `greet` with a default name of "Guest". Call it with no argument, with one argument, and write down each result.
4. Write a function that takes a list of marks and returns the average. Test it with `[80, 90, 70]` — the answer should be 80.0.
5. Write a function `perimeter(length, width)` that returns the perimeter of a rectangle. Test it with (5, 4) — the answer should be 18.
6. Write a function `biggest(a, b, c)` that returns the largest of three numbers, using `if` inside it. Test it with (4, 9, 2) and with (10, 10, 3).
7. Write a function `count_vowels(word)` that returns how many vowels a word has, using a loop. Test it with `"python"` (answer: 1) and `"education"` (answer: 5).

Practice quiz

1. What does def do in Python?

- Deletes a file
- Defines a new function
- Describes a variable
- Delays the program

Answer: Defines a new function

2. What does return do in a function?

- Prints the answer
- Sends a value back to the caller
- Repeats the function
- Stops Python

Answer: Sends a value back to the caller

3. Why do we write functions?

- To make programs longer
- To reuse code instead of writing the same lines again and again

- Python requires them
- To hide errors

Answer: To reuse code instead of writing the same lines again and again

---

Watch the video lesson: https://youtu.be/v8StTrFzvnk

Read online: https://scopetreasure.com/notes/py-functions · More free tools and notes at https://scopetreasure.com