# What Are Functions?

## Introduction

As Python programs become bigger, writing all the code in one place becomes difficult.

You may find yourself writing the same instructions again and again.

For example, imagine you need to calculate the total price of a product many times:

```
price = 500
quantity = 3

total = price * quantity
print(total)
```

If you need this calculation in 10 different places, you could repeat the code 10 times.

But repeating code creates problems:

- More code to write
- More chances for mistakes
- Harder maintenance
- Difficult debugging
- Changes must be made in multiple places

Python gives us **functions** to solve this problem.

A function allows you to group related instructions together and give them a name.

Then, whenever you need those instructions, you can simply call the function.

---

# 1. What Is a Function?

A **function** is a reusable block of code designed to perform a specific task.

Think of a function like a small machine.

You give it something:

```
Input → Function → Output
```

For example:

```
Number → Calculate Square → Result
```

In Python:

```
def square(number):
    return number * number
```

Now we can use it:

```
print(square(5))
print(square(8))
```

Output:

```
25
64
```

The same function can be reused with different values.

---

# 2. Why Do We Need Functions?

Functions help us write programs that are:

### Reusable

Write the logic once and use it many times.

### Organized

Break a large program into smaller pieces.

### Easier to Understand

Each function can have one clear responsibility.

### Easier to Debug

If something goes wrong, you can focus on a smaller section of code.

### Easier to Maintain

Change the function once instead of changing repeated code everywhere.

---

# 3. A Real Life Example

Think about a calculator.

A calculator may have separate operations:

```
Addition
Subtraction
Multiplication
Division
```

Instead of building one huge piece of logic, we can create separate functions:

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

```
def subtract(a, b):
    return a - b


def multiply(a, b):
    return a * b


def divide(a, b):
    return a / b
```

Now each function has one job.

```
print(add(10, 5))
print(multiply(4, 3))
```

Output:

```
15
12
```

This makes the program easier to understand.

---

# 4. Creating a Function

Python uses the `def` keyword to create a function.

Basic syntax:

```
def function_name():
    # code
```

Example:

```
def greet():
    print("Hello!")
```

Here:

```
def       → tells Python we are defining a function
greet     → function name
()        → parameters would go here
:         → starts the function body
print()   → code that belongs to the function
```

---

# 5. Defining a Function vs Calling a Function

This is an important distinction.

When we write:

```
def greet():
    print("Hello!")
```

we are **defining** the function.

We are telling Python:

> "Create a function called `greet`."

But Python does not run the function just because we defined it.

To execute it, we need to **call** it.

```
greet()
```

Complete example:

```
def greet():
    print("Hello!")


greet()
```

Output:

```
Hello!
```

---

# 6. Function Calls

A function is called by writing its name followed by parentheses.

```
greet()
```

If the function requires information, we put that information inside the parentheses.

```
square(5)
```

The value `5` is passed to the function.

---

# 7. A Function Without Parameters

A function does not always need input.

Example:

```
def show_message():
    print("Welcome to haas.dev!")
```

Call it:

```
show_message()
```

Output:

```
Welcome to haas.dev!
```

This function always performs the same task.

---

# 8. Functions With Parameters

Sometimes a function needs information to perform its task.

We can give it **parameters**.

Example:

```
def greet(name):
    print("Hello", name)
```

Call it:

```
greet("Hafsa")
```

Output:

```
Hello Hafsa
```

Here:

```
name
```

is a **parameter**.

And:

```
"Hafsa"
```

is an **argument**.

---

# 9. Parameter vs Argument

These two terms are related but different.

### Parameter

A parameter is the variable written in the function definition.

```
def greet(name):
```

`name` is a parameter.

### Argument

An argument is the actual value passed when calling the function.

```
greet("Hafsa")
```

`"Hafsa"` is an argument.

Think of it like this:

```
Parameter = placeholder
Argument  = actual value
```

Example:

```
def introduce(name):
    print("My name is", name)


introduce("Hafsa")
introduce("Asim")
```

Output:

```
My name is Hafsa
My name is Asim
```

The same function works with different arguments.

---

# 10. Multiple Parameters

A function can have multiple parameters.

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

Call it:

```
add(10, 20)
```

Output:

```
30
```

Here:

```
a → 10
b → 20
```

Another call:

```
add(100, 50)
```

Output:

```
150
```

---

# 11. Returning a Value

A function can calculate something and give the result back.

Python uses the `return` keyword for this.

Example:

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

Now:

```
result = add(10, 20)

print(result)
```

Output:

```
30
```

The function calculates:

```
a + b
```

and returns the result.

---

# 12. `print()` vs `return`

This is one of the most important concepts when learning functions.

### `print()`

Displays something on the screen.

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

### `return`

Sends a value back to the code that called the function.

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

With `return`, we can store or reuse the result.

```
result = add(10, 20)

double = result * 2

print(double)
```

Output:

```
60
```

A useful way to remember:

```
print()  → show the result
return   → give the result back
```

---

## 13. What Happens After `return`?

When Python reaches `return`, the function immediately stops.

Example:

```
def test():
    print("Start")
    return
    print("End")
```

Call:

```
test()
```

Output:

```
Start
```

`"End"` is never printed because the function already returned.

---

## 14. Functions Can Be Reused

One of the biggest benefits of functions is reuse.

Without a function:

```
print("Hello Hafsa")
print("Hello Asim")
print("Hello Ali")
```

With a function:

```
def greet(name):
    print("Hello", name)


greet("Hafsa")
greet("Asim")
greet("Ali")
```

We write the logic once and reuse it.

---

# 15. Functions Make Large Programs Easier

Imagine building a website.

You might need different tasks:

```
validate_user()
calculate_total()
send_email()
save_user()
create_report()
```

Instead of putting everything into one huge block, each task can have its own function.

Example:

```
def calculate_total(price, quantity):
    return price * quantity


def apply_discount(total, discount):
    return total - discount


def show_total(total):
    print("Final total:", total)
```

Now the program becomes easier to understand.

---

# 16. A Function Should Usually Have One Clear Job

A good function usually performs one specific task.

Instead of:

```
def do_everything():
    # create user
    # calculate salary
    # send email
    # create report
    # save database
```

Prefer smaller functions:

```
def create_user():
    pass


def calculate_salary():
    pass


def send_email():
    pass


def create_report():
    pass
```

This principle is often described as:

> **One function, one clear responsibility.**

---

# 17. Function Naming

Function names should clearly describe what the function does.

Good names:

```
calculate_total()
get_user()
send_email()
validate_password()
calculate_average()
```

Poor names:

```
do_it()
thing()
abc()
function1()
```

Use `snake_case` for Python function names.

```
calculate_total()
get_user_name()
check_password()
```

---

# 18. Functions and Variables

Functions can work with variables.

Example:

```
def calculate_area(length, width):
    area = length * width
    return area
```

Then:

```
result = calculate_area(10, 5)

print(result)
```

Output:

```
50
```

The variable `area` belongs to the function.

---

# 19. Local Variables

A variable created inside a function is generally a **local variable**.

Example:

```
def calculate():
    result = 10 + 20
    print(result)
```

Here `result` exists inside the function.

Trying to use it outside:

```
def calculate():
    result = 10 + 20


calculate()

print(result)
```

will cause an error because `result` is not available in that outside scope.

This introduces an important concept called **scope**, which will be covered in more detail when we study function scope.

---

# 20. Functions Can Call Other Functions

One function can call another function.

Example:

```
def greet():
    print("Hello!")


def start_program():
    greet()
    print("Program started")


start_program()
```

Output:

```
Hello!
Program started
```

This allows larger programs to be built from smaller reusable pieces.

---

# 21. A Practical Example

Imagine a shopping application.

We want to calculate the total cost.

```
def calculate_total(price, quantity):
    return price * quantity
```

Now:

```
total = calculate_total(500, 3)

print("Total:", total)
```

Output:

```
Total: 1500
```

We can reuse the same function:

```
print(calculate_total(100, 2))
print(calculate_total(750, 4))
print(calculate_total(1200, 1))
```

Output:

```
200
3000
1200
```

The function contains the calculation logic once.

---

# 22. Functions in a Real Project

Suppose haas.dev needs to display information about a resource.

We might have functions like:

```
def get_resource_title():
    return "Python Functions"


def get_resource_category():
    return "Python"


def display_resource():
    print(get_resource_title())
    print(get_resource_category())
```

Then:

```
display_resource()
```

Output:

```
Python Functions
Python
```

In real applications, functions can become the building blocks of much larger systems.

---

# 23. A Function as a Building Block

Think of a program like a house.

A house is not built as one giant piece.

It has:

```
Kitchen
Bedroom
Bathroom
Garage
```

Similarly, a large program can be divided into:

```
Login Function
Payment Function
Search Function
Database Function
Email Function
```

Each function handles a smaller part of the application.

Together, they create the complete program.

---

# 24. Common Beginner Mistakes

## Mistake 1: Forgetting to Call the Function

```
def greet():
    print("Hello")
```

Nothing is displayed because the function was only defined.

Correct:

```
greet()
```

---

## Mistake 2: Forgetting Parentheses

Incorrect:

```
greet
```

Correct:

```
greet()
```

---

### Mistake 3: Forgetting the Colon

Incorrect:

```
def greet()
```

Correct:

```
def greet():
```

---

### Mistake 4: Incorrect Indentation

Incorrect:

```
def greet():
print("Hello")
```

Correct:

```
def greet():
    print("Hello")
```

Python uses indentation to identify the function body.

---

### Mistake 5: Confusing `print()` and `return`

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

This displays the result but does not return it for further use.

If you need the value:

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

---

### Mistake 6: Wrong Number of Arguments

If a function has two parameters:

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

calling:

```
add(10)
```

will cause an error because one argument is missing.

Correct:

```
add(10, 20)
```

---

# 25. How to Think Before Creating a Function

When solving a problem, ask:

### Step 1: What task am I repeating?

Example:

```
Calculate total price
```

### Step 2: What information does the task need?

```
price
quantity
```

### Step 3: What should the task give back?

```
total price
```

### Step 4: Create the function

```
def calculate_total(price, quantity):
    return price * quantity
```

This way, you design the function around the problem instead of randomly creating functions.

---

# 26. Function Thinking Framework

Whenever you see a repeated task, think:

```
What is the task?
       ↓
What input does it need?
       ↓
What processing should happen?
       ↓
What output should it produce?
       ↓
Can I reuse this logic?
```

```
        ↓
Create a function
```

For example:

```
Task
Calculate average marks

Input
marks

Processing
Add marks ÷ number of marks

Output
average

Function
calculate_average()
```

---

# 27. Mini Project: Student Result Calculator

Create a function that calculates the average of three marks.

```
def calculate_average(mark1, mark2, mark3):
    total = mark1 + mark2 + mark3
    average = total / 3
    return average
```

Use it:

```
result = calculate_average(80, 75, 90)

print("Average:", result)
```

Output:

```
Average: 81.66666666666667
```

The function handles the calculation, while the rest of the program handles how the result is displayed.

---

# 28. 5 Minute Challenge

Create a function called:

```
calculate_rectangle_area()
```

It should:

1. Accept `length`
2. Accept `width`
3. Calculate the area
4. Return the result

Expected usage:

```
area = calculate_rectangle_area(10, 5)

print(area)
```

Expected output:

```
50
```

Then test it with different values.

---

# 29. Practice Exercises

## Exercise 1

Create a function:

```
greet_user(name)
```

It should print:

```
Welcome, Hafsa!
```

when called with:

```
greet_user("Hafsa")
```

## Exercise 2

Create:

```
square(number)
```

It should return the square of a number.

Example:

```
print(square(6))
```

Output:

```
36
```

### Exercise 3

Create:

```
is_even(number)
```

It should return whether a number is even.

### Exercise 4

Create:

```
calculate_total(price, quantity)
```

It should return the total price.

### Exercise 5

Create:

```
calculate_discount(price, discount)
```

It should calculate and return the final price after discount.

---

# 30. Mini Quiz

### Question 1

Which keyword is used to define a function?

A. `function`

B. `def`

C. `func`

D. `create`

### Question 2

What is the difference between a parameter and an argument?

A. They are exactly the same

B. A parameter is a placeholder; an argument is the actual value

C. An argument defines the function

D. A parameter is always a number

### Question 3

What does `return` do?

A. Prints text automatically

B. Stops the entire Python program

C. Sends a value back from the function

D. Creates a variable

**Answers**

```
1. B
2. B
3. C
```

---

# 31. Interview Questions

### Beginner

1. What is a function in Python?
2. Why are functions useful?
3. How do you define a function?
4. How do you call a function?
5. What is a parameter?
6. What is an argument?
7. What does `return` do?
8. What is the difference between `print()` and `return`?

### Intermediate

9. Can a function have multiple parameters?
10. Can one function call another function?
11. What happens when Python reaches `return`?
12. What is a local variable?
13. Why should functions usually have a clear responsibility?
14. What happens if you provide the wrong number of arguments?

---

# 32. Function Cheat Sheet

```
# Define a function
def greet():
    print("Hello")


# Call a function
```

```
greet()


# Function with a parameter
def greet(name):
    print("Hello", name)


# Call with an argument
greet("Hafsa")


# Multiple parameters
def add(a, b):
    return a + b


# Store returned value
result = add(10, 20)


# Use returned value
print(result)
```

Remember:

```
def        → define a function
function() → call a function
parameter  → input placeholder
argument   → actual input value
return     → send a value back
```

---

# 33. Key Takeaways

- A function is a reusable block of code.
- Functions help reduce repeated code.
- Use `def` to define a function.
- Call a function using its name and parentheses.
- Parameters define the inputs a function expects.
- Arguments are the actual values passed to those parameters.
- `return` sends a value back from a function.
- `print()` displays a value but does not return it.

- Functions can have zero, one, or multiple parameters.
- A function can call another function.
- Good functions usually have one clear responsibility.
- Functions are fundamental building blocks of larger Python programs.

The important idea is:

```
Don't repeat the same logic everywhere.

Identify the task.
Turn it into a function.
Give it inputs.
Let it do the work.
Return the result.
Reuse it whenever needed.
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

## Next Step

After understanding what functions are, the next concepts to learn are:

```
Function Parameters and Arguments
        ↓
Return Values
        ↓
Default Arguments
        ↓
Keyword Arguments
        ↓
Variable Length Arguments
        ↓
Scope
        ↓
Lambda Functions
        ↓
Higher Order Functions
```

These concepts will turn basic functions into powerful tools for building real Python programs.